

Java™ Programming Language

SL-275

Student Guide **With Instructor Notes**



Sun Microsystems, Inc.
MS BRM01-209
500 Eldorado Boulevard
Broomfield, Colorado 80021
U.S.A.

Revision D, April 2000

Copyright 2000 Sun Microsystems, Inc., 901 San Antonio Road, California 94303 U.S.A. All rights reserved.

This product or document is protected by copyright and distributed under licenses restricting its use, copying, distribution, and decompilation. No part of this product or document may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any.

Parts of this product may be derived from Berkeley BSD systems, licensed from the University of California. UNIX is a registered trademark in the U.S. and other countries, exclusively licensed through X/Open Company Ltd. Third-party software, including font technology in this product, is protected by copyright and licensed from Sun's suppliers.

RESTRICTED RIGHTS: Use, duplication, or disclosure by the U.S. Government is subject to restrictions of FAR 52.227-14(g)(2)(6/87) and FAR 52.227-19(6/87), or DFAR 252.227-7015(b)(6/95) and DFAR 227.7202-1(a).

Sun, Sun Microsystems, the Sun logo, Solstice, Java, JavaBeans, JavaChip, Java HotSpot, JavaOS, JavaSoft, JDBC, JDK, JVM, OpenWindows, Write Once, Run Anywhere and Solaris are trademarks or registered trademarks of Sun Microsystems, Inc. in the U. S. and other countries.

UNIX is a registered trademark in the U.S. and other countries, exclusively licensed through X/Open Company, Ltd.

All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U. S. and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

Netscape Navigator is a trademark of Netscape Communications Corporation.

The OPEN LOOK[®] and Sun[™] Graphical User Interfaces were developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees who implement OPEN LOOK GUIs and otherwise comply with Sun's written license agreements.

X Window System is a product of the X Consortium, Inc.

DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.



Please
Recycle



Adobe PostScript

Contents

About This Course	xix
Course Goal	xix
Course Overview	xx
Course Map	xxi
Module-by-Module Overview	xxii
Course Objectives.....	xxvi
Skills Gained by Module.....	xxvii
Guidelines for Module Pacing	xxviii
Topics Not Covered	xxix
How Prepared Are You?	xxx
Introductions	xxxii
How to Use Course Materials	xxxii
Course Icons and Typographical Conventions	xxxiv
Course Icons.....	xxxiv
Typographical Conventions	xxxv
Notes to the Instructor.....	xxxvii
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
A Basic Java Application.....	1-8
TestGreeting.java	1-8
Greeting.java.....	1-8
TestGreeting Described	1-9
Greeting Described	1-12
Compiling and Running TestGreeting.....	1-14
Troubleshooting the Compilation	1-16
Java - Behind the Scenes.....	1-18
The Java Runtime Environment.....	1-18
The Java Virtual Machine	1-19

Garbage Collection	1-22
Code Security.....	1-24
Exercise: Performing Basic Tasks.....	1-30
Preparation.....	1-30
Tasks	1-30
Check Your Progress	1-31
Think Beyond	1-32
Object-Oriented Programming	2-1
Objectives	2-1
Relevance.....	2-2
What Is Object-Oriented Programming?.....	2-3
Analysis and Design.....	2-5
Analysis and Design Example	2-7
Abstraction.....	2-8
Classes as Blueprints for Objects	2-9
Declaring Java Classes.....	2-10
Declaring Attributes	2-11
Declaring Methods.....	2-12
Accessing Object Members.....	2-14
Information Hiding.....	2-15
The Problem.....	2-15
The Solution	2-16
Encapsulation	2-17
Declaring Constructors	2-18
The Default Constructor	2-20
Source File Layout	2-21
Software Packages	2-23
The package Statement.....	2-24
The import Statement.....	2-26
Directory Layout and Packages	2-28
Development	2-29
Deployment	2-30
Terminology Recap.....	2-31
Using the Java Technology API Documentation.....	2-32
Exercise: Using Objects and Classes.....	2-35
Preparation.....	2-35
Tasks	2-35
Exercise Summary.....	2-36
Check Your Progress	2-37
Think Beyond	2-38
Identifiers, Keywords, and Types.....	3-1
Objectives	3-1
Relevance.....	3-2
Comments	3-3
Semicolons, Blocks, and Whitespace.....	3-4

Identifiers	3-7
Java Keywords.....	3-9
Basic Java Types	3-10
Primitive Types	3-10
Logical – boolean.....	3-11
Textual – char and String	3-12
Integral – byte, short, int, and long.....	3-14
Floating Point – float and double.....	3-17
Variables, Declarations, and Assignments.....	3-19
Java Reference Types.....	3-20
Constructing and Initializing Objects	3-21
Memory Allocation and Layout.....	3-22
Explicit Attribute Initialization	3-23
Executing the Constructor	3-24
Variable Assignment	3-25
This Is Not the Whole Story	3-26
Assignment of Reference Types.....	3-27
Pass-by-Value	3-29
The this Reference.....	3-32
Java Coding Conventions	3-35
Exercise: Using Objects	3-38
Preparation.....	3-38
Tasks	3-38
Exercise Summary.....	3-39
Check Your Progress	3-40
Think Beyond	3-41
Expressions and Flow Control.....	4-1
Objectives	4-1
Relevance.....	4-2
Expressions	4-3
Variables and Scope.....	4-3
Variable Scope Example.....	4-5
Variable Initialization.....	4-6
Operators.....	4-7
Logical Operators.....	4-8
Short-Circuit Logical Operators.....	4-9
Bitwise Logical Operators.....	4-10
Right-Shift Operators >> and >>>.....	4-11
Left-Shift Operator (<<)	4-13
Shift Operator Examples.....	4-14
String Concatenation With +	4-15
Casting.....	4-16
Promotion and Casting of Expressions.....	4-18

Branching Statements	4-20
if, else Statements.....	4-20
switch Statement	4-22
Looping Statements	4-25
for Loops	4-25
while Loops.....	4-27
do Loops	4-29
Special Loop Flow Control	4-31
Example.....	4-33
Exercise: Using Expressions	4-34
Preparation.....	4-34
Tasks	4-34
Exercise Summary.....	4-35
Check Your Progress	4-36
Think Beyond	4-37
Arrays	5-1
Objectives	5-1
Relevance.....	5-2
Declaring Arrays.....	5-3
Creating Arrays.....	5-5
Initializing Arrays.....	5-7
Multi-Dimensional Arrays.....	5-9
Array Bounds	5-11
Array Resizing.....	5-12
Copying Arrays.....	5-13
Exercise: Using Arrays	5-14
Preparation.....	5-14
Tasks	5-14
Exercise Summary.....	5-15
Check Your Progress	5-16
Think Beyond	5-17
Inheritance.....	6-1
Objectives	6-1
Relevance.....	6-2
Subclassing.....	6-3
The is a Relationship.....	6-3
Single Inheritance.....	6-6
Constructors Are Not Inherited.....	6-8
Polymorphism	6-9
Heterogeneous Collections.....	6-11
Polymorphic Arguments	6-13
The instanceof Operator.....	6-14
Casting Objects.....	6-16
The has a Relationship	6-18
Access Control.....	6-19

Overloading Method Names.....	6-20
Overloading Constructors	6-22
Overriding Methods.....	6-24
Invoking Overridden Methods.....	6-28
Rules About Overridden Methods.....	6-28
The super Keyword	6-30
Invoking Parent Class Constructors.....	6-32
Constructing and Initializing Objects: A Slight Reprise.....	6-34
Implications of the Initialization Process	6-37
The Object Class	6-39
The == Operator Compared With the equals Method.....	6-40
Example.....	6-41
The toString Method	6-43
Wrapper Classes.....	6-44
Exercise: Using Objects and Classes.....	6-46
Preparation.....	6-46
Tasks	6-46
Exercise Summary.....	6-47
Check Your Progress	6-48
Think Beyond	6-49
Advanced Class Features	7-1
Objectives	7-1
Relevance.....	7-2
The static Keyword.....	7-3
Class Attributes	7-4
Class Methods	7-6
Static Initializers.....	7-8
Implementing the Singleton Design Pattern.....	7-10
The final Keyword	7-12
Final Classes.....	7-12
Final Methods.....	7-13
Final Variables.....	7-14
Exercise: Working With the static and final Keywords.....	7-15
Preparation.....	7-15
Tasks	7-15
Exercise Summary.....	7-16
Abstract Classes.....	7-17
The Scenario.....	7-17
The Problem.....	7-18
The Solution.....	7-19
Template Method Design Pattern.....	7-21
Interfaces	7-22
The Flyer Example	7-23
Multiple Interface Example	7-27
Inner Classes	7-29

Properties of Inner Classes	7-34
Exercise: Working With Interfaces and Abstract Classes	7-37
Preparation.....	7-37
Tasks	7-37
Exercise Summary.....	7-38
Check Your Progress	7-39
Think Beyond	7-40
Exceptions.....	8-1
Objectives	8-1
Relevance.....	8-2
Exceptions	8-3
Introduction	8-3
Example.....	8-5
Exception Handling	8-6
Introduction	8-6
try and catch Statements.....	8-7
The Call Stack Mechanism.....	8-8
finally Statement	8-9
Example Revisited	8-11
Exception Categories	8-13
Common Exceptions.....	8-15
The Handle or Declare Rule	8-17
Method Overriding and Exceptions.....	8-19
Creating Your Own Exceptions	8-23
Introduction	8-23
Example.....	8-24
Exercise: Handling and Creating Exceptions.....	8-26
Preparation.....	8-26
Tasks	8-26
Exercise Summary.....	8-27
Check Your Progress	8-28
Think Beyond	8-29
Text-Based Applications.....	9-1
Objectives	9-1
Relevance.....	9-2
Command-Line Arguments	9-3
System Properties.....	9-4
The Properties Class	9-5
Console I/O	9-7
Writing to Standard Output.....	9-8
Reading From Standard Input	9-9
Files and File I/O	9-11
Creating a New File Object	9-12
File Tests and Utilities.....	9-14

File Stream I/O.....	9-16
File Output.....	9-18
Exercise: File Input and Output.....	9-20
Preparation.....	9-20
Tasks.....	9-20
The Math Class.....	9-21
The String Class.....	9-23
The StringBuffer Class.....	9-25
The Collections API.....	9-27
A Set Example.....	9-29
A List Example.....	9-30
Iterators.....	9-31
Maps.....	9-33
A Map Example.....	9-34
Sorting.....	9-35
Sorting Examples.....	9-37
Collections in JDK 1.1.....	9-40
Exercise: Using Collections to Represent Aggregation.....	9-41
Preparation.....	9-41
Tasks.....	9-41
Using the javadoc Tool.....	9-42
Documentation Tags.....	9-43
Example.....	9-44
Deprecation.....	9-47
Using the jar Tool.....	9-51
Exercise: Building a System.....	9-52
Preparation.....	9-52
Tasks.....	9-52
Exercise Summary.....	9-53
Check Your Progress.....	9-54
Think Beyond.....	9-55
Building Java GUIs.....	10-1
Objectives.....	10-1
Relevance.....	10-2
The AWT.....	10-3
The java.awt Package.....	10-5
Building Graphical User Interfaces.....	10-6
Containers.....	10-6
Positioning Components.....	10-8
Component Sizing.....	10-9
Frames.....	10-10
Panels.....	10-12
Container Layouts.....	10-14
Layout Managers.....	10-15
Default Layout Managers.....	10-16

A Simple FlowLayout Example.....	10-17
The main Method	10-18
Layout Managers	10-20
FlowLayout Manager.....	10-20
BorderLayout Manager	10-24
GridLayout Manager.....	10-29
CardLayout Manager.....	10-33
GridBagLayout Manager	10-36
Creating Panels and Complex Layouts	10-37
Drawing in AWT.....	10-39
Exercise: Building Java GUIs.....	10-41
Preparation.....	10-41
Tasks	10-41
Exercise Summary.....	10-42
Check Your Progress	10-43
Think Beyond	10-44
GUI Event Handling.....	11-1
Objectives	11-1
Relevance.....	11-2
What Is an Event?	11-3
Event Sources.....	11-4
Event Handlers.....	11-4
Java 2 SDK Event Model	11-5
Delegation Model.....	11-5
GUI Behavior	11-9
Event Categories	11-9
Complex Example.....	11-12
Multiple Listeners	11-16
Event Adapters.....	11-18
Event Handling Using Inner Classes	11-20
Event Handling Using Anonymous Classes.....	11-21
Exercise: Working With Events	11-22
Preparation.....	11-22
Tasks	11-22
Exercise Summary.....	11-23
Check Your Progress	11-24
Think Beyond	11-25
Introduction to Java Applets.....	12-1
Objectives	12-1
Relevance.....	12-2
What Is an Applet?	12-3
Loading an Applet.....	12-3
Applet Security Restrictions	12-5

Writing an Applet	12-7
Applet Class Hierarchy	12-7
Key Applet Methods	12-8
Applet Methods and the Applet Life Cycle	12-9
init	12-9
start	12-10
stop	12-10
Applet Display	12-11
The paint Method and the Graphics Object	12-12
AWT Painting	12-13
The paint Method	12-14
The repaint Method	12-14
The update Method	12-14
Method Interaction	12-15
Applet Display Strategies	12-16
An Example Paint Model	12-18
What Is the appletviewer?	12-21
Starting Applets With the appletviewer	12-22
Synopsis	12-23
Example	12-23
The applet Tag	12-24
Syntax	12-24
Description	12-25
Additional Applet Features	12-26
A Simple Image Test	12-28
Audio Clips	12-29
Playing a Clip	12-29
A Simple Audio Test	12-30
Looping an Audio Clip	12-31
Loading an Audio Clip	12-31
Playing an Audio Clip	12-32
Stopping an Audio Clip	12-32
A Simple Audio Looping Test	12-33
Mouse Input	12-34
A Simple Mouse Test	12-35
Reading Parameters	12-36
Exercise: Creating Applets	12-38
Preparation	12-38
Tasks	12-38
Exercise Summary	12-39
Check Your Progress	12-40
Think Beyond	12-41

GUI-Based Applications	13-1
Objectives	13-1
Relevance.....	13-2
AWT Components	13-3
Component Events.....	13-4
How to Create a Menu	13-5
The Help Menu.....	13-5
Creating a MenuBar.....	13-6
Creating a Menu	13-7
Creating a MenuItem.....	13-8
Creating a CheckboxMenuItem.....	13-9
Controlling Visual Aspects.....	13-10
Colors.....	13-10
Fonts.....	13-12
The Toolkit Class.....	13-14
Printing.....	13-15
Dual-Purpose Code.....	13-17
Example.....	13-18
Discussion of Dual-Purpose Code.....	13-22
Swing	13-23
Exercise: Building GUI-Based Applications.....	13-24
Preparation.....	13-24
Tasks	13-24
Exercise Summary.....	13-25
Check Your Progress	13-26
Think Beyond	13-27
Threads	14-1
Objectives	14-1
Relevance.....	14-2
Threads	14-3
What Are Threads?	14-3
Threads in Java Programming	14-4
Three Parts of a Thread	14-4
Creating the Thread.....	14-6
Starting the Thread	14-9
Thread Scheduling.....	14-10
Basic Control of Threads.....	14-13
Terminating a Thread.....	14-13
Testing a Thread.....	14-15
Accessing Thread Priority	14-15
Putting Threads on Hold	14-16
Other Ways to Create Threads.....	14-19
Selecting a Way to Create Threads.....	14-20

Exercise: Using Basic Threads	14-22
Preparation.....	14-22
Tasks	14-22
Using synchronized in Java Technology	14-23
The Problem.....	14-23
The Object Lock Flag	14-26
Releasing the Lock Flag.....	14-30
synchronized – Putting It Together	14-31
Thread States	14-33
Deadlock.....	14-34
Thread Interaction – wait and notify.....	14-35
Scenario	14-35
The Problem.....	14-36
The Solution	14-36
Thread Interaction.....	14-37
wait and notify.....	14-37
Thread States	14-39
Monitor Model for Synchronization	14-40
Putting It Together.....	14-41
Producer	14-42
Consumer	14-43
SyncStack Class	14-44
Complete Code.....	14-48
Thread Control in Java 2 SDK.....	14-53
The suspend and resume Methods.....	14-53
The stop Method	14-55
Proper Thread Control	14-56
Exercise: Using Multithreaded Programming.....	14-59
Preparation.....	14-59
Tasks	14-59
Exercise Summary.....	14-60
Check Your Progress	14-61
Think Beyond	14-62
Advanced I/O Streams	15-1
Objectives	15-1
Relevance.....	15-2
I/O Fundamentals	15-3
Byte Streams	15-5
InputStream Methods.....	15-5
OutputStream Methods	15-7
Character Streams	15-8
Reader Methods.....	15-8
Writer Methods.....	15-10
Node Streams.....	15-11
A Reader/Writer Example	15-12

A Buffered Reader/Writer Example	15-13
I/O Stream Chaining.....	15-14
Processing Streams	15-15
Processing Streams as Decorators	15-16
Basic Byte Stream Classes	15-18
FileInputStream and FileOutputStream.....	15-19
BufferedInputStream and BufferedOutputStream...	15-19
PipedInputStream and PipedOutputStream.....	15-19
DataInputStream and DataOutputStream.....	15-20
Basic Character Stream Classes.....	15-21
InputStreamReader and OutputStreamWriter.....	15-22
Byte and Character Conversions	15-22
Using Other Character Encoding	15-22
FileReader and FileWriter.....	15-23
BufferedReader and BufferedWriter	15-23
StringReader and StringWriter	15-23
PipedReader and PipedWriter	15-23
URL Input Streams	15-24
Opening an Input Stream	15-25
Random Access Files	15-26
Creating a Random Access File.....	15-26
Accessing Information.....	15-28
Appending Information.....	15-29
Serialization	15-30
Object Graphs	15-31
Writing and Reading an Object Stream	15-32
Writing.....	15-32
Reading.....	15-33
Exercise: Getting Acquainted With I/O	15-34
Preparation.....	15-34
Tasks	15-34
Exercise Summary.....	15-35
Check Your Progress	15-36
Think Beyond	15-37
Networking	16-1
Objectives	16-1
Relevance.....	16-2
Networking	16-3
Sockets	16-3
Setting up the Connection	16-4
Addressing the Connection	16-5
Port Numbers	16-6
Java Networking Model.....	16-7
Minimal TCP/IP Server	16-8
Minimal TCP/IP Client.....	16-9

Exercise: Using Socket Programming	16-10
Preparation.....	16-10
Tasks	16-10
Exercise Summary.....	16-11
Check Your Progress	16-12
Think Beyond	16-13
Elements of Advanced Java Programming	A-1
Objectives	A-1
Introduction to Two- and Three-Tier Architectures	A-2
The Three-Tier Architecture	A-3
Three -Tier Client-Server Definition	A-3
A Database Frontend.....	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-10
Creating an RMI Application.....	A-11
CORBA	A-12
The Java IDL	A-13
RMI Compared With CORBA.....	A-14
The JavaBeans Component Model	A-15
Bean Architecture.....	A-16
Bean Introspection	A-18
A Sample Bean Interaction	A-19
The Beans Development Kit (BDK).....	A-19
JAR Files	A-20
Check Your Progress	A-21
JDK 1.0 GUI Event Model	B-1
Additional Resources	B-2
Event Handling	B-3
Event Handling Before JDK 1.1.....	B-3
Event Handling in JDK 1.1	B-3
JDK 1.0 Event Model Compared to Java 2 SDK Event Model... B-4	
Hierarchical Model (JDK 1.0)	B-4
Converting 1.0 Event Handling to 1.1.....	B-6
Making a Component a Listener	B-10
The AWT Component Library	C-1
Features of the AWT.....	C-1
Button.....	C-2
Checkbox	C-3
Checkbox Group – Radio Buttons	C-4

Choice	C-5
Canvas	C-6
Label.....	C-8
TextField.....	C-9
TextArea	C-11
Text Components	C-12
List.....	C-13
Dialog.....	C-14
FileDialog	C-16
ScrollPane.....	C-18
PopupMenu	C-19
Using the GridBagLayout	D-1
Layout Managers	D-2
The GridBagLayout	D-4
The GridBagConstraints Class.....	D-8
Designing With GridBagLayout.....	D-10
Design Steps.....	D-10
Example.....	D-18
RELATIVE and REMAINDER	D-22
Java Foundation Classes	E-1
Objectives	E-1
Additional Resources	E-2
Swing Introduction.....	E-3
Pluggable Look and Feel.....	E-4
Swing Architecture	E-5
The Swing Hierarchy.....	E-6
Swing Components.....	E-7
Basic Swing Application	E-9
HelloSwing	E-10
Importing Swing Packages.....	E-12
Choosing the Look and Feel.....	E-12
Setting up Windows	E-13
Setting up Swing Components	E-14
Supporting Assistive Technologies	E-15
Building a Swing GUI	E-16
The JComponent Class	E-21
Java Native Interface	F-1
Native Methods.....	F-1
Native HelloWorld.....	F-2
Defining Native Methods	F-2
Calling Native Methods.....	F-3
The javah Utility	F-4
Coding C Functions for Native Methods	F-5
Putting It Together.....	F-6

Passing Information to a Native Method	F-7
Passing a Java Primitive as an Argument	F-7
Accessing a Java Primitive as an Object Data Member	F-8
Accessing Strings	F-11
Summary	F-14
UML Modeling and Java	G-1
What Is UML?	G-1
Package Diagrams.....	G-2
Class Diagrams.....	G-4
Class Nodes.....	G-4
Inheritance and Interface Implementation.....	G-6
Association and Aggregation.....	G-7
Object Diagrams	G-9
State Diagrams.....	G-10
Transitions.....	G-11
Other UML Elements.....	G-12
Stereotypes	G-12
Diagram Annotation.....	G-12
Index	Index-1

About This Course

Course Goal

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 and applets. In this course, you will learn Java programming language syntax and object-oriented concepts, as well as more sophisticated features of the Java runtime environment, such as support for graphical user interfaces (GUIs), multithreading, and networking. This course covers prerequisite knowledge to prepare you for the Sun Microsystems™ Certified Java Programmer and Certified Java Developer examinations. Please review the Web site:

http://suned.sun.com/usa/cert_test.html
for details about the exam.

- ✓ *It is important to make the students understand that they will need to study for the SCJP exam and that this course is NOT a substitute for proper studying.*
- ✓ *Use this module to get the students excited about this course.*
- ✓ *With regard to the overheads: To avoid confusion among the students, it is very important to tell them that the page numbers on the overheads have no relation to the page numbers in their course materials. They should use the title of each overhead as a reference.*
- ✓ *The strategy provided by the “About This Course” is to introduce students to the course before they introduce themselves to you and one another. By familiarizing them with the content of the course first, their introductions will have more meaning in relation to the course prerequisites and objectives.*
- ✓ *Use this introduction to the course to determine how well students are equipped with the prerequisite knowledge and skills.*



Course Overview

This course covers the following areas:

- Syntax of the Java programming language
- Object-oriented concepts as they apply to the Java programming language
- Graphical user interface (GUI) programming
- Applet creation
- Multithreading
- Networking

Course Overview

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

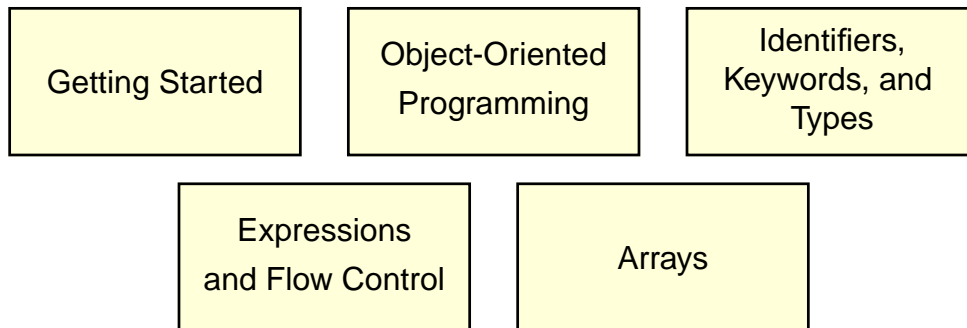
The audience for this course includes people who are familiar with implementing elementary programming concepts using the Java programming language or other languages. This is the follow-up course to *Java Programming for Non-Programmers* (SL-110).

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. The course material code examples were run in the Solaris™ Operating Environment and in the Microsoft Windows operating environment; therefore, the graphics in this guide have both a Motif and a Microsoft Windows GUI. The content of this course is applicable to all Java operating system ports.

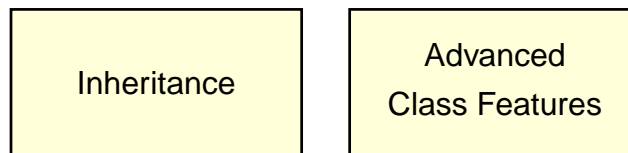
Course Map

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

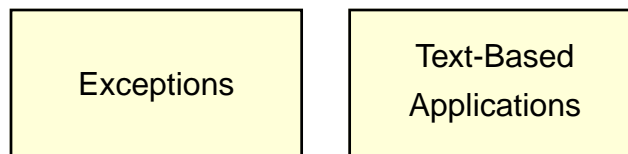
The Java Programming Language Basics



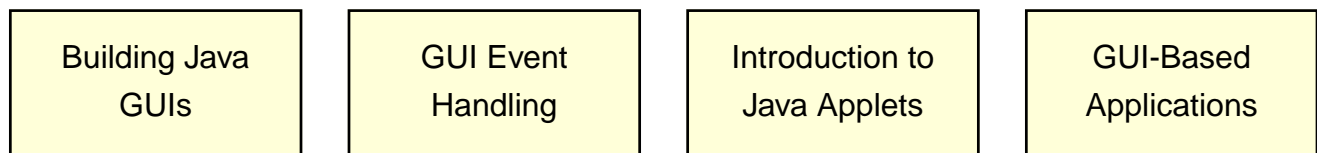
More Object-Oriented Programming



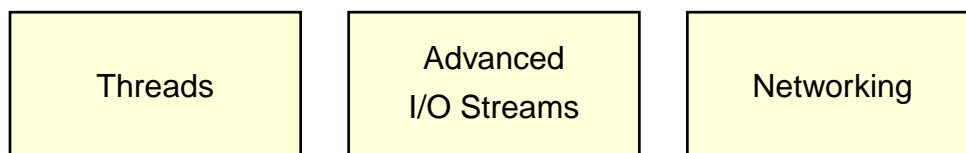
Building Applications



Developing Graphical User Interfaces



Advanced Java Programming





Module-by-Module Overview

- Module 1 – "Getting Started"
- Module 2 – "Object-Oriented Programming"
- Module 3 – "Identifiers, Keywords, and Types"
- Module 4 – "Expressions and Flow Control"
- Module 5 – "Arrays"
- Module 6 – "Inheritance"
- Module 7 – "Advanced Class Features"
- Module 8 – "Exceptions"
- Module 9 – "Text-Based Applications"

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 Java applications. This module also reviews the concepts of classes and packages and some of the more commonly used Java packages.

- Module 2 – "Object-Oriented Programming"

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

Module-by-Module Overview

- 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 – "Inheritance"

This module takes the introduction to Java object concepts described in Module 2 to the next level, including a discussion on subclassing, overloading, and overriding.

- Module 7 – "Advanced Class Features"

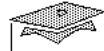
This module completes the Java object-oriented programming model by discussing the concepts of abstract classes, interfaces, and inner classes.

- Module 8 – "Exceptions"

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

- Module 9 – "Text-Based Applications"

This module introduces topics that are useful in implementing large, text-based applications; such as console and file I/O, collections and sorting algorithms, and the use of two Java 2 SDK tools: `javadoc` and `jar`.



Module-by-Module Overview

- Module 10 – "Building Java GUIs"
- Module 11 – "GUI Event Handling"
- Module 12 – "Introduction to Java Applets"
- Module 13 – "GUI-Based Applications"
- Module 14 – "Threads"
- Module 15 – "Advanced I/O Streams"
- Module 16 – "Networking"

Module-by-Module Overview

- Module 10 – "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.

- Module 11 – "GUI Event Handling"

Creating a layout of GUI components in a frame is not enough. Code must be written 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 12 – "Introduction to Java Applets"

This module demonstrates how to program Java technology applets; including the use of images and audio clips.

Module-by-Module Overview

- Module 13 – "GUI-Based Applications"

This module discusses a variety of GUI elements: menus, color and font control, printing, and the difference between applet and application development.

- Module 14 – "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 15 – "Advanced I/O Streams"

This module explains the classes available for reading and writing both data and text files, and introduces object serialization.

- Module 16 – "Networking"

This module introduces the Java network programming package and demonstrates a Transmission Control Protocol/Internet Protocol (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 application
- Understand and use the online hypertext Java technology documentation
- Describe language syntactic elements and constructs
- Understand the object-oriented paradigm
- Use object-oriented features of Java
- Understand and use exceptions
- Understand and use the Collections API
- Read and write to files
- Develop a graphical user interface
- Describe the Java technology platform's Abstract Window Toolkit
- Develop a program to take input from a GUI
- Understand event handling
- Develop Java applets
- Understand and use the `java.io` package
- Understand the basics of multithreading
- Develop multithreaded Java applications and applets
- Develop Java client and server programs using TCP/IP

✓ Ask the students how many signed up for this course because of the information in the Sun Educational Services course catalog, what their knowledge and expectations of the objectives stated there are, and use this information as a tool to manage your time in covering the material in this course.

Skills Gained by Module

The skills for *Java™ Programming Language* are shown in column 1 of the matrix below. The black boxes indicate the main coverage for a topic; the gray boxes indicate the topic is briefly discussed.

Skills Gained	Module															
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Describe key language features	■															
Compile and run a Java application	■															
Understand and use the online hypertext Java technology documentation		■							■							
Describe language syntactic elements and constructs		■	■	■	■	■										
Understand the object-oriented paradigm		■	■			■	■									
Use object-oriented features of Java		■	■			■	■									
Understand and use exceptions								■								
Understand and use the Collections API									■							
Read and write to files									■							
Develop a GUI										■	■	■	■			
Describe the Java technology platform's Abstract Window Toolkit										■	■	■	■			
Create a program to take input from a graphical user interface										■	■	■	■			
Understand event handling										■	■	■	■			
Develop Java applets											■	■	■			
Understand and use the <code>java.io</code> package														■		
Understand the basics of multithreading															■	
Develop multithreaded Java applications and applets															■	
Develop Java client and server programs using TCP/IP																■

✓ **Refer students to this matrix as you progress through the course to show them the progress they are making in learning the skills advertised for this course.**

Guidelines for Module Pacing

The table below provides a rough estimate of pacing for this course.

Module	Day 1	Day 2	Day 3	Day 4	Day 5
About This Course	A.M.				
Module 1 – "Getting Started"	A.M.				
Module 2 – "Object-Oriented Programming"	P.M.				
Module 3 – "Identifiers, Keywords, and Types"	P.M.				
Module 4 – "Expressions and Flow Control"		A.M.			
Module 5 – "Arrays"		A.M.			
Module 6 – "Inheritance"		P.M.			
Module 7 – "Advanced Class Features"			A.M.		
Module 8 – "Exceptions"			A.M.		
Module 9 – "Text-Based Applications"			P.M.		
Module 10 – "Building Java GUIs"				A.M.	
Module 11 – "GUI Event Handling"				A.M.	
Module 12 – "Introduction to Java Applets"				P.M.	
Module 13 – "GUI-Based Applications"				P.M.	
Module 14 – "Threads"					A.M.
Module 15 – "Advanced I/O Streams"					P.M.
Module 16 – "Networking"					P.M.



Topics Not Covered

- General programming concepts. This is not a course for people who have never programmed before.
- General object-oriented concepts.

Topics Not Covered

This course does not cover the topics shown on the above overhead. Many of the topics listed on the overhead are covered in other courses offered by Sun Educational Services:

- Object-oriented concepts – Covered in OO-100: *Object-Oriented Technology and Concepts*
- Object-oriented design and analysis – Covered in OO-120: *Object-Oriented Design and Analysis*
- General programming concepts – Covered in SL-110: *Java Programming for Non-Programmers*



How Prepared Are You?

Before attending this course, you should have completed:

- SL-110: *Java Programming For Non-Programmers*

or have:

- Created 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™

How Prepared Are You?

Before attending this course, you should have completed:

- SL-110: *Java Programming For Non-Programmers*

or have:

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

- ✓ ***If any students indicate they cannot do these requirements, meet with them at the first break to decide how to proceed with the class. Do they want to take the class at a later date? Is there some way to get the extra help needed during the week?***
- ✓ ***It might be appropriate here to recommend resources from the Sun Educational Services catalog that provide training for topics not covered in this course.***



Introductions

- Name
- Company affiliation
- Title, function, and job responsibility
- Programming experience
- Reasons for enrolling in this course
- Expectations for this course

Introductions

Now that you have been introduced to the course, introduce yourself to each other and to the instructor, addressing the items shown on the above overhead.



How to Use Course Materials

- Course Map
- Relevance
- Overhead Image
- Lecture
- Exercise
- Check Your Progress
- Think Beyond

How to Use Course Materials

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

- **Course Map** – This preface contains an overview of the content so you can see how the modules fit into the overall course goal.
- **Objectives** - What you should be able to accomplish after completing this module is listed here.
- **Relevance** – The Relevance section for each module provides scenarios or questions that introduce you to the information contained in the module and encourage you to think about how the module content relates to your interest in Java applications programming.
- **Overhead Image** – Reduced overhead images for the course are included in the course materials to help you easily follow where the instructor is at any point in time. Overheads do not appear on every page.

How to Use Course Materials

- **Lecture** – The instructor will present information specific to the topic of the module. This information will help you learn the knowledge and skills necessary to succeed with the exercises.
- **Exercise** – Lab exercises will give you the opportunity to practice your skills and apply the concepts presented in the lecture.
- **Check Your Progress** – Module objectives are restated, sometimes in question format, so that before moving on to the next module you are sure that you can accomplish the objectives of the current module.
- **Think Beyond** – Thought-provoking questions are posed to help you apply the content of the module or predict the content in the next module.

Course Icons and Typographical Conventions

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

Course Icons



Additional resources – Indicates additional reference materials are available.



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



Exercise objective – Indicates the objective for the lab exercises that follow. The exercises are appropriate for the material being discussed.

Note – Additional important, reinforcing, interesting or special information.



Caution – A potential hazard to data or machinery.

Typographical Conventions

Courier is used for the names of commands, files, and directories, as well as on-screen computer output. For example:

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

It is also used to represent parts of the Java™ programming language such as class names, methods, and keywords. For example:

The `getServletInfo` method is used to...
The `java.awt.Dialog` class contains `Dialog` (Frame parent)

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

```
system% su
Password:
```

It is also used for each code line that will be referenced in text. For example:

```
while ( (s = input.readLine()) != null ) {
    // process input string
}
```

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

To delete a file, type `rm filename`.

Palatino italics is used for book titles, new words or terms, or words that are emphasized. For example:

Read Chapter 6 in *User's Guide*.
These are called *class* options.
You *must* be root to do this.

The 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` which 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 is different on the Solaris Operating Environment and Microsoft Windows platforms, both commands are shown. For example:

On Solaris

```
cd server_root/bin
```

On Microsoft Windows

```
cd server_root\bin
```

Notes to the Instructor

Philosophy

The *Java Programming Language* course has been created to allow for interactions between the instructor and the student as well as between the students themselves. In an effort to enable you to accomplish the course objectives easily, and in the time frame given, a series of tools has been developed and support materials created for your discretionary use.

A consistent structure has been used throughout this course. This structure is outlined in the “Course Goal” section. The suggested flow for each module is:

1. Module objectives
2. Context questions/module rationale
3. Lecture information with appropriate overheads
4. Lab exercises
5. Discussion: either as whole class or in small groups

To allow the instructor flexibility and give time for meaningful discussions during the lectures and the small-group discussions, a timing table is included in the “Course Tools” section.

About the Labs

Almost every module in this course has a set of exercises. Each exercise is marked as either a Level 1, Level 2, or Level 3 lab. The Level 1 labs are designed to reinforce the material presented in the module. Level 2 labs extend the material presented in the module, providing additional practice. Level 3 labs require some additional research to complete—this might include external materials like books on the Java programming language, or material that is available in the Java API documentation.

Every module will have a Level 2 or Level 3 lab or both.

Course Tools

To enable you to follow this structure, the following supplementary materials are provided with this course:

- **Relevance**

These questions or scenarios set the context of the module. It is suggested that the instructor ask these questions and discuss the answers. The answers are provided only in the instructor's guide.

- **Course map**

The course map allows the students to get a visual picture of the course. It also helps students know where they have been, where they are, and where they are going. The course map is presented in the "About This Course" in the student's guide.

- **Lecture overheads**

Overheads for the course are provided in two formats:

The paper-based format can be copied onto standard transparencies and used on a standard overhead projector. These overheads are also provided in the student's guide.

The Web browser-based format is in HTML and can be projected using a projection system which displays from a workstation. This format gives the instructor the ability to allow the students to view the overhead information on individual workstations. It also allows better random access to the overheads.

Course Tools (Continued)

- **Small-group discussion**

After the lab exercises, it is a good idea to debrief the students. Gather them back into the classroom and have them discuss their discoveries, problems, and issues in programming the solution to the problem in small groups of four or five, one-on-one, or one-on-many.

- **General timing recommendations**

Each module contains a “Relevance” section. This section may present a scenario relating to the content presented in the module, or it may present questions that stimulate students to think about the content that will be presented. Engage the students in relating experiences or posing possible answers to the questions. Spend no more than 10–15 minutes on this section

Module	Day 1	Day 2	Day 3	Day 4	Day 5
About This Course	A.M.				
Module 1 – Getting Started	A.M.				
Module 2 – Object-Oriented Programming	P.M.				
Module 3 – Identifiers, Keywords, and Types	P.M.				
Module 4 – Expressions and Flow Control		A.M.			
Module 5 – Arrays		A.M.			
Module 6 – Inheritance		P.M.			
Module 7 – Advanced Class Features			A.M.		
Module 8 – Exceptions			A.M.		
Module 9 – Text-Based Applications			P.M.		
Module 10 – Building Java GUIs				A.M.	
Module 11 – GUI Event Handling				A.M.	
Module 12 – Introduction to Java Applets				P.M.	
Module 13 – GUI-Based Applications				P.M.	
Module 14 – Threads					A.M.
Module 15 – Advanced I/O Streams					P.M.
Module 16 – Networking					P.M.

- **Module self-check**

Each module contains a checklist for students in the “Check Your Progress” section. Give them a little time to read through this checklist before going on to the next lecture. Ask them to see you for items they do not feel comfortable checking off.

Instructor Setup Notes

Purpose of This Guide

This guide provides general information about setting up the classroom. Refer to the `SL275.D.setup.txt` file in the `SL275_IN` directory for specific information about how to set up this course.

Projection System and Workstation

If you have a projection system for projecting HTML slides and are planning to use the HTML slides, you need to do the following:

- Install the HTML overheads on the workstation connected to the projection system so you can display them with a browser during lecture.

To install the HTML overheads on the machine connected to your overhead projection system, copy the `HTML` and `images` subdirectories provided in the `SL275_OH` directory to any directory on the overhead workstation machine.

Display the overheads in the browser by choosing `Open > File` and typing the following in the Selection field of the pop-up window:

```
/SL275_revX_XXXX/SL275_OH/HTML/OH.Title.doc.html
```

- Set up an overhead-projection system that can project instructor workstation screens.

Note – This document does not describe the steps necessary to set up an overhead projection system because it is unknown what will be available in each training center. This setup is the responsibility of each training center.

Course Files

All of the course files for this course are available from the education.central server. You can use ftp or the education.central Web site, <http://education.central/Released.html>, to download the files from education.central. Either of these methods requires you to know the user ID and password for FTP access. See your manager for these if you have not done this before.

Course Components

This course consists of the following components:

- Instructor guide

The SL275_IG directory contains the FrameMaker files for the instructor's guide (student's guide with instructor notes). The ART directory is required for printing this guide.

- Student guide

The SL275_SG directory contains the FrameMaker files for the student's guide. The ART directory is required for printing this guide.

- Art

The SL275_ART directory contains the supporting images and artwork for the student's and instructor's guides. This directory is *required* for the printing of the student's and instructor's guides and should be located in the same directory as SL275_IG and SL275_SG.

- Instructor notes

The SL275_IN directory contains the text file SL275.D.setup.txt.

- Overheads

The SL275_OH directory contains the instructor overheads. There are both HTML and FrameMaker versions of the overheads.

- Lab files

The SL275_LF directory contains the lab files for this course.

Objectives

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

- Describe key features of Java technology
- Define the terms *class* and *application*
- Write, compile, and run a simple Java application
- Describe the Java virtual machine's (JVM™) function
- Describe how garbage collection works
- List the three tasks performed by the Java platform that handle code security

This module provides a general overview of Java technology including the Java virtual machine, garbage collection, and security features.

Relevance

- ✓ **Present the following questions to stimulate the students and get them thinking about the issues and topics presented in this module. They are not expected to know the answers to all of these questions. Hold discussions where students have input; otherwise, if no one can propose answers, begin the lecture for this module.**



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

- Is the Java programming language a complete language or is it just useful for writing programs for the Web?
- ✓ **There might still be some attitudes that the Java programming language is for Web programs. There are certain features (such as better security) that make this language ideal for the Web, but it is a complete object-oriented programming language useful for mainstream applications as well.**
 - Why is another programming language needed?
 - How does the Java technology platform improve on other language platforms?
 - ✓ **The Java platform was written with security in mind from the inception of the language thus security issues are at the core of the platform. Other languages address security issues later in the development cycle, and might still have security leaks.**
 - ✓ **Other issues you might discuss are platform-independence and ease of use. The Java programming language is easier to use than other programming languages (such as C or C++) because it has automatic garbage collection, stronger typing of objects and variables, and removal of problem areas such as pointers. You might want to use this question as an introduction to what will be discussed in this module.**

Additional Resources



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

- Gosling, Jay, and Steele. *The Java Language Specification*. Addison-Wesley. 1996.
[Also online at: <http://java.sun.com/docs/books/jls/>]
- Lindholm and Yellin. *The Java Virtual Machine Specification*. Addison-Wesley. 1997.
- Yellin, Frank. Low-Level Security in Java, white paper. [Online]. Available: <http://www.javasoft.com/sfaq/verifier.html>.

What Is the Java Technology?

Java technology is:

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

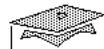
The syntax of the Java programming language is similar to C++ syntax and the semantics are similar to SmallTalk semantics. You can use the Java programming language to create all kinds of applications that you could create using any conventional programming language.

The Java programming language is usually mentioned in the context of the World Wide Web (WWW) and browsers that are capable of running programs called *applets*. Applets are programs written in the Java programming language that reside on WWW 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 *applications* are standalone programs that do not require a Web browser to execute. They are typically general-purpose programs that run on any machine where the Java runtime environment (JRE) is installed.

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

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



Primary Goals of the Java Technology

- Provides an easy-to-use language by:
 - ▼ Avoiding the pitfalls of other languages
 - ▼ Being object-oriented
 - ▼ Enabling users to create streamlined and clear code

What Is the Java Technology?

Primary Goals of the Java Technology

Java technology provides the following:

- A language that is easy to program because it:
 - ▼ Eliminates the pitfalls of other languages, such as pointer arithmetic and memory management that affect code robustness
 - ▼ Is object-oriented to help the programmer visualize the program in real-life terms
 - ▼ Provides a means to make code as streamlined and clear as possible



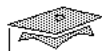
Primary Goals of the Java Technology

- Provides an interpreted environment for:
 - ▼ Improved speed of development
 - ▼ Code portability
- Enables users to run more than one thread of activity
- Loads classes dynamically, that is, at the time they are actually needed
- Supports dynamically changing programs during runtime by loading classes from disparate sources
- Furnishes better security

What Is the Java Technology?

Primary Goals of the Java Technology (Continued)

- An interpreted environment resulting in the following two benefits:
 - ▼ Speed of development – Reduces the compile-link-load-test cycle
 - ▼ Code portability – Allows user code to be written once and 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 allowing them to download code modules
- A means of checking code modules that are loaded to ensure security



Primary Goals of the Java Technology

The following features fulfill these goals:

- The Java virtual machine (JVM)
- Garbage collection
- Code security

What Is the Java Technology?

Primary Goals of the Java Technology (Continued)

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

- The Java virtual machine
- Garbage collection
- Code security

These topics will be covered in more detail in the section "Java - Behind the Scenes" but first let's take a look at a simple Java application and applet.

A Basic Java Application

Like any other programming language, the Java programming language is used to create applications. The following code shows a simple Java application that prints a greeting to the world.

TestGreeting.java

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

Greeting.java

```
1 // The Greeting class declaration.
2 public class Greeting {
3     private String salutation;
4
5     Greeting(String s) {
6         salutation = s;
7     }
8
9     public void greet(String whom) {
10        System.out.println(salutation + " " + whom);
11    }
12 }
```

✓ *The following pages describe this program line by line.*

A Basic Java Application

TestGreeting *Described*

Lines 1–3

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

Lines 1–3 in the program are comment lines.

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 specify no 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`.

Line 5

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

Line 5 is where the execution of the program starts. The Java technology interpreter must find this defined exactly as given or it will refuse 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 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.

A Basic Java Application

TestGreeting *Described (Continued)*

Line 5 (Continued)

Let's look at each element of this line in more detail:

- `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.

✓ **Static members are loaded first and are immediately accessible at runtime.**

✓ **As of JDK1.2, you are not allowed to override a static method. Therefore, `main` is visible to the Java runtime environment only if it is defined correctly.**

- `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 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] . . .
```

✓ **The Java programming language does not pass the name of the class as an argument to a program.**

A Basic Java Application

TestGreeting Described (Continued)

Line 6

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

Line 6 illustrates the creation of 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`. The implementation of this constructor is shown on lines 5-7 of the `Greeting.java` file.

Lines 7

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

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 9-11 of the `Greeting.java` file.

Lines 8-9

```
8      }  
9  }
```

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

A Basic Java Application

Greeting Described

Lines 1-2

```
1 // The Greeting class declaration.  
2 public class Greeting {
```

Line 2 declares the `Greeting` class.

Line 3

```
3     private String salutation;
```

Line 3 declares an attribute, `salutation`, of the `Greeting` class. This attribute is declared as `private` to hide it from the `TestGreeting` program. The data type of this attribute is `String`, a sequence of characters.

Lines 4-7

```
4  
5     public Greeting(String s) {  
6         salutation = s;  
7     }
```

Lines 5-7 of the program declare a constructor for this class. This code is called when a new `Greeting` object is created, as was done on line 6 of the `TestGreeting.java` file. It takes a single `String` parameter, which is the salutation for the new greeting. Line 6 initializes the `salutation` attribute to that value.

A Basic Java Application

Greeting *Described (Continued)*

Lines 8-11

```
8
9   public void greet(String whom) {
10      System.out.println(salutation + " " + whom);
11   }
```

Lines 9-11 demonstrates 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 takes one parameter, `whom`. This parameter is of type: `String`.

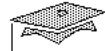
- ✓ **The static variable `out` is defined in the `System` class to be of type `PrintStream`; `PrintStream` is a class defined in the `java.io` package, and the source file is called `PrintStream.java`. This is where the `println` method is declared.**

The purpose of the `greet` method is to send a message to standard output stream. The message is a string that is concatenated from the value of the `salutation` variable, a space character, and the value of the `whom` parameter. For example, if `salutation` is "Hello" and `whom` is "World", then the message is "Hello World". The `println()` method is used to write this message to the standard output stream.

Line 12

```
12 }
```

Line 12 closes the class declaration for `Greeting`.



Compiling and Running TestGreeting

- Compiling `TestGreeting.java`
`javac TestGreeting.java`
- `Greeting.java` is compiled automatically
- Running an application
`java TestGreeting`
- Locating common compile and runtime errors

A Basic Java Application

Compiling and Running TestGreeting

Once 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. Also notice that the `Greeting.java` file has been compiled into `Greeting.class`. This is done automatically by the compiler, because the `TestGreeting` class uses the `Greet` class.

If you have a problem compiling the application, check the troubleshooting messages on page 1-16.

A Basic Java Application

Compiling and Running TestGreeting (Continued)

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

```
java TestGreeting
Hello World
```

Note – The `PATH` environment variable must be set to find `java` and `javac`; make sure it includes `java_root/bin` (where `java_root` represents the directory root where Java is installed).

✓ **As of JDK 1.1, `PATH` is all that is required.**

A Basic Java Application

Troubleshooting the Compilation

Compile-Time Errors

The following are common errors seen at compile time:

- 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 JDK™ directory.

- Greeting.java:10: Method println(java.lang.String) not found in class java.io.PrintStream.
System.out.println(salutation + " " + whom);

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 therefore 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

Only one top level, non-static class can be declared 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 above for every public class in the file that does not have the same name as the file.

A Basic Java Application

Troubleshooting the Compilation

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`) 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.

Java - Behind the Scenes

The Java Runtime Environment

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

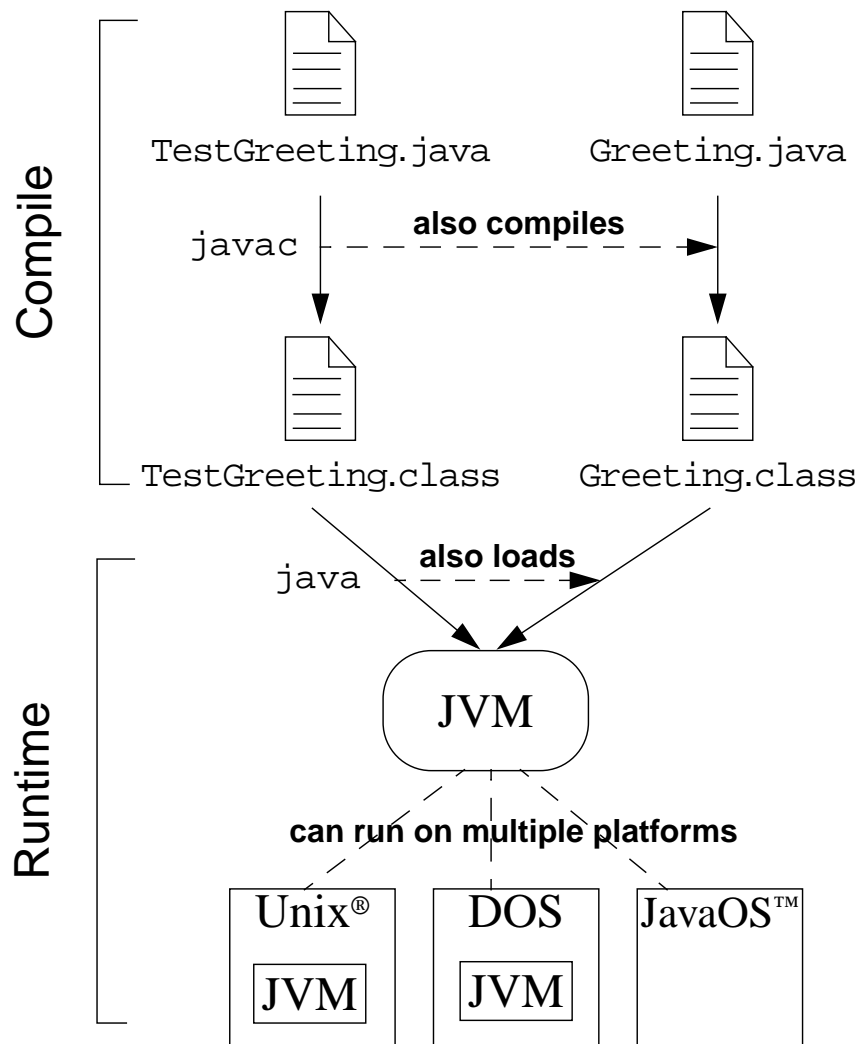
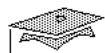


Figure 1-1 Java Technology Runtime Environment

- ✓ This diagram is meant to show how code is compiled into class files (using `javac`) and then executed on a JVM (using `java`) and how the JVM can be implemented on multiple platforms. It is a bit complicated by our `TestGreeting` example because there are two files, but this gives you the opportunity to explain how the JVM will load classes as needed.
- ✓ Show other platforms: `JavaChip™`, in a browser, `PalmPilot (KVM)`, and so on.



The Java Virtual Machine

- Provides hardware platform specifications
- Reads compiled byte codes that are platform independent
- Is implemented as software or hardware
- Is implemented in a Java technology development tool or a Web browser

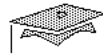
Java - Behind the Scenes

The Java Virtual Machine

The Java Virtual Machine Specification defines the Java virtual machine (JVM) as:

An imaginary machine that is implemented by emulating it in software on a real machine. Code for the Java virtual machine 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 all Java technology code is compiled. This specification enables Java software to be platform independent because the compilation is done for a generic machine known as the Java virtual machine (JVM). You can emulate this “generic machine” in software to run on various existing computer systems or implemented in hardware.



The Java Virtual Machine

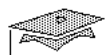
- JVM provides definitions for the:
 - ▼ Instruction set (central processing unit [CPU])
 - ▼ Register set
 - ▼ Class file format
 - ▼ Stack
 - ▼ Garbage-collected heap
 - ▼ Memory area

Java - Behind the Scenes

The Java Virtual Machine (Continued)

The compiler takes the Java application source code and generates bytecodes. Bytecodes are machine code instructions for the JVM. Every Java 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, and a memory area.



Sun Educational Services

The Java Virtual Machine

- The majority of type checking is done when the code is compiled.
- Every Sun Microsystems approved implementation of the JVM must be able to run any compliant class file.

Java - Behind the Scenes

The Java Virtual Machine (Continued)

The code format of the JVM 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*.



Garbage Collection

- Allocated memory that is no longer needed should be deallocated
- In other languages, deallocation is the programmer's responsibility
- The Java programming language provides a system-level thread to track memory allocation
- Garbage collection:
 - ▼ Checks for and frees memory no longer needed
 - ▼ Is done automatically
 - ▼ Can vary dramatically across JVM implementations

Java - Behind the Scenes

Garbage Collection

Many programming languages allow the dynamic allocation of memory 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.

Once the allocated memory is no longer required (the pointer that references the memory has gone *out of scope*), the program or runtime environment should deallocate the memory.

In C, C++, and other languages, the program developer is responsible for deallocating the memory. This can be a difficult exercise at times, because it is not always known in advance when memory should be released. Programs that do not deallocate memory can eventually crash when there is no memory left on the system to allocate. These programs are said to have memory leaks.

Java - Behind the Scenes

Garbage Collection (Continued)

The Java programming language removes the responsibility for allocating and deallocating memory from the programmer. 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 allocate and deallocate memory and avoiding memory leaks. However, garbage collection schemes can vary dramatically across JVM implementations.

Java - Behind the Scenes

Code Security

Overview

Figure 1-2 illustrates the Java technology runtime environment (JRE) and how it enforces code security.

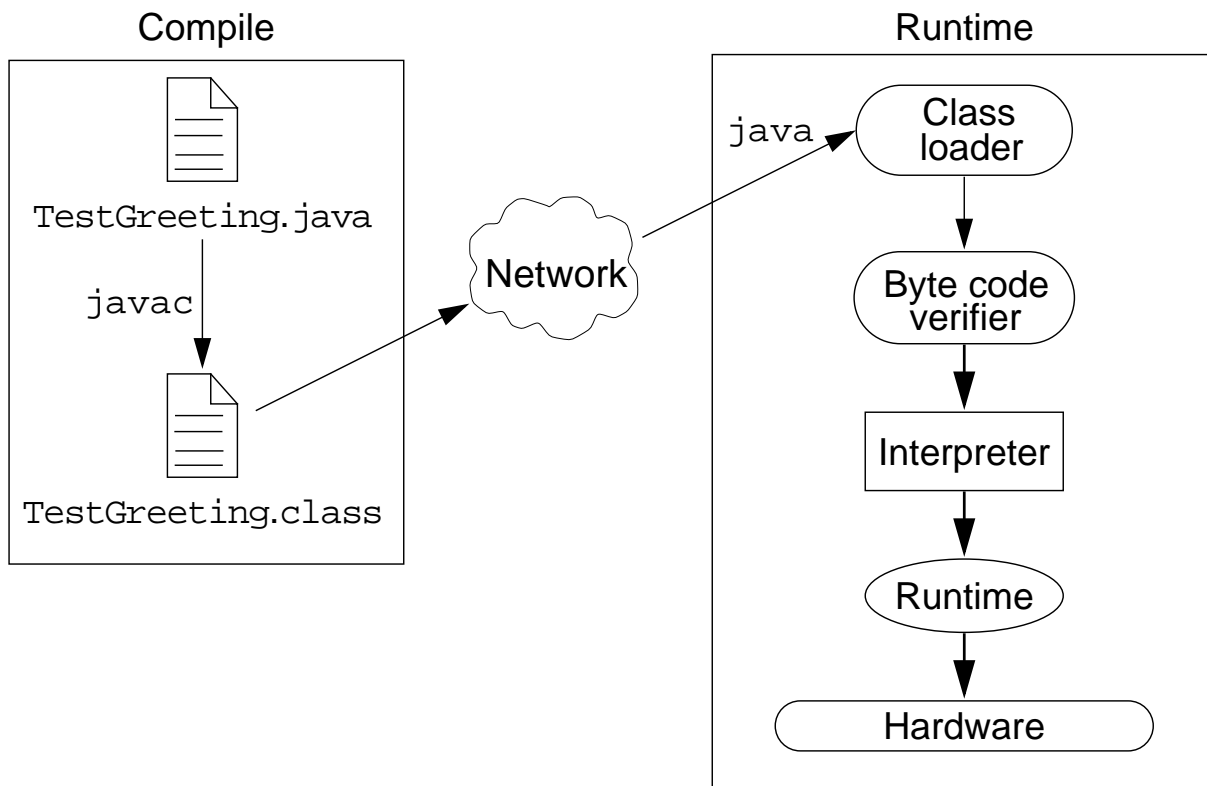


Figure 1-2 Operation of the JRE

Java software source files are “compiled” in the sense that they are converted into a set of bytecodes from the text format in which programmers 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, the bytecodes can be downloaded and then interpreted by the JVM built into the browser. The interpreter has two functions: It executes bytecodes and makes the appropriate calls to the underlying hardware.

Java - Behind the Scenes

Code Security

Overview (Continued)

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 allows Java software code to run close to the speed of C or C++ with a small delay at loadtime to allow compilation to the native machine code.

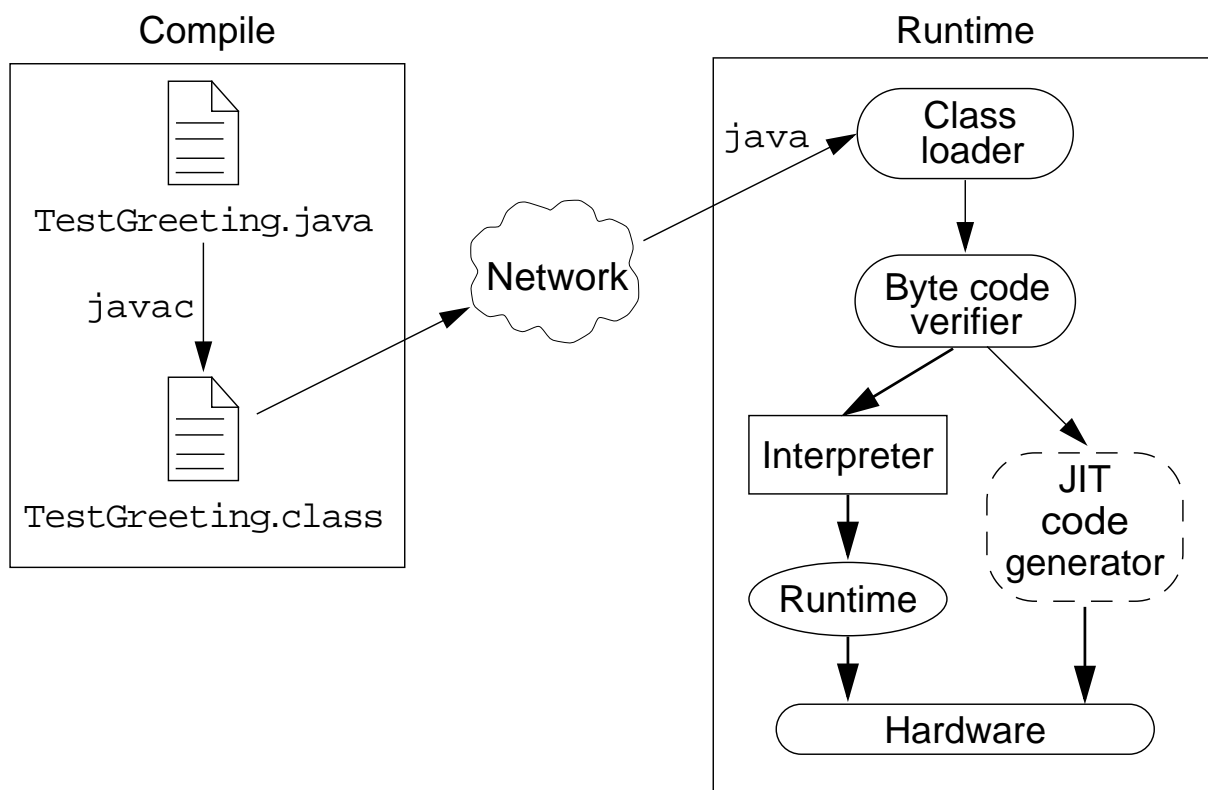


Figure 1-3 Operation of the JRE With a Just-In-Time Compiler

Note – Sun Microsystems has enhanced the Java virtual 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++.



Java Runtime Environment

- Performs three main tasks:
 - ▼ Loads code
 - ▼ Verifies code
 - ▼ Executes code

Java - Behind the Scenes

Code Security

The Java Runtime Environment

A Java technology runtime environment runs code compiled for a JVM and performs three main tasks:

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



Class Loader

- Loads all classes necessary for the execution of a program
- Maintains classes of the local file system in separate "namespaces"
- Prevents spoofing

Java - Behind the Scenes

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.

- ✓ ***Classes that are imported from across the network are loaded into a private namespace associated with the origin. When a class from the private namespace accesses another class, the built-in (local system) classes are checked first, then those in the namespace of the referencing class. This prevents a class from spoofing (creating a hoax of) a built-in class.***

Once 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.



Bytecode Verifier

Ensures that:

- The code adheres to the JVM specification
- The code does not violate system integrity
- The code causes no operand stack overflows or underflows
- The parameter types for all operational code are correct
- No illegal data conversions (the conversion of integers to pointers) have occurred

Java - Behind the Scenes

Code Security (Continued)

Bytecode Verifier

Java software code passes several tests before actually 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—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.

Java - Behind the Scenes

Code Security (Continued)

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 known to always be correct.
- No illegal data conversions, such as converting integers to object references, have occurred.

Exercise: Performing Basic Tasks



Exercise objective – You will compile and debug several test programs. You will also write, compile, and run a simple Java program.

Preparation

An understanding of the concepts and terminology presented in this module is critical to being able to troubleshoot several compilation and runtime errors.

Tasks

In a Web browser view the `lab_files.html` page that is at the top-level of the SL275 directory on your computer. There will be a summary of each exercise and a link to a page that gives a detailed explanation of the exercise.

Go to the SL275 directory on your computer and change to the directory for this module (`mod01`). A listing of this directory will show two subdirectories: one for each of the exercises below.

Exercise 1: Explore Java Errors (Level 1)

In this exercise you will solve compilation and runtime errors by fixing several example Java technology programs.

Exercise 2: Write, Compile, and Run TestGreeting (Level 2)

In this exercise you will write, compile, and run the `TestGreeting` program from "A Basic Java Application" on page 1-8.

Check Your Progress

Before continuing on to the next module, check to be sure that you can:

- Describe key features of Java technology
- Define the terms *class* and *application*
- Write, compile, and run a simple Java application
- Describe the JVM function
- Describe how garbage collection works
- List the three tasks performed by the Java platform that handle code security

Think Beyond

How can you benefit from using this programming language in your work environment?

Objectives

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

- Define modeling concepts: *abstraction*, *encapsulation*, and *packages*
- Discuss why Java technology application code is reusable
- 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
- In a Java technology program, identify the following:
 - ▼ The `package` statement
 - ▼ The `import` statements
 - ▼ Classes, methods, and attributes
 - ▼ Constructors
- Use the Java technology application programming interface (API) online documentation

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

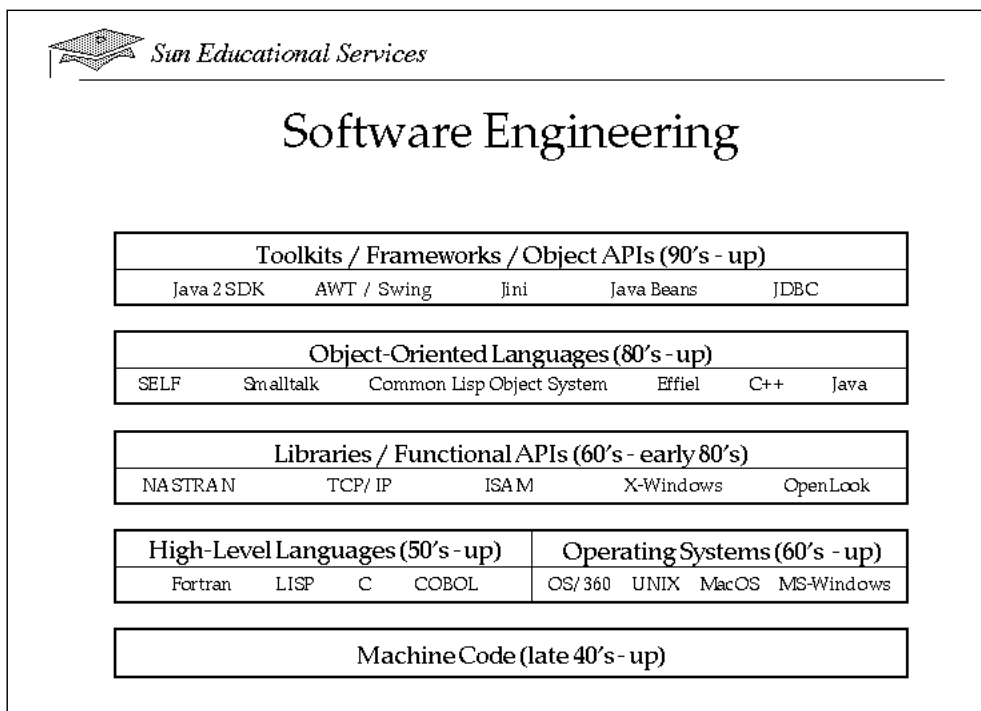
Relevance

- ✓ **Present the following questions to stimulate the students and get them thinking about the issues and topics presented in this module. They are not expected to know the answers to all of these questions. Hold discussions where students have input; otherwise, if no one can propose answers, begin the lecture for this module.**



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?
- ✓ **There are many forms and levels of abstraction. What the students should understand is the importance of functional and object abstraction (and inheritance). In addition, knowing about design patterns is a plus.**
 - What features does the Java programming language possess which make it an object-oriented language?
 - What does the term *object-oriented* really mean?
 - ✓ **This module is the first of three that discusses the object-oriented features of the language. The object-oriented features help make development and maintenance of a Java software program an easier task than if the program were written in another language. For example, data elements in the form of objects are easier to conceptualize because they model real-world objects better; this helps to cut a project's development time. Other object-oriented ideas, such as data hiding help, make future efforts (maintenance, upgrading, and so on) a more efficient and reliable procedure.**



What Is Object-Oriented Programming?

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. 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 file system (ISAM), and creating windows, graphics and other GUI widgets on a bit-mapped monitor (X-Windows and Open Look®).

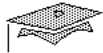
What is Object-Oriented Programming?

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 could not hide the implementation of the data used in the procedures. This made it difficult to modify the implementation of the library without affecting the client code, because that code was often tied to the particular details of the data structures.

By the late 1980's, 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 would not affect client code (assuming the interfaces had not changed). The other important advantage was that procedures were associated with the data structure. The combination of data attributes and procedures (called methods) were named a *class*.

The 1990'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 can be implemented by many different vendors to allow client programmers the choice of flexibility and performance suitable to their applications.

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



Analysis and Design

- Analysis describes *what* the system needs to do
 - ▼ Modeling the real-world: actors and activities, objects and behaviors
- Design describes *how* the system does it
 - ▼ Modeling the relationships and interactions between objects and actors in the system
 - ▼ Finding useful abstractions to help simplify the problem or solution

What Is Object-Oriented Programming?

Analysis and Design

There are five phases in a software development project: analysis, design, implementation, test, and deployment. They are all important, but it is critical that enough time is given to analysis and design.

Analysis is the definition of *what* the system is supposed to accomplish. This is done by defining the set of actors (users and other systems that interact with the proposed system) and activities that the proposed system must accommodate. Also, the analysis 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 must be supported by the proposed system. The description of the activities should be detailed enough to create baseline criteria for the testing phase.

What Is Object-Oriented Programming?

Analysis and Design (Continued)

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

Note – UML is a very large and complex language. You will only be using a small portion of it. Appendix G, “UML Modeling and Java” is a reference to the UML elements that we will be using. It also shows you how to implement Java technology code from a UML class diagram.

- ✓ ***We will be using only class models. If a student is interested in a deeper understanding of UML, then you should recommend the OO-226 course, "OO Application Analysis & Design for Java Technology (UML)."***

What Is Object-Oriented Programming?

Analysis and Design Example

In this module, we will use the example of a shipping company. We will 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.

- A graphical user interface (GUI) will be used to keep track of adding boxes to vehicles.
- Several reports need to be generated from the fleet records.

From these requirements we can isolate 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

- Functions – Write an algorithm once to be used in many situations
- Objects – Group a related set of attributes and behaviors into a class
- Frameworks and APIs – Large groups of objects that support a complex activity
 - ▼ Frameworks can be used "as is" or be modified to extend the basic behavior

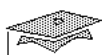
What Is Object-Oriented Programming?

Abstraction

As you saw in the chart on Software Engineering, software design has moved from low-level constructs, such as writing in machine code, towards 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.

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 Application Programming Interfaces (APIs).

✓ ***In this text, we will not distinguishing between functions, procedures, sub-routines, and methods.***



Classes as Blueprints for Objects

- In manufacturing, a blueprint is a description of a device from which many physical devices are constructed
- In software, a class is a description of an object:
 - ▼ A class describes the data that each object includes
 - ▼ A class describes the behaviors that each object exhibits
- In Java, classes support three key features of OOP:
 - ▼ encapsulation
 - ▼ inheritance
 - ▼ polymorphism

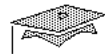
What Is Object-Oriented Programming?

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 Object-Oriented Programming: encapsulation, inheritance, and polymorphism.

Note – Encapsulation is covered in the Section "Encapsulation" on page 2-17. Inheritance and polymorphism are discussed in Module 6, "Inheritance."



Declaring Java Classes

- Basic syntax of a Java class:

```
<class_declaration> ::=
    <modifier> class <name> {
        <attribute_declaration>*
        <constructor_declaration>*
        <method_declaration>*
    }
```

- Example:

```
public class Vehicle {
    private double maxLoad;
    public void setMaxLoad(double value) {
        maxLoad = value;
    }
}
```

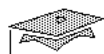
Declaring Java Classes

The Java technology class declaration takes the following form:

```
<class_declaration> ::=
    <modifier> class <name> {
        <attribute_declaration>*
        <constructor_declaration>*
        <method_declaration>*
    }
```

- ✓ **This is a modified Backus-Naur Form (BNF). "::<=" means "is defined by," [] is a grouping construct and means "is optional," unless followed by a * or +, * means "zero or more," and + means "one or more." The tokens in angle brackets <> are non-terminals and all other tokens are terminals, including curly brackets {}. In BNF grammar, terminals are final tokens in the sentence and non-terminals are nodes that expand into other terminals and non-terminals.**
- ✓ **The syntax diagrams in this module are not complete. They are meant to give the student an overview of the "layout" of the code for a given declaration. A complete BNF grammar of Java is beyond the scope of this course and is readily available in the JLS.**

The `<name>` can be any legal identifier. It is the name of the class being declared. There are several possible `<modifier>`, 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.



Declaring Attributes

- Basic syntax of an attribute:

```
<attribute_declaration> ::=  
    <modifier> <type> <name> [= <default_value>];
```

```
<type> ::= byte | short | int | long | char  
         float | double | boolean | <class>
```

- Examples:

```
public class Foo {  
    public int    x;  
    private float y = 10000.0F;  
    private String name = "Fred Flintstone";  
}
```

Declaring Attributes

The declaration of an object attribute takes the following form:

```
<attribute_declaration> ::=  
    <modifier> <type> <name> [= <default_value>];
```

```
<type> ::= byte | short | int | long | char |  
         float | double | boolean | <class_name>
```

The *<name>* can be any legal identifier. It is the name of the attribute being declared. There are several possible *<modifier>*, but for now, use either *public* or *private*; this declares that the attribute is accessible only to the methods within this class. The *<type>* of the attribute can be any primitive type or any class.



Declaring Methods

- Basic syntax of a method:

```

<method_declaration> ::=
    <modifier> <return_type> <name> (<parameter>*) {
        <statement>*
    }
<parameter> ::= <parameter_type> <parameter_name>,
  
```

- Examples:

```

public class Thing {
    private int x;
    public int getX() {
        return x;
    }
    public void setX(int new_x) {
        x = new_x;
    }
}
  
```

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 form:

```

<method_declaration> ::=
    <modifier> <return_type> <name> (<parameter>*) {
        <statement>*
    }
<parameter> ::= <parameter_type> <parameter_name>,
  
```

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

The *<modifier>* segment 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. *private* indicates that a method can be called only by the other methods in the class. *protected* is discussed later in this course.

Declaring Methods

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 *<parameter>* 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.

For example:

```
1 public class Thing {
2     private int x;
3     public int getX() {
4         return x;
5     }
6     public void setX(int new_x) {
7         x = new_x;
8     }
9 }
```

The `Thing` class has a single instance variable `x`. The method `getX` returns the `x` data attribute; it uses no parameters. A value is returned from a method using the `return` statement (line 4). The method `setX` modifies the `x` value with the parameter `new_x`; it does not return any value. Here is how this method can be used:

```
1 public class TestThing {
2     public static void main(String[] args) {
3         Thing thing1 = new Thing();
4
5         thing1.setX(47);
6         System.out.println("thing1.x is " + thing1.getX());
7     }
8 }
```

The output is:

```
thing1.x is 47
```



Accessing Object Members

- The "dot" notation: `<object>.<member>`
- This is used to access object members including attributes and methods
- Examples:

```
thing1.setX(47);  
thing1.x = 47; // only permissible if x is public
```

Accessing Object Members

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

```
thing1.setX(47);
```

This line of code tells the `thing1` object (actually a variable, `thing1`, holding a reference to an object of type `Thing`) to execute its `setX` method. This is called "dot notation" because it allows the programmer to access non-private attribute and method members of that class.

Within the definition of a method, you do not need to use the dot notation for accessing local members. For example, in the `setX` method of the `Thing` class we did not use the dot notation to access the `x` attribute.



Information Hiding

The Problem:

MyDate
+day : int
+month : int
+year : int

Client code has direct access to internal data:

```
MyDate d = new MyDate()
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

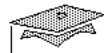
The Problem

Suppose that you have a MyDate class that includes the attributes: day, month, and year. A naive implementation would be to allow direct access to these data attributes. For example:

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

Client code could then access the attributes directly and make mistakes. For example:

```
MyDate d = new MyDate()
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

The Solution:

MyDate
-day : int -month : int -year : int
+getDay() : int +getMonth() : int +getYear() : int +setDay(d : int) : boolean +setMonth(m : int) +setYear(y : int) -validDay(d: int) : boolean

verify days in month

Client code must use setters/getters to access internal data:

```
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
```

Information Hiding

The Solution

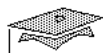
To solve the problem, hide the data attributes by making them private and supply retrieval access methods, `getXxx()`, that are often called "getters" and storage access methods, `setXxx()`, that are often called "setters." These methods allow the class to modify the internal data, but more importantly 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

- Hides the implementation details of a class
- Forces the user to use an interface to access data
- Makes the code more maintainable

MyDate
-date : long
+getDay() : int
+getMonth() : int
+getYear() : int
+setDay(d : int)
+setMonth(m : int)
+setYear(y : int)
-validDay(d: int) : boolean

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.



Declaring Constructors

- Basic syntax of a constructor:

```
<constructor_declaration> ::=  
    <modifier> <class_name> (<parameter>*) {  
        <statement>*  
    }
```

- Examples:

```
public class Thing {  
    private int x;  
    public Thing() {  
        x = 47;  
    }  
    public Thing(int new_x) {  
        x = new_x;  
    }  
}
```

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 declaration takes the following form:

```
<constructor_declaration> ::=  
    [<modifier>] <class_name> (<parameter>*) {  
        <statement>*  
    }
```

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

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

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

Declaring Constructors

For example:

```
1 public class Thing {
2     private int x;
3
4     public Thing() {
5         x = 47;
6     }
7     public Thing(int new_x) {
8         x = new_x;
9     }
10
11    public int getX() {
12        return x;
13    }
14    public void setX(int new_x) {
15        x = new_x;
16    }
17 }
```

The Thing class has a single instance variable `x`. The first constructor (with no parameters) initializes `x` to 47. The second constructor uses a parameter, `new_x`, to initialize `x`.

Here is how these method may be used:

```
1 public class TestThing {
2     public static void main(String[] args) {
3         Thing thing1 = new Thing();
4         Thing thing2 = new Thing(42);
5
6         System.out.println("thing1.x is " + thing1.getX());
7         System.out.println("thing2.x is " + thing2.getX());
8     }
9 }
```

The output is:

```
thing1.x is 47
thing2.x is 42
```



The Default Constructor

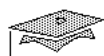
- There is always at least one constructor in every class
- If the writer does not supply any constructors, the default constructor will be present automatically
 - ▼ The default constructor takes no arguments
 - ▼ The default constructor has no body
- Enables you to create object instances with `new Xxx()` without having to write a constructor

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 Xxx()`; otherwise, you would be required to provide a constructor for every class.

Note – If you add a constructor declaration with arguments to a class that previously had no explicit constructors, you lose the default constructor. From that point, calls to `new Xxx()` will cause compiler errors.



Source File Layout

- Basic syntax of a Java source file:

```
<source_file> ::=  
    [<package_declaration>]  
    <import_declaration>*  
    <class_declaration>+
```

- Example, the `VehicleCapacityReport.java` file:

```
package shipping.reports.Web;  
  
import shipping.domain.*;  
import java.util.List;  
import java.io.*;  
  
public class VehicleCapacityReport {  
    private List vehicles;  
    public void generateReport(Writer output) {...}  
}
```

Source File Layout

A Java technology source file takes the following form:

```
<source_file> ::=  
    [<package_declaration>]  
    <import_declaration>*  
    <class_declaration>+
```

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 may include more than one class declaration, but *only one* class may 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.

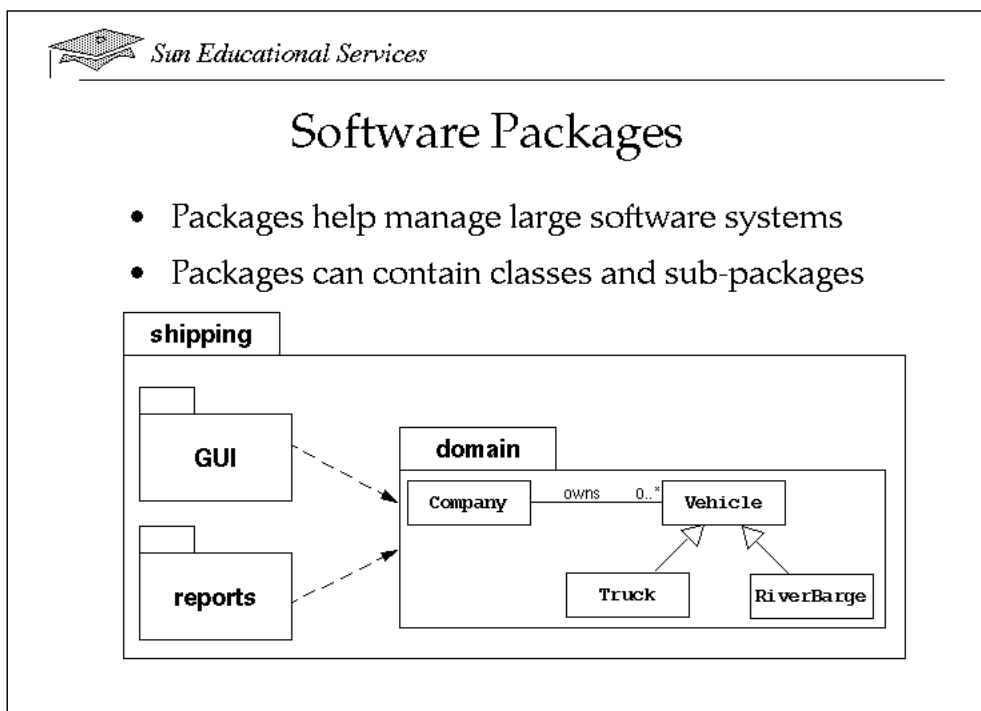
Source File Layout

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

```
package shipping.reports.Web;

import shipping.objects.*;
import java.util.List;
import java.io.*;

public class VehicleCapacityReport {
    private List vehicles;
    public void generateReport(Writer output) {
        ...
    }
}
```

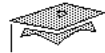


Software Packages

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 forming a hierarchy of 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 (another name for package in UML) are dependent upon the objects package. All of these packages are contained in the top-level package called shipping.



The package Statement

- Basic syntax of the package statement:

```
<package_declaration> ::=  
package <top_pkg_name>[.<sub_pkg_name>]*;
```

- Example:

```
package shipping.reports.Web;
```

- Specify the package declaration at the beginning of the source file
- Only one package declaration per source file
- If no package is declared, then the class "belongs" to the default package
- Package names must be hierarchical and separated by dots

The package Statement

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

```
<package_declaration> ::=  
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:

```
// Class Vehicle of 'objects' sub-package for the  
// 'shipping' application package.  
package shipping.objects;  
  
public class Vehicle {  
    ...  
}
```

The package Statement

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

Package names are hierarchical, separated by dots. It is usual for the elements of the package name to be entirely lowercase. However, the classname usually starts with a capital letter and you can capitalize the first letter of each additional word to distinguish words in the classname. These naming conventions and others are discussed in "Java Coding Conventions" on page 3-35.

Note – If no package statement is 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

- Basic syntax of the package statement:

```
<import_declaration> ::=  
import <pkg_name>[.<sub_pkg_name>]*.<class_name | *>;
```

- Examples:

```
import shipping.domain.*;  
import java.util.List;  
import java.io.*;
```

- Precedes all class declarations
- Tells the compiler where to find classes to use

The import Statement

The import statement takes the following form:

```
<import_declaration> ::=  
import <pkg_name>[.<sub_pkg_name>]*.<class_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.objects`) forms part of the name of the classes within the package. You could refer to the `Company` class as `shipping.objects.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 import Statement

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

```
package shipping.reports.Web;
```

```
import shipping.objects.*;
```

✓ **The * indicates that the program can import any class in the shipping.objects package.**

```
import java.util.List;
```

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

```
public class VehicleCapacityReport {  
    private Company    companyForReport;  
    ...  
}
```

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 bring classes in other packages into the current namespace. The current package, whether explicit or implicit, is always part of the current namespace.

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

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

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

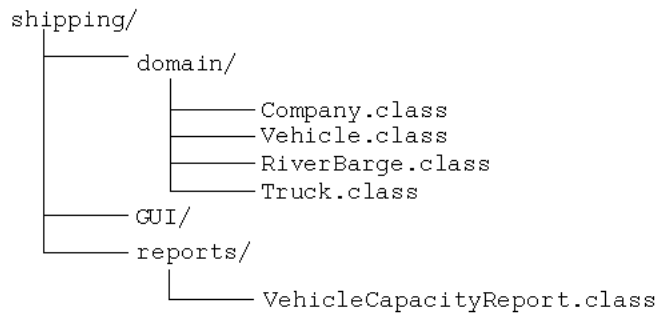
```
import java.io.*;
```

Note – The use of the import statement specifies a path for the compiler to find code, not actually load it as an `#include` statement would do in C or C++. Using the import statement with `"*"` does not affect performance.



Directory Layout and Packages

- Packages are stored in the directory tree containing the package name
- Example, the "shipping" application packages:



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 following directories for the Solaris Operating Environment and the Microsoft Windows operating environment, respectively:

`path/shipping/domain`
`path\shipping\domain`

Solaris Operating Environment
 Microsoft Windows environment

Directory Layout and Packages

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 method.

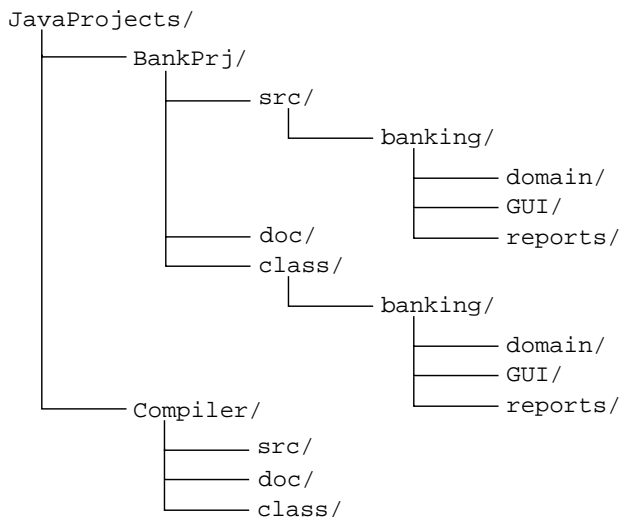


Figure 2-1 Example Project Development Directory Structure

Figure 2-1 demonstrates an example development directory hierarchy for multiple projects. The important element of this hierarchy is that the source files of each project are separated from the compiled (.class) files.

Normally the Java compiler will place the class files in the same directory as the source files. The class files can be rerouted to another directory using the `-d` option of the `javac` command. Also, if you are compiling a set of files within the package hierarchy (that is, not at the top-level source directory), then you must use the `-sourcepath` option. For example:

```

> cd JavaProject/BankPrj/src/banking/domain
> javac -sourcepath JavaProject/BankPrj/src \
        -d JavaProject/BankPrj/class *.java
  
```

Directory Layout and Packages

Deployment

An application can be deployed on a client machine without manipulating the user's CLASSPATH environment variable.

If the application is deployed as a JAR file, then that file should be copied into the "library extension" directory. This directory exists in `path/jre/lib/ext/`; for example:

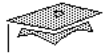
<code>/usr/jdk1.2/jre/lib/ext/</code>	Solaris Operating Environment
<code>C:\jdk1.2\jre\lib\ext\</code>	Microsoft Windows environment

If the application is deployed as a hierarchy of `.class` files, then place the complete package hierarchy under the "JRE classes" directory. This directory exists in `path/jre/classes/`; for example:

<code>/usr/jdk1.2/jre/classes/</code>	Solaris Operating Environment
<code>C:\jdk1.2\jre\classes\</code>	Microsoft Windows environment

There are no special environment variables (such as `JDK_HOME`, `JAVA_HOME` or `CLASSPATH`) required; moreover, existing settings might result in improper operation. Check these settings for possible conflicts.

- ✓ **In releases prior to Java 2 SDK, the CLASSPATH environment variable was required. For the compiler to locate the `shipping.domain.Company class while compiling Vehicle.java`, the CLASSPATH environment had to include the following package path (prior to JDK1.2): For Solaris Operating Environments, `CLASSPATH=home/anton/mypackages:.` For Microsoft Windows environments, set `CLASSPATH=\home\anton\mypackages;` had to be included.**



Terminology Recap

- Class – The source-code blueprint for a run-time object
- Object – An entity of a class
AKA: instance
- Attribute – A data element of an object
AKA: data member, instance variable, data field
- Method – A behavioral element of an object
AKA: algorithm, function, procedure
- Constructor – A "method-like" construct used to initialize a new object
- Package – A grouping of classes and/or sub-packages

Terminology Recap

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

- Class – A way to define new types in the Java programming language. The class can be considered as a blueprint—a model of the object 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 are not members (for example, they are not inherited).
- Package – A grouping of classes and/or subpackages.



Using the Java API Documentation

- A set of hypertext markup language (HTML) files provides information about the API
- One package contains hyperlinks to information on all of the classes
- A class document includes the class hierarchy, a description of the class, a list of member variables, a list of constructors, and so on

Using the Java Technology API Documentation

A set of HTML files document the supplied application programming interface (API). The layout of this documentation is hierarchical, so that the home page lists all the packages as hyperlinks. If a particular package hotlink is selected, the classes that are members of that package are listed. Selecting a class hotlink from a package page presents a page of information about that class. Figure 2-2 shows one such class.

Using the Java Technology API Documentation



Figure 2-2 Java Technology API Documentation With HTML

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 member variables
- A list of constructors
- A list of methods
- A detailed list of variables, with descriptions of the purpose and use of each variable
- A detailed list of constructors, with descriptions
- A detailed list of methods, with descriptions

Exercise: Using Objects and Classes



Exercise objective – You will explore the Java 2 SDK API documentation. You will write, compile, and run three versions of a program that explores the use of object-oriented data hiding and encapsulation. You will create a program that models a simple bank account.

Preparation

To successfully complete this lab, you must understand the concepts of classes and objects.

Tasks

In a Web browser, view the `lab_files.html` page that is at the top-level of the SL275 directory on your computer. There will be a summary of each exercise and a link to a page that gives a detailed explanation of the exercise.

Go to the SL275 directory on your computer and change to the directory for this module (`mod02`). A listing of this directory will show three subdirectories: one for each of the exercises below.

Exercise 1: Use the Java API Documentation (Level 1 Lab)

In this exercise you will explore the Java 2 SDK class API documentation using a Web browser. You will hunt for a specific method in the `String` class.

Exercise 2: Explore Encapsulation (Level 2 Lab)

In this exercise you will explore the purpose of proper *object encapsulation*. You will create a class in three steps demonstrating the use of information hiding.

Exercise 3: Create a Simple Banking Package (Level 2 Lab)

In this exercise you will create the `Account` class in the banking package. This will introduce you to the Banking project which we will return to in several labs up to Module 9.

Exercise: Using Objects and Classes

Exercise Summary



Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercises.

✓ *If you do not have time to spend on discussion, just highlight the key concepts students should have learned from the lab exercise.*

- Experiences

✓ *Ask students what their overall experiences with this exercise have been. You might want to go over any trouble spots or especially confusing areas at this time.*

- Interpretations

✓ *Ask students to interpret what they observed during any aspects of this exercise.*

- Conclusions

✓ *Have students articulate any conclusions they have reached as a result of this exercise experience.*

- Applications

✓ *Explore with the students how they might apply what they learned in this exercise to situations at their workplace.*

Check Your Progress

Before continuing on to the next module, check to be sure that you can:

- Define modeling concepts: *abstraction*, *encapsulation*, and *packages*
- Discuss why Java technology application code is reusable
- 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
- In a Java technology program, identify the following:
 - ▼ The `package` statement
 - ▼ The `import` statements
 - ▼ Classes, methods, and attributes
 - ▼ Constructors
- Use the Java technology application programming interface (API) online documentation

Think Beyond

Does your organization spend enough time on analysis and design?

What domain objects and relationships appear in your existing applications?

Objectives

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

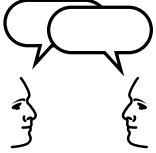
- Use comments in a source program
- Distinguish between valid and invalid identifiers
- Recognize Java technology keywords
- List the eight primitive types
- Define literal values for numeric and textual types
- Define the terms *primitive variable* and *reference variable*
- Declare variables of class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

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

- ✓ **Documentation comments are explained in this module. The complete definition of the documentation system used by the JavaSoft™ team and created by javadoc is defined in “The Design of Distributed Hyperlinked Programming Documentation,” a paper by Lisa Friendly. It is available from http://www.javasoft.com/doc/api_documentation.html.**

Relevance

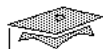
- ✓ *Present the following questions to stimulate the students and get them thinking about the issues and topics presented in this module. They are not expected to know the answers to all of these questions. Hold discussions where students have input; otherwise, if no one can propose answers, begin the lecture for this module.*



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

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

- ✓ *Java is not a pure OO language. It uses primitive data types to improve performance of low-level operations such as integer and floating point arithmetic and comparisons.*



Comments

- Three permissible styles of comment in a Java technology program are:

```
// comment on one line
```

```
/* comment on one  
or more lines */
```

```
/** documentation comment */
```

Comments

There are three permissible styles for inserting comments:

```
// comment on one line  
/* comment on one or more lines */  
/** documentation comment */
```

✓ **The last comment containing two asterisks at the beginning and one at the end is correct.**

Documentation comments placed immediately before a declaration (of a variable, method, or class) indicate that the comments should be included in any automatically generated documentation (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 discussed in the [docs/tooldocs/solaris](#) directory of the API documentation for Java 2 SDK.



Semicolons, Blocks, and Whitespace

- A *statement* is one or more lines of code terminated by a semicolon (;):

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

- A *block* is a collection of statements bound by opening and closing braces:

```
{
  x = y + 1;
  y = x + 1;
}
```

Semicolons, Blocks, and Whitespace

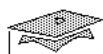
In the Java programming language, a statement is one or more lines of code terminated with a semicolon (;).

For example,

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

is the same as

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



Semicolons, Blocks, and Whitespace

- You can use a *block* in a *class* definition:

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

- You can nest block statements
- Any amount of *whitespace* is allowed in a Java program

Semicolons, Blocks, and Whitespace

A *block* or a compound statement is a collection of statements bound by opening and closing braces (`{ }`). *Block* statements are also used to group declarations belonging to a class.

You can nest blocks of statements. Consider the `TestGreeting` class, which consists of the `main` method. The body of this method is a block of statements that is a single unit, the unit itself being one of a group of items in the class `TestGreeting` block.

The following are other examples of block statements or groupings:

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

Semicolons, Blocks, and Whitespace

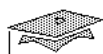
```
// a block used in a class definition
public class MyDate {
    private int day;
    private int month;
    private int year;
}

// an example of a block statement nested within
// another block statement
while ( i < large ) {
    a = a + i;
    // nested block
    if ( a == max ) {
        b = b + a;
        a = 0;
    }
}
```

You can have *whitespace* between elements of the source code. Any amount of whitespace is allowed. You can use whitespace, including spaces, tabs, and newlines, to enhance the visual appearance of your source code.

```
{
    int x;

    x = 23 * 54;
}
```

Sun Educational Services

Identifiers

- Are names given to a variable, class, or method
- Can start with a Unicode letter, underscore(_), or dollar sign(\$)
- Are case sensitive and have no maximum length
- Examples:

```
identifier  
userName  
user_name  
_sys_var1  
$change
```

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.

- ✓ **Specifically, the Java Language Specification states that an identifier starts with a Unicode letter and is followed by any number of Unicode letters or digits. The specification lists the Unicode letters and digits. Unicode represents an extended ASCII set capable of handling international characters.**

The following are valid identifiers:

- identifier
- userName
- user_name
- _sys_var1
- \$change

Identifiers

Java technology sources are in 16-bit Unicode rather than 8-bit ASCII text, so a letter is a considerably wider definition than just a to z and A to 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 character.

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 keyword. Java keywords are discussed next.

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 Keywords

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

Table 3-1 Java Keywords

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

✓ **A discussion of `strictfp` is beyond the scope of this course.**

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

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; the size and representation of all types is fixed and is *not* implementation dependent.
- `goto` and `const` are keywords that are not used in the Java programming language.

✓ **There is no longer a keyword `byvalue`. The keywords `const` and `goto` still exist, but remain unused. While `true` and `false` would seem to be keywords, they are actually boolean literals. The word `null` is also a literal. You can confirm this by reading “The Java Language Specification,” ISBN 0-201-63451-1 available from <http://java.sun.com/docs/books/jls/index.html>**



Primitive Types

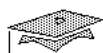
- The Java programming language defines eight primitive types:
 - ▼ Logical `boolean`
 - ▼ Textual `char`
 - ▼ Integral `byte, short, int, and long`
 - ▼ Floating `double and float`

Basic Java Types

Primitive Types

The Java programming language defines literal values for eight *primitive* data types and one special type. The primitive types can be considered in four categories:

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



Logical – boolean

- The boolean data type has two literals, `true` and `false`.
- For example, the statement:

```
boolean truth = true;
```

declares the variable `truth` as boolean type and assigns it a value of `true`.

Basic Java Types

Logical – boolean

Logical values have two states: on or off, true or false, or yes or no. Such a value is represented by the boolean type. 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 only boolean values can be used.



Textual – char and String

char

- Represents a 16-bit Unicode character
- Must have its literal enclosed in single quotes(' ')
- Uses the following notations:

'a'	The letter <i>a</i>
'\t'	A tab
'\u????'	A specific Unicode character, ????, is replaced with exactly four hexadecimal digits (for example, '\u03A6' is the Greek letter phi Φ)

Basic Java Types

Textual – char and String

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 (for example, '\u03A6' is the Greek letter phi Φ)

The String type, which is not a primitive but a class, is used 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.

✓ **For more on escape codes refer students to the Java Language Specification Web site:**
<http://java.sun.com/docs/books/jls/html/3.doc.html#101089>



Textual – char and String

String

- Is not a primitive data type; it is a class
- Has its literal enclosed in double quotes (" ")

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

- Can be used as follows:

```
String greeting = "Good Morning !! \n";
String errorMessage = "Record Not Found !";
```

Basic Java Types

Textual – char and String (Continued)

A String literal is enclosed in double quote marks:

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

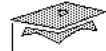
Some examples of the declarations and initialization of char and String type variables are:

```
// 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;
```



Integral – byte, short, int, and long

- Uses three forms – Decimal, octal, or hexadecimal
 - 2 The decimal value is two
 - 077 The leading zero indicates an octal value
 - 0xBAAC The leading 0x indicates a hexadecimal value
- Has a default `int`
- Defines `long` by using the letter `L` or `l`

Basic Java 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 The decimal value is two
- 077 The leading zero indicates an octal value
- 0xBAAC The leading 0x indicates a hexadecimal value

Note – All integral types in the Java programming language are signed numbers.

Basic Java Types

Integral – byte, short, int, and long (Continued)

Integral literals are of type `int` unless explicitly followed by the letter "L." The L indicates a `long` value. In the Java programming language, you can use either an uppercase or lowercase L, but lowercase is a poor choice because it is usually hard to distinguish it from the digit 1. Long versions of the literals shown previously are:

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



Integral – byte, short, int, and long

- Integral data types have the following ranges:

Integer Length	Name or Type	Range
8 bits	byte	-2^7 to 2^7-1
16 bits	short	-2^{15} to $2^{15}-1$
32 bits	int	-2^{31} to $2^{31}-1$
64 bits	long	-2^{63} to $2^{63}-1$

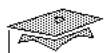
Basic Java Types

Integral – byte, short, int, and long (Continued)

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 two's complement and is platform independent.

Table 3-2 Integral Data Types – Size and Range

Integer Length	Name or Type	Range
8 bits	byte	-2^7 to 2^7-1
16 bits	short	-2^{15} to $2^{15}-1$
32 bits	int	-2^{31} to $2^{31}-1$
64 bits	long	-2^{63} to $2^{63}-1$



Floating Point – float and double

- Default is double
- Floating point literal includes either a decimal point or one of the following:
 - ▼ E or e (add exponential value)
 - ▼ F or f (float)
 - ▼ D or d (double)

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

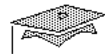
Basic Java Types

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).

- 3.14 A simple floating-point value (a double)
- 6.02E23 A large floating-point value
- 2.718F A simple float size value
- 123.4E-306D A large double 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 – float and double

- Floating point data types have the following ranges:

Float Length	Name or Type
32 bits	float
64 bits	double

Basic Java Types

Floating Point – float and double (Continued)

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	float
64 bits	double

Note – Floating point literals are double unless explicitly declared 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

- Beyond primitive types all others are reference types
- A *reference variable* contains a "handle" to an object
- Example:

```
1 public class MyDate {
2     private int day = 1;
3     private int month = 1;
4     private int year = 2000;
5 }

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

Java Reference Types

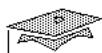
As you have seen, there are eight primitive Java types: boolean, char, byte, short, int, long, float, and double. All other types refer to objects rather than primitives. Variables that refer to objects are *reference variables*. For example, we can define a class `MyDate`:

```
1 public class MyDate {
2     private int day = 1;
3     private int month = 1;
4     private int year = 2000;
5 }
```

We can then use `MyDate` like this:

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

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



Constructing and Initializing Objects

- Calling `new Xxx()` to allocate space for the new object results in:
 - ▼ Memory Allocation: Space for the new object is allocated and instance variables are initialized to their default values (e.g. 0, false, null, and so on)
 - ▼ Explicit attribute initialization is performed
 - ▼ A constructor is executed
 - ▼ Variable assignment is made to reference the object
- Example:


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

Constructing and Initializing Objects

You have seen how you must execute a call to `new Xxx()` 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:

- First, the space for the new object is allocated and initialized to the form of zero or null. In the Java programming language, this phase is indivisible to ensure that you cannot have an object with random values in it.
- Second, any explicit initialization is performed.
- 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).
- Last, the variable is assigned the reference to the new object in heap memory.

This section investigates these phases.



Sun Educational Services

Memory Allocation and Layout

- A declaration allocates storage only for a reference:


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

my_birth	????
----------	------
- Use the `new` operator to allocate space for `MyDate`:


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

my_birth	????
day	0
month	0
year	0

Constructing and Initializing Objects

Memory Allocation and Layout

In a method body, the declaration

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


allocates storage only for the reference:

my_birth	????
----------	------

The keyword `new` implies allocation and initialization of storage.

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

my_birth	????
day	0
month	0
year	0



Sun Educational Services

Explicit Attribute Initialization

- Initialize the attributes:

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

my_birth	????
day	1
month	1
year	2000

- The default values are taken from the attribute declaration in the class

Constructing and Initializing Objects

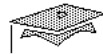
Explicit Attribute Initialization

If you place simple assignment expressions in your member declarations, you can perform explicit member initialization during construction of your object.

In our MyDate class we declared the explicit initialization of all three attributes:

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

my_birth	????
day	1
month	1
year	2000



Executing the Constructor

- Execute the matching constructor:

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

my_birth	????
day	22
month	7
year	1964

- In the case of an overloaded constructor, the first constructor may call another

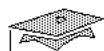
Constructing and Initializing Objects

Executing the Constructor

The final stage of initialization of a new object is to call the constructor. The constructor allows 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.

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

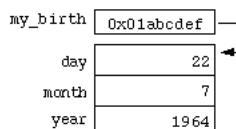
my_birth	????
day	22
month	7
year	1964



Variable Assignment

- Assign newly created object to reference variable:

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

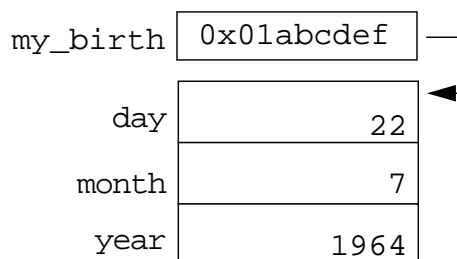


Constructing and Initializing Objects

Variable Assignment

The assignment then sets up the reference variable so that it refers properly to the newly created object.

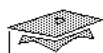
```
MyDate my_birth = new MyDate(22, 7, 1964);
```



Constructing and Initializing Objects

This Is Not the Whole Story

It turns out that object construction and initialization is much more complex than we have describe here. We will revisit this topic in "Constructing and Initializing Objects: A Slight Reprise" on page 6-34 in Module 6, "Inheritance."



Assignment of Reference Variables

- Consider the following code fragment:

```
int x = 7;
int y = x;
MyDate s = new MyDate(22, 7, 1964);
MyDate t = s;
t = new MyDate(22, 12, 1964);
```

Assignment of Reference Types

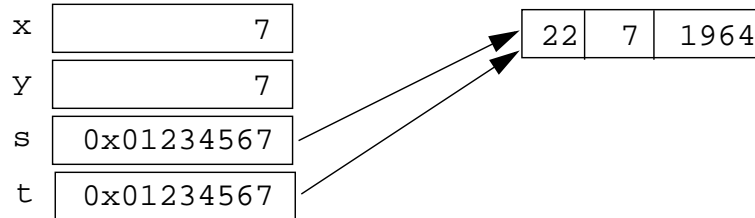
In the Java programming language, a variable declared with a type of class is referred to as a reference type. This is because it refers to a non-primitive type. This has consequences for the meaning of assignment. Consider this code fragment:

```
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 seven, and this value is copied into `y`. Both `x` and `y` are independent variables and further changes to either do not affect the other.

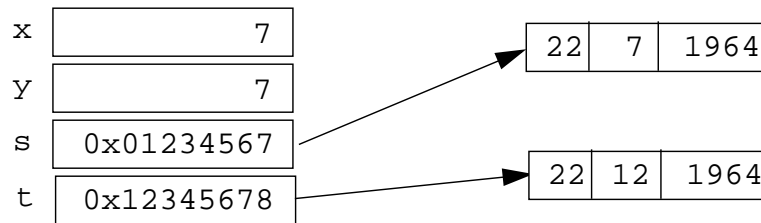
Assignment of Reference Types

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.



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

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





Pass-by-Value

- The Java programming language only passes arguments by value
- When an object instance is passed as an argument to a method, the value of the argument is a *reference* to the object
- The *contents* of the object can be changed in the called method, but the object reference is never changed

Pass-by-Value

The Java programming language passes arguments only “by value” that is, the argument *cannot be changed* by the method called. When an object instance is passed as an argument to a method, the value of the argument is a reference to the object. The *contents* of the object can be changed in the called method, but the object reference is never changed.

- ✓ **There has been much discussion on the `ses_java` e-mail alias about “is Java pass-by-value, or pass-by-reference for objects, and so on.” The main issue is that pass-by-reference allows a method to change the value of a variable in the context that the method was called. For example, in C you can pass a reference to an integer variable and modify the value of the calling-variable. This is prohibited in Java.**

```
void changeInt(int* variable) {
    *variable = 5;
}
void main() {
    int val = 4;
    changeInt(&val);
    printf("val = %d\n", val);
}
```

Pass-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        date.print();
31
32        // Now change the day attribute
33        // through the object reference
34        changeObjectAttr(date);
35        // What is the current value?
36        date.print();
37    }
38 }
39

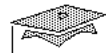
```


Pass-by-Value

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 day attribute of the `MyDate` object is changed by the `changeObjectAttr` method.



The `this` Reference

Here are a few uses of the `this` keyword:

- To reference local attribute and method members within a local method or constructor
 - ▼ This is used to disambiguate a local method or constructor variable from an instance variable
- To pass the current object as a parameter to another method or constructor

The `this` Reference

There are a few uses of the `this` keyword:

- To reference local attribute and method members within a local method or constructor
- To pass the current object as a parameter to another method

The following class definition demonstrates all three uses.

```
1 public class MyDate {
2     private int day = 1;
3     private int month = 1;
4     private int year = 2000;
```

Initially, we create a simple date class with three attributes.

The this Reference

```
5
6   public MyDate(int day, int month, int year) {
7       this.day    = day;
8       this.month  = month;
9       this.year   = year;
10  }
```

In this first constructor, we use the `this` reference to distinguish the parameters from the object's attributes.

```
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 more_days) {
18      MyDate new_date = new MyDate(this);
19
20      new_date.day = new_date.day + more_days;
21      // Not Yet Implemented: wrap around code...
22
23      return new_date;
24  }
```

This final example shows a method `addDays` that constructs and returns a new `MyDate` object. This new date is constructed from the current object using the third constructor.

The this Reference

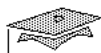
```
25 public void print() {
26     System.out.println("MyDate: " + day + "-" + month
27                         + "-" + year);
28 }
29 }
```

This last method will be used to test our `MyDate` code. The following `TestMyDate` class can be used to test these constructors and methods:

```
1 public class TestMyDate {
2     public static void main(String[] args) {
3         MyDate my_birth = new MyDate(22, 7, 1964);
4         MyDate the_next_week = my_birth.addDays(7);
5
6         the_next_week.print();
7     }
8 }
```

Running `TestMyDate` gives the following output:

```
MyDate: 29-7-1964
```



Java Coding Conventions

- Packages:

```
package banking.objects;
```

- Classes:

```
class SavingsAccount
```

- Interfaces:

```
interface Account
```

- Methods:

```
balanceAccount()
```

Java Coding Conventions

The following are coding conventions of the Java programming language:

- *Packages* – Package names should be nouns in lower case.

```
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()
```

 Sun Educational Services

Java Coding Conventions

- Variables:
`currentCustomer`
- Constants:
`HEAD_COUNT`
`MAXIMUM_SIZE`

Java Coding Conventions

✓ **Native methods use underscores to create complex names. For example, `java.lang.String` becomes `java_lang_String`.**

- *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`

Java Coding Conventions

- *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 ) {
    do something
} else {
    do something else
}
```

- *Spacing* – Place only a single statement on any line, and use two or four-space indentations to make your code readable. The number of spaces can vary depending on what code standards are used.
- *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 – @see 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 from http://www.javasoft.com/doc/api_documentation.html.

Note – For more information on Sun's Java coding conventions refer to the Web page:

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

Exercise: Using Objects



Exercise objective – Using the correct Java keywords, write a program to create a class and an object from the class. Compile and run the program; then verify that the references are assigned and manipulated as described in this module.

Preparation

To successfully complete this lab, you must be able to compile and run a Java program. You also need to be familiar with the object-oriented concepts of classes and objects, and with the concept of references.

Tasks

In a Web browser view the `lab_files.html` page that is at the top-level of the SL275 directory on your computer. There will be a summary of each exercise and a link to a page that gives a detailed explanation of the exercise.

Go to the SL275 directory on your computer and change to the directory for this module (`mod03`). A listing of this directory will show two subdirectories: one for each of the exercises below.

Exercise 1: Investigate Reference Assignment (Level 2 Lab)

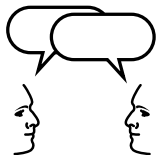
In this exercise you will investigate Java reference variables; object creation and reference variable assignment.

Exercise 2: Create Customer Accounts (Level 2 Lab)

In this exercise you will expand the Banking project by adding a Customer class. A customer will contain one Account object.

Exercise: Using Objects

Exercise Summary



Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercises.

- ✓ **If you do not have time to spend on discussion, just highlight the key concepts students should have learned from the lab exercise.**

- Experiences

- ✓ **Ask students what their overall experiences with this exercise have been. You might want to go over any trouble spots or especially confusing areas at this time.**

- Interpretations

- ✓ **Ask students to interpret what they observed during any aspects of this exercise.**

- Conclusions

- ✓ **Have students articulate any conclusions they have reached as a result of this exercise experience.**

- Applications

- ✓ **Explore with the students how they might apply what they learned in this exercise to situations at their workplace.**

Check Your Progress

Before continuing on to the next module, check to be sure that you can:

- 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 consequences of assigning variables of class type

Think Beyond

Can you think of examples of classes and objects in your existing applications?

Objectives

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

- Distinguish between instance and local variables
- Describe how instance variables are initialized
- 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 discusses variables, operators, and arithmetic expressions, and lays out the different control structures governing the path of execution.

Relevance

- ✓ **Present the following questions to stimulate the students and get them thinking about the issues and topics presented in this module. They are not expected to know the answers to all of these questions. Hold discussions where students have input; otherwise, if no one can propose answers, begin the lecture for this module.**



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

- What types of variables are useful to programmers (for instance, programmers of other languages will want to know how the Java programming language defines and handles global and local variables)?
 - Can multiple classes have variables with the same name and, if so, what is their scope?
- ✓ **Good use of variables greatly improves a computer program all around, from execution efficiency to maintainability. This module discusses the implications of where and how variables are declared, and the implications of how they are used in expressions.**
 - What types of control structures are used in other languages? What methods do languages in general employ for flow control and for discontinuing the flow (such as in a loop or switch)?
 - ✓ **Control structures are what govern the path of program execution. Like variables, good use of control structures yields efficient, straightforward programs. In `while` loops, certain conditions have to be met in order to exit the loop. In `switch` controls, a `break` is used to exit. Older languages used the `goto` statement, for lack of a better way, to control the program flow. The Java programming language does not use `goto`.**
 - ✓ **Focus the students' attention on the following areas:**
 - **Use variables effectively.**
 - **Determine the implications of how and where variables are declared.**
 - **Learn the implications of how variables are used.**
 - **Learn about operator usage and expressions in a Java software program.**
 - **Control a Java software program's execution path with control structures.**



Variables and Scope

Local variables are:

- Variables that are defined inside a method and are called *local*, *automatic*, *temporary*, or *stack* variables
- Variables that are created when the method is executed are destroyed when the method is exited
- Variables that must be initialized before they are used or compile-time errors will occur

Expressions

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. You can also define variables as method parameters or constructor parameters.

Variables defined inside a method are called *local* variables, but are sometimes referred to as *automatic*, *temporary*, or *stack* variables.

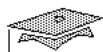
Expressions

Variables and Scope (Continued)

Variables defined outside a method are created when the object is constructed using the `new Xxx()` call. There are two possible kinds of variables. The first kind is a class variable that is declared using the `static` keyword. This is done when the class is loaded. Class variables continue to exist for as long as the class exists. The second kind is an instance variable that is declared without the `static` keyword. Instance variables continue to exist for as long as the object is referenced. Instance variables are sometimes referred to as member variables, because they are members of the class. The `static` variable is discussed later in this course in more detail.

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 page 4-5.



Variable Scope Example

```

public class ScopeExample {
    private int i=1;

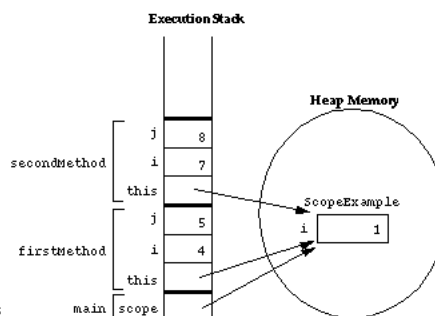
    public void firstMethod() {
        int i=4, j=5;

        this.i = i + j;
        secondMethod(7);
    }
    public void secondMethod(int i) {
        int j=8;
        this.i = i + j;
    }
}

public class TestScoping {
    public static void main(String[] args) {
        ScopeExample scope = new ScopeExample();

        scope.firstMethod();
    }
}

```



Expressions

Variable Scope Example

```

public class ScopeExample {
    private int i=1;

    public void firstMethod() {
        int i=4, j=5;

        this.i = i + j;
        secondMethod(7);
    }
    public void secondMethod(int i) {
        int j=8;
        this.i = i + j;
    }
}

public class TestScoping {
    public static void main(String[] args) {
        ScopeExample scope = new ScopeExample();

        scope.firstMethod();
    }
}

```

Expressions

Variable Initialization

No variable in a Java program can be used before being initialized. When an object is created, instance variables are initialized with the following values at the time the storage is allocated:

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 refers to no object. An attempt to use the object it refers to causes an exception. Exceptions are errors that occur at runtime and are discussed in a later module.

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
}
```

Expressions

Operators

The Java software 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 in Order of Precedence

Separator	. [] () ; ,
-----------	-------------

Associative	Operators
R to L	++ -- + - ~ ! (data type)
L to R	* / %
L to R	+ -
L to R	<< >> >>>
L to R	< > <= >= instanceof
L to R	== !=
L to R	&
L to R	^
L to R	
L to R	&&
L to R	
R to L	?:
R to L	= *= /= %= += -= <<= >>= >>>= &= ^= =

Note – The instanceof operator is unique to the Java programming language and is discussed in Module 6, "Inheritance."



Logical Operators

- The boolean operators are:

! – NOT & – AND
| – OR ^ – XOR

- The Short-Circuit boolean operators are:

&& – AND || – OR

- These operators can be used as follows:

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

Expressions

Logical Operators

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

Relational and logical operators return a boolean result. There is no automatic conversion of `int` to `boolean`.

```
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 are discussed on the following page.


Expressions

Short-Circuit Logical Operators

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

```
MyDate d;  
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.



Sun Educational Services

Bitwise Logical Operators

- The Integer *bitwise* operators are:

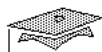
~ - Complement	& - AND
^ - XOR	- OR
- Byte-sized examples:

$\sim \begin{array}{r} \boxed{01001111} \\ \hline \boxed{10111000} \end{array}$	$\& \begin{array}{r} \boxed{00101101} \\ \boxed{01001111} \\ \hline \boxed{00001101} \end{array}$
$\wedge \begin{array}{r} \boxed{00101101} \\ \boxed{01001111} \\ \hline \boxed{01100010} \end{array}$	$\mid \begin{array}{r} \boxed{00101101} \\ \boxed{01001111} \\ \hline \boxed{01101111} \end{array}$

Expressions

Bitwise Logical Operators

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. The bit shift operators are discussed later in this course.



Right-Shift Operators \gg and \ggg

- *Arithmetic* or *signed* right shift (\gg) is used as follows:

128 \gg 1 returns $128/2^1 = 64$
256 \gg 4 returns $256/2^4 = 16$
-256 \gg 4 returns $-256/2^4 = -16$

- ▼ The sign bit is copied during the shift
- A *logical* or *unsigned right shift* operator (\ggg) is:
 - ▼ Used for bit patterns
 - ▼ Not copied during the shift

Expressions

Right-Shift Operators \gg and \ggg

The Java programming language provides two right-shift operators.

The operator \gg performs an *arithmetic* or *signed* right shift. The result of this shift is that the first operand is divided by two raised to the number of times specified by the second operand. For example:

128 \gg 1 returns $128/2^1 = 64$
256 \gg 4 returns $256/2^4 = 16$
-256 \gg 4 returns $-256/2^4 = -16$

The \gg operator results in the sign bit being copied during the shift.

The *logical* or *unsigned* right shift operator \ggg works on the bit pattern rather than the arithmetic meaning of a value and always places zeros in the most significant bits. For example:

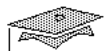
1010 ... \gg 2 gives 111010 ...
1010 ... \ggg 2 gives 001010 ...

Expressions

Right-Shift Operators >> and >>> (Continued)

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

Note – The `>>>` operator is permitted only on integral types, and 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 has usually become a signed shift.



Left-Shift Operator (<<)

- Left-shift works as follows:

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

Expressions

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
```



Shift Operator Examples

1357 = `0000000000000000000000000000000001010101001101`

-1357 = `1111111111111111111111111111111110101010110011`

1357 >> 5 = `00000000000000000000000000000000000000010101010`

-1357 >> 5 = `11111111111111111111111111111111101010101`

1357 >>> 5 = `0010101010`

-1357 >>> 5 = `00000011111111111111111111111111101010101`

1357 << 5 = `00000000000000000000000000001010101001110100000`

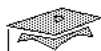
-1357 << 5 = `1111111111111111111111111010101011001100000`

Expressions

Shift Operator Examples

These examples show the bit patterns of a positive and a negative number and the bit patterns resulting from the three shift operators: `>>`, `>>>`, and `<<`.

Note – The code that generated these examples (including printing out the complete bit pattern) can be found in file: `mod04/examples/TestShift.java`.



String Concatenation With +

- The + operator:
 - ▼ Performs String concatenation
 - ▼ Produces a new String:

```
String salutation = "Dr.";
String name = "Pete" + " " + "Seymour";
String title = salutation + " " + name;
```
- One argument must be a String object
- Non-strings are converted to String objects automatically

Expressions

String Concatenation With +

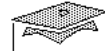
The + operator performs a concatenation of String objects, producing a new String.

```
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 is converted to a String. All objects can be converted to a String automatically, although the result might be rather cryptic. The object that is not a string is converted to a string equivalent using the toString() member function.



Casting

- If information is lost in an assignment, the programmer must confirm the assignment with a typecast.
- The assignment between `long` and `int` requires an explicit cast.

```
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
```

Expressions

Casting

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

Where information would be lost in an assignment, the compiler requires that you confirm the assignment with a typecast. This can be done, for example, by “squeezing” a `long` value into an `int` variable. Explicit casting is done like this:

```
long bigValue = 99L;
int squashed = (int)(bigValue);
```

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 "Casting Objects" on page 6-16 in Module 6, "Inheritance."



Promotion and Casting of Expressions

- Variables are automatically promoted to a longer form (such as `int` to `long`).
- Expression is *assignment compatible* if the variable type is at least as large (the same number of bits) as the expression type.

```
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
```

Expressions

Promotion and Casting of Expressions

Variables can be automatically promoted 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 (the number of bits) as the expression type.

Expressions

Promotion and Casting of Expressions (Continued)

For the + operator, when the two operands are of primitive numeric types, the result is at least an `int` and has a value calculated by promoting the operands to the result type or promoting the result to the wider type of the operands. This might result in overflow or loss of precision.

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 typecast is done as:

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

then the code works.



Branching Statements

The `if, else` statement syntax:

```
if (boolean expression) {  
    statement or block;  
}  
  
if (boolean expression) {  
    statement or block;  
} else if (boolean expression) {  
    statement or block;  
} else {  
    statement or block;  
}
```

Branching Statements

Conditional statements allow for 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.

`if, else` Statements

The basic syntax for `if, else` statements is:

```
if (boolean expression) {  
    statement or block;  
}  
  
if (boolean expression) {  
    statement or block;  
} else if (boolean expression) {  
    statement or block;  
} else {  
    statement or block;  
}
```


Branching Statements

if, else Statements (Continued)

Example

```
int count;
count = getCount(); // a method defined in the program
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/C++ because an `if()` 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 is optional and you can omit it if no action is to be taken when the tested condition is false.



Branching Statements

The `switch` statement syntax:

```
switch (expr1) {  
    case constant2:  
        statements;  
        break;  
    case constant3:  
        statements;  
        break;  
    default:  
        statements;  
        break;  
}
```

Branching Statements

`switch` Statement

The `switch` statement syntax is:

```
switch (expr1) {  
    case constant2:  
        statements;  
        break;  
    case constant3:  
        statements;  
        break;  
    default:  
        statements;  
        break;  
}
```

Branching Statements

`switch` Statement (Continued)

Note – In the `switch (expr1)` statement, `expr1` must be assignment compatible with an `int` type. Promotion occurs with `byte`, `short`, or `char` types. Floating point, long expressions, or class references (including `Strings`) are not permitted.

The optional `default` label is used to specify 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.

Example 1

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

Example 1 configures a car object based on the `carModel`. If `carModel` is the integer constant `DELUXE`, then A/C 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.

Branching Statements

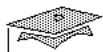
switch Statements (Continued)

Example 2

```
switch ( carModel ) {
    case THE_WORKS:
        addGoldPackage();
        add7WayAdjustableSeats();
    case DELUXE:
        addFloorMats();
        addAirConditioning();
    case STANDARD:
        addRadio();
        addDefroster();
    default:
        addWheels();
        addEngine();
}
```

Example 2 solves the redundant method calls in the previous example by allowing the flow of control descend through multiple case blocks. For example, if the `carModel` is the `THE_WORKS`, then the gold package and 7-way adjustable seats are adding to this car, plus floor mats, A/C, a radio, the defroster, and of course, wheels and an engine. However, if the `carModel` is only a `STANDARD`, then only a radio, defroster, wheels, and an engine are added.

Note – Nine out of ten `switch` statements will need breaks in each case block. Forgetting the `break` statement is the number one programming error in using `switch` statements.



Looping Statements

The `for` statement:

```
for (init_expr; boolean testexpr; alter_expr) {  
    statement or block;  
}
```

Example:

```
for (int i = 0; i < 10; i++) {  
    System.out.println("Are you finished yet?");  
}  
System.out.println("Finally!");
```

Looping Statements

Loop statements allow for the repeated execution of blocks of statements. The Java programming language supports three types of loop constructs: `for`, `while`, and `do` loops. `for` and `while` loops test the loop condition before executing the loop body, whereas `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.

for Loops

The `for` loop syntax is:

```
for (init_expr; boolean testexpr; alter_expr3) {  
    statement or block;  
}
```

Looping Statements

for Loops (Continued)

Example

```
for (int i = 0; i < 10; i++) {  
    System.out.println("Are you finished yet?");  
}  
System.out.println("Finally!");
```

Note – The Java programming language allows 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.

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.



Looping Statements

The while loop:

```
while (boolean) {  
    statement or block;  
}
```

Example:

```
int i = 0;  
  
while (i < 10) {  
    System.out.println("Are you finished yet?");  
    i++;  
}  
System.out.println("Done");
```

Looping Statements

while Loops

The while loop syntax is:

```
while (boolean) {  
    statement or block;  
}
```

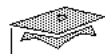
Example

```
1 int i = 0;  
2  
3 while (i < 10) {  
4     System.out.println("Are you finished yet?");  
5     i++;  
6 }  
7 System.out.println("Done");
```

Looping Statements

while Loops (Continued)

Ensure that the loop control variable is appropriately initialized before the loop body begins execution, and ensure that the loop condition is true to begin with. You must update the control variable appropriately to prevent an infinite loop.



Looping Statements

The do/while statement:

```
do {  
    statement or block;  
} while (boolean test);
```

Example:

```
int i = 0;  
  
do {  
    System.out.println("Are you finished yet?");  
    i++;  
} while (i < 10);  
System.out.println("Done");
```

Looping Statements

do Loops

The syntax for the do loop is:

```
do {  
    statement or block;  
} while (boolean test);
```

Example

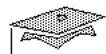
```
1 int i = 0;  
2 do {  
3     System.out.println("Are you finished yet?");  
4     i++;  
5 } while (i < 10);  
6 System.out.println("Done");
```

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

Looping Statements

do Loops (Continued)

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

- `break [label];`
- `continue [label];`
- `label: statement; // Where statement should
// be a loop`

Special Loop Flow Control

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

- `break [label];`
- `continue [label];`
- `label: statement; // Where statement should be a loop`

The `break` statement is used to prematurely exit from `switch` statements, loop statements, and labeled blocks.

The `continue` statement is used to skip over and jump to the end of the loop body.

The `label` statement identifies any valid statement to which control needs to be transferred. It is used to identify a compound statement that is a loop construct.

Special Loop Flow Control

The `break`, `continue`, and `label` statements can be used as follows:

- The `break` statement:

```
do {
    statement;
    if (condition is true) {
        break;
    }
    statement;
} while (boolean expression);
```

- The `continue` statement:

```
do {
    statement;
    if (boolean expression) {
        continue;
    }
    statement;
} while (boolean expression);
```

- The `break` statement with a label named `outer`:

```
outer:
do {
    statement;
    do {
        statement;
        if (boolean expression) {
            break outer;
        }
        statement;
    } while (boolean expression);
    statement;
} while (boolean expression);
```

Special Loop Flow Control

- The continue statement with a label named test :

```
test:
do {
    statement;
do {
    statement;
    if (condition is true) {
        continue test;
    }
    statement;
} while (condition is true);
statement;
} while (condition is true);
```

- ✓ **Labeled break and continue statements are used to jump directly out of nested loops. This facility removes one of the legitimate reasons for using goto. The Java programming language does not use goto, although it is a reserved word.**

Example

```
1 loop:while ( true ) {
2     for ( int i = 0; i < 100; i++ ) {
3         switch ( c = System.in.read() ) {
4             case -1:
5                 case '\n':
6                     // jumps out of while loop to line 13
7                     break loop;
8                 ...
9             }
10    }
11 }
12
13 test:for ( ... ) {
14     ...
15     while ( ... ) {
16         if ( j > 10 ) {
17             // jumps to the increment portion of for loop
18             // at line 13
19             continue test;
20         }
21     }
22 }
```

Exercise: Using Expressions



Exercise objective – You will write, compile, and run three programs that use identifiers, expressions, and control structures.

Preparation

In order to successfully complete this lab, you must be able to compile and run a Java program, and have a familiarity with flow control constructs.

Tasks

In a Web browser view the `lab_files.html` page that is at the top-level of the SL275 directory on your computer. There will be a summary of each exercise and a link to a page that gives a detailed explanation of the exercise.

Go to the SL275 directory on your computer and change to the directory for this module (`mod04`). A listing of this directory will show three subdirectories: one for each of the exercises below.

Exercise 1: Use Loops and Branching Statements (Level 1)

In this exercise you will use a simple integer loop and branching statements to play a fictitious game of “foo bar baz.”

Exercise 2: Conditionalize the `withdraw` Method (Level 2)

In this exercise you will modify the `withdraw` method to return a boolean value to specify if the transaction was successful.

Exercise 3: Use Nested Loops (Level 3)

In this exercise you will use nested loops to implement a string search operation.

Exercise: Using Expressions

Exercise Summary



Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercises.

- ✓ *If you do not have time to spend on discussion, just highlight the key concepts students should have learned from the lab exercise.*

- Experiences

- ✓ *Ask students what their overall experiences with this exercise have been. You might want to go over any trouble spots or especially confusing areas at this time.*

- Interpretations

- ✓ *Ask students to interpret what they observed during any aspects of this exercise.*

- Conclusions

- ✓ *Have students articulate any conclusions they have reached as a result of this exercise experience.*

- Applications

- ✓ *Explore with the students how they might apply what they learned in this exercise to situations at their workplace.*

Check Your Progress

Before continuing on to the next module, check to be sure that you can:

- Distinguish between instance and local variables
- Describe how instance variables are initialized
- 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

Think Beyond

What data types do most programming languages use to group similar data elements together?

How do you perform the same operation on all elements of a group (for example, a matrix)?

What data types does the Java programming language use?

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 multi-dimensional array
- Write code to copy array values from one array type to another

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

Relevance

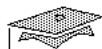
- ✓ **Present the following question to stimulate the students and get them thinking about the issues and topics presented in this module. They are not expected to know the answers to this question. Hold discussions where students have input; otherwise, if no one can propose answers, begin the lecture for this module.**



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

- What is the purpose of an array?

- ✓ **An array is often used to group similar data elements together, and perform the same operation on all elements of the group (for example, a matrix). This module discusses the Java programming language implementation of arrays, or groupings of similar elements.**
- ✓ **This module covers declaration and initialization of arrays, as well as their use. Implementation of a multi-dimensional array as an array of arrays is examined. Use of arrays' built-in `length` attribute for bounds checking is also discussed.**



Declaring Arrays

- Group data objects of the same type
- Declare arrays of primitive or class types

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

- Create space for a reference
- An array is an object; it is created with `new`

Declaring Arrays

Arrays are typically used to group objects of the same type. Arrays allow 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
```

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 dynamically allocated either by a `new` statement or by an array initializer.

The next section describes how to create and initialize the actual array.

Declaring Arrays

The code shown on page page 5-3, with square brackets after the variable name, is standard for C, C++, and the Java programming language. This format leads to complex forms of declaration that can be difficult to read. Therefore, the Java programming language allows an alternative form with the square brackets on the left:

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

The result is that you can consider a declaration as having the type part at the left, and the variable name at 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.

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



Creating Arrays

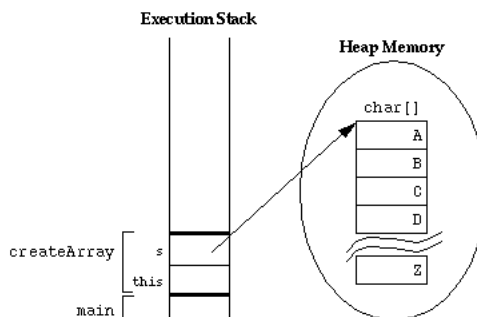
Use the `new` keyword to create an array object.

For example, a primitive (`char`) array:

```
public char[] createArray() {
    char[] s;

    s = new char[26];
    for ( int i=0; i<26; i++ ) {
        s[i] = 'A' + i;
    }

    return s;
}
```



Creating Arrays

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

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

The first line creates an array of 26 `char` values. Once created, 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:

```
s[0] = 'A';
s[1] = 'B';
...
```

The subscript used to index the individual array elements always begins from 0, and must be maintained in the legal range—greater than or equal to zero and less than the array length. Any attempt to access an array element outside these bounds causes a runtime exception. More elegant ways of initializing arrays are described in the next section.



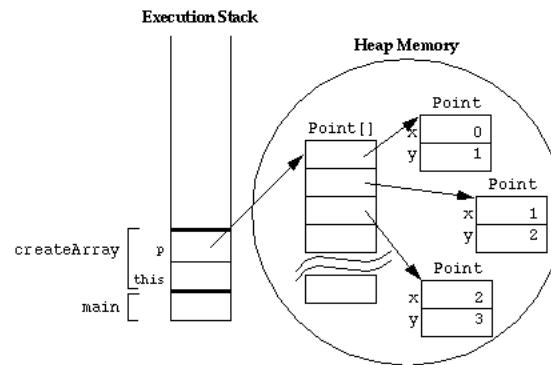
Creating Arrays

Another example, an object array:

```
public Point[] createArray() {
    Point[] p;

    p = new char[10];
    for ( int i=0; i<10; i++ ) {
        p[i] = new Point(i, i+1);
    }

    return p;
}
```



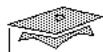
Creating Arrays

You can create arrays of objects as well. You would 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. These must be created separately as follows:

```
p[0] = new Point(0, 1);
p[1] = new Point(1, 2);
...
```

Initializing Arrays

- Initialize an array element
- Create an array with initial values:

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

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

MyDate dates[];
dates = new MyDate[3];
dates[0] = new MyDate(22, 7, 1964);
dates[1] = new MyDate(1, 1, 2000);
dates[2] = new MyDate(22, 12, 1964);

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

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 zero (`'\u0000'` - null) 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"
};
```

Initializing Arrays

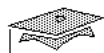
- ✓ **Make sure students understand that in this example the `names` array has exactly three elements.**

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)
};
```



Multi-Dimensional Arrays

- Arrays of arrays:

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

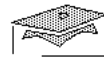
Multi-Dimensional Arrays

The Java programming language does not provide multi-dimensional 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 each element must be initialized 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.



Multi-Dimensional Arrays

- Non-rectangular arrays of arrays:

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

- Array of four arrays of five integers each:

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

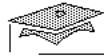
Multi-Dimensional Arrays

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

All array subscripts begin at 0:

```
int list[] = new int [10];
for (int i = 0; i < list.length; i++) {
    System.out.println(list[i]);
}
```

Array Bounds

In the Java programming language, all array indices begin at zero. The number of elements in an array is stored as part of the array object, as the `length` attribute. This value is used to perform bounds checking of all runtime accesses. If an out-of-bounds access occurs, then a runtime exception occurs.

Use the `length` attribute to iterate on an array as follows:

```
int list[] = new int [10];
for (int i = 0; i < list.length; i++) {
    System.out.println(list[i]);
}
```

Using the `length` attribute makes program maintenance easier.

✓ **Ask the students about the multi-dimensional example:**

```
int multiDim[][] = new int [10][5];

System.out.println("multiDim.length is " + multiDim.length);

System.out.println("multiDim[0].length is " + multiDim[0].length);
```



Array Resizing

- Cannot resize an array
- Can use the same reference variable to refer to an entirely new array:

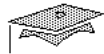
```
int elements[] = new int[6];  
elements = new int[10];
```

Array Resizing

Once created, an array cannot be resized. 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 is effectively lost unless another reference to it is retained elsewhere.



Copying Arrays

The `System.arraycopy()` method:

```
1 //original array
2 int elements[] = { 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 elements array to the hold
8 // array, starting with the 0th index
9 System.arraycopy(elements, 0, hold, 0, elements.length);
```

Copying Arrays

The Java programming language provides a special method in the `System` class, `arraycopy()`, to copy arrays. For example, you can use `arraycopy()` 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 – `System.arraycopy()` copies references, not objects, when dealing with arrays of objects. The objects themselves do not change.

Exercise: Using Arrays



Exercise objective – After defining and initializing arrays, you will use them in a program. You will also use them to implement multiplicity in an object association.

Preparation

To successfully complete this lab, you must understand basic matrix concepts and know how to index an array to obtain its value.

Tasks

In a Web browser view the `lab_files.html` page that is at the top-level of the SL275 directory on your computer. There will be a summary of each exercise and a link to a page that gives a detailed explanation of the exercise.

Go to the SL275 directory on your computer and change to the directory for this module (`mod05`). A listing of this directory will show two subdirectories: one for each of the exercises below.

Exercise 1: Manipulate Arrays (Level 1)

In this exercise you will have hands-on experience in declaring, creating, and manipulating one- and two-dimensional arrays of primitive types.

Exercise 2: Use Arrays to Represent Multiplicity (Level 2)

In this exercise you will use arrays to implement the multiplicity on the association between a bank and its customers.

Exercise: Using Arrays

Exercise Summary



Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercises.

✓ *If you do not have time to spend on discussion, just highlight the key concepts students should have learned from the lab exercise.*

- Experiences

✓ *Ask students what their overall experiences with this exercise have been. You might want to go over any trouble spots or especially confusing areas at this time.*

- Interpretations

✓ *Ask students to interpret what they observed during any aspects of this exercise.*

- Conclusions

✓ *Have students articulate any conclusions they have reached as a result of this exercise experience.*

- Applications

✓ *Explore with the students how they might apply what they learned in this exercise to situations at their workplace.*

Check Your Progress

Before continuing on to the next module, check to be sure that you can:

- 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 any array
- Create a multi-dimensional array
- Write code to copy array values from one array type to another

Think Beyond

How can you create a three-dimensional array?

What is a disadvantage of using arrays?

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 "package-friendly"
- Describe constructor and method overloading
- Describe the complete object construction and initialization operation
- In a Java program, identify the following:
 - ▼ Overloaded methods and constructors
 - ▼ The use of `this` to call overloaded constructors
 - ▼ Overridden methods
 - ▼ Invocation of super class methods
 - ▼ Parent class constructors
 - ▼ Invocation of parent class constructors

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


- ✓ **Present the following question to stimulate the students and get them thinking about the issues and topics presented in this module. They are not expected to know the answers to all of this question. Hold discussions where students have input; otherwise, if no one can propose answers, begin the lecture for this module.**



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

- How does the Java programming language support object inheritance?

- ✓ **The most important aspects of inheritance are code reuse, polymorphism, and virtual method invocation.**



Sun Educational Services

The *is* a Relationship

The Employee class:

Employee
+name : String = ""
+salary : double
+birthDate : Date
+getDetails() : String

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

    public String getDetails() {...}
}
```

Subclassing

The is a Relationship

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. Clearly a manager actually *is an* employee, but an employee with additional features.

Consider the following sample class declarations that demonstrate this:

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

    public String getDetails() {...}
}
```



The is a Relationship

The Manager class:

Manager
+name : String = ""
+salary : double
+birthDate : Date
+department : String
+getDetails() : String

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

    public String getDetails() {...}
}
```

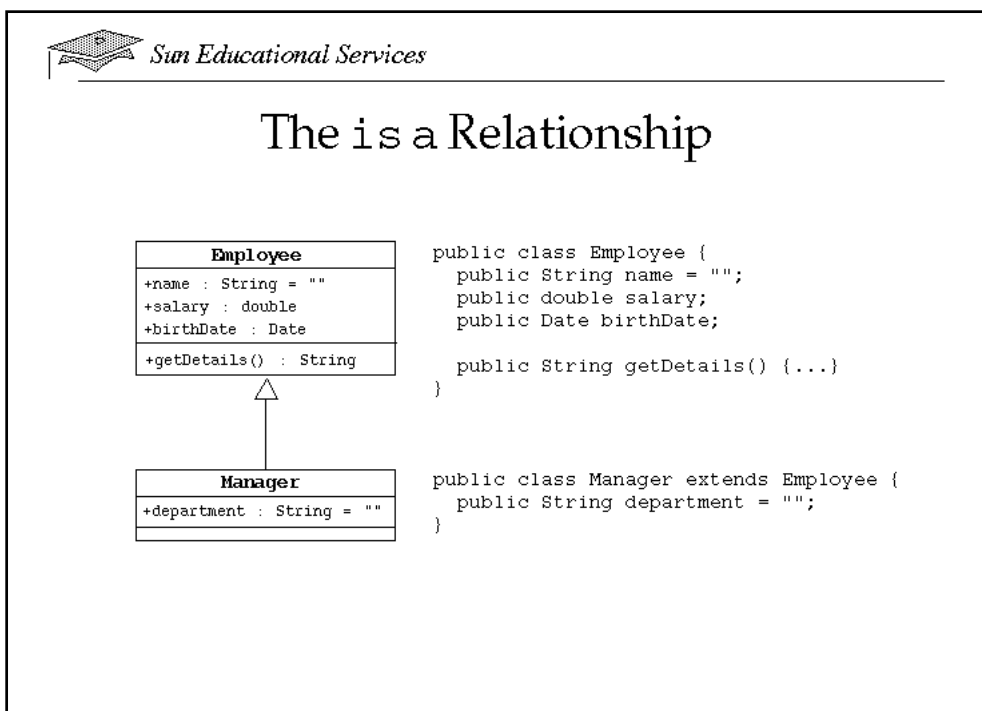
Subclassing

The is a Relationship (Continued)

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

    public String getDetails() {...}
}
```

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*.



Subclassing

The is a Relationship (Continued)

In object-oriented languages, special mechanisms are provided that allow the programmer to define a class in terms of a previously defined class. In the Java technology programming language, this is achieved by the keyword `extends` as follows:

```
public class Manager extends Employee {
    private String department = "";
}
```



Single Inheritance

- When a class inherits from only one class, it is called *single inheritance*.
- Single inheritance makes code more reliable.
- *Interfaces* provide the benefits of multiple inheritance without drawbacks.
- Syntax of a Java class:

```
<class_declaration> ::=  
    <modifier> class <name> [extends <superclass>] {  
        <declarations>*  
    }
```

Subclassing

Single Inheritance

The Java programming language allows a class to extend only one other class. This restriction is called *single inheritance*. The relative merits of single and multiple inheritance are the subject of extensive discussions among object-oriented programmers. The Java programming language imposes the single inheritance restriction to make the resulting code more reliable, although this sometimes makes more work for the programmer. Module 7, "Advanced Class Features," examines a language feature called *interfaces* that allows most of the benefits of multiple inheritance without suffering from any of its drawbacks.

Subclassing

Single Inheritance (Continued)

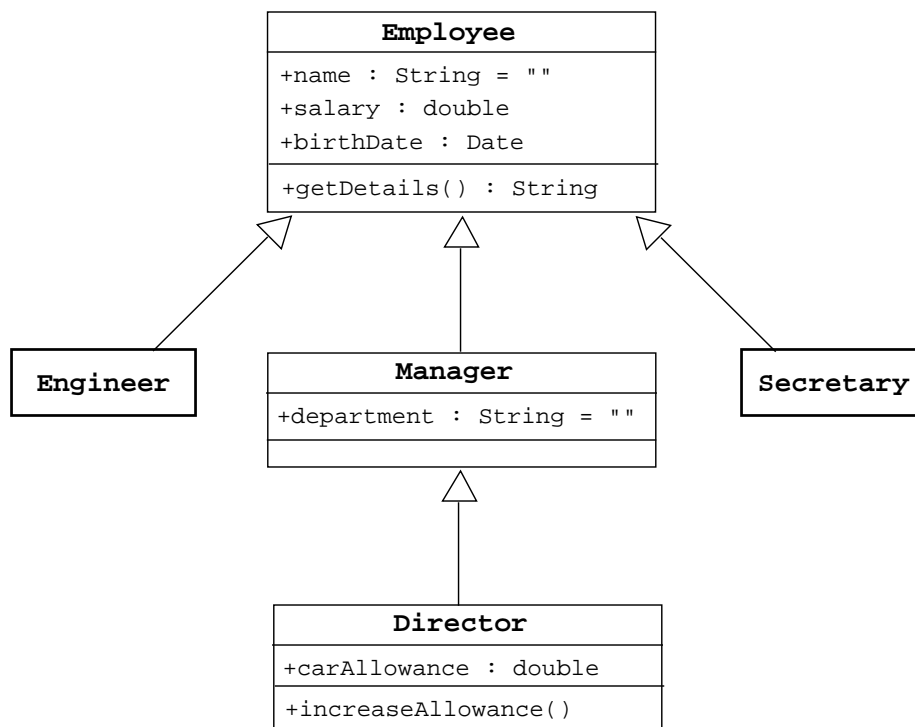


Figure 6-1 An Example Inheritance Tree

This example shows the base class `Employee` and three subclasses: `Engineer`, `Manager`, and `Secretary`. The `Manager` is also subclassed by `Director`.

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`, and overrides 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).



Constructors Are Not Inherited

- A subclass inherits all methods and variables from the superclass (parent class)
- A subclass does not inherit the constructor from the superclass
- Two ways to include a constructor are:
 - ▼ Use the default constructor
 - ▼ Write one or more explicit constructors

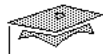
Subclassing

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 that 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.

Note – A parent constructor is always called in addition to a child constructor. This is discussed in detail later in this module.



Polymorphism

- *Polymorphism* is the ability to have many different forms; for example, the `Manager` class has access to methods from `Employee` class
- An object has only one form
- A reference variable can refer to objects of different forms

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.

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.

Polymorphism

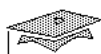
The Java programming language, like most object-oriented languages, actually allows you to refer to an object with a variable that is one of the parent class types. So you can say:

```
Employee employee = new Manager();
```

Using the variable `employee` 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, `employee` is an `Employee`, not a `Manager`. Therefore, the following is not allowed:

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



Heterogeneous Collections

- Collections of objects with the same class type are called *homogenous* collections.

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

- Collections of objects with different class types are called *heterogeneous* collections.

```
Employee [] staff = new Employee[1024];
staff[0] = new Manager();
staff[1] = new Employee();
staff[2] = new Engineer();
```

Polymorphism

Heterogeneous Collections

You can create collections of objects that have a common class. Such collections are called *homogenous* collections.

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

It might seem unrealistic to create a `Manager` and deliberately assign 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.

Polymorphism

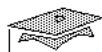
Heterogeneous Collections (Continued)

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

```
Employee [] staff = new Employee[1024];
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 are managers.

Note – Every class is a subclass of `Object`, so you can use an array of `Object` as a container for any 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, discussed on page 6-44.



Polymorphic Arguments

- Because a Manager is an Employee:

```
// In the Employee class
public TaxRate findTaxRate(Employee e) {
}
// Meanwhile, elsewhere in the application class
Manager m = new Manager();
:
TaxRate t = findTaxRate(m);
```

Polymorphism

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:

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

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

This is legal, because a `Manager` is an `Employee`.



The instanceof Operator

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

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

Polymorphism

The instanceof Operator

Given that you can pass objects around using references to their parent classes, sometimes you might want to know what you actually 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 actually redundant. It is shown here only as a reminder.

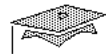
Polymorphism

The instanceof Operator (Continued)

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), but `instanceof` in the Java programming language is more powerful.



Casting Objects

- Use `instanceof` to test the type of an object
- Restore full functionality of an object by casting
- Check for proper casting using the following guidelines:
 - ▼ Casts up hierarchy are done implicitly
 - ▼ Downward casts must be to a subclass and checked by the compiler
 - ▼ The object type is checked at runtime when runtime errors can occur

Polymorphism

Casting Objects

In circumstances where you have received a reference to a parent class, and you have determined that the object is actually a particular subclass by using the `instanceof` operator, you can restore 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.

Polymorphism

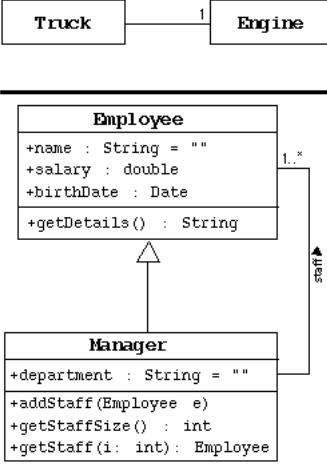
Casting Objects (Continued)

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 “up” 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 `Contractor` reference is definitely not permitted, because the `Contractor` 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 allows 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.


Sun Educational Services

The has a Relationship



```

classDiagram
    class Truck
    class Engine
    class Employee {
        +name : String = ""
        +salary : double
        +birthDate : Date
        +getDetails() : String
    }
    class Manager {
        +department : String = ""
        +addStaff(Employee e)
        +getStaffSize() : int
        +getStaff(i: int) : Employee
    }
    Truck "1" -- "1" Engine
    Employee <|-- Manager
    Employee "1..*" -- "1" Manager : staff
        
```

```

public class Vehicle {
    private Engine theEngine;
    public Engine getEngine() {
        return theEngine;
    }
}

public class Manager extends Employee {
    public String department = "";
    public Employee[] staff = new Employee[20];
    public int staffSize = 0;

    public void addStaff(Employee e) {
        staff[staffSize++] = e;
    }

    public int getStaffSize() {
        return staffSize;
    }

    public Employee getStaff(int i) {
        return staff[i];
    }

    ...
}
        
```

The has a Relationship

Objects can contain other objects. This is often called *association* or the “has a” relationship. For example, a manager *has a* staff of employees. An association is implemented in Java using a data attribute.

If multiplicity of an association is 1 (a “one-to-one” relationship), then data attribute is a single reference to the object. For example, a vehicle may only have one engine. Therefore, there is a one-to-one relationship between the Vehicle and Engine classes.

However, if the multiplicity might be greater than one, then a collection (possibly implemented as an array) should be used. The staff association is a one-to-many relationship between the manager and his set of employees.

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.

A variable, method, or class has default accessibility if it does not have an explicit protection 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."

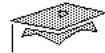
A variable or method marked with the modifier `protected` is actually more accessible than one with default access control. A `protected` method or variable is accessible from methods in classes that are members of the same package and from any method in any *subclass*. You should use the `protected`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

Table 6-1 Accessibility Criteria

Modifier	Same Class	Same Package	Subclass	Universe
public	Yes	Yes	Yes	Yes
protected	Yes	Yes	Yes	
<i>default</i>	Yes	Yes		
private	Yes			

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

- ✓ **A design that makes extensive use of protected or default access elements is probably either very well designed or very poorly designed.**
- ✓ **`protected` is particularly useful in JavaBeans. Introspection works only on public methods; `protected` allows member variables to be accessed by subclasses and is invisible to the introspector.**



Overloading Method Names

- It can be used as follows:

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

- Argument lists *must* differ
- Return types *can* be different

Overloading Method Names

In some circumstances, you might want to write several methods in the same class that do basically the same 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, allows 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 actually needed. In the case of the three print methods, this distinction is based on the number and type of the arguments.

Overloading Method Names

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()
```

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:

- 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.
- 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.



Overloading Constructors

- As with methods, constructors can be overloaded

- Example:

```
public Employee(String name, double salary, Date DoB)
public Employee(String name, double salary)
public Employee(String name, Date DoB)
```

- Argument lists *must* differ
- The `this` reference can be used at the first line of a constructor to call another constructor

Overloading Constructors

When an object is instantiated, the program might want to supply multiple constructors based on what data is known about the object being created. For example, a payroll system might want 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 may not know the starting salary or the date of birth.

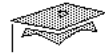
```
1 public class Employee {
2     private static final double BASE_SALARY = 15000.00;
3     private String name;
4     private double salary;
5     private Date    birthDate;
6
7     public Employee(String name, double salary, Date DoB) {
8         this.name = name;
9         this.salary = salary;
10        this.birthDate = DoB;
11    }
```

Overloading Constructors

```
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 }
```

In the code fragment above, we have coded four, overloaded constructors. The first one (lines 7-11) initializes all instance variables. In the second one (lines 12-14), the date of birth is not provided. Notice the use of the `this` reference: It 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.

Note – The `this` command in a constructor must be the first line of code in the constructor. There may be more initialization code after the `this` call, but not before.



Overriding Methods

- A subclass can modify behavior inherited from a parent class
- A subclass can create a method with different functionality than the parent's method but with the same:
 - ▼ Name
 - ▼ Return type
 - ▼ Argument list

Overriding Methods

In addition to being able to produce 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 such that the name, return type, and argument list exactly match those of a method in the parent class, then the new method is said to *override* the old one.

Note – Remember that methods with the same name, but with different argument lists in the same class, are simply overloaded. This causes the compiler to use the supplied arguments to determine which method to call.

Overriding Methods

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

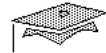
```
public class Employee {
    protected String name;
    protected double salary;
    protected Date    birthDate;

    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

- Virtual method invocation:

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

- Compile-time type and runtime type

Overriding Methods

Assume that the example on the previous page and the following scenario are 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();
```

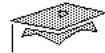
or some similar effect, such as a general method argument or an item from a heterogeneous collection.

Overriding Methods

In fact, you get the behavior associated with the runtime type of the variable (that is, the type of the object referred to by the variable), not the behavior associated with the compile time type of the variable. This is an important feature of object-oriented languages. It is also another important feature of polymorphism and 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`.

Note – 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.



Rules About Overridden Methods

- Must have a return type that is identical to the method it overrides
- Cannot be less accessible than the method it overrides

Invoking Overridden Methods

Rules About Overridden Methods

Remember that the method name, and 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. The following rules apply to overridden methods:

- The return type of the overriding method must be identical to the method it overrides.
- An overriding method cannot be less accessible than the method it overrides.

Note – An overriding method cannot throw different types of exceptions than the method it overrides. This will be discussed in more detail in Module 8, "Exceptions."

Invoking Overridden Methods

Rules About Overridden Methods (Continued)

These rules result from the nature of polymorphism combined with the need for the Java programming language to be “typesafe.” Consider this invalid scenario:

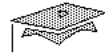
```
public class Parent {
    public void doSomething() {
    }
}

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

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.

✓ ***This code fails at compile time, in spite of the fact that true type resolution takes effect at runtime.***



The super Keyword

- `super` is used in a class to refer to its superclass
- `super` is used to refer to the members of superclass, both data attributes and methods
- Behavior invoked does not have to be in the superclass; it can be further up in the hierarchy

Invoking Overridden Methods

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.

Quite often when you override a method, your real goal is not to replace the existing behavior but to extend that behavior in some way.

Invoking Overridden Methods

The `super` Keyword (Continued)

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 generally good programming practice.



Invoking Parent Class Constructors

- To invoke a parent constructor you must place a call to `super` in the first line of the constructor
- You can call a specific parent constructor by the arguments that you use in the call to `super`
- If no `this` or `super` call is used in a constructor, then the compiler adds an implicit call to `super()` which calls the parent "default" constructor
 - ▼ If the parent class does not supply a non-private "default" constructor, then a compiler warning will be issued

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 "calling" 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 default parent constructor (that is, the constructor with zero arguments) is called implicitly. In this case, if there is no default parent constructor, a compiler error results.

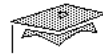
Note – The call to `super()` can take any number of arguments appropriate to the various constructors available in the parent class, but it must be the first statement in the constructor.

Invoking Parent Class Constructors

Assuming that the `Employee` class has the set of constructors that were defined in the “Overloading Constructors” section on page 6-22, then the following constructors in `Manager` might be defined. Note that 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 with no 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 n, String dept) {
9         super(name);
10        department = dept;
11    }
12    public Manager(String dept) { // This code fails: no super()
13        department = d;
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 automatically inserts a call to the parent class constructor with no arguments. Other constructors can also call `super(...)` or `this(...)`, invoking a chain of constructors. What ultimately happens is the parent class constructor (or possibly several) executes before any child class constructor in the chain.



Constructing and Initializing Objects: A Slight Reprise

- Memory is allocated and default initialization occurs
- Instance variable initialization uses these steps recursively:
 - 1 Bind constructor parameters.
 - 2 If explicit `this()`, call recursively and then skip to step 5.
 - 3 Call recursively the implicit or explicit `super` call, except for `Object`.
 - 4 Execute explicit instance variable initializers.
 - 5 Execute body of current constructor.

Constructing and Initializing Objects: A Slight Reprise

Object initialization is a rather complex process. In "Constructing and Initializing Objects" on page 3-21 in Module 3, "Identifiers, Keywords, and Types," you were exposed to a very rudimentary explanation. In this section you will 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 explicit instance variable initializers.
5. Execute body of current constructor.

Constructing and Initializing Objects: A Slight Reprise

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

```
public class Object {
    ...
    public Object() {}
    ...
}

public class Employee extends Object {
    private String name;
    private double salary = 15000.00;
    private Date    birthDate;

    public Employee(String n, Date DoB) {
        // implicit super();
        name = n;
        birthDate = DoB;
    }
    public Employee(String n) {
        this(n, null);
    }
}

public class Manager extends Employee {
    private String department;

    public Manager(String n, String d) {
        super(n);
        department = d;
    }
}
```

Constructing and Initializing Objects: A Slight Reprise

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 (0 or null)
- 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"`

Constructing and Initializing Objects: A Slight Reprise

Implications of the Initialization Process

Suppose the `Employee` class was defined as follows:

```
1 public class Employee extends Object {
2     private String name;
3     private double salary = 15000.00;
4     private Date    birthDate;
5     private String summary;
6
7     public Employee(String n, Date DoB) {
8         name = n;
9         birthDate = DoB;
10        summary = getDetails();
11    }
12    public Employee(String n) {
13        this(n, null);
14    }
15
16    public String getDetails() {
17        return "Name: " + name + "\nSalary: " + salary
18            + "\nBirth Date: " + birthDate;
19    }
20 }
```

Notice that the `summary` data attribute is being initialized to the details of the employee (line 10).

Constructing and Initializing Objects: A Slight Reprise

Implications of the Initialization Process (Continued)

And the Manager class as:

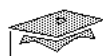
```
1 public class Manager extends Employee {
2     private String department;
3
4     public Manager(String n, String d) {
5         super(n);
6         department = d;
7     }
8
9     public String getDetails() {
10        return super.getDetails() + "\nDept: " + department;
11    }
12 }
```

Notice that the `getDetails` method is being overridden (line 9-11) and that the parent constructor is being called (line 5).

The problem arises from the virtual method invocation in the `Employee` constructor on line 10. If a `Manager` object is being constructed, then the `getDetails` method on line 8 is called which uses the `department` attribute before it has been initialized (line 5).

This example is not too awful because the `department` attribute will be added to the details string as “Dept: null”. However, if the `getDetails` method had called a method on an object that had not been initialized, then an exception would have been thrown which might cause the program to halt.

As a rule of thumb: If you must call a method in a constructor make that method private.



The Object Class

- The `Object` class is the root of all classes in Java
- A class declaration with no `extends` clause, implicitly uses "`extends Object`"

```
public class Employee {  
    ...  
}
```

is equivalent to:

```
public class Employee extends Object {  
    ...  
}
```

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 implicitly adds "`extends Object`" to the declaration. For example:

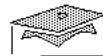
```
public class Employee {  
    ...  
}
```

is equivalent to:

```
public class Employee extends Object {  
    ...  
}
```

This allows you to override several methods inherited from the `Object` class. The following sections discuss two important `Object` methods.

✓ **Encourage students to review the `Object` class in the API documentation.**



The == Operator Compared With the equals Method

- The == operator determines if two references are identical to each other (that is, refer to the same object)
- The equals method determines if objects are "equal" but not necessarily identical
- The Object implementation of the equals method uses the == operator
- User classes can override the equals method to implement a domain-specific test for equality
- Note: You should override the hashCode method, if you override the equals method

The == Operator Compared With 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 – It is recommended that you override the hashCode method whenever you override the equals method. A decent, but naive, implementation could use a bitwise XOR on the hash codes of the elements tested for equality.

The == Operator Compared With the equals Method

Example

In this example, the `Employee` class has been modified to include an `equals` method that tests against the employee name and date of birth.

```
1 class Employee {
2
3     private String name;
4     private MyDate birthDate;
5     private float salary;
6
7     // Constructor
8     public Employee(String name, MyDate DoB, float salary) {
9         this.name = name;
10        this.birthDate = DoB;
11        this.salary = salary;
12    }
13
14    public boolean equals(Object o) {
15        boolean result = false;
16        if ( (o != null) && (o instanceof Employee) ) {
17            Employee e = (Employee) o;
18            if ( name.equals(e.name) && birthDate.equals(e.birthDate) ) {
19                result = true;
20            }
21        }
22        return result;
23    }
24
25    public int hashCode() {
26        return ( name.hashCode() ^ birthDate.hashCode() );
27    }
28 }
```

We overrode the `hashCode` method that implements a bitwise XOR of the hash codes for the name string and the `birthDate` date. This will guarantee that hash code for equal `Employee` objects will have the same value.

✓ **Implementing an efficient hash code algorithm is non-trivial, but a decent implementation must guarantee that two "equal" objects have the same hash value.**

The == Operator Compared With the equals Method

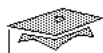
Example (Continued)

The following program tests two Employee objects that are not identical, but are equal relative to the name/birthdate test.

```
1 class TestEquals {
2     public static void main(String[] args) {
3         Employee emp1 = new Employee("Fred Smith",
4                                     new MyDate(14, 3, 1976),
5                                     25000.0F);
6         Employee emp2 = new Employee("Fred Smith",
7                                     new MyDate(14, 3, 1976),
8                                     25000.0F);
9
10        if ( emp1 == emp2 ) {
11            System.out.println("emp1 is identical to emp2");
12        } else {
13            System.out.println("emp1 is not identical to emp2");
14        }
15
16        if ( emp1.equals(emp2) ) {
17            System.out.println("emp1 is equal to emp2");
18        } else {
19            System.out.println("emp1 is not equal to emp2");
20        }
21
22        emp2 = emp1;
23        System.out.println("set emp2 = emp1;");
24        if ( emp1 == emp2 ) {
25            System.out.println("emp1 is identical to emp2");
26        } else {
27            System.out.println("emp1 is not identical to emp2");
28        }
29    }
30 }
31
```

The execution of this test program generates the following output:

```
emp1 is not identical to emp2
emp1 is equal to emp2
set emp2 = emp1;
emp1 is identical to emp2
```



toString Method

- Converts an object to a `String`
- Used during string concatenation
- Override this method to provide information about a user-defined object in readable format
- Primitive types are converted to a `String` using the wrapper class's `toString` static method

The toString Method

The `toString` method is used to convert 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 roughly 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

- Look at primitive data elements as objects

Primitive Data Type	Wrapper Class
boolean	Boolean
byte	Byte
char	Character
short	Short
int	Integer
long	Long
float	Float
double	Double

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 once the primitive value is initialized in the wrapper object, then there is no means to change that value.

Wrapper Classes

Table 6-2 Wrapper Classes

Primitive Data Type	Wrapper Class
boolean	Boolean
byte	Byte
char	Character
short	Short
int	Integer
long	Long
float	Float
double	Double

You can construct a wrapper class object by passing the value to be wrapped into the appropriate constructor. For example:

```
int pInt = 500;
Integer wInt = new Integer(pInt);
int p2 = wInt.intValue();
```

- ✓ ***It can also be constructed by passing a string that represents a value to be wrapped. If the string does not represent a valid value, a `NumberFormatException` is thrown, except in the case of a `boolean`. Wrapped values can be extracted as a numeric type using the appropriate call. For example, a `long` value can be extracted using `public long longValue()`.***

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);
```

Exercise: Using Objects and Classes



Exercise objective – You will write, compile, and run a program that uses inheritance to implement two types of bank accounts. You will write, compile, and run a program that uses a heterogeneous collection of objects.

Preparation

To successfully complete this lab, you must understand the concepts of inheritance, polymorphism, method overriding, and heterogeneous collections.

Tasks

In a Web browser view the `lab_files.html` page that is at the top-level of the SL275 directory on your computer. There will be a summary of each exercise and a link to a page that gives a detailed explanation of the exercise.

Go to the SL275 directory on your computer and change to the directory for this module (`mod06`). A listing of this directory will show two subdirectories: one for each of the exercises below.

Exercise 1: Create Subclasses of Bank Accounts (Level 1)

In this exercise you will create two subclasses of the `Account` class in the Banking project: `SavingsAccount` and `CheckingAccount`.

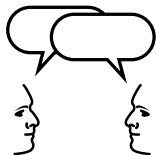
✓ **For more advanced students use the alternate exercise #1. This lab can be found in the `alternatel` directory.**

Exercise 2: Use a Heterogeneous Collection (Level 2)

In this exercise you will create a heterogeneous array to represent the aggregation of customers to accounts. That is, a given customer may have several accounts of different types.

Exercise: Using Objects and Classes

Exercise Summary



Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercises.

- ✓ *If you do not have time to spend on discussion, just highlight the key concepts students should have learned from the lab exercise.*

- Experiences

- ✓ *Ask students what their overall experiences with this exercise have been. You might want to go over any trouble spots or especially confusing areas at this time.*

- Interpretations

- ✓ *Ask students to interpret what they observed during any aspects of this exercise.*

- Conclusions

- ✓ *Have students articulate any conclusions they have reached as a result of this exercise experience.*

- Applications

- ✓ *Explore with the students how they might apply what they learned in this exercise to situations at their workplace.*

Check Your Progress

Before continuing on to the next module, check to be sure that you can:

- Define *inheritance, polymorphism, overloading, overriding, and virtual method invocation*
- Use the access modifiers `protected` and "package-friendly"
- Describe constructor and method overloading
- Describe the complete object construction and initialization operation
- In a Java program, identify the following:
 - ▼ Overloaded methods and constructors
 - ▼ The use of `this` to call overloaded constructors
 - ▼ Overridden methods
 - ▼ Invocation of super class methods
 - ▼ Parent class constructors
 - ▼ Invocation of parent class constructors

Think Beyond

Now that you understand inheritance and polymorphism, how can you use this information on a current or future project?

Objectives

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

- Describe `static` variables, methods, and initializers
- Describe `final` classes, methods, and variables
- Explain how and when to use abstract classes and methods
- Explain how and when to use inner classes
- Distinguish between static and non-static inner classes
- Explain how and when to use an interface
- In a Java software program, identify:
 - ▼ `static` methods and attributes
 - ▼ `final` methods and attributes
 - ▼ inner classes
 - ▼ interface and abstract classes
 - ▼ abstract methods

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

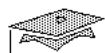
Relevance

- ✓ **Present the following questions to stimulate the students and get them thinking about the issues and topics presented in this module. They are not expected to know the answers to all of these questions. Hold discussions where students have input; otherwise, if no one can propose answers, begin the lecture for this module.**



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

- How can you create a constant?
 - How can you create an instance variable that is set once and can not be reset, even internally?
 - 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?
 - How can you create several classes that implement a common interface yet not be part of a common inheritance tree?
- ✓ **A programmer needs to know all of the object-oriented features of the Java programming language in order to fully use the language's object-oriented power. This module discusses additional features of the language, picking up where the last module left off.**
 - ✓ **Topics in this module are more advanced. It includes a discussion of inner classes; interfaces; abstract class; and the keywords `static` and `final`. Knowledge of these language features greatly helps in the implementation of well-written Java software programs.**



The static Keyword

- The `static` keyword is used as a modifier on variables, methods, and inner classes
- The `static` keyword declares the attribute or method is associated with the class as a whole rather than any particular instance of that class
- Thus static members are often called "class members," such as "class attributes" or "class methods"

The static Keyword

The static keyword is used to declare members (attributes, methods, and inner 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. The discussion of static inner classes comes later in this module.

Sun Educational Services

Class Attributes

- Are shared among all instances of a class

```

classDiagram
    class Count {
        -counter : int = 0
        -serialNumber : int
    }
    class c1 {
        serialNumber=1
    }
    class c2 {
        serialNumber=2
    }
    Count <|-- c1
    Count <|-- c2
    
```

```

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

Class Attributes

Sometimes it is desirable to have a variable that is shared among all instances of a class. For example, this can be used as the basis for communication between instances or to keep track of the number of instances that have been created.

You achieve this 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

Class Attributes (Continued)

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 globals as such, but a `static` variable is a single variable 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         Count.counter++;
4     }
5 }
```

The static Keyword

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 Count {
2     private int serialNumber;
3     private static int counter = 0;
4
5     public static int getTotalCount() {
6         return counter;
7     }
8
9     public Count() {
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                             + Count.getTotalCount());
5         Count count1 = new Count();
6         System.out.println("Number of counter is "
7                             + Count.getTotalCount());
8     }
9 }
```

The output of the `TestCounter` program is:

```

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

The static Keyword

Class Methods (Continued)

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 apart from the local variables, static attributes, and its parameters. Attempting to access non-static attributes causes a compiler error.

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

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

Important points to remember about static methods are:

- A static method cannot be overridden.
- `main()` is static 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 a *static block* that does not exist within a method body
- Static block code executes only once, when the class is loaded
- A static block is usually used to initialize static (class) attributes

The static Keyword

Static Initializers

It is perfectly valid for a class to contain code in a “static block” that does not exist within a method body. The static block code executes only once, when the class is loaded. Different static blocks within a class are executed in the order of their appearance in the class.

```
1 public class Count {
2     public static int counter;
3     static {
4         counter = Integer.getInteger("myApp.Count.counter").intValue();
5     }
6 }
7
8 public class TestStaticInit {
9     public static void main(String[] args) {
10        System.out.println("counter = "+ Count.counter);
11    }
12 }
```

The static Keyword

Static Initializers (Continued)

Note – The code on line 4 of the `StaticInit` class uses a static method on the `Integer` class `getInteger(String)`, which returns an `Integer` object that represents the value of a system property. This property, named `myApp.Count.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.Count.counter=47 TestStaticInit
counter = 47
```

The static Keyword

Implementing the Singleton Design Pattern

Recall the shipping application from "Analysis and Design" on page 2-5. There was a requirement that the system deal with *only one* "company." If the software were written carelessly, it would be possible for the system to create two company objects each with a different fleet of vehicles. This is a problem that is addressed by the Singleton design pattern. The goal of the Singleton is to ensure that—throughout the software system—only one instance of a given class exists and that there is a single point of access to that object.

Figure 7-1 shows the generic model of the Singleton design pattern.

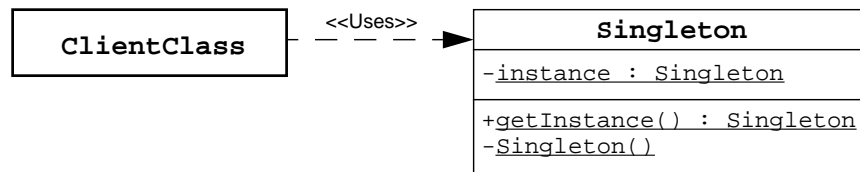


Figure 7-1 The Generic Singleton Design Pattern Model

✓ **Design patterns are solutions to common problems in OO design and they are implementation-independent. Visit <http://hillside.net/patterns/> for more information.**

Figure 7-2 demonstrates how the Company class can implement the Singleton design pattern. In particular, we have used a private class attribute instance that is initialized to an instance of Company. We have used public class method getCompany to access that single instance. The constructor is private to ensure that no other client code can construct a new company object.

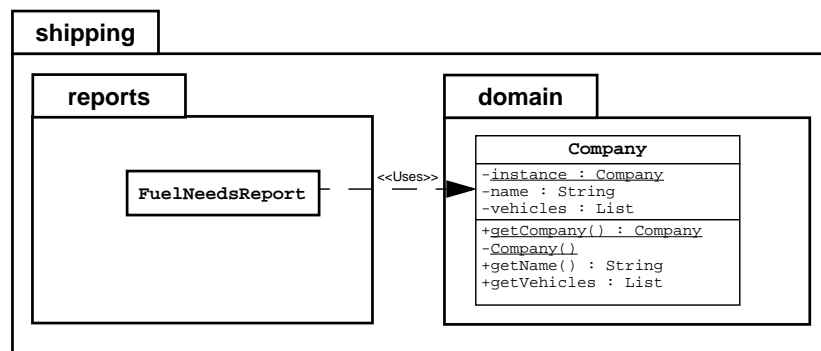


Figure 7-2 The Company Class Implements the Singleton Pattern

The static Keyword

Implementing the Singleton Design Pattern (Continued)

The following code fragment implements the Singleton pattern as modeled in the diagram above:

```
1 package shipping.domain;
2
3 public class Company {
4     private static Company instance = new Company();
5     private String name;
6     private Vehicle[] fleet;
7
8     public static Company getCompany() {
9         return instance;
10    }
11
12    private Company() {...}
13
14    // more Company code ...
15 }
```

The following code fragment demonstrate how to use the Singleton object:

```
1 package shipping.reports;
2
3 import shipping.domain.*;
4
5 public class FuelNeedsReport {
6     public void generateText(PrintStream output) {
7         Company c = Company.getCompany();
8         // use Company object to retrieve the fleet vehicles
9     }
10 }
```



The `final` Keyword

- You cannot subclass a `final` class
- You cannot override a `final` method
- A `final` variable is a constant
- A `final` variable can only be set once, but that assignment can occur independently of the declaration; this is called "blank final variable"
 - ▼ A blank final instance attribute must be set in every constructor
 - ▼ A blank final method variable must be set in the method body before being used

The `final` Keyword

Final Classes

The Java programming language allows you to apply the keyword `final` to classes. If this is done, 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 has a reference to a string, it is definitely a string of class `String` and not a string of a class that is a modified subclass of `String` that might have been maliciously changed.

The final Keyword

Final Methods

You can also mark individual methods as `final`. Methods marked `final` cannot be overridden. This is done 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` are sometimes used for optimization. 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` are `final` automatically because dynamic binding cannot be applied in either case.

The final Keyword

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 properly defined `final` variable:

```
public class Bank {
    private static final double  DEFAULT_INTEREST_RATE=3.2;
    ... // more declarations
}
```

✓ **Final variables are roughly equivalent to `const` in C and C++.**

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`.

A "blank final variable" is a `final` variable that is not initialized in its declaration. The initialization is delayed. Typically, a blank final instance variable should be assigned in a constructor. A blank final local variable can be set at any time in the body of the method, but it can only be set once. The code fragment below 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
}
```

Exercise: Working With the `static` and `final` Keywords



Exercise objective – You will rewrite, compile, and run a program that use the bank account model and employ the `static` and `final` keywords.

Preparation

To successfully complete this lab, you must be familiar with the use of the `static` and `final` keywords and the Singleton design pattern.

Tasks

In a Web browser view the `lab_files.html` page that is at the top-level of the SL275 directory on your computer. There will be a summary of each exercise and a link to a page that gives a detailed explanation of the exercise.

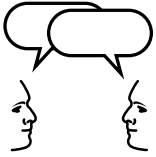
Go to the SL275 directory on your computer and change to the directory for this module (`mod07`). A listing of this directory will show two subdirectories. This exercise is found in the directory called `exercisel`.

Exercise 1: Use the Singleton Design Pattern (Level 2)

In this exercise you will modify the `Bank` class to implement the Singleton design pattern.

Exercise: Working With The static and final Keywords

Exercise Summary



Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercises.

✓ *If you do not have time to spend on discussion, just highlight the key concepts students should have learned from the lab exercise.*

- Experiences

✓ *Ask students what their overall experiences with this exercise have been. You might want to go over any trouble spots or especially confusing areas at this time.*

- Interpretations

✓ *Ask students to interpret what they observed during any aspects of this exercise.*

- Conclusions

✓ *Have students articulate any conclusions they have reached as a result of this exercise experience.*

- Applications

✓ *Explore with the students how they might apply what they learned in this exercise to situations at their workplace.*

Abstract Classes

The Scenario

In our 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 up-coming trips. Let's assume that the Shipping System has a ShippingMain class that populates the company's vehicle fleet list and generates the "Fuel Needs" report.

```

1 public class ShippingMain {
2     public static void main(String[] args) {
3         Company c = Company.getCompany();
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();
13        report.generateText(System.out);
14    }
15 }

```

- ✓ **Point out that the company fleet is a heterogeneous collection of vehicles. Virtual method invocation is the key to this example.**

Figure 7-3 shows the UML model of the Company and its heterogeneous collection of vehicles (the fleet association).

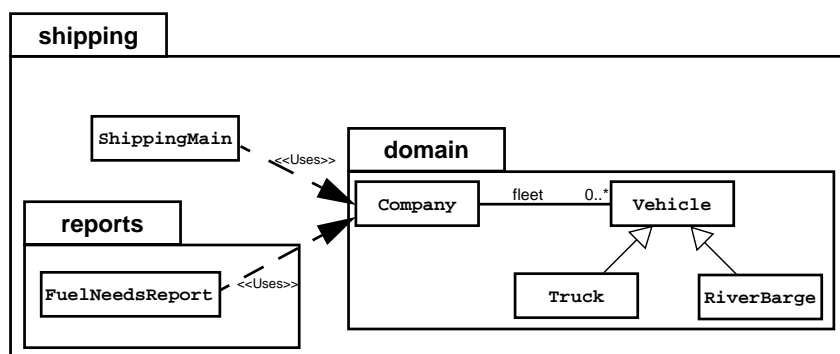


Figure 7-3 UML Model of the Company Fleet

Abstract Classes

The Scenario (Continued)

You would like to be able to write the report code as follows:

```

1 public class FuelNeedsReport {
2     public void generateText(PrintStream output) {
3         Company c = Company.getCompany();
4         Vehicle v;
5         double fuel;
6         double total_fuel = 0.0;
7
8         for ( int i = 0; i < c.getFleetSize(); i++ ) {
9             v = c.getVehicle(i);
10
11             // Calculate the fuel needed for this trip
12             fuel = v.calcTripDistance() / v.calcFuelEfficiency();
13
14             output.println("Vehicle " + v.getName() + " needs "
15                 + fuel + " liters of fuel.");
16             total_fuel += fuel;
17         }
18         output.println("Total fuel needs is " + total_fuel + " liters.");
19     }
20 }

```

The “fuel needed” calculation is the trip distance (in kilometers) divided by the vehicle’s fuel efficiency (in kilometers/liter).

The Problem

The calculations to determine fuel efficiency of a truck as compared with a river barge might be radically different. The `Vehicle` class can not supply these two methods, but its subclasses (`Truck` and `RiverBarge`) can.

✓ *I have not supplied a slide for presenting “The Problem” because it would simply replicate the words used above.*

Abstract Classes

The Solution

The Java language allows a class designer to specify that a superclass declares a method that does not supply an implementation. The implementation of this method is supplied by the subclasses. This is called an *abstract method*. Any class with one or more abstract methods is called an *abstract class*.

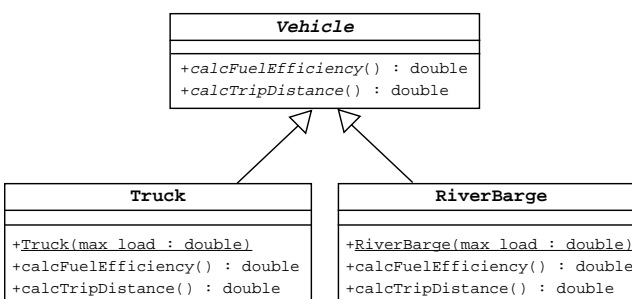


Figure 7-4 UML Model of the Abstract Vehicle Class

Figure 7-4 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.

The Java compiler prevents the programmer from instantiating an abstract class. For example, the statement “`new Vehicle()`” is illegal.

However, abstract classes may 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.

Abstract Classes

The Solution (Continued)

You can declare a class (or method) abstract with the `abstract` keyword:

```
1 public abstract class Vehicle {
2     public abstract double calcFuelEfficiency();
3     public abstract double calcTripDistance();
4 }
```

Therefore, the subclasses of `Vehicle` must supply an implementation of the abstract methods. For example:

```
1 public class Truck extends Vehicle {
2     public Truck(double max_load) {...}
3
4     public double calcFuelEfficiency() {
5         /* calculate the fuel consumption of a truck at a given load */
6     }
7     public double calcTripDistance() {
8         /* calculate the distance of this trip on highway */
9     }
10 }
```

```
1 public class RiverBarge extends Vehicle {
2     public RiverBarge(double max_load) {...}
3
4     public double calcFuelEfficiency() {
5         /* calculate the fuel efficiency of a river barge */
6     }
7     public double calcTripDistance() {
8         /* calculate the distance of this trip along the river-ways */
9     }
10 }
```

Note – If a subclass does not supply an implementation, it must also be declared `abstract` or a compiler warning will occur.

Abstract Classes

Template Method Design Pattern

Recall that on line 12 of the `FuelNeedsReport` class a calculation is performed to determine how much fuel a vehicle will need.

```

1 public class FuelNeedsReport {
2     public void generateText(PrintStream output) {
3         ...
11        // Calculate the fuel needed for this trip
12        fuel = v.calcTripDistance() / v.calcFuelEfficiency();
13        ...
19    }
20 }

```

As simple as it was, this calculation should not be performed in this class. It belongs in the `Vehicle` class:

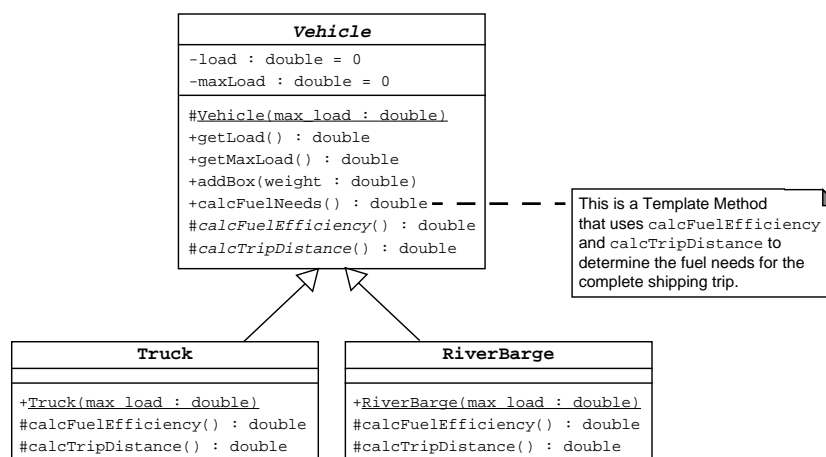


Figure 7-5 The Abstract Vehicle Using a Template Method

```

1 public class FuelNeedsReport {
2     public void generateText(PrintStream output) {
3         ...
11        // Calculate the fuel needed for this trip
12        fuel = v.calcFuelNeeds();
13        ...
19    }
20 }

```

This is called a Template Method because it is a concrete method that is supplied in an abstract class that uses abstract methods, which are implemented by the subclasses.



Interfaces

- A "public interface" is a contract between client code and the class that implements that interface
- A Java *interface* is a formal declaration of such a contract in which all methods contain no implementation
- Many, unrelated classes can implement the same interface
- A class can implement many, unrelated interfaces
- Syntax of a Java class:

```

<class_declaration> ::=
    <modifier> class <name> [extends <superclass>]
        [implements <interface> [,<interface>]* ] {
        <declarations>*
    }
    
```

Interfaces

The “public interface” of a class is a contract between the “client code” and the class that provides the service. Concrete classes provide an implementation for each method, an abstract class can defer the implementation by declaring the method to be abstract, a Java interface declares only the contract and no implementation what so ever.

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. We will see how all of this works in the following pages.

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 – An interface can also declare constants: public static final

✓ **The next few pages/slides present a step-wise example to demonstrate the last two points.**

Interfaces

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`.

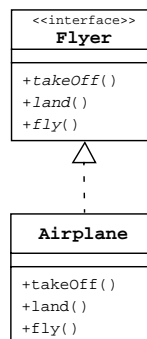


Figure 7-6 The Flyer Interface and Airplane Implementation

```

public interface Flyer {
    public void takeOff();
    public void land();
    public void fly();
}

public class Airplane implements Flyer {
    public void takeOff() {
        // accelerate until lift-off
        // raise landing gear
    }
    public void land() {
        // lower landing gear
        // decelerate and lower flaps until touch-down
        // apply breaks
    }
    public void takeOff() {
        // keep those engines running
    }
}
  
```

- ✓ **At this point ask: “What other objects can fly?” Typical answers might be: a bird, a helicopter, and maybe even Superman. If you get Superman, point out that the Superman class is a Singleton.**

Interfaces

The Flyer Example (Continued)

There can be multiple classes that implement the Flyer interface. We have seen that an airplane can fly, but a bird can also fly, Superman can fly, and so on.

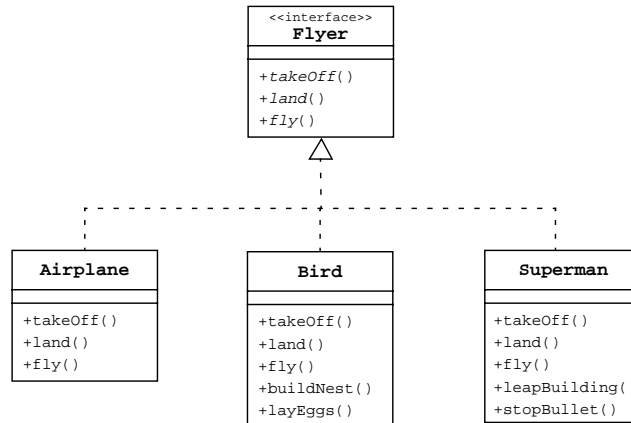


Figure 7-7 Multiple Implementations of The Flyer Interface

✓ **At this point ask: “What is the superclass of Bird?” Try to phrase it such that they say “a bird is an animal.”**

Interfaces

The Flyer Example (Continued)

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.

This sounds like multiple inheritance. Not quite. 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.

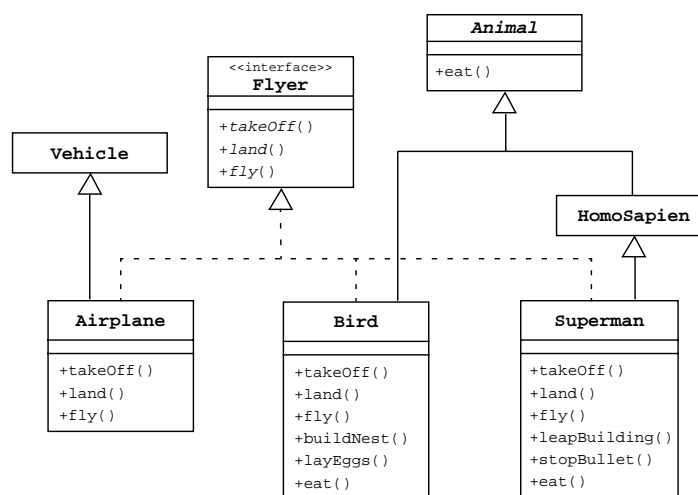


Figure 7-8 A Mixture of Inheritance and Implementation

Let's look at the code outline for 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. Notice that the Bird class can supply its own methods (buildNest and layEggs) as well as override the Animal class methods (eat).

Interfaces

The Flyer Example (Continued)

Let's look at how interfaces can be used. Suppose that you are constructing an aircraft control software system. It needs to grant permission to land and take-off for flying objects of all types.

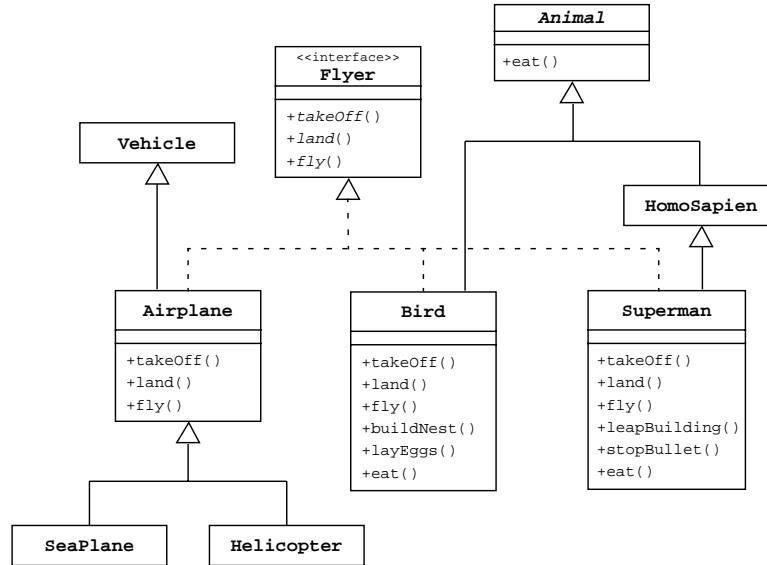


Figure 7-9 Class Hierarchy for the Airport Example

The code for our airport could look like the following:

```

public class Airport {
    public static void main(String[] args) {
        Airport metropolisAirport = new Airport();
        Helicopter copter = new Helicopter();
        SeaPlane sPlane = new SeaPlane();
        Flyer S = Superman.getSuperman(); // Superman is a Singleton

        metropolisAirport.givePermissionToLand(copter);
        metropolisAirport.givePermissionToLand(sPlane);
        metropolisAirport.givePermissionToLand(S);
    }

    private void givePermissionToLand(Flyer f) {
        f.land();
    }
}

```


Interfaces

Multiple Interface Example

A class can implement more than one interface. Not only can our 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.

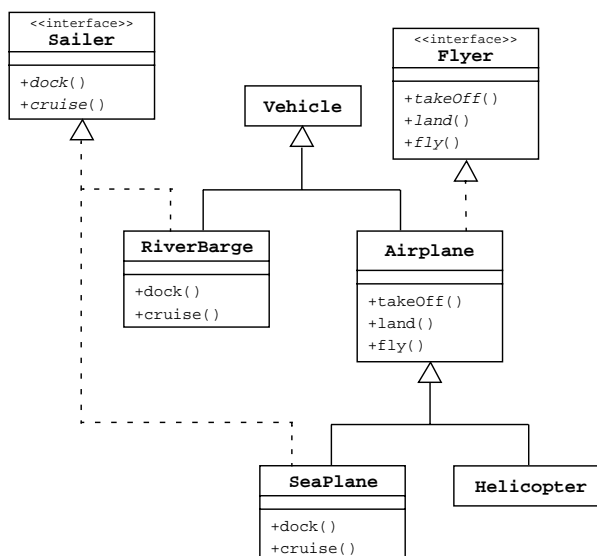


Figure 7-10 An Example of Multiple Implementations

Let's now write the Harbor class that gives docking permission:

```

public class Harbor {
    public static void main(String[] args) {
        Harbor bostonHarbor = new Harbor();
        RiverBarge barge = new RiverBarge();
        SeaPlane sPlane = new SeaPlane();

        bostonHarbor.givePermissionToDock(barge);
        bostonHarbor.givePermissionToDock(sPlane);
    }
    private void givePermissionToDock(Sailer s) {
        s.dock();
    }
}
  
```

Notice that our seaplane can take off from Metropolis airport and dock in Boston harbor.



Uses of Interfaces

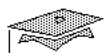
- Declaring methods that one or more classes are expected to implement
- Determining an object's programming interface without revealing the actual body of the class
- Capturing similarities between unrelated classes without forcing a class relationship
- Simulating multiple inheritance by declaring a class that implements several interfaces

Interfaces

Interfaces are useful for:

- Declaring methods that one or more classes are expected to implement.
- Revealing 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.)
- Capturing similarities between unrelated classes without forcing a class relationship.
- Simulating multiple inheritance by declaring a class that implements several interfaces.

✓ **Interfaces are often considered as an alternative to multiple inheritance, even though they provide a different functionality. The concept of interfaces was borrowed from Objective - C where they were called protocols.**



Inner Classes

- Added to JDK 1.1
- Allow a class definition to be placed inside another class definition
- Group classes that logically belong together
- Have access to their enclosing class's scope

Inner Classes

Inner classes, sometimes called nested classes, were added to JDK 1.1 and all subsequent versions. Inner classes allow a class definition to be placed inside another class definition. Inner classes are a useful feature because they allow you to group classes that logically belong together and to control the visibility of one within another. They are also used to implement details of an implementation that should not be shared by any other class.

- ✓ ***The next few pages/slides present a step-wise example to demonstrate the syntactic and semantic elements of inner classes. These examples are stripped down to their most abstract level; that is, they do not show any useful behavior.***
- ✓ ***While many instructors wait until the introduction of GUI event handlers to discuss this topic, I feel that this is the proper place for this material within the context of the book. First, this is an “Advanced Class Feature” topic. Second, I am trying to group all of the syntax-related material within the first three days of the course. Third, inner classes can be used for more than event handlers and should be discussed independent of any specific use.***

Inner Classes

Consider the following class definition:

```

1 public class Outer1 {
2     private int size;
3
4     /* Declare an inner class called "Inner" */
5     public class Inner {
6         public void doStuff() {
7             // The inner class has access to 'size' from Outer
8             size++;
9         }
10    }
11
12    public void testTheInner() {
13        Inner i = new Inner();
14        i.doStuff();
15    }
16 }

```

The Outer class declares a data attribute called `size`, an inner class called `Inner`, and a method called `testTheInner`. The Inner class declares a method called `doStuff`. This method has access to the scope of the Outer class. That is, the `size` variable in the `doStuff` method (line 8) refers to the data attribute of the Outer object (line 2).

This implies that the inner object keeps a reference to the outer object. Figure 7-11 demonstrates how this might be implemented in the JVM.

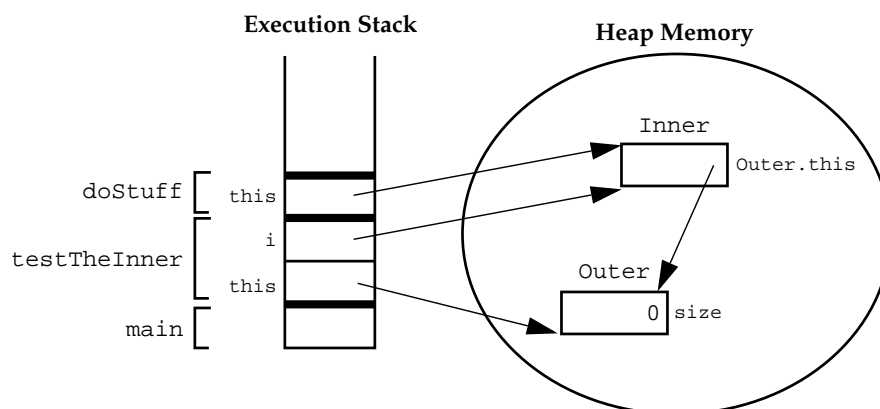


Figure 7-11 Memory Representation of Inner Classes

Inner Classes

In this example, we show how to instantiate objects of a public inner class outside of the definition of the Outer class.

```

1 public class Outer2 {
2     private int size;
3
4     public class Inner {
5         public void doStuff() {
6             size++;
7         }
8     }
9 }

1 public class TestInner {
2     public static void main(String[] args) {
3         Outer2 outer = new Outer2();
4
5         // Must create an Inner object relative to an Outer
6         Inner inner = outer.new Inner();
7         inner.doStuff();
8     }
9 }

```

An Inner class object is instantiated within the context of an instance of the Outer class (line 6 of the TestInner main method).

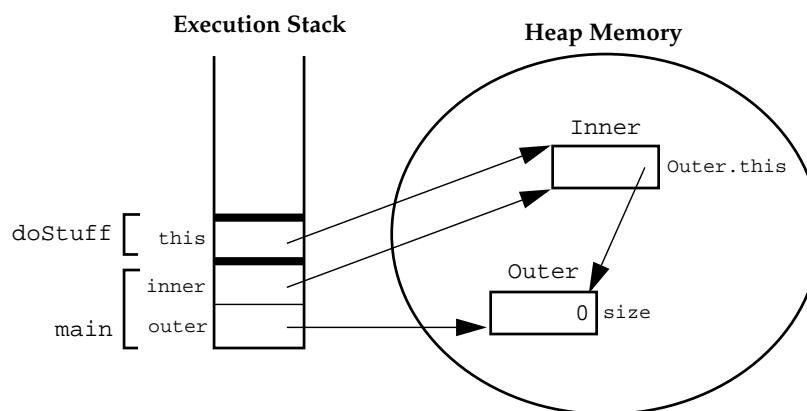


Figure 7-12 Access to an Inner Class Object From Another Class

Inner Classes

In this example, we show how to disambiguate variables with the same name:

```

1 public class Outer3 {
2     private int size;
3
4     public class Inner {
5         private int size;
6
7         public void doStuff(int size) {
8             size++;           // the local parameter
9             this.size++;     // the Inner object attribute
10            Outer.this.size++; // the Outer object attribute
11        }
12    }
13 }

```

The size variable is used in three contexts: as a data attribute for the Outer class, as a data attribute for the Inner class, and as a local variable of the doStuff method. This is perfectly valid, but the code must be written such that the compiler can distinguish each variable.

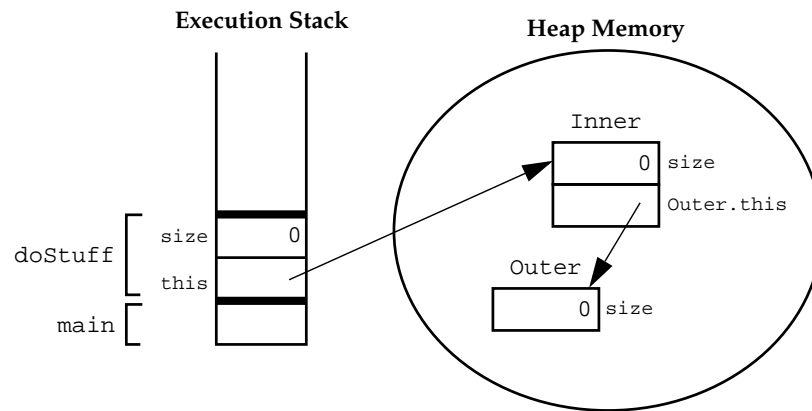


Figure 7-13 Disambiguation of Variables in Multiple Contexts

Inner Classes

In this example, we show that an inner class can be defined within the scope of a method:

```
1 public class Outer4 {
2     private int size = 5;
3
4     public Object makeTheInner(int localVar) {
5         final int finalLocalVar = 6;
6
7         // Declare a class within a method!?!
8         class Inner {
9             public String toString() {
10                return ("#<Inner size=" + size +
11                    // " localVar=" + localVar + // ERROR: ILLEGAL
12                    " finalLocalVar=" + finalLocalVar + ">");
13            }
14        }
15
16        return new Inner();
17    }
18
19    public static void main(String[] args) {
20        Outer4 outer = new Outer4();
21        Object obj = outer.makeTheInner(47);
22        System.out.println("The object is " + obj);
23    }
24 }
```

The important thing to note in this example is that methods of the inner class do not have complete access to the scope of the outer-method. For example, suppose you have a method that creates an inner class object and returns that object (line 16). Upon exiting the `makeTheInner` method, the local variable no longer exists, so methods of the inner class (such as `doStuff`) cannot have access to these variables at run-time. Therefore, line 11 issues a compiler error. However, the final variable exists beyond the run-time life of a method; therefore, line 12 is fine.

Note – This discussion of inner classes only shows the syntax of this mechanism. You will see how they are commonly used in Module 11, "GUI Event Handling."



Properties of Inner Classes

- You can use the class name only within the defined scope, except when used in a qualified name. The name of the inner class must differ from the enclosing class
- The inner class can be defined inside a method. Only local variables marked as `final`, can be accessed by methods within an inner class.

Properties of Inner Classes

Inner classes have the following properties:

- You can use the class name only within the defined scope, except when referenced by its qualified name. The name of the inner class must differ from the enclosing class.
- You can define the inner class inside a method. The rule that governs access to variables of enclosing methods is simple. Any variable, either a local variable or a formal parameter, cannot be accessed by methods within an inner class, unless the variable is marked as `final`.



Properties of Inner Classes

- The inner class can use both class and instance variables of enclosing classes and local variables of enclosing blocks
- The inner class can be defined as `abstract`
- The inner class can have any access mode
- The inner class can act as an interface implemented by another inner class

Properties of Inner Classes

- The inner class can use both static and instance variables of enclosing classes and final local variables of enclosing blocks.
- You can define the inner class as `abstract`. Therefore, you can have complete inner class hierarchies.
- You can declare inner classes with any level of access protection. For example, a `private` inner class can only be accessed with outer class scope; a `protected` inner class can be used by subclasses; and so on. Access protection does not prevent the inner class from using any member of another class as long as one encloses the other.
- An inner class can be an interface that is implemented by another inner class.



Properties of Inner Classes

- Inner classes that are declared `static` automatically become top-level classes
- Inner classes cannot declare any `static` members; only top-level classes can declare `static` members
- An inner class wanting to use a `static` member must be declared `static`

Properties of Inner Classes

- Inner classes that are declared `static` automatically become top-level classes. Static inner classes are not created relative to an object of the outer class. Therefore, methods of a static inner class do not have access to the outer class scope.
- Inner classes cannot declare any `static` members; only top-level classes can declare `static` members. Therefore, an inner class requiring a `static` member must use one from the top-level class.

Note – When a top-level class is compiled, the inner class is compiled as well. A `.class` file is created for it with a name that takes the form: `OuterClass$InnerClass`. For example, the `Inner` inner class is compiled to the file: `Outer$Inner.class`.

Exercise: Working With Interfaces and Abstract Classes



Exercise objective – You will rewrite, compile, and run a program that uses an abstract class and an interface.

Preparation

To successfully complete this lab, you must be familiar with the object-oriented concepts presented in this module.

Tasks

In a Web browser view the `lab_files.html` page that is at the top-level of the SL275 directory on your computer. There will be a summary of each exercise and a link to a page that gives a detailed explanation of the exercise.

Go to the SL275 directory on your computer and change to the directory for this module (`mod07`). A listing of this directory will show two subdirectories. This exercise is found in the directory called `exercise2`.

Exercise 2: Use Interfaces and Abstract Classes (Level 2)

In this exercise you will create a hierarchy of animals that is rooted in an abstract class `Animal`. Several of the animal classes will implement an interface called `Pet`. You will experiment with variations of these animals, their methods, and polymorphism.

Exercise: Working With Interfaces and Abstract Classes

Exercise Summary



Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercises.

✓ *If you do not have time to spend on discussion, just highlight the key concepts students should have learned from the lab exercise.*

- Experiences

✓ *Ask students what their overall experiences with this exercise have been. You might want to go over any trouble spots or especially confusing areas at this time.*

- Interpretations

✓ *Ask students to interpret what they observed during any aspects of this exercise.*

- Conclusions

✓ *Have students articulate any conclusions they have reached as a result of this exercise experience.*

- Applications

✓ *Explore with the students how they might apply what they learned in this exercise to situations at their workplace.*

Check Your Progress

Before continuing on to the next module, check to be sure that you can:

- Describe `static` variables, methods, and initializers
- Describe `final` classes, methods, and variables
- Explain how and when to use abstract classes and methods
- Explain how and when to use inner classes
- Distinguish between static and non-static inner classes
- Explain how and when to use an interface
- In a Java software program, identify:
 - ▼ `static` methods and attributes
 - ▼ `final` methods and attributes
 - ▼ inner classes
 - ▼ interface and abstract classes
 - ▼ abstract methods

Think Beyond

What features of the Java programming language are used to deal with runtime error conditions?

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

This module covers the error handling facilities built into the Java programming language.

Relevance

- ✓ **Present the following question to stimulate the students and get them thinking about the issues and topics presented in this module. They are not expected to know the answer to the question. Hold discussions where students have input; otherwise, if no one can propose answers, begin the lecture for this module.**



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

- In most programming languages, how are runtime errors resolved?

- ✓ **Writing a good computer program includes proper error handling and recovery. The Exception mechanism in the Java programming language provides a simple, streamlined, organized approach. This module covers the Exception mechanism built into the Java programming language, and how to use it effectively.**



Exceptions

- The `Exception` class defines mild error conditions that your program encounters.
- Exceptions can occur when:
 - The file you try to open does not exist
 - The network connection is disrupted
 - Operands being manipulated are out of prescribed ranges
 - The class file you are interested in loading is missing
- An error class defines serious error conditions

Exceptions

Introduction

What is an exception? In the Java programming language, the `Exception` class defines mild error conditions that your programs might encounter. Rather than letting the program terminate, you can write code to handle your exceptions and continue program execution.

Any abnormal condition that disturbs the normal program flow while the program is in execution is an error or exception. For example, exceptions can occur when:

- The file you try to open does not exist.
- The network connection is disrupted.
- Operands being manipulated are out of prescribed ranges.
- The class file you are interested in loading is missing.

Exceptions

Introduction (Continued)

In the Java programming language, the `Error` class defines what are considered to be serious error conditions from which you should not attempt to recover. In most cases, you should let the program terminate when such an error is encountered.

The Java programming language implements C++ style exceptions to help you build resilient code. When an error occurs in your program, the method that finds the error can “throw” an exception back to its caller to signal that a problem has occurred. The calling method then has the opportunity to “catch” the thrown exception and, when possible, recovers from it. This scheme gives the programmer the option of writing a “handler” to deal with the exception.

You can determine the exceptions a method throws by browsing the API.

Exceptions

Example

The following is an example of a version of the HelloWorld.java program that cycles through messages.

```
1 public class HelloWorld {
2     public static void main(String[] args) {
3         int i = 0;
4
5         String greetings [] = {
6             "Hello world!",
7             "No, I mean it!",
8             "HELLO WORLD!!"
9         };
10
11        while (i < 4) {
12            System.out.println(greetings[i]);
13            i++;
14        }
15    }
16 }
```

- ✓ ***This program quickly produces an exception. Ask students where the exception comes from.***
- ✓ ***The exception produced is the `ArrayIndexOutOfBoundsException`. It is produced in the `System.out.println` method when `i` has a value of 3.***

Exception Handling

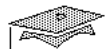
Introduction

Normally, a program terminates with an error message when an exception is thrown, as does the program shown previously after its loop has executed four times.

```
java HelloWorld
Hello world!
No, I mean it!
HELLO WORLD!!
java.lang.ArrayIndexOutOfBoundsException: 3
    at HelloWorld.main(HelloWorld.java:12)
```

Exception handling allows a program to catch exceptions, handle them, and then continue program execution. It is structured so that error cases do not get in the way of the normal flow of a program. These special cases are handled when they occur, in separate code blocks associated with the code for normal execution. This produces more legible and manageable code.

- ✓ **To see the line numbers in the stack trace, the student must set the `JAVA_COMPILER` environment variable to `NONE` or use the java command option `"-Djava.compiler=NONE"`.**



try and catch Statements

```
1 try {
2     // code that might throw a particular exception
3 } catch (MyExceptionType myExcept) {
4     // code to execute if a MyExceptionType exception is thrown
5 } catch (Exception otherExcept) {
6     // code to execute if a general Exception exception is thrown
7 }
```

Exception Handling

The Java programming language provides a mechanism for figuring out which exception was thrown and how to recover from it.

try and catch Statements

To handle a particular exception, place code, which when invoked throws exceptions, inside a `try` block and create a list of adjoining `catch` blocks, one for each possible exception that can be thrown. The block statement of a `catch` clause is executed if the exception generated matches the one listed in the `catch`. There can be multiple `catch` blocks after a `try` block, each handling a different exception type.

```
1 try {
2     // code that might throw a particular exception
3 } catch (MyExceptionType myExcept) {
4     // code to execute if a MyExceptionType exception is thrown
5 } catch (Exception otherExcept) {
6     // code to execute if a general Exception exception is thrown
7 }
```



Call Stack Mechanism

- If an exception is not handled in the current `try-catch` block, it is thrown to the caller of that method.
- If the exception gets back to the main method and is not handled there, the program is terminated abnormally.

Exception Handling

The Call Stack Mechanism

If a statement within a method throws an exception, 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 until the exception is handled. If an exception is not handled by the time it gets back to `main()` and `main()` does not handle it, the exception terminates the program abnormally.

Consider a case where the `main()` method calls another method named `first()`, and this in turn calls another method named `second()`. If an exception occurs in `second()`, it is thrown back to `first()`, where a check is made to see if there is a catch for that type of exception. If no catch exists in `first()`, the next method in the call stack, `main()`, is checked. If the exception is not handled by the last method on the call stack, then a runtime error occurs and the program stops executing.



finally Statement

```
1  try {
2    startFaucet();
3    waterLawn();
4  } catch (BrokenPipeException e) {
5    logProblem(e);
6  } finally {
7    stopFaucet();
8  }
```

Exception Handling

finally Statement

The finally statement 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  }
```

✓ **Use of `catch()` is optional, depending on the situation.**

Exception Handling

finally Statement (Continued)

In the previous example, the faucet is turned off whether or not 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 time the `finally` statement would not be executed is if the `System.exit()` method, which terminates the program, is executed within the protected code. This implies that the control flow can deviate from normal sequential execution. If, for example, a `return` statement is embedded in the code inside the `try` block, the code in the `finally` block executes before the `return`.

Exception Handling

Example Revisited

The following example is a rewrite of the `main()` method from page 7-5. The exception generated in the earlier version of the program is caught and the array index is reset, allowing the program to continue.

```
1 public class HelloWorld {
2     public static void main(String[] args) {
3         int i = 0;
4         String[] greetings = {
5             "Hello world!",
6             "No, I mean it!",
7             "HELLO WORLD!!"
8         };
9
10        while (i < 4) {
11            try {
12                System.out.println(greetings[i]);
13            } catch (ArrayIndexOutOfBoundsException e){
14                System.out.println("Re-setting Index Value");
15                i = -1;
16            } finally {
17                System.out.println("This is always printed");
18            }
19            i++;
20        }
21    }
22 }
```

✓ **The `try` statements can be nested and the exceptions can migrate upward.**

Exception Handling

Example Revisited (Continued)

The message displayed on the screen alternates among the following messages as the loop is executed:

```
Hello world!  
This is always printed  
No, I mean it!  
This is always printed  
HELLO WORLD!!  
This is always printed  
Re-setting Index Value  
This is always printed
```

- ✓ ***The fourth message displayed is due to the generated exception being caught and handled. The array index is reset to allow continuation of the alternating messages.***

Exception Categories

There are three broad categories of exceptions in the Java technology programming language. The class `java.lang.Throwable` acts as the parent class for all objects that are 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 essential subclasses of this, `Error`, `RuntimeException`, and `Exception`, which are shown in Figure 8-1.

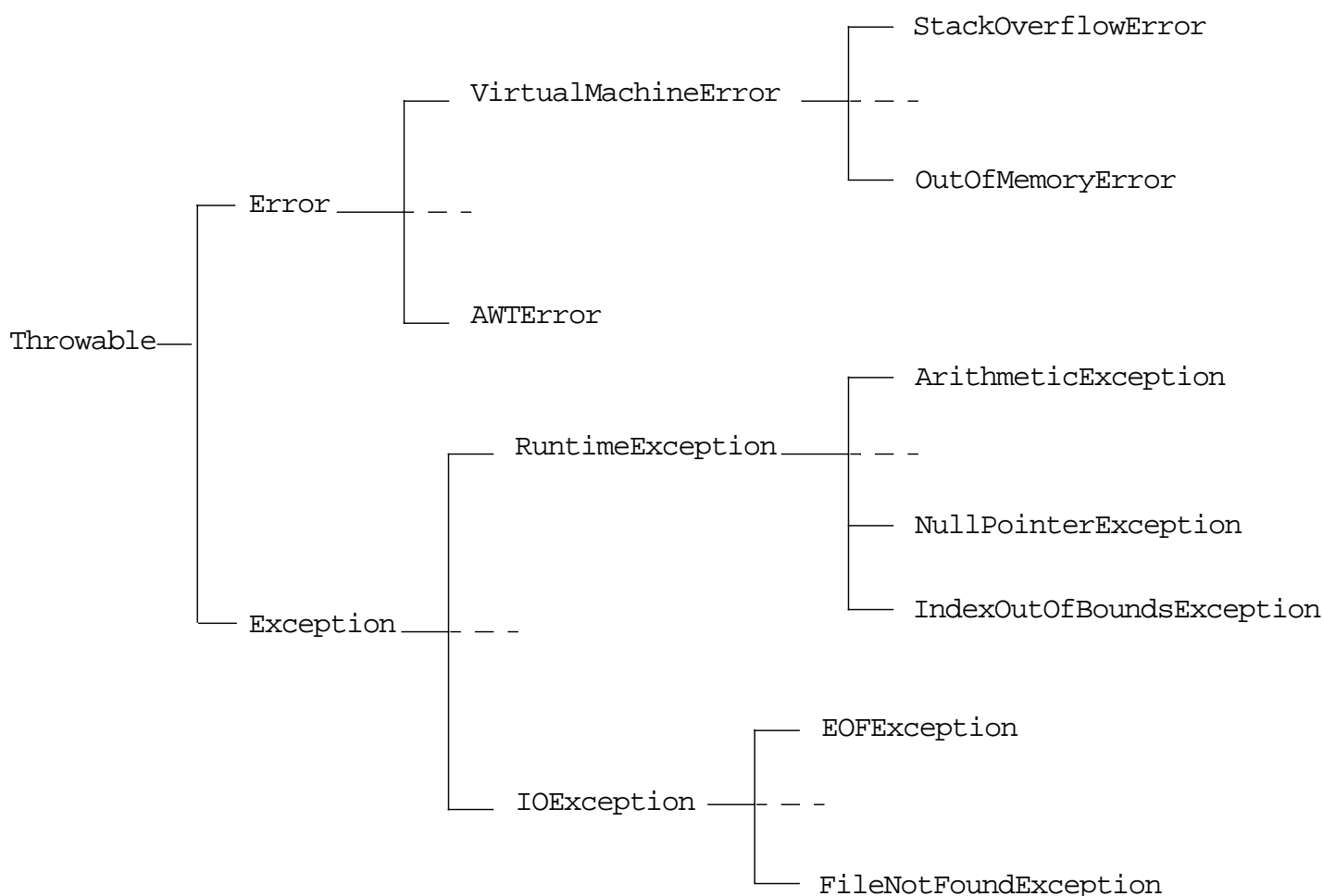


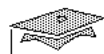
Figure 8-1 Subclasses and Exceptions

✓ **The `IndexOutOfBoundsException` is the superclass of `ArrayIndexOutOfBoundsException`, which is the exception thrown by the example.**

Exception Categories

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:

- `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.
- `RuntimeException` indicates a design or implementation problem. That is, it indicates conditions that should never happen if the program is operating properly. An `ArrayIndexOutOfBoundsException` exception, for example, should never be thrown if the array indices do not extend past the array bounds. This would also apply, for example, to referencing a null object variable. Because a correctly designed and implemented program 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 usually caused by environmental effects and can be handled. Examples include a file not found or invalid URL exceptions (user typed a wrong URL), both of which could easily occur if the user mistyped something. Because these usually occur as a result of user error, programmers are encouraged to handle them.



Common Exceptions

- ArithmeticException
- NullPointerException
- NegativeArraySizeException
- ArrayIndexOutOfBoundsException
- SecurityException

Common Exceptions

The Java programming language provides several predefined exceptions. Some of the more common exceptions are:

- ArithmeticException – The result of a divide-by-zero operation for integers:

```
int i = 12 / 0;
```

✓ **This exception is not generated by arithmetic overflow.**

✓ **ArithmeticExceptions are not generated by dividing floating point numbers by zero; these values are defined by the IEEE and are constants (final static), which are declared in the Float class.**

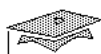
- **0.0/0 = NaN (not a number)**
- **FPN/0 = POSITIVE_INFINITY**
- **-FPN/0 = NEGATIVE_INFINITY**

- NullPointerException – An attempt to access an object's attribute or method when the object is not instantiated:

```
Date d = null;  
System.out.println(d.toString());
```

Common Exceptions

- `NegativeArraySizeException` – An attempt to create an array with a negative dimension size.
- `ArrayIndexOutOfBoundsException` – An attempt to access an element of an array beyond the array's size.
- `SecurityException` – Typically thrown in a browser, the `SecurityManager` class throws an exception for applets that attempt to do any of the following (unless explicitly allowed):
 - ▼ Access a local file
 - ▼ Open a socket to the host that is not the same host that served the applet
 - ▼ Execute another program in a runtime environment



The Handle or Declare Rule

- Handle the exception by using the `try-catch-finally` block
- Declare that the code causes an exception by using the `throws` clause
- A method may declare that it throws more than one exception

```
1 public void readDatabaseFile(String file)
2     throws FileNotFoundException, UTFDataFormatException {
3     // open file stream; may cause FileNotFoundException
4     FileInputStream fis = new FileInputStream(file);
5     // read a string from fis may cause UTFDataFormatException...
6 }
```

- You do not need to handle or declare run-time exceptions or errors

The Handle or Declare Rule

To encourage the writing of robust code, the Java programming language requires that if an `Exception` occurs while a method is on the stack (that is, it has been called), then the *caller* of that method must determine what action is to be taken if a problem arises.

The programmer can do the following to satisfy this requirement:

- Have the calling method handle the exception by including in its code a `try {} catch() {}` block where the `catch` names any superclass of the thrown exception. This counts as handling the situation, even if the `catch` block is empty.
- Have the calling method indicate that it does not handle the exception, and that the exception will be thrown back to *its* calling method.

Note – A method does not have to handle or declare run-time exceptions or errors.

The Handle or Declare Rule

A method can declare that an exception might be thrown in the body of the method with a throws clause as follows:

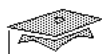
```
public void callsTroublesome() throws IOException
```

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.

✓ ***Even after an exception is caught, it can still be rethrown.***

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 as other classes are, 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 `IOException`s (`EOFException`, `FileNotFoundException`, and so on), by trapping `IOException` you can also catch instances of any subclass of `IOException`.



Method Overriding and Exceptions

- Must throw exceptions that are the same class as the exceptions being thrown by the overridden method
- May throw exceptions that are subclasses of the exceptions being thrown by the overridden method
- If a superclass method throws multiple exceptions, the overriding method must throw a proper subset of exceptions thrown by the overridden method

Method Overriding and Exceptions

When overriding a method that throws exceptions, the overriding method must also declare a throws clause that is a proper subset of exception classes thrown by the superclass method.

- The declared exception class *must be* the same class or a subclass. 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`)
- Fewer exceptions *may be* declared in the throws clause.
- New exceptions *may not be* added to the throws clause.

Method Overriding and Exceptions

In the example below, we declare three classes: TestA, TestB1, and TestB2. TestA is the superclass of TestB1 and TestB2.

```
1 public class TestA {
2     public void methodA() throws RuntimeException {
3         // do some number crunching
4     }
5 }
```

✓ ***It is not necessary to declare the RuntimeException class, but it is useful for these examples.***

```
1 public class TestB1 extends TestA {
2     public void methodA() throws ArithmeticException {
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 since ArithmeticException is a subclass of RuntimeException. However, class TestB2 fails to compile because Exception is a superclass of RuntimeException.

Method Overriding and Exceptions

In this example, we demonstrate the rules for declaring multiple exceptions in overriding methods.

```
1 import java.io.*;
2
3 public class TestMultiA {
4     public void methodA()
5         throws IOException, RuntimeException {
6         // do some IO stuff
7     }
8 }
```

The class `TestMultiA` declares a method that throws two classes of exceptions: `IOException` and `RuntimeException`.

```
1 import java.io.*;
2
3 public class TestMultiB1 extends TestMultiA {
4     public void methodA()
5         throws FileNotFoundException, UTFDataFormatException,
6         ArithmeticException {
7         // do some IO and number crunching stuff
8     }
9 }
```

The class `TestMultiB1` compiles because both `FileNotFoundException` and `UTFDataFormatException` are subclasses of `IOException` and `ArithmeticException` is a subclass of `RuntimeException`.

Method Overriding and Exceptions

```
1 import java.io.*;
2 import java.sql.*;
3
4 public class TestMultiB2 extends TestMultiA {
5     public void methodA()
6         throws FileNotFoundException, UTFDataFormatException,
7             ArithmeticException, SQLException {
8         // do some IO, number crunching, and SQL stuff
9     }
10 }
```

Class `TestMultiB2` fails to compile because `SQLException` is not consistent with any of the exceptions declared in the `TestMultiA` method; that is, you *cannot* add new exceptions to the overriding method.

```
1 public class TestMultiB3 extends TestMultiA {
2     public void methodA() throws java.io.FileNotFoundException {
3         // do some file IO
4     }
5 }
```

Class `TestMultiB3` demonstrates that you can declare that an overriding method throws fewer exceptions than the superclass method.

Creating Your Own Exceptions

Introduction

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 ServerTimeoutException extends Exception {
2     private int port;
3
4     public ServerTimeoutException(String message, int port) {
5         super(message);
6         this.port = port;
7     }
8
9     // Use the getMessage method to get the reason the exception was made
10
11     public int getPort() {
12         return port;
13     }
14 }
```

To throw an exception that you have created, use the following syntax:

```
throw new ServerTimeoutException("Could not connect", 80);
```

Creating Your Own Exceptions

Example

Consider a client-server program. In the client code, you try to connect to the server and expect the server to respond within 5 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     int success;
4     int portToConnect = 80;
5
6     success = open(serverName, portToConnect);
7
8     if (success == -1) {
9         throw new ServerTimedOutException("Could not connect",
10                                         portToConnect);
11     }
12 }
```

- ✓ **Point out that although the `open()` call here is fictitious, the `throw` command uses the correct syntax.**
- ✓ **Point out that you should create the exception at the point it is thrown. This is because the stack trace and line number information are added during construction and will be misleading otherwise.**

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 }
```

Creating Your Own Exceptions

Example (Continued)

Note – You can nest the try and catch blocks, as shown in the previous example.

You can also partially process an exception and then throw it as well. For example:

```
try {
    connectMe(defaultServer);
} catch (ServerTimeoutException e) {
    System.out.println("Error caught and rethrown");
    throw e;
}
```

Exercise: Handling and Creating Exceptions



Exercise objective – You will gain experience with the exception mechanism by writing Java software programs that create and handle exceptions.

Preparation

To successfully complete this lab, you must understand the concepts of handling runtime errors called exceptions.

Tasks

In a Web browser view the `lab_files.html` page that is at the top-level of the SL275 directory on your computer. There will be a summary of each exercise and a link to a page that gives a detailed explanation of the exercise.

Go to the SL275 directory on your computer and change to the directory for this module (`mod08`). A listing of this directory will show two subdirectories: one for each of the exercises below.

Exercise 1: Handle an Exception (Level 1)

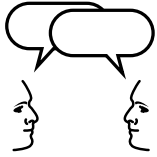
In this exercise you will use the `try-catch` block to handle a simple runtime exception.

Exercise 2: Create Your Own Exception (Level 1)

In this exercise you will create an `OverdraftException` that is thrown by the `withdraw` method in the `Account` class.

Exercise: Handling and Creating Exceptions

Exercise Summary



Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercises.

- ✓ *If you do not have time to spend on discussion, just highlight the key concepts students should have learned from the lab exercise.*

- Experiences

- ✓ *Ask students what their overall experiences with this exercise have been. You might want to go over any trouble spots or especially confusing areas at this time.*

- Interpretations

- ✓ *Ask students to interpret what they observed during any aspects of this exercise.*

- Conclusions

- ✓ *Have students articulate any conclusions they have reached as a result of this exercise experience.*

- Applications

- ✓ *Explore with the students how they might apply what they learned in this exercise to situations at their workplace.*

Check Your Progress

Before continuing on to the next module, check to be sure that you can:

- Define exceptions
- Use `try`, `catch`, and `finally` statements
- Describe exception categories
- Identify common exceptions
- Develop programs to handle your own exceptions

Think Beyond

How many situations can you think of that would require you to create new classes of exceptions?

Can you think of situations where a constructor would throw an exception?

Objectives

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

- Write a program that uses command-line arguments and system properties
- Write a program that reads from *standard input*
- Write a program that can create, read, and write files
- Describe the basic hierarchy of collections in Java 2 SDK
- Write a program that uses sets and lists
- Write a program to iterate over a collection
- Write a program to sort an array or a list
- Describe the collection classes that existed before Java 2 SDK
- Describe and use the javadoc and jar tools
- Identify deprecated classes and explain how to migrate from JDK 1.0 to JDK 1.1 to Java 2 JDK

This module covers a variety of topics that extend your knowledge of the Java 2 SDK. These topics cover elements of parameterizing the run-time behavior of a program, reading and writing text files, handling collections, sorting arrays and collections, and using Java technology tools to document your code and deploy systems.

- ✓ ***This is a long module. The goal is to present information useful for a programmer building full-scale applications. A few topics are included that are covered in the SCJP exam and are not covered elsewhere in this course. The instructor is free to skip certain sections if time is running short or to tailor the content to the needs or requests of the class.***

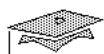
Relevance

- ✓ **Present the following questions to stimulate the students and get them thinking about the issues and topics presented in this module. They are not expected to know the answers to the questions. Hold discussions where students have input; otherwise, if no one can propose answers, begin the lecture for this module.**



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

- It is often the case that certain elements of a program should not be hard coded, such as file names or the name of a database. How can a program be coded to supply these elements at runtime?
- ✓ **A program can be parameterized by command-line arguments and system properties.**
 - Simple arrays are far too static for most collections (that is, a fixed number of elements). What Java technology features exist to support more flexible collections?
 - Besides computation, what are key elements of any text-based application?
- ✓ **Writing an application is much more than performing some calculations. A successful text-based application might use input from the keyboard or a file and produce output. It might also be parameterized by command-line arguments and system properties.**
 - Documentation is a key source of technology transfer. What Java technology tools support package and class API documentation?
- ✓ **Why javadoc, of course!**



Command-Line Arguments

- Any Java technology application can use command-line arguments
- These string arguments are placed on the command line to launch the Java interpreter, after the class name:

```
java TestArgs arg1 arg2 "another arg"
```

- Each command-line argument is placed in the `args` array that is passed to the static `main` method:

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

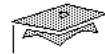
Command-Line Arguments

When a Java technology program is launched from a terminal window the user may provide the program with zero or more *command-line arguments*. These arguments are strings: either stand-alone 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. For example:

```
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. Try this:

```
java TestArgs arg1 arg2 "another arg"
```



System Properties

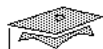
- System properties is a feature that replaces the concept of *environment variables* (which is platform-specific)
- The `System.getProperties` method returns a `Properties` object
- The `getProperty` method returns a `String` representing the value of the named property
- Use the `-D` option to include a new property

System Properties

System properties are another mechanism used to parameterize a program at run-time. 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 allows you to supply a default string value (the second parameter), which is returned if the named property does not exist.

Note – There is a default set of properties that every JVM must supply. (See the documentation for the `System.getProperties` method for details.) Moreover, a particular JVM vendor may 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, the `false` or `null` (respectively) is returned.



The Properties Class

- The `Properties` class implements a mapping of names to values (a `String` to `String` map)
- The `propertyNames` method returns an `Enumeration` of all property names
- The `getProperty` method returns a `String` representing the value of the named property
- You can also read and write a properties collection into a file using `load` and `store`

System Properties

The Properties Class

As mentioned above, 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 allows a default value to be specified which is returned if the named property does not exist.

You can iterate through the complete set of property names using the `propertyNames` method; and 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.

System Properties

The following program lists the complete set of properties that exist when the program executes:

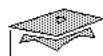
```
1 import java.util.Properties;
2 import java.util.Enumeration;
3
4 public class TestProperties {
5     public static void main(String[] args) {
6         Properties props = System.getProperties();
7         Enumeration prop_names = props.propertyNames();
8
9         while ( prop_names.hasMoreElements() ) {
10            String prop_name = (String) prop_names.nextElement();
11            String property = props.getProperty(prop_name);
12            System.out.println("property '" + prop_name
13                + "' is '" + property + "'");
14        }
15    }
16 }
```

Line 6 retrieves the set of system properties and line 7 retrieves an "enumeration" over all of the property names in the set of properties. An Enumeration object allows the program to loop over elements in a collection. This is very similar to an Iterator, which we will discuss in "Iterators" on page 9-31 The `hasMoreElements` method returns true if there are more elements to be iterated over and the `nextElement` method returns the next element in the enumeration. Line 11 retrieves the property value and lines 12-13 print out the property name/value pair.

```
> java -DmyProp=theValue TestProperties
```

Here is an excerpt of the output:

```
property 'java.vm.version' is '1.2.2'
property 'java.compiler' is 'NONE'
property 'path.separator' is ':'
property 'file.separator' is '/'
property 'user.home' is '/home/basham'
property 'java.specification.vendor' is 'Sun Microsystems Inc.'
property 'user.language' is 'en'
property 'user.name' is 'basham'
property 'myProp' is 'theValue'
```



Console I/O

- `System.out` allows you to write to "standard output"
 - ▼ It is an object of type `PrintStream`
- `System.in` allows you to read from "standard input"
 - ▼ It is an object of type `InputStream`
- `System.err` allows you to write to "standard error"
 - ▼ It is an object of type `PrintStream`

Console I/O

Most applications must interact with the user. Such interaction is often 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 2 SDK supports console I/O with three public variables on the `java.lang.System` class:

- `System.out` is a `PrintStream` object that (initially) refers to the terminal window that launched the Java technology application.
- `System.in` is an `InputStream` object that (initially) refers to the users keyboard.
- `System.err` is a `PrintStream` object that (initially) refers 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

- The `println` methods print the argument and a newline (`\n`)
- The `print` methods print the argument without a newline
- The `print` and `println` methods are overloaded for most primitive types (boolean, char, int, long, float, and double) and for `char[]`, `Object`, and `String`
- The `print(Object)` and `println(Object)` methods call the `toString` method on the argument

Console I/O

Writing to Standard Output

As we have seen, it is possible to 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 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.

Console I/O

Reading From Standard Input

The following example shows a technique that you should use to read String information from the console standard input:

```
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 = new InputStreamReader(System.in);
9         BufferedReader in = new BufferedReader(ir);
```

Line 5 declares a String variable, *s*, that the program uses to hold each line read from standard input.

Lines 8-9 wrap `System.in` with two support objects that massage the stream of bytes coming from standard input. The `InputStreamReader` (*ir*) reads characters from and converts the raw bytes into Unicode characters. The `BufferedReader` (*in*) provides the `getLine` method which allows the program to read from standard input a line at a time.

✓ **Stream chaining is covered in Module 15, "Advanced I/O Streams."**

```
10
11     System.out.println("Type ctrl-d or ctrl-c to exit.");
```

Note – The `ctrl-d` character on UNIX indicates the “end of file” condition. On Microsoft Windows use the keystroke sequence: `ctrl-z` <Enter> to indicate end of file (EOF).

Console I/O

Reading From Standard Input (Continued)

```
12
13 try {
14     // Read each input line and echo it to the screen.
15     s = in.readLine();
16     while ( s != null ) {
17         System.out.println("Read: " + s);
18         s = in.readLine();
19     }
```

Line 15 reads the first line of text from standard input. The `while` loop (lines 16-19) 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);
}
```

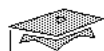
Because the `readLine` method may throw an I/O exception, all of this code needs to be wrapped in a `try-catch` block.

```
20
21     // Close the buffered reader.
22     in.close();
```

Line 22 closes the outer most input stream. This is done to release any system resources related to creating these stream objects.

```
23     } catch (IOException e) { // Catch any IO exceptions.
24         e.printStackTrace();
25     }
26 }
27 }
```

Finally, the program handles any I/O exceptions that may arise.



Files and File I/O

- The `java.io` package
- Creating `File` objects
- Manipulating `File` objects
- Reading and writing to file streams

Files and File I/O

Input/Output (I/O) is one of the most important elements of programming. Java technology includes a rich set of I/O "streams." In the previous section we saw how to use streams to communicate with the user through standard output (usually a terminal window) and standard input (usually the keyboard). In this section we will examine a few simple techniques for reading and writing to files with a focus on character data. We will cover:

- Creating `File` objects
- Manipulating `File` objects
- Reading and writing to file streams

Module 15, "Advanced I/O Streams," covers the details of Java technology I/O streams.



Creating a New File Object

- `File myFile;`
- `myFile = new File("myfile.txt");`
- `myFile = new File("MyDocs", "myfile.txt");`
- Directories are treated just like files in Java; the `File` class supports methods for retrieving an array of files in the directory
- `File myDir = new File("MyDocs");`
`myFile = new File(myDir, "myfile.txt");`

Files and File I/O

Creating a New File Object

The `File` class provides several utilities for dealing with files and obtaining information about them. In Java, 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 is done in the third example below.

- `File myFile;`
`myFile = new File("myfile.txt");`
- `myFile = new File("MyDocs", "myfile.txt");`
- `File myDir = new File("MyDocs");`
`myFile = new File(myDir, "myfile.txt");`

Files and File I/O

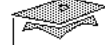
Creating a New File Object (Continued)

The constructor you use often depends on the other file objects 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 allow 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 generally recommended.

- ✓ **Although Microsoft Windows uses `\` as directory delimiters, `/` works within Java `File` and other streaming classes. If no device is specified (`/thisDevice/outputFile`), the paths are considered relative to the current device. If you need to specify a different device, preface the path with the device name (`D:/otherDevice/inputFile`). You use `/`; even though you are referencing files on a Microsoft Windows system.**



File Tests and Utilities

- File names:

```
String getName()  
String getPath()  
String getAbsolutePath()  
String getParent()  
boolean renameTo(File newName)
```

- File tests:

```
boolean exists()  
boolean canWrite()  
boolean canRead()  
boolean isFile()  
boolean isDirectory()  
boolean isAbsolute();
```

Files and File I/O

File Tests and Utilities

Once you have created a `File` object, you can use any of the following methods 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)`

Files and File I/O

File Tests and Utilities (Continued)

File Tests

The following methods return information about file attributes.

- `boolean exists()`
- `boolean canWrite()`
- `boolean canRead()`
- `boolean isFile()`
- `boolean isDirectory()`
- `boolean isAbsolute()`

General File Information and Utilities

The following methods return general file information.

- `long lastModified()`
- `long length()`
- `boolean delete()`

Directory Utilities

The following methods provide directory utilities.

- `boolean mkdir()`
- `String[] list()`



File Stream I/O

- File Input:
 - ▼ Use the `FileReader` class to read characters
 - ▼ Use the `BufferedReader` class to use the `readLine` method
- File Output:
 - ▼ Use the `FileWriter` class to write characters
 - ▼ Use the `PrintWriter` class to use the `print` and `println` methods

Files and File I/O

File Stream I/O

The following program reads a text file and echoes each line to standard output; thus printing the 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 to read each line from a file.
9             BufferedReader in = new BufferedReader(new FileReader(file));
10            String s;
```

Line 5 creates a new `File` object based on the first command-line argument to the program. Line 10 creates a buffered reader that wraps around a file reader. This code may throw a `FileNotFoundException` if the file does not exist.

Files and File I/O

File Stream I/O (Continued)

The following program reads a text file and echoes each line to standard output; thus printing the file.

```
11
12     // Read each line from the file and echo it to the screen.
13     s = in.readLine();
14     while ( s != null ) {
15         System.out.println("Read: " + s);
16         s = in.readLine();
17     }
18     // Close the buffered reader, which also closes the file reader.
19     in.close();
20
21 } catch (FileNotFoundException e1) {
22     // If this file does not exist
23     System.err.println("File not found: " + file);
24
25 } catch (IOException e2) {
26     // Catch any other IO exceptions.
27     e2.printStackTrace();
28 }
29 }
30 }
```

The while loop in lines 13 through 15 is exactly the same as in the `KeyboardInput` program; it reads each text line in the buffered reader and echoes it to standard output.

Line 17 closes the buffered reader, which in turn closes the file reader that the buffered reader object decorates.

The exception handling code in lines 19 through 21 is used to catch the `FileNotFoundException` that might be thrown by the `FileReader` constructor. Lines 23 through 25 handle any other I/O-based exception that might be thrown (by the `readLine` and `close` methods).

Files and File I/O

File Output

The following program reads input lines from the keyboard and echoes each line to a file.

```
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 to read each line from standard in.
10            BufferedReader in
11                = new BufferedReader(new InputStreamReader(System.in));
12            // Create a print writer on this file.
13            PrintWriter out
14                = new PrintWriter(new FileWriter(file));
15            String s;
```

Just as in the previous example, line 6 creates a File object based on the first command-line argument. Lines 10-11 create a buffered reader for the standard input. Lines 13-14 create a print writer that decorates a file writer for the file created in line 6.

```
16
17            System.out.print("Enter file text. ");
18            System.out.println("[Type ctrl-d to stop.]");
19
20            // Read each input line and echo it to the screen.
21            while ((s = in.readLine()) != null) {
22                out.println(s);
23            }
```

Line 17 and 18 prompt the user to enter lines of text to be placed in the file and to type ctrl-d to stop.

Note – The ctrl-d character (which represents the "end of file/input") must be used in this example and not ctrl-c because ctrl-c terminates the JVM before the program properly closes the file stream.

Files and File I/O

File Output (Continued)

```
24
25     // Close the buffered reader and the file print writer.
26     in.close();
27     out.close();
28
29     } catch (IOException e) {
30     // Catch any IO exceptions.
31     e.printStackTrace();
32     }
33 }
34 }
```

Lines 26 and 27 dutifully close the input and output streams. Lines 29 through 31 handle any I/O exceptions that might be thrown.

Exercise: File Input and Output



Exercise objective – You will gain experience with reading from the standard input and writing to a file.

Preparation

To successfully complete this lab, you must understand the concepts of reading text lines from standard input and sending text to a file output stream.

Tasks

In a Web browser view the `lab_files.html` page that is at the top-level of the SL275 directory on your computer. There will be a summary of each exercise and a link to a page that gives a detailed explanation of the exercise.

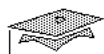
Go to the SL275 directory on your computer and change to the directory for this module (`mod09`). A listing of this directory will show six subdirectories. These two will be found in a directory called `exercisel` and `exercise2`.

Exercise 1: Write a File with Numbered Lines (Level 1 Lab)

In this exercise you will create a program to read text from standard input and write it to a file with each line prefixed with a line-number count. This file will be specified by a command-line argument.

Exercise 2: Develop a Directory Listing Program (Level 3 Lab)

In this exercise you will create a program to print out a directory listing for the directory specified by the last command-line argument.



The Math Class

The `Math` class contains a group of static math functions:

- truncation: `ceil`, `floor`, and `round`
- variations on `max`, `min`, and `abs` (absolute value)
- trigonometry: `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `toDegrees`, and `toRadians`
- logarithms: `log` and `exp`
- others: `sqrt`, `pow`, and `random`
- constants: `PI` and `E`

The Math Class

✓ *The next three sections are reference material for the SCJP exam.*

The `Math` class in the `java.lang` package contains a group of static methods and two constants that support mathematical calculations. This class cannot be extended (`final`) and no instance can be made (private constructor).

Truncation Methods

- `double ceil(double d)` – Returns the smallest integer that is not less than `d`
- `double floor(double d)` – Returns the largest integer that is not greater than `d`
- `int round(float f)` – Returns the closest `int` to `f`
- `long round(double d)` – Returns the closest `long` to `d`

The Math Class

Variations on min, max, and abs

- `double abs(double d)` – Returns the absolute value of `d`; likewise for `float`, `int`, and `long`
- `double min(double d1, double d2)` – Returns the smaller of `d1` and `d2`; likewise for `float`, `int`, and `long`
- `double max(double d1, double d2)` – Returns the greater of `d1` and `d2`; likewise for `float`, `int`, and `long`

Trigonometry Functions

- `double sin(double d)` – Returns the sine of `d`; likewise for `cos` (cosine), and `tan` (tangent)
- `double asin(double d)` – Returns the arc sine of `d`; likewise for `acos` (arc cosine), and `atan` (arc tangent)
- `double toDegrees(double r)` – Converts radians to degrees
- `double toRadians(double d)` – Converts degrees to radians

Logarithm Functions

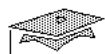
- `double log(double d)` – Returns the natural logarithm of `d`
- `double exp(double d)` – Returns e raised to the power of `d`

Other Functions

- `double sqrt(double d)` – Returns the square root of `d`
- `double pow(double d1, double d2)` – Returns the value of `d1` raised to the power of `d2`
- `double random()` – Returns a random number between 0.0 and 1.0

Constants

- `PI` – The double value that is closer to any other than to π , the ratio of a circle's circumference to its diameter
- `E` – The double value that is closer to any other than to e , the base of the natural logarithms



The String Class

- String objects are *immutable* sequences of Unicode characters
- Operations that create new strings: `concat`, `replace`, `substring`, `toLowerCase`, `toUpperCase`, and `trim`
- Search operations: `endsWith`, `startsWith`, `indexOf`, and `lastIndexOf`
- Comparisons: `equals`, `equalsIgnoreCase`, and `compareTo`
- Others: `charAt` and `length`

The String Class

A String object is an *immutable* (cannot be changed) sequence of Unicode characters.

Methods That Create New Strings

- `String concat(String s)` – Returns a new string consisting of this string followed by the `s` string
- `String replace(String old, String new)` – Returns a new string which is a copy of this string with the new string replacing all occurrences of the old string
- `String substring(int start, int end)` – Returns a portion of this string starting at the `start` index and ending at `end` (An alternate method defaults `end` to the length of the string)
- `String toLowerCase()` – Returns a new string consisting of this string converted to lower case
- `String toUpperCase()` – Returns a new string consisting of this string converted to upper case

The String Class

Search Methods

- `boolean endsWith(String s)` – Returns true if this string ends with `s`
- `boolean startsWith(String s)` – Returns true if this string starts with `s`
- `int indexOf(String s)` – Returns index within this string that starts with `s`; likewise for `lastIndexOf` but a reverse search
- `int indexOf(int ch)` – Returns index within this string of the first occurrence of the character `ch`; likewise for `lastIndexOf` but a reverse search
- `int indexOf(String s, int offset)` – Returns index within this string that matches with `s` starting at `offset`; likewise for `lastIndexOf` but a reverse search starting at `offset`
- `int indexOf(int ch, int offset)` – Returns index within this string of the first occurrence of the character `ch` starting at `offset`; likewise for `lastIndexOf` but a reverse search starting at `offset`

Comparison Methods

- `boolean equals(String s)` – Returns true if this string is equal to (character by character) the string `s`
- `boolean equalsIgnoreCase(String s)` – Returns true if this string is equal to (ignoring case) the string `s`
- `int compareTo(String s)` – Performs a lexical comparison between this string and `s`; returns a negative `int` if this string is less than `s`, a positive `int` if this is greater than `s`, or zero if the two strings are equal

Other Methods

- `char charAt(int index)` – Returns the character at the `index`
- `int length()` – Returns the length of the string



The StringBuffer Class

- `StringBuffer` objects are mutable sequences of Unicode characters
- Constructors:
 - ▼ `StringBuffer()` – Creates an empty buffer
 - ▼ `StringBuffer(int capacity)` – Creates an empty buffer with a specified initial capacity
 - ▼ `StringBuffer(String initialString)` – Creates a buffer that initially contains the specified string
- Modification operations: `append`, `insert`, `reverse`, `setCharAt`, and `setLength`

The StringBuffer Class

A `StringBuffer` object is a mutable sequence of Unicode characters. Never confuse a `String` and a `StringBuffer`; there is no inheritance relationship between them. You cannot assign a `String` object to a variable that is declared as a `StringBuffer`, nor can you assign a `StringBuffer` to a `String` variable. However, you can create a new `String` from a `StringBuffer` by calling the `toString` method on the buffer object. You can create a `StringBuffer` from a `String` by using the third constructor listed below.

Constructors

- `StringBuffer()` – Creates an empty string buffer
- `StringBuffer(int capacity)` – Creates an empty string buffer with a specified initial capacity
- `StringBuffer(String initialString)` – Creates a string buffer that initially contains the specified string

The StringBuffer Class

Modification Methods

- `StringBuffer append(String s)` – Modifies this string buffer by appending the `s` string onto the end of the buffer; likewise, there are overloaded methods for the following parameter types: `boolean`, `char`, `char[]`, `double`, `float`, `int`, `long`, and `Object`
- `StringBuffer insert(int offset, String s)` – Modifies this string buffer by inserting the `s` string into the buffer at the specified offset location; likewise, there are overloaded methods for the following parameter types: `boolean`, `char`, `char[]`, `double`, `float`, `int`, `long`, and `Object`
- `StringBuffer reverse()` – Reverses the order of the string buffer

Note – These modification methods return the string buffer itself, so that they can be sequenced together:

```
buffer.append(3).append(" blind ").append("mice;")
        .append('\n').append("see how they run.");
```

- `void setCharAt(int index, char ch)` – Modifies this string buffer by changing the character at the location specified by `index` to the specified character, `ch`
- `void setLength(int newLength)` – Sets the length of the string buffer



Collections

- A *collection* is a single object representing a group of objects known as its elements
- The Collection API contains interfaces that group objects as a:
 - ▼ *Collection* – A group of objects with no specific ordering; duplicates are permitted
 - ▼ *Set* – An unordered collection; no duplicates are permitted
 - ▼ *List* – An ordered collection; duplicates are permitted

The Collections API

A *collection* is a single object representing a group of objects. The objects in the collection are called *elements*. Collections typically deal with many types of objects, all of which are of a particular kind (that is, they all descend from a common parent type).

- *Collection* – A group of objects with no specific ordering; duplicates are permitted. Also known as a *bag* or *multiset*.
 - *Set* – An unordered collection; no duplicates are permitted.
 - *List* – An ordered collection; duplicates are permitted.
- ✓ ***If a structured query language (SQL) database API furnishes a collection and the GUI toolkit expects a collection, these APIs interoperate seamlessly because they take collections as input and return collections as output.***

Collections maintain references to objects of type `Object`. This allows 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.

The Collections API

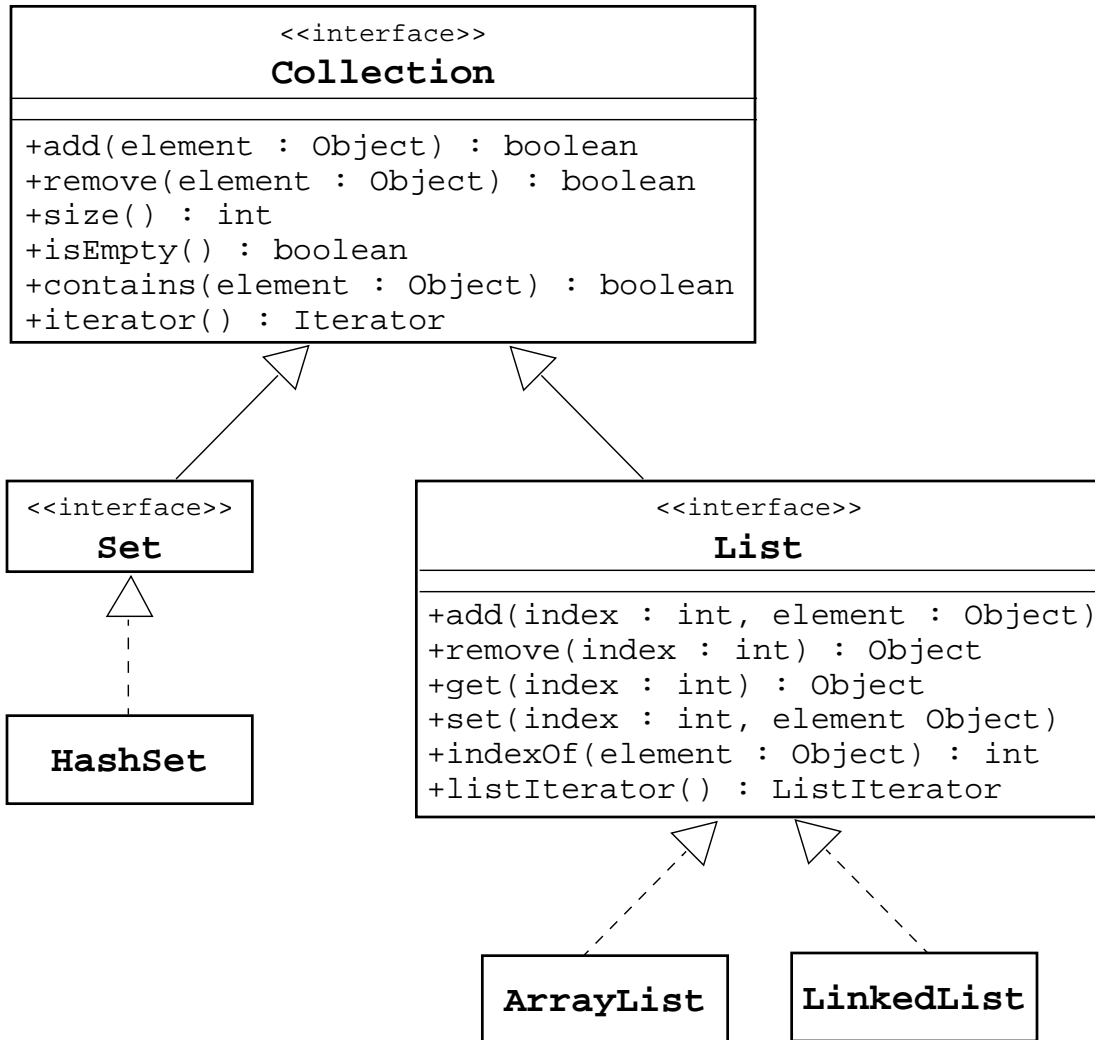


Figure 9-1 The Collection Interface and Class Hierarchy

The HashSet class supplies an implementation of the Set 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 URL:
<http://developer.java.sun.com/developer/onlineTraining/collections/>

The Collections API

A Set Example

In the following example, the program declares a variable (`set`) of type `Set` and is initialized to a new `HashSet` object. It then adds a few elements and prints the set to standard output. The attempt to add the duplicate items on lines 11 and 12 fails; thus the `add` method returns `false`.

```
1 import java.util.*;
2
3 public class SetExample {
4     public static void main(String[] args) {
5         Set set = new HashSet();
6         set.add("one");
7         set.add("second");
8         set.add("3rd");
9         set.add(new Integer(4));
10        set.add(new Float(5.0F));
11        set.add("second");           // duplicate, not added
12        set.add(new Integer(4))     // duplicate, not added
13        System.out.println(set);
14    }
15 }
```

The output generated from this program might be:

```
[one, second, 5.0, 3rd, 4]
```

Note – On line 13 the program prints the `set` object to standard output. This works because the `HashSet` class overrides the `toString` method that creates a comma-separated sequence of the items delimited by the open and close braces.

The Collections API

A List Example

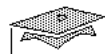
In the following example, the program declares a variable (`list`) of type `List` and is initialized to a new `ArrayList` object. It then adds a few elements and prints the list to standard output. Because lists allow duplicates, lines 11 and 12 succeed.

```
1 import java.util.*;
2
3 public class ListExample {
4     public static void main(String[] args) {
5         List list = new ArrayList();
6         list.add("one");
7         list.add("second");
8         list.add("3rd");
9         list.add(new Integer(4));
10        list.add(new Float(5.0F));
11        list.add("second");           // duplicate, is added
12        list.add(new Integer(4));    // duplicate, is added
13        System.out.println(list);
14    }
15 }
```

The output generated from this program is:

```
[one, second, 3rd, 4, 5.0, second, 4]
```

✓ **Notice that we could have used the declaration `Collection list` on line 5 and the rest of the code would still be the same.**



Iterators

- Iteration is the process of retrieving every element in a collection
- An Iterator of a Set is unordered
- A ListIterator of a List can be scanned forwards (using the `next` method) or backwards (using the `previous` method)

```
List list = new ArrayList();
// add some elements
Iterator elements = list.iterator();
while ( elements.hasNext() ) {
    System.out.println(elements.next());
}
```

The Collections API

Iterators

A collection can be scanned using an *iterator*. The basic `Iterator` interface allows scanning 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 allows the list to be scanned backwards.

The following code fragment demonstrates the use of an iterator:

```
List list = new ArrayList();
// add some elements
Iterator elements = list.iterator();
while ( elements.hasNext() ) {
    System.out.println(elements.next());
}
```

The Collections API

Iterators (Continued)

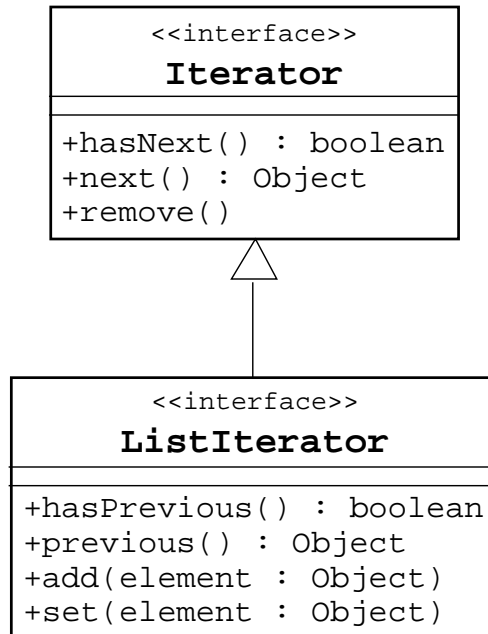
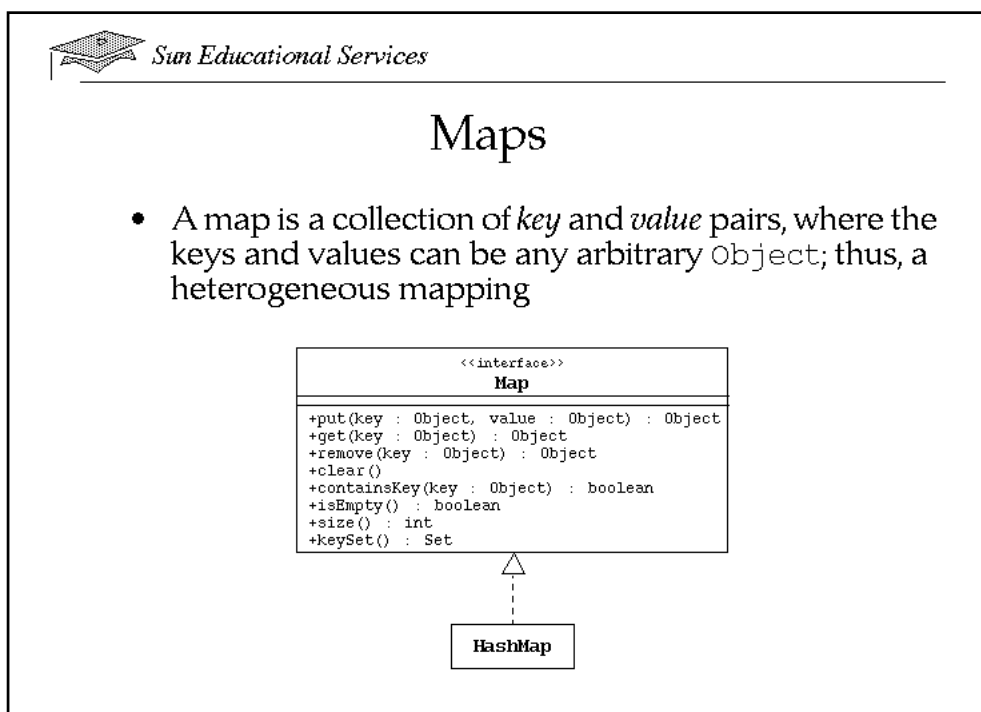


Figure 9-2 The Iterator Interface Hierarchy

The `remove` method allows 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: forward using `next` and backward using `previous`. If you use `previous` immediately after `next`, then you will get back 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 will return the newly added element. However, a call to `next` will not be affected. If setting or adding is not supported by the underlying collection, then an `UnsupportedOperationException` is thrown.



The Collections API

Maps

A map is a collection of arbitrary associations between a key object and a value object. In a given map, there may only be one entry for a given key.

The `put` method inserts a key and value pair into the map. If the key already exists, then the new value replaces the old value. The `get` method returns the value associated with a given key, or `null` if the key does not exist in the map.

The `HashMap` class implements the `Map` interface.

Iteration over a map can be accomplished by retrieving the set of keys, using the `keySet` method, and then iterating over the key set and retrieving the values as you go along.

The Collections API

A Map Example

The following program demonstrates the use of a map. In this case the map is between a word (`String`) and the number of times the word has been used (`Integer`).

```

1 import java.util.Map;
2 import java.util.HashMap;
3 import java.util.Iterator;
4 import java.io.FileReader;
5
6 public class MapExample {
7     public static void main(String[] args)
8         throws java.io.FileNotFoundException {
9         Map        word_count_map = new HashMap();
10        FileReader reader = new FileReader(args[0]);
11        Iterator   words = new WordStreamIterator(reader);
12
13        while ( words.hasNext() ) {
14            String word = (String) words.next();
15            String word_lowercase = word.toLowerCase(); // this is the key
16            Integer frequency = (Integer)word_count_map.get(word_lowercase);
17
18            if ( frequency == null ) {
19                frequency = new Integer(1);
20            } else {
21                int value = frequency.intValue();
22                frequency = new Integer(value + 1);
23            }
24            word_count_map.put(word_lowercase, frequency);
25        }
26        System.out.println(word_count_map);
27    }
28 }

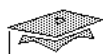
```

Running this program on the first paragraph of Shakespeare's *Romeo and Juliet* generates the following output:

```

> java MapExample romeo_and_juliet.txt
{unclean=1, with=2, scene=1, passage=1, our=3, ancient=1, two=3, these=1, mark'd=1,
patient=1, do=1, cross'd=1, where=2, lovers=1, fatal=1, stage=1, verona=1, new=1,
bury=1, forth=1, strife=1, lay=1, fair=1, we=1, alike=1, could=1, piteous=1, is=1,
hands=1, mend=1, in=2, nought=1, both=1, continuance=1, life=1, if=1, shall=2, the=5,
traffic=1, and=1, a=1, toil=1, take=1, which=2, loins=1, of=5, here=1, end=1, what=1,
civil=2, their=6, love=1, but=1, makes=1, miss=1, rage=1, foes=1, you=1, ears=1,
whose=1, now=1, to=2, dignity=1, fearful=1, pair=1, star=1, strive=1, households=1,
hours'=1, grudge=1, break=1, misadventured=1, mutiny=1, attend=1, overthrows=1,
parents'=2, blood=1, from=2, children's=1, remove=1, death=2}

```



Sorting Arrays and Collections

- Sorting arrays using `Arrays.sort` methods:
 - ▼ `void sort(<type> array[])`
 - ▼ `void sort(<type> array[], int fromIndex, int toIndex)`
where `<type>` is any primitive type (except `boolean`)
- Sorting lists using `Collections.sort` methods:
 - ▼ `void sort(List)`
 - ▼ `void sort(List, Comparator)`
- The `Comparable` and `Comparator` interfaces
- Sorting a `Set` using a `SortedSet` implementation

The Collections API

Sorting

Sorting is fundamental to any complex business application, especially when generating reports. In the new Java 2 SDK Collections API, several implementations of sorting have been included.

Sorting arrays is accomplished by the set of overloaded `Arrays.sort` method. There are two variations: `sort(<type> array[])` which sorts the whole array and `sort(<type> array[], int fromIndex, int toIndex)` which sorts a portion of the array. The `<type>` variable can be any primitive type except `boolean`. There are four methods for sorting arrays of Objects:

- `Arrays.sort(Object array[])`
- `Arrays.sort(Object array[], int fromIndex, int toIndex)`
- `Arrays.sort(Object array[], Comparator comparator)`
- `Arrays.sort(Object array[], int fromIndex, int toIndex, Comparator comparator)`

The Collections API

Sorting (Continued)

Since the sorting algorithm must compare two objects (to determine their order), the class of the objects in the array must implement the `Comparable` interface or an object implementing the `Comparator` interface must be passed into the sorting algorithm (using the second or fourth variation listed above).

The `Comparable` interface supports one method: `compareTo`. This method compares this object to the element parameter. If this is a "less than" element, then a negative value is returned; if this is a "greater than" element, then a positive value is returned; otherwise zero is returned.

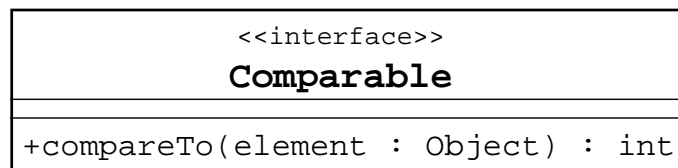


Figure 9-3 The `Comparable` Interface

The `Comparator` interface supports two methods: `compare` and `equals`. The `compare` method takes two parameters, `e1` and `e2`, both of type `Object`. It returns a negative value if `e1` "is less than" `e2`, a positive value if `e1` "is greater than" `e2`, otherwise zero. The `equals` method is used to determine if two comparator objects are equal. You do not have to implement this method because the class that implements the `Comparator` interface can inherit the default implementation supplied by the `Object` class.

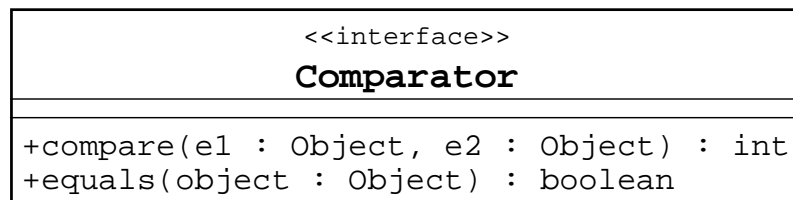


Figure 9-4 The `Comparator` Interface

The Collections API

Sorting Examples

The following program demonstrates sorting an array of doubles:

```
1 import java.util.Arrays;
2 import java.text.DecimalFormat;
3
4 public class SortingExample1 {
5     public static void main(String[] args) {
6         double[] random_values = new double[10];
7
8         // populate the array with random numbers
9         for ( int i = 0; i < random_values.length; i++ ) {
10            random_values[i] = Math.random();
11        }
12
13        // print out unsorted array
14        System.out.println("Unsorted Array:");
15        printArray(random_values);
16
17        // print out sorted array
18        Arrays.sort(random_values);
19        System.out.println("Sorted Array:");
20        printArray(random_values);
21    }
22
23    private static void printArray(double[] array) {
24        System.out.print('[ ');
25        for ( int i = 0; i < array.length; i++ ) {
26            System.out.print(DecimalFormat.format(array[i]));
27            if ( (i + 1) < array.length ) {
28                System.out.print(", ");
29            }
30        }
31        System.out.println(']');
32    }
33    private static DecimalFormat FORMAT = new DecimalFormat("0.000");
34 }
```

Running this program generates the following output:

```
Unsorted Array:
[0.604, 0.717, 0.835, 0.050, 0.662, 0.168, 0.919, 0.958, 0.637, 0.011]
Sorted Array:
[0.011, 0.050, 0.168, 0.604, 0.637, 0.662, 0.717, 0.835, 0.919, 0.958]
```

Note – The `java.util.Arrays` class should not be confused with the `java.lang.reflect.Array` class or a simple array.

The Collections API

Sorting Examples (Continued)

The following program demonstrates sorting a list of the word count map entries:

```

1  import java.util.*;
2
3  public class SortingExample2 {
4      public static void main(String[] args)
5          throws java.io.FileNotFoundException {
6          Map    word_count_map = new WordCountMap(args[0]);
7          Set    entry_set = word_count_map.entrySet();
8
9          System.out.println("Unsorted Entry Set:\n" + entry_set);
10
11         // Create a list of the entries and sort it alphabetically
12         List    entry_list = new ArrayList(entry_set);
13         Collections.sort(entry_list, new AlphaComparator());
14         System.out.println("\nEntry Set (sorted alpha):\n" + entry_list);
15
16         // Sort the list by frequency
17         Collections.sort(entry_list, new FreqComparator());
18         System.out.println("\nEntry Set (sorted by freq):\n" + entry_list);
19     }
20
21     private static class AlphaComparator implements Comparator {
22         public int compare(Object e1, Object e2) {
23             String word1 = (String) ((Map.Entry) e1).getKey();
24             String word2 = (String) ((Map.Entry) e2).getKey();
25             return word1.compareTo(word2);
26         }
27     }
28     private static class FreqComparator implements Comparator {
29         public int compare(Object e1, Object e2) {
30             Integer freq1 = (Integer) ((Map.Entry) e1).getValue();
31             Integer freq2 = (Integer) ((Map.Entry) e2).getValue();
32             return freq2.compareTo(freq1);
33         }
34     }
35 }

```

Note – The `WordCountMap` class encapsulates the basic code on “A Map Example” section on page 9-34. The constructor takes a file name as its argument and processes the “word count” of the text in that file.

The Collections API

Sorting Examples (Continued)

Running this program on the first paragraph of Shakespeare's *Romeo and Juliet* generates the following output:

```
> java SortingExample2 romeo_and_juliet.txt
```

Unsorted Entry Set:

```
[unclean=1, with=2, scene=1, passage=1, our=3, ancient=1, two=3, these=1, mark'd=1, patient=1, do=1, cross'd=1, where=2, lovers=1, fatal=1, stage=1, verona=1, new=1, bury=1, forth=1, strife=1, lay=1, fair=1, we=1, alike=1, could=1, piteous=1, is=1, hands=1, mend=1, in=2, nought=1, both=1, continuance=1, life=1, if=1, shall=2, the=5, traffic=1, and=1, a=1, toil=1, take=1, which=2, loins=1, of=5, here=1, end=1, what=1, civil=2, their=6, love=1, but=1, makes=1, miss=1, rage=1, foes=1, you=1, ears=1, whose=1, now=1, to=2, dignity=1, fearful=1, pair=1, star=1, strive=1, households=1, hours'=1, grudge=1, break=1, misadventured=1, mutiny=1, attend=1, overthrows=1, parents'=2, blood=1, from=2, children's=1, remove=1, death=2]
```

Entry Set (sorted alpha):

```
[a=1, alike=1, ancient=1, and=1, attend=1, blood=1, both=1, break=1, bury=1, but=1, children's=1, civil=2, continuance=1, could=1, cross'd=1, death=2, dignity=1, do=1, ears=1, end=1, fair=1, fatal=1, fearful=1, foes=1, forth=1, from=2, grudge=1, hands=1, here=1, hours'=1, households=1, if=1, in=2, is=1, lay=1, life=1, loins=1, love=1, lovers=1, makes=1, mark'd=1, mend=1, misadventured=1, miss=1, mutiny=1, new=1, nought=1, now=1, of=5, our=3, overthrows=1, pair=1, parents'=2, passage=1, patient=1, piteous=1, rage=1, remove=1, scene=1, shall=2, stage=1, star=1, strife=1, strive=1, take=1, the=5, their=6, these=1, to=2, toil=1, traffic=1, two=3, unclean=1, verona=1, we=1, what=1, where=2, which=2, whose=1, with=2, you=1]
```

Entry Set (sorted by freq):

```
[their=6, of=5, the=5, our=3, two=3, civil=2, death=2, from=2, in=2, parents'=2, shall=2, to=2, where=2, which=2, with=2, a=1, alike=1, ancient=1, and=1, attend=1, blood=1, both=1, break=1, bury=1, but=1, children's=1, continuance=1, could=1, cross'd=1, dignity=1, do=1, ears=1, end=1, fair=1, fatal=1, fearful=1, foes=1, forth=1, grudge=1, hands=1, here=1, hours'=1, households=1, if=1, is=1, lay=1, life=1, loins=1, love=1, lovers=1, makes=1, mark'd=1, mend=1, misadventured=1, miss=1, mutiny=1, new=1, nought=1, now=1, overthrows=1, pair=1, passage=1, patient=1, piteous=1, rage=1, remove=1, scene=1, stage=1, star=1, strife=1, strive=1, take=1, these=1, toil=1, traffic=1, unclean=1, verona=1, we=1, what=1, whose=1, you=1]
```



Collections in JDK 1.1

- Vector implements the List interface
- Stack is a subclass of Vector and supports the push, pop, and peek methods
- Hashtable implements the Map interface
- Enumeration is a variation on the Iterator interface
 - ▼ An enumeration is returned by the elements method in Vector, Stack, and Hashtable
- These classes are thread-safe, and therefore, "heavy-weight"

The Collections API

Collections in JDK 1.1

The Collections API is a Java 2 SDK feature, but there were a few collections classes in the JDK 1.0 and JDK 1.1. These classes still exist in the SDK with the same interface, but they have also 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 (reviewed in "System Properties" on page 9-4) is an extension of Hashtable that only uses Strings for keys and values. Each of these collections has an elements method which returns an enumeration object. Enumeration is an interface similar, but incompatible with the Iterator interface. For example, hasNext is replaced by hasMoreElements in the Enumeration interface.

All of these collection classes are thread-safe, which makes them a "heavy weight" implementation.

Exercise: Using Collections to Represent Aggregation



Exercise objective – Become familiar with collections and iterators by rewriting the Banking Project to use the Java 2 SDK Collections API instead of arrays.

Preparation

To successfully complete this lab, you must understand the concepts of a set, iteration, and sorting.

Tasks

In a Web browser view the `lab_files.html` page that is at the top-level of the SL275 directory on your computer. There will be a summary of each exercise and a link to a page that gives a detailed explanation of the exercise.

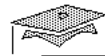
Go to the SL275 directory on your computer and change to the directory for this module (`mod09`). A listing of this directory will show six subdirectories. These two will be found in directories called `exercise3` and `exercise4`.

Exercise 3: Use Collections to Represent Multiplicity (Level 2 Lab)

In this exercise, you will replace the arrays code that you used to implement multiplicity in the relationships between bank and customer, and customer and their accounts.

Exercise 4: Sort Customers (Level 3 Lab)

In this exercise, you will sort the list of bank customers by their names. This will require you to modify the `Customer` class to implement the `Comparable` interface.



Using the javadoc Tool

- This Java 2 SDK tool generates HTML documentation pages
- Usage: `javadoc [options] [packages|files]`

This example generates the API documentation for the complete Banking project:

```
javadoc -d ../doc/api banking banking.domain /  
banking.reports
```

Option	Value	Description
-d	output path	The directory where the generated HTML files should be placed.
-sourcepath	directory path	The root directory where the source file package tree.
-public		Specifies that only public declarations be included. (<i>default</i>)
-private		Specifies that all declarations be included.

Using the javadoc Tool

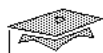
Documentation of your code is important to the success of future maintenance efforts for stand-alone applications and is critical to the use of APIs. In this section we will briefly describe the javadoc tool, comment tags, and how to use the tool.

Hopefully you already have some experience in using the Java 2 SDK documentation. Let's now look at how you can generate documentation HTML pages for your projects.

```
> javadoc -private -d ../doc/api banking 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 on-line documentation for more details.



Documentation Tags

- Comments starting with `/**` are parsed by the javadoc tool
- These comments should immediately precede the declaration they reflect

Tag	Purpose	class/ interface	constructor	method	attribute
@see	To create a link to another declaration (or any other HTML page)	✓	✓	✓	✓
@deprecated	Documents that the declaration has been deprecated in this release	✓	✓	✓	✓
@author	The author of the class or interface	✓			
@param	Documents a parameter		✓	✓	
@throws @exception	Documents why an exception might be thrown		✓	✓	
@return	Documents the return value/type			✓	

Using the javadoc Tool

Documentation Tags

The javadoc tool parses the specified source files for comment lines that start with `/**` and end with `*/`. These are used by the tool 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. Text following the summary sentence can be used 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, `` (etc) 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 above shows a set of the most common javadoc tags, their meaning, and with which declarations they may be used.

Using the javadoc Tool

Example

We will use the following example in our 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 }
```


Using the javadoc Tool

Example (Continued)

```
> javadoc -d doc/api/public DocExample.java
```

The screenshot shows a Netscape browser window titled "Netscape: Generated Documentation (Untitled)". The address bar shows the file path: "file:/home/basham/Courses/SL275/SL275_revD/SL275_SOL_LF/text". The browser displays the Javadoc output for the class `DocExample`.

All Classes
[DocExample](#)

[Class](#) [Tree](#) [Deprecated](#) [Index](#) [Help](#)

PREV CLASS NEXT CLASS
 SUMMARY: [DOER](#) | [FIELD](#) | [CONST](#) | [METHOD](#)

FRAMES NO FRAMES
 DETAIL: [FIELD](#) | [CONST](#) | [METHOD](#)

mypack
Class DocExample
 java.lang.Object
 |
 +--mypack.DocExample

public class **DocExample**
 extends java.lang.Object

This class contains a bunch of documentation tags.

Constructor Summary

[DocExample](#)(int x_value)
 This constructor initializes the x attribute.

Method Summary

java.util.List [getStuff](#)()
 This method return some stuff.

Methods inherited from class java.lang.Object
 clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

DocExample

public **DocExample**(int x_value)

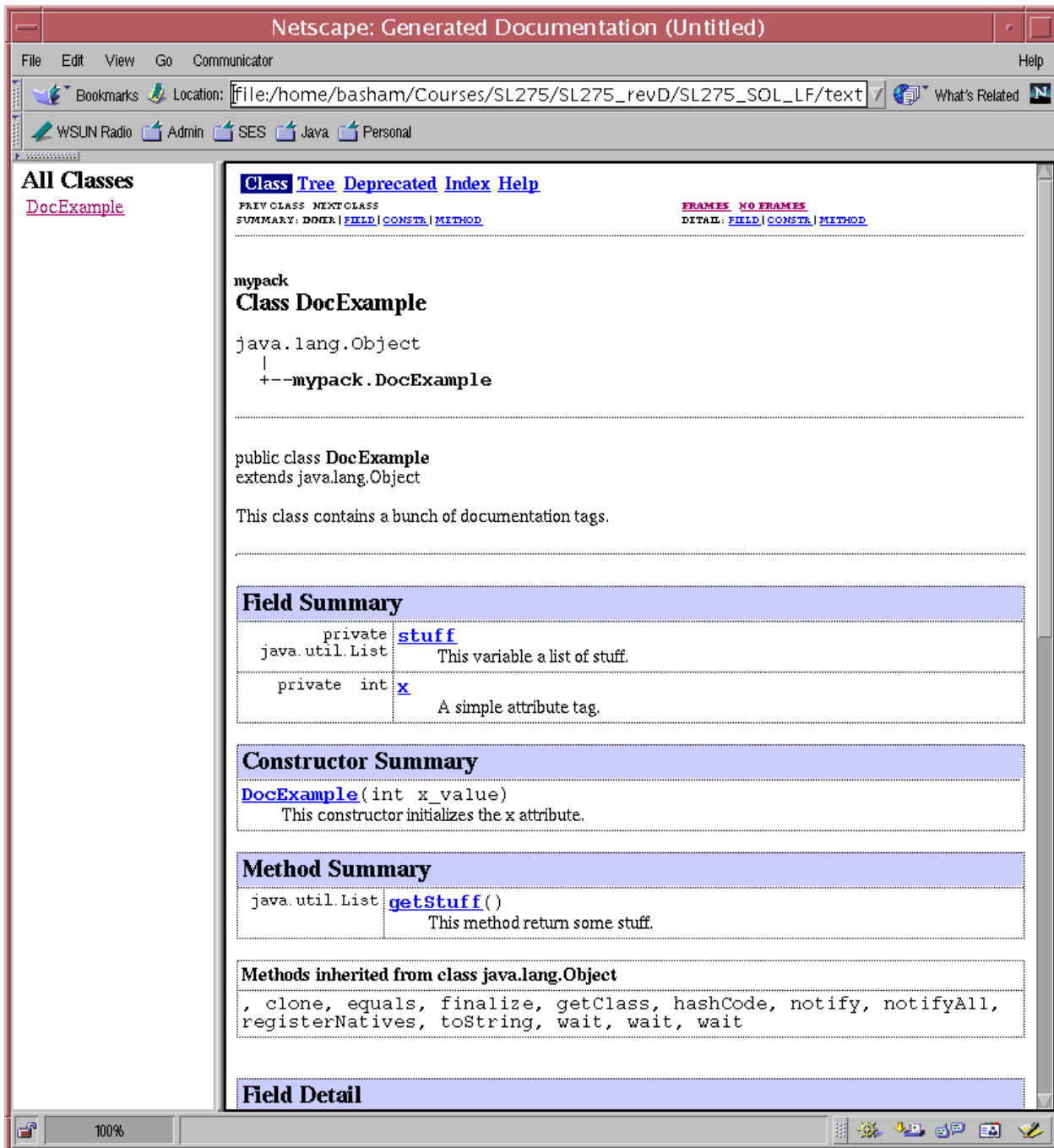
This constructor initializes the x attribute.

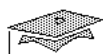
Parameters:
 x_value - the value of x

Using the javadoc Tool

Example (Continued)

```
> javadoc -private -d doc/api/private DocExample.java
```





Deprecation

- Deprecation makes classes, attributes, methods, constructors, and so on, obsolete
- Obsolete declarations are replaced by methods with a more standardized naming convention
- When migrating code, compile the code with the `-deprecation` flag:

```
javac -deprecation MyFile.java
```

Deprecation

In JDK 1.1, a significant effort was made to standardize the names of methods. As a result, in Java 2 SDK, a significant number of class constructors and method calls are obsolete. They have been replaced by method names that follow a more standardized naming convention and, in general, make life less complicated for the programmer.

For example, in the JDK 1.0 version of the `java.awt.Component` class:

- The methods for changing or getting the size of a component are `resize()` and `size()`.
- The methods for changing or getting the bounding box of a component are `reshape()` and `bounds()`.

Deprecation

In the JDK 1.1 version of `java.awt.Component` :

- Method names that begin with the words `set` and `get` indicate the primary operation of the method, respectively. For example:
 - ▼ `setSize()` and `getSize()`
 - ▼ `setBounds()` and `getBounds()`

Whenever you are moving code from JDK 1.0 to JDK 1.1 or higher, or even if you are using code that previously worked with JDK 1.0, you should compile the code with the `-deprecation` flag.

```
javac -deprecation MyFile.java
```

- ✓ *Sometimes whole classes are changed between JDK 1.0 and JDK 1.1. Unfortunately, these were not deprecated, so the compiler issues an error message stating that the class cannot be found. This is annoying, but unavoidable.*

Deprecation

The `-deprecation` flag reports any methods used within the class that are deprecated. For example, consider a utility class called `DateConverter`, which converts a date in the format `mm/dd/yy` to the day of the week.

```
1 package myutilities;
2
3 import java.util.*;
4 import java.text.*;
5
6 public final class DateConverter {
7     private static final String DAY_OF_THE_WEEK [] =
8         {"Sunday", "Monday", "Tuesday", "Wednesday",
9          "Thursday", "Friday", "Saturday"};
10
11     public static String getDayOfWeek (String theDate){
12         int month, day, year;
13
14         StringTokenizer st = new StringTokenizer (theDate, "/");
15
16         month = Integer.parseInt(st.nextToken ());
17         day = Integer.parseInt(st.nextToken());
18         year = Integer.parseInt(st.nextToken());
19         Date d = new Date (year, month, day);
20
21         return (DAY_OF_THE_WEEK[d.getDay()]);
22     }
23 }
```

When this code is compiled under Java 2 SDK with the `-deprecation` flag, you get the following:

```
javac -deprecation DateConverter.java
```

```
DateConverter.java:19: Note: The constructor java.util.Date(int,int,int)
has been deprecated.
```

```
    Date d = new Date (year, month, day);
                ^
```

```
DateConverter.java:21: Note: The method int getDay() in class
java.util.Date has been deprecated.
```

```
    return (day_of_the_week[d.getDay()]);
                ^
```

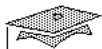
Note: `DateConverter.java` uses or overrides a deprecated API. Please consult the documentation for a better alternative. 1 warning

Deprecation

The rewritten `DateConverter` class looks like the following:

```
1 package myutilities;
2
3 import java.util.*;
4 import java.text.*;
5
6 public final class DateConverter {
7     private static String day_Of_The_Week[] =
8         {"Sunday", "Monday", "Tuesday", "Wednesday",
9          "Thursday", "Friday", "Saturday"};
10
11 public static String getDayOfWeek (String theDate) {
12     Date d = null;
13     SimpleDateFormat sdf = new SimpleDateFormat("MM/dd/yy");
14
15     try {
16         d = sdf.parse (theDate);
17     } catch (ParseException e) {
18         System.out.println (e);
19         e.printStackTrace();
20     }
21
22     // Create a GregorianCalendar object
23     Calendar c =
24         new GregorianCalendar(
25             TimeZone.getTimeZone("EST"), Locale.US);
26     c.setTime (d);
27
28     return(
29         day_Of_The_Week[(c.get(Calendar.DAY_OF_WEEK)-1)]);
30 }
31 }
```

Here the 1.2 version uses two new classes: `SimpleDateFormat`, a class used to take any `String` date format and create a `Date` object, and the `GregorianCalendar` class, used to create a calendar with the local time zone and locale.



Using the jar Tool

- This Java 2 SDK tool generates a compressed archive of `.class` and media files
- Usage: `jar [options][archive_file] [files]`

This generates an archive for the Banking project:

```
jar cvf banking.jar banking/domain/*.class banking/reports/*.class
```

This extracts an archive for the Banking project:

```
jar xvf banking.jar
```

Option	Value	Description
c		This option creates a new archive.
f	filepath	This option specifies the filepath of the JAR (Java archive) file.
x		This option extracts an archive to the current directory.
v		This option specifies verbose output from the jar tool.

Using the jar Tool

✓ **JAR stands for Java ARchive.**

All applications and frameworks need to be deployed. Because Java technology programs are not linked into an independent executable (or shared library), Java 2 SDK supports a means to combine a set of files (usually `.class` files) into a single archive file (called a JAR file) that can be delivered using a floppy, file transfer, and even downloaded from a Web site. The `jar` tool is used to create (and extract, if necessary) these archive files.

The `jar` tool is a general purpose tool that can archive more than just class files; it can include HTML pages, media files (such as images, sounds, and video), and even text files. Not only does it create an archive of multiple files, but it also compresses these files as they are loaded into the JAR.

As mentioned in "Directory Layout and Packages" on page 2-28, a JAR file can be used in your CLASSPATH or stored in the JRE library directory.

Exercise: Building a System



Exercise objective – You will gain experience using the javadoc and jar tools to build a complete system.

Preparation

To successfully complete this lab, you must understand the concepts of using javadoc comment tags, and running the javadoc and jar tools.

Tasks

In a Web browser view the `lab_files.html` page that is at the top-level of the SL275 directory on your computer. There will be a summary of each exercise and a link to a page that gives a detailed explanation of the exercise.

Go to the SL275 directory on your computer and change to the directory for this module (`mod09`). A listing of this directory will show six subdirectories. These two will be found in the directories called `exercise5` and `exercise6`.

Exercise 5: Document the Customer Class (Level 2 Lab)

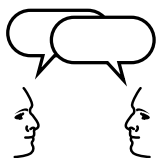
In this exercise you will use javadoc comment tags to document the customer class. You will then generate the API documentation for the entire banking package.

Exercise 6: Build an Archive of the Bank Project (Level 3 Lab)

In this exercise you will create a JAR file of the Banking project. You will then run the reports program using only the JAR file.

Exercise: Building Text-Based Applications

Exercise Summary



Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercises.

✓ **If you do not have time to spend on discussion, just highlight the key concepts students should have learned from the lab exercise.**

- Experiences

✓ **Ask students what their overall experiences with this exercise have been. You might want to go over any trouble spots or especially confusing areas at this time.**

- Interpretations

✓ **Ask students to interpret what they observed during any aspects of this exercise.**

- Conclusions

✓ **Have students articulate any conclusions they have reached as a result of this exercise experience.**

- Applications

✓ **Explore with the students how they might apply what they learned in this exercise to situations at their workplace.**

Check Your Progress

Before continuing on to the next module, check to be sure that you can:

- Write a program that uses command-line arguments and system properties
- Write a program that reads from *standard input*
- Write a program that can create, read, and write files
- Describe the basic hierarchy of collections in Java 2 SDK
- Write a program that uses sets and lists
- Write a program to iterate over a collection
- Write a program to sort an array or a list
- Describe the collection classes that existed before Java 2 SDK
- Describe and use the `javadoc` and `jar` tools
- Identify deprecated classes and explain how to migrate from JDK 1.0 to JDK 1.1 to Java 2 JDK

Think Beyond

Many applications are text-based. What other styles of programs exist?

What features does the Java application environment have that support user interface development?

How were interfaces used in this module? Could they have been replaced by some other mechanism, such as abstract classes?

Objectives

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

- Describe the Abstract Windowing Toolkit (AWT) package and its components
- Define the terms *containers*, *components*, and *layout managers*, and describe how they work together to build a graphical user interface (GUI)
- Use layout managers
- Use the `FlowLayout`, `BorderLayout`, `GridLayout`, and `CardLayout` managers to achieve a desired dynamic layout
- Add components to a container
- Use the `Frame` and `Panel` containers appropriately
- Describe how complex layouts with nested containers work
- In a Java technology program, identify the following:
 - ▼ Containers
 - ▼ The associated layout managers
 - ▼ The layout hierarchy of all components

This module covers the setup and layout of graphical user interfaces. It introduces the AWT, a package of classes from which GUIs are built.

Relevance

- ✓ **Present the following question to stimulate the students and get them thinking about the issues and topics presented in this module. They are not expected to know the answer to this question. Hold discussions where students have input; otherwise, if no one can propose answers, begin the lecture for this module.**



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

- As a platform-independent programming language, how is Java technology used to make the GUI platform independent?
- ✓ **Most interactive user input and output is done through a GUI. The Java platform provides a number of classes in the AWT package to make programming and setup of GUIs very easy. The AWT contains classes of `Components` from which GUIs are pieced together, and contains other classes which deal with other aspects, such as color, font, and layout of components. This module explains how GUIs are put together and formatted, and describes some of the `Components` from which GUIs are built.**



Abstract Window Toolkit (AWT)

- Provides graphical user interface (GUI) components that are used in all Java applets and applications
- Contains classes that can be extended and their properties inherited; classes can also be abstract
- Ensures that every GUI component that is displayed on the screen is a subclass of the abstract class `Component` or `MenuComponent`
- Has `Container`, which is an abstract subclass of `Component` and includes two subclasses:
 - ▼ `Panel`
 - ▼ `Window`

The AWT

The AWT provides basic GUI components that are used in Java applets and applications. The AWT provides a machine-independent interface for applications. This ensures that what appears on one computer is comparable to what appears on another.

Before looking at the AWT, briefly review the object hierarchy. Remember that super classes can be extended and their properties inherited. Also, classes can be abstract, meaning that they are templates that are subclassed to provide concrete implementations of the class.

Every GUI component that appears on the screen is a subclass of the abstract class `Component` or `MenuComponent`. That is, basic GUI components inherit from the `Component` class a number of methods and instance variables. Likewise, menu components inherit from the `MenuComponent` class.

The AWT

Container is an abstract subclass of Component, which allows other components to be nested inside it. These components can also be containers allowing other components to be nested inside, which creates a complete hierarchical structure. Containers are helpful in arranging GUI components on the screen. A Panel is the simplest concrete subclass of Container. Another subclass of Container is a Window.

The java.awt Package

The java.awt package contains classes that generate GUI components. A basic overview of this package is shown in Figure 10-1. The classes shown in bold type highlight the main focus of this module.

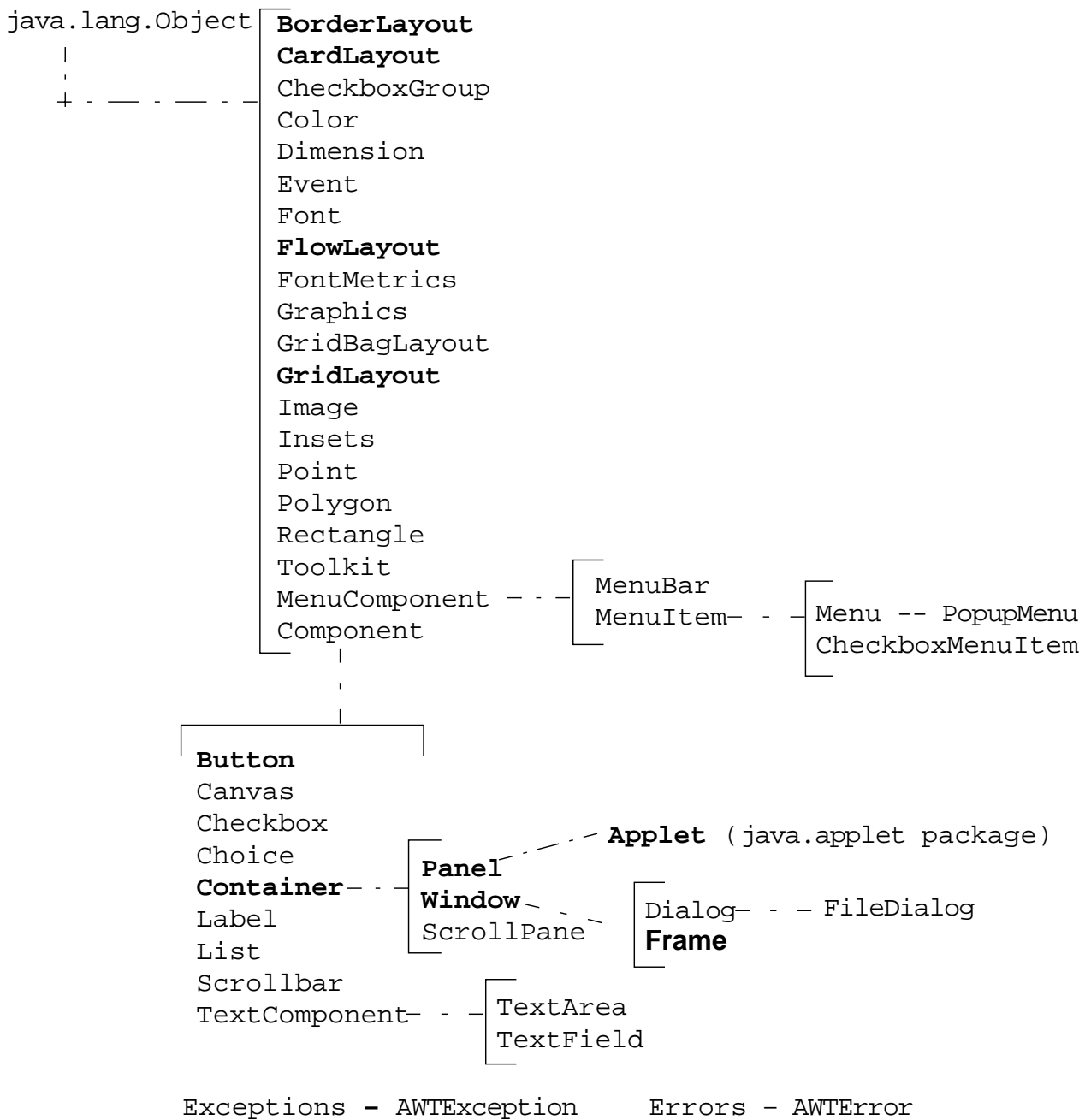


Figure 10-1 java.awt Package



Containers

- Add components with the `add()` method
- The two main types of containers are `Window` and `Panel`
- A `Window` is a free floating window on the display
- A `Panel` is a container of GUI components that must exist in the context of some other container, such as a window or applet

Building Graphical User Interfaces

Containers

GUI components are added to a container using the `add` method.

There are two main types of containers: `Window` and `Panel`.

A `Window` is a free-standing native window on the display that is independent of other containers. There are two important types of `Window`: `Frame` and `Dialog`. `Frame` is a window with a title and resizing corners. `Dialog` does not have a menu bar. Although you can move it, you cannot resize it.

Building Graphical User Interfaces

Containers (Continued)

A `Panel` is contained within another `Container`, or inside a Web browser's window. `Panel` identifies a rectangular area into which you can place other components. You must place `Panel` into a `Window` (or a subclass of `Window`) to be displayed.

Note – The fact that a container can hold not only components, but also other containers, is critical and fundamental to building complex layouts.

`ScrollPane` is also a subclass of `Container`. It is discussed in Appendix C, "The AWT Component Library."



Building Graphical User Interfaces

- The position and size of a component in a container is determined by a layout manager.
- You can control the size or position of components by disabling the layout manager.

You must then use `setLocation()`, `setSize()`, or `setBounds()` on components to locate them in the container.

Building Graphical User Interfaces

Positioning Components

The position and size of a component in a container is determined by a layout manager. A container keeps a reference to a particular instance of a layout manager. When the container needs to position a component, it invokes the layout manager to do so. The same delegation occurs when deciding on the size of a component. The layout manager takes full control over all of the components within the container. It is responsible for computing and defining the preferred size of the object in the context of the actual screen size.

- ✓ ***The preferred size expresses how big a component will be displayed. For example, the preferred size of a button is the size of the label text plus the border space and the shadowed decorations that mark the boundary of the button. The preferred size is platform dependent.***

Building Graphical User Interfaces

Component Sizing

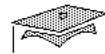
Because the layout manager generally is responsible for the size and position of components on its container, you should not normally attempt to set the size or position of components yourself. If you try to do so (using any of the methods `setLocation`, `setSize` or `setBounds`), the layout manager can override your decision.

If you must control the size or position of components in a way that cannot be done using the standard layout managers, you can disable the layout manager by issuing the following method call to your container:

```
cont.setLayout(null);
```

After this step, you must use `setLocation`, `setSize` or `setBounds` on all components to locate them in the container.

This approach results in platform-dependent layouts due to the differences between window systems and font sizes. A better approach is to create a new subclass of `LayoutManager`.



Frames

- Are a subclass of `Window`
- Have title and resizing corners
- Inherit from `Container` and add components with the `add()` method
- Are initially invisible, use `setVisible(true)` to expose the frame
- Have `BorderLayout` as the default layout manager
- Use the `setLayout` method to change the default layout manager

Frames

Frame is a subclass of `Window`. It is a `Window` with a title and resizing corners. Frame inherits from `Container` so you can add components to a Frame using the `add` method. The default layout manager for a Frame is `BorderLayout`. You can change this using the `setLayout` method.

The constructor `Frame(String)` in the `Frame` class creates a new, invisible `Frame` object with the title specified by `String`. You add all the components to the `Frame` while it is still invisible.

- ✓ ***The code on the next page (and all of the GUI examples in this course) has been rewritten with Sun Educational Services' new "GUI Coding Guidelines" which is currently in draft. Please be aware of two things. First, make sure that students understand that these GUI examples demonstrate one possible style of coding; it is not a standard and not the final word. Second, this style was created based on a lot of feedback accumulated through the ses_java e-mail list. The goal was to make our GUI coding examples more object-oriented and flexible.***

Frames

The following program creates a Frame that has a specific title, size, and background color (Figure 10-2).

```
1 import java.awt.*;
2
3 public class FrameExample {
4     private Frame f;
5
6     public FrameExample() {
7         f = new Frame("Hello Out There!");
8     }
9
10    public void launchFrame() {
11        f.setSize(170,170);
12        f.setBackground(Color.blue);
13        f.setVisible(true);
14    }
15
16    public static void main(String args[]) {
17        FrameExample guiWindow = new FrameExample();
18        guiWindow.launchFrame();
19    }
20 }
```

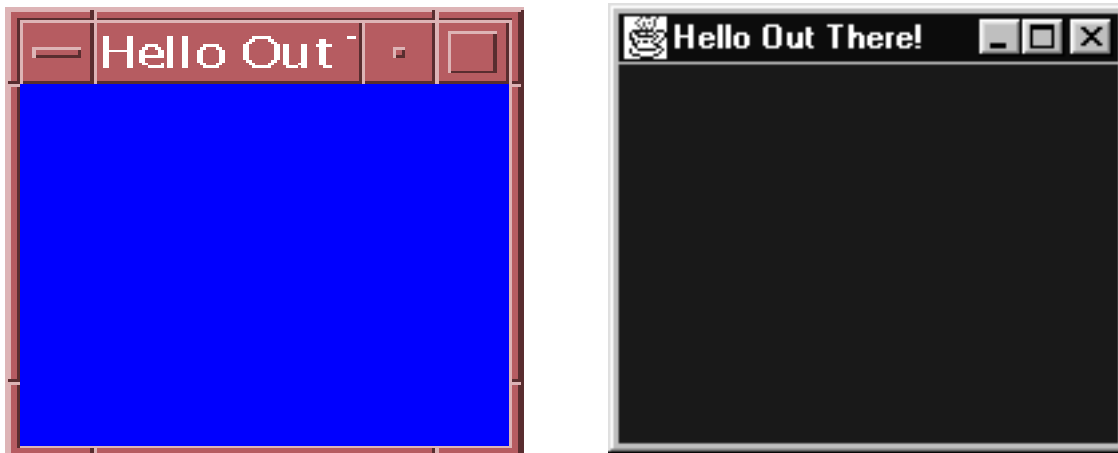
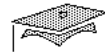


Figure 10-2 Example Frame

Note – A Frame must be made visible (using a call to `setVisible(true)`) and its size defined (using a call to `setSize` or `pack`) before it is displayed on screen.



Panels

- Provide a space for components
- Allow subpanels to have their own layout manager

Panels

`Panel`, like `Frame`, provides the space for you to attach any GUI component, including other panels. Each `Panel`, which inherits from `Container`, can have its own layout manager.

Once a `Panel` is created, you must add it to a `Window` or `Frame` to be visible. This is done using the `add` method of the `Container` class.


The following program creates a small yellow `Panel`, and adds it to a `Frame` (Figure 10-3).

Panels

```
1 import java.awt.*;
2
3 public class FrameWithPanel {
4     private Frame f;
5     private Panel pan;
6
7     public FrameWithPanel(String title) {
8         f = new Frame(title);
9         pan = new Panel();
10    }
11
12    public void launchFrame() {
13        f.setSize(200,200);
14        f.setBackground(Color.blue);
15        f.setLayout(null); // Override default layout mgr
16
17        pan.setSize(100,100);
18        pan.setBackground(Color.yellow);
19        f.add(pan);
20        f.setVisible(true);
21    }
22
23    public static void main(String args[]) {
24        FrameWithPanel guiWindow =
25            new FrameWithPanel("Frame with Panel");
26        guiWindow.launchFrame();
27    }
28 }
```



Figure 10-3 Example Panel

 *Sun Educational Services*

Container Layouts

- FlowLayout
- BorderLayout
- GridLayout
- CardLayout
- GridBagLayout

Container Layouts

The layout of components in a container is usually governed by a layout manager. Each container (such as `Panel` or `Frame`) has a default layout manager associated with it, which can be changed by calling `setLayout`.

The layout manager is responsible for deciding the layout policy and size of each of its container's child components.

- ✓ ***The layout manager gives first preference to the layout policy. If honoring a component's preferred size means violating the layout policy, the layout manager overrules the component's preferred size.***

Container Layouts

Layout Managers

The following layout managers are included with the Java programming language:

- `FlowLayout` – The default layout manager of `Panel` and `Applet`
- `BorderLayout` – The default layout manager of `Window`, `Dialog`, and `Frame`
- `GridLayout`
- `CardLayout`
- `GridBagLayout`

Note – The `GridBagLayout` manager is not discussed in-depth in this module. For details about `GridBagLayout`, see Appendix D, “Using the `GridBagLayout`.”

Container Layouts

Default Layout Managers

Figure 10-4 illustrates the default Layout Managers.

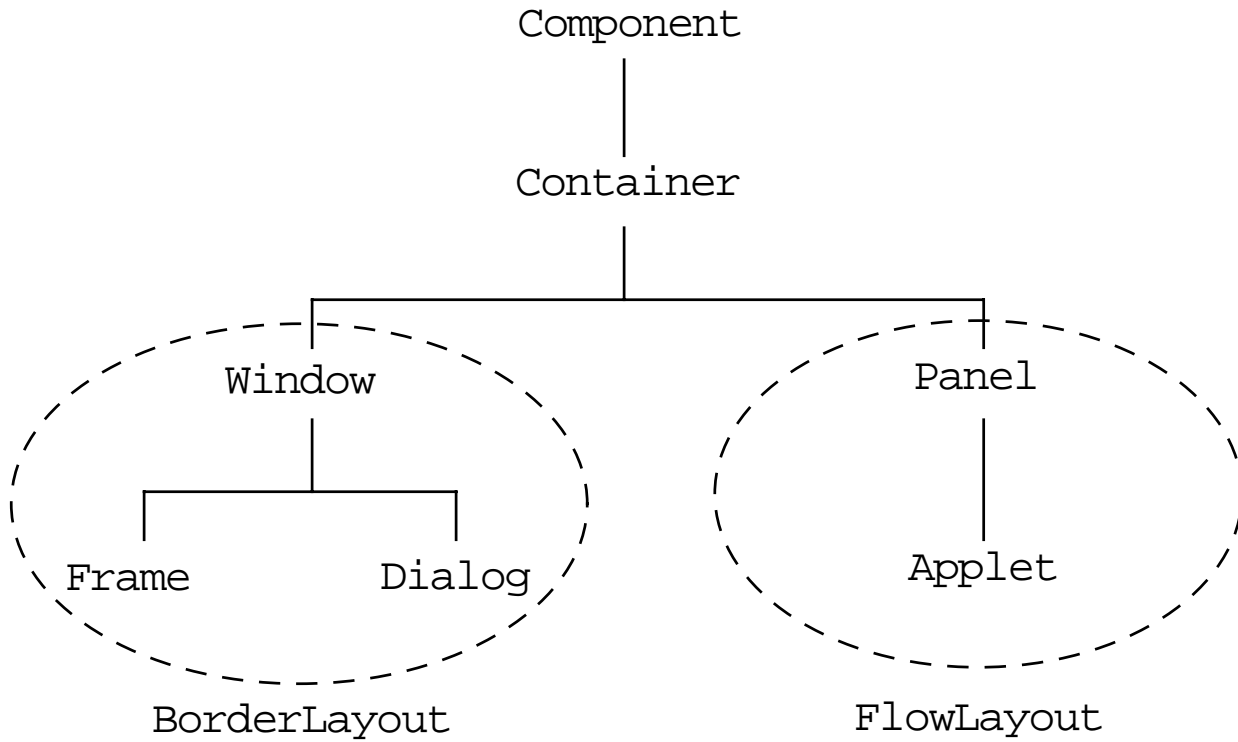


Figure 10-4 Layout Managers

A Simple `FlowLayout` Example

The following sample code demonstrates several important points, which are discussed in the following sections.

```
1 import java.awt.*;
2
3 public class LayoutExample {
4     private Frame f;
5     private Button b1;
6     private Button b2;
7
8     public LayoutExample() {
9         f = new Frame("GUI example");
10        b1 = new Button("Press Me");
11        b2 = new Button("Don't press Me");
12    }
13
14    public void launchFrame() {
15        f.setLayout(new FlowLayout());
16        f.add(b1);
17        f.add(b2);
18        f.pack();
19        f.setVisible(true);
20    }
21
22    public static void main(String args[]) {
23        LayoutExample guiWindow = new LayoutExample();
24        guiWindow.launchFrame();
25    }
26 }
```

This code creates the following:

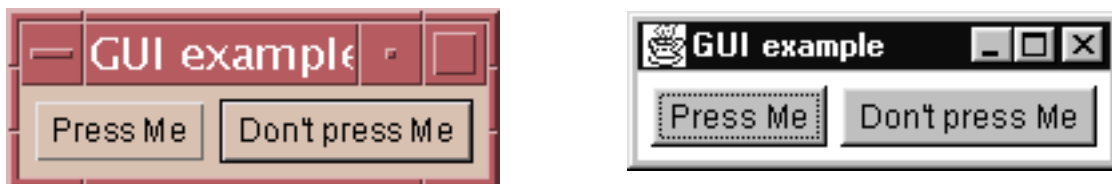


Figure 10-5 Example of `FlowLayout`

A Simple FlowLayout Example

The main Method

The main method in line 23 of this example performs two actions. First, it creates an instance of the `LayoutExample` object. Recall that until an instance exists, there are no real data items called `f`, `b1`, and `b2` for use. The constructor (lines 8-12) creates three GUI objects:

```
f = new Frame( "GUI Example" )
```

This constructor creates an instance of the class `Frame`. `Frame` in the Java programming language is a top-level window, with a title bar that is defined by the constructor argument; "GUI Example" in this case, resize handles, and other decorations, according to local conventions. `Frame` has a 0x0 size and currently is not visible.

```
b1 = new Button( "Press Me" )  
b2 = new Button( "Don't Press Me" )
```

These constructors create instances of the class `Button`. A button is the standard pushbutton taken from the local window toolkit. The button label is defined by the string argument to the constructor.

Second, when the data space has been created, `main` calls the instance method `launchFrame` in the context of that instance. In `launchFrame`, the real action occurs.

```
f.setLayout( new FlowLayout( ) )
```

This method creates an instance of the flow layout manager and installs it in the `Frame`. The default layout manager for every frame, `BorderLayout`, is not used for this example. The `FlowLayout` manager is the simplest in the AWT and positions components somewhat like words on a page, line by line. The `FlowLayout` centers each line by default.

A Simple FlowLayout Example

The main Method (Continued)

```
f.add(b1)
f.add(b2)
```

These method calls tell `Frame f` (which is a container) that it is to contain the components `b1` and `b2`. The size and position of `b1` and `b2` are under the control of the frame's layout manager from this point onward.

```
f.pack()
```

This method tells the frame to set a size that “neatly encloses” the components that it contains. To determine what size to use for the `Frame`, `f.pack()` queries the layout manager, which is responsible for the size and position of all the components within the `Frame`.

```
f.setVisible(true)
```

This method causes the `Frame`, and all its contents, to become visible to the user.



FlowLayout Manager

- Default layout for the `Panel` class
- Components added from left to right
- Default alignment is centered
- Uses components' preferred sizes
- Uses the constructor to tune behavior

Layout Managers

FlowLayout Manager

The `FlowLayout` used in the example for this topic positions components on a line-by-line basis. Each time a line is filled, a new line is started.

Unlike other layout managers, the `FlowLayout` manager does not constrain the size of the components it manages, but instead allows them to have their preferred size.

`FlowLayout` constructor arguments allow you to flush the components to the left or to the right if you prefer that to the default centering behavior.

You can specify gaps if you want to create a larger minimum space between the components.

Layout Managers

FlowLayout Manager (Continued)

When the area being managed by a FlowLayout is resized by the user, the layout can change. See Figure 10-6.

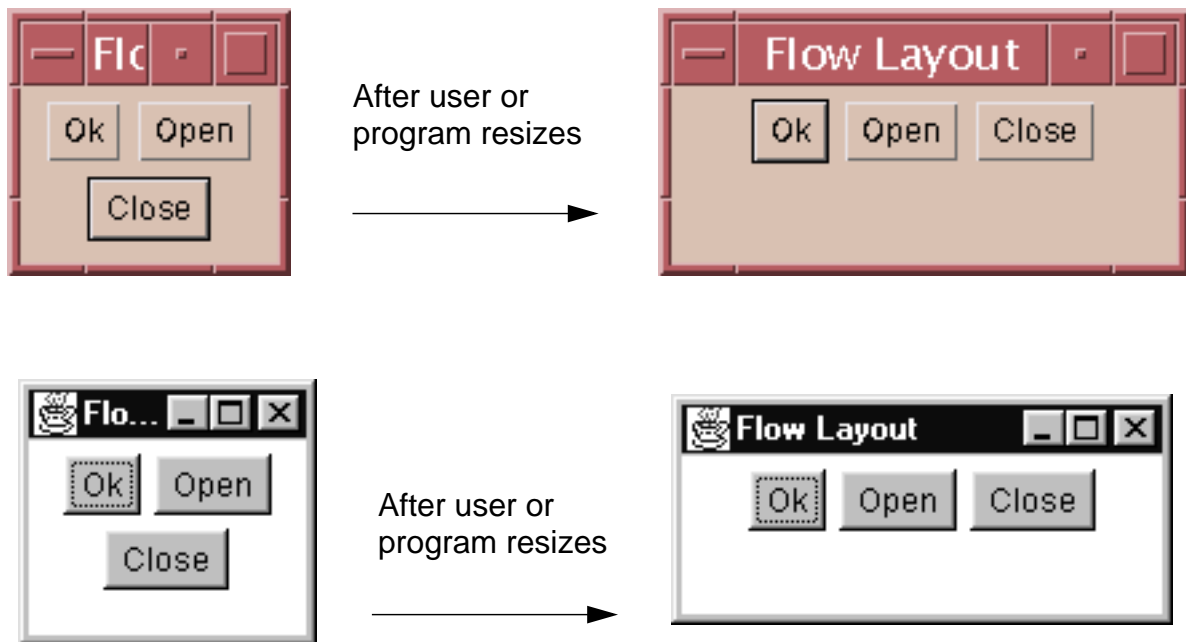


Figure 10-6 FlowLayout Resizing

Layout Managers

FlowLayout Manager (Continued)

The following are examples of how to create `FlowLayout` objects and install them using the `setLayout` method of class `Container`:

```
setLayout(new FlowLayout(  
            int align,int hgap,int vgap));
```

The value of `align` must be `FlowLayout.LEFT`, `FlowLayout.RIGHT`, or `FlowLayout.CENTER`. For example:

```
setLayout(new FlowLayout(FlowLayout.RIGHT, 20, 40));
```

The following constructs and installs a new `FlowLayout` with the specified alignment and a default five-unit horizontal and vertical gap.

```
setLayout(new FlowLayout(int align);  
setLayout(new FlowLayout(FlowLayout.LEFT));
```

The following constructs and installs a new `FlowLayout` with centered alignment and a default five-unit horizontal and vertical gap:

```
setLayout(new FlowLayout());
```

Layout Managers

FlowLayout Manager (Continued)

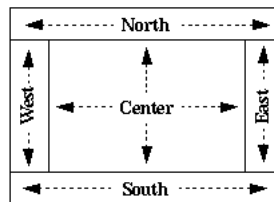
The following sample code adds several buttons to a flow layout on a Frame:

```
1 import java.awt.*;
2
3 public class FlowExample {
4     private Frame f;
5     private Button button1;
6     private Button button2;
7     private Button button3;
8
9     public FlowExample() {
10        f = new Frame("Flow Layout");
11        button1 = new Button("Ok");
12        button2 = new Button("Open");
13        button3 = new Button("Close");
14    }
15
16    public void launchFrame() {
17        f.setLayout(new FlowLayout());
18        f.add(button1);
19        f.add(button2);
20        f.add(button3);
21        f.setSize(100,100);
22        f.setVisible(true);
23    }
24
25    public static void main(String args[]) {
26        FlowExample guiWindow = new FlowExample();
27        guiWindow.launchFrame();
28    }
29 }
```



BorderLayout Manager

- Default layout for the `Frame` class
- Components added to specific regions
- The resizing behavior:
 - ▼ North, South, and Center regions adjust horizontally
 - ▼ East, West, and Center regions adjust vertically



Layout Managers

BorderLayout Manager

The `BorderLayout` manager provides a more complex scheme for placing your components within a container. It is the default layout manager for `Frame` and `Dialog`. The `BorderLayout` manager contains five distinct areas: `NORTH`, `SOUTH`, `EAST`, `WEST`, and `CENTER`, indicated by `BorderLayout.NORTH`, and so on.

`NORTH` occupies the top of a frame, `EAST` occupies the right side, and so on. The `CENTER` area represents everything left over once the `NORTH`, `SOUTH`, `EAST`, and `WEST` areas are filled. When the window is stretched vertically, the `EAST`, `WEST`, and `CENTER` regions are stretched, whereas when the window is stretched horizontally, the `NORTH`, `SOUTH`, and `CENTER` regions are stretched.

Note – The relative positions of the buttons do not change as the window resizes, but the sizes of the buttons do change.

Layout Managers

BorderLayout *Manager (Continued)*

The following line:

```
setLayout(new BorderLayout());
```

constructs and installs a new BorderLayout with no gaps between the components.

The following line:

```
setLayout(new BorderLayout(int hgap, int vgap);
```

constructs and installs a BorderLayout with the gaps between components as specified by hgap and vgap.

You must add components to named regions in the BorderLayout manager; otherwise, they will not be visible. Spell the region names correctly; especially if you choose not to use constants (for instance, `add(button, "Center")` instead of `add(button, BorderLayout.CENTER)`). Spelling and capitalization are crucial.

You can use a BorderLayout manager to produce layouts with elements that stretch in one direction, the opposite direction, or both, when resized.

If you leave a region of a BorderLayout unused, it behaves as if its preferred size is zero by zero. The CENTER region still appears as background even if it contains no components.

Layout Managers

BorderLayout Manager (Continued)

You can add only a single component to each of the five regions of the `BorderLayout` manager. If you try to add more than one, only the last one added is visible. A later example shows how you can use intermediate containers to allow more than one component to be laid out in the space of a single `BorderLayout` manager region.

Note – The layout manager honors the preferred height of the `NORTH` and `SOUTH` components, but forces them to be as wide as the container. In the case of the `EAST` and `WEST` components, the preferred width is honored and the height is constrained.

Layout Managers

BorderLayout *Manager (Continued)*

The following code is a modification of the previous example and demonstrates the behavior of the BorderLayout manager. You can set the layout to use BorderLayout by using the `setLayout` method inherited from the class `Container`.

```
1 import java.awt.*;
2
3 public class BorderExample {
4     private Frame f;
5     private Button bn, bs, bw, be, bc;
6
7     public BorderExample() {
8         f = new Frame("Border Layout");
9         bn = new Button("B1");
10        bs = new Button("B2");
11        bw = new Button("B3");
12        be = new Button("B4");
13        bc = new Button("B5");
14    }
15
16    public void launchFrame() {
17        f.add(bn, BorderLayout.NORTH);
18        f.add(bs, BorderLayout.SOUTH);
19        f.add(bw, BorderLayout.WEST);
20        f.add(be, BorderLayout.EAST);
21        f.add(bc, BorderLayout.CENTER);
22        f.setSize(200,200);
23        f.setVisible(true);
24    }
25
26    public static void main(String args[]) {
27        BorderExample guiWindow2 = new BorderExample();
28        guiWindow2.launchFrame();
29    }
30 }
```

Layout Managers

BorderLayout Manager (Continued)

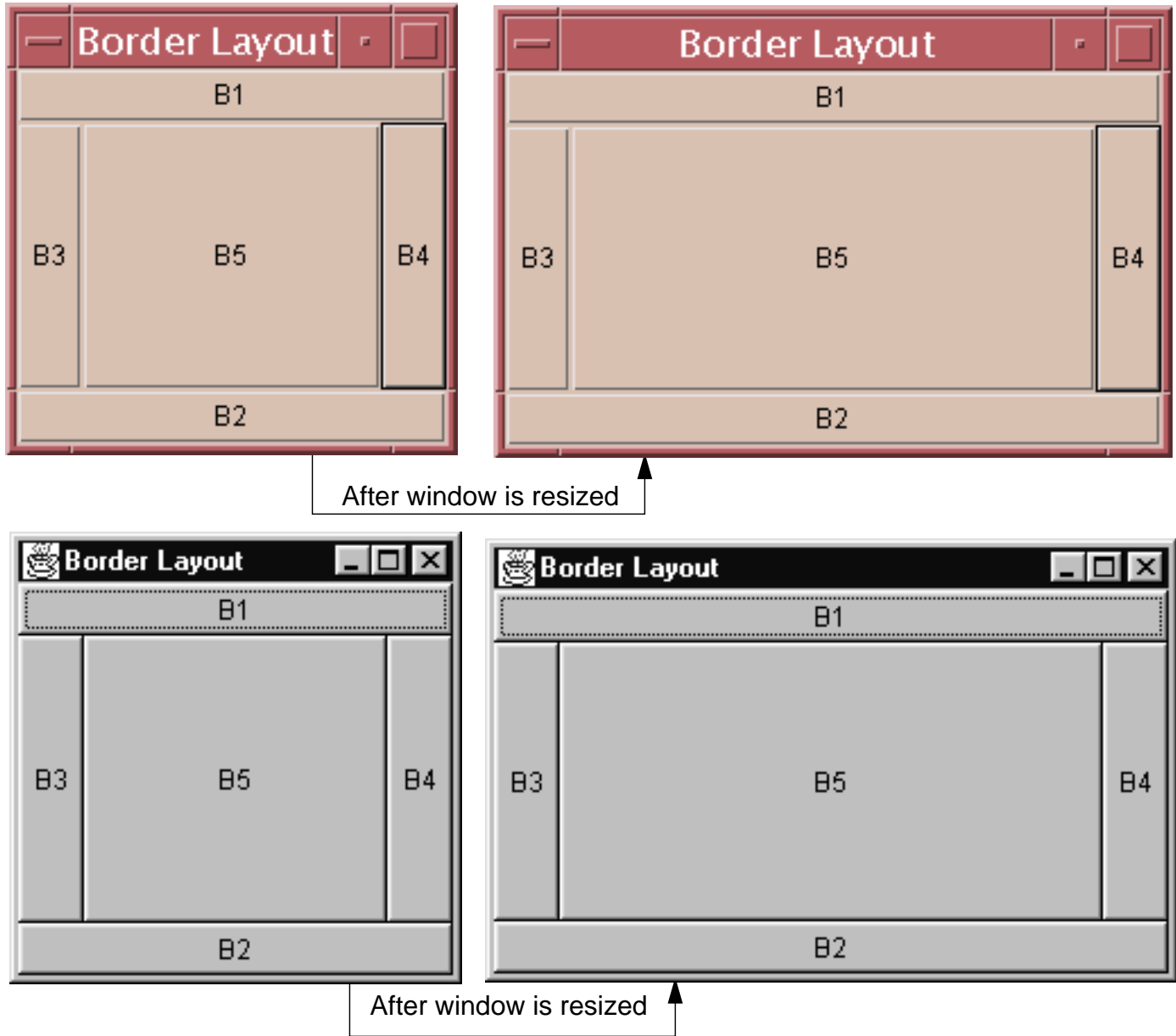
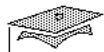


Figure 10-7 Example of BorderLayout

- ✓ BorderLayout **regions can also be spelled out (such as, `f.add(bn, "North")`). However, runtime exceptions can occur for misspellings (including capitalization mistakes). Using predefined constants such as `borderLayout.NORTH` is preferred, because errors will be caught at compile time instead of runtime.**



GridLayout Manager

- Components are added left to right, top to bottom
- All regions are equally sized
- The constructor specifies the rows and columns

Layout Managers

GridLayout Manager

The GridLayout manager provides flexibility for placing components. You create the manager with a number of rows and columns. Components then fill up the cells defined by the manager. For example, a GridLayout with three rows and two columns created by the statement `new GridLayout(3, 2)` would create six cells.

As with the BorderLayout manager, the relative position of components does not change as the area is resized. Only the sizes of the components change.

The GridLayout manager always ignores the component's preferred size. The width of all cells is identical and is determined by dividing the available width by the number of columns. Similarly, the height of all cells is determined by dividing the available height by the number of rows.

Layout Managers

GridLayout *Manager (Continued)*

The order in which components are added to the grid determines the cell that they occupy. Lines of cells are filled left to right, like text, and the “page” is filled with lines from top to bottom.

The following line:

```
setLayout(new GridLayout());
```

creates and installs a `GridLayout` with a default of one column per component in a single row.

The following line:

```
setLayout(new GridLayout(int rows, int cols);
```

creates and installs a grid layout with the specified number of rows and columns. All components in the layout are given equal size.

The following lines:

```
setLayout(new GridLayout(  
    int rows, int cols, int hgap, int vgap);
```

create and install a `GridLayout` with the specified number of rows and columns. All components in the layout are given equal size. `hgap` and `vgap` specify the respective gaps between components. Horizontal gaps are placed between each of the columns. Vertical gaps are placed between each of the rows.

Note – One, but not both, of the rows and columns can be zero. The zero value parameter is treated as a flexible guideline. For example, zero rows and two columns mean that there are always two columns, but as few or many rows as are needed to hold all of the added components.

Layout Managers

GridLayout Manager (Continued)

The following code produces the objects displayed in Figure 10-8:

```
1 import java.awt.*;
2
3 public class GridExample {
4     private Frame f;
5     private Button b1, b2, b3, b4, b5, b6;
6
7     public GridExample() {
8         f = new Frame("Grid Example");
9         b1 = new Button("1");
10        b2 = new Button("2");
11        b3 = new Button("3");
12        b4 = new Button("4");
13        b5 = new Button("5");
14        b6 = new Button("6");
15    }
16
17    public void launchFrame() {
18        f.setLayout (new GridLayout(3,2));
19
20        f.add(b1);
21        f.add(b2);
22        f.add(b3);
23        f.add(b4);
24        f.add(b5);
25        f.add(b6);
26
27        f.pack();
28        f.setVisible(true);
29    }
30
31    public static void main(String args[]) {
32        GridExample grid = new GridExample();
33        grid.launchFrame();
34    }
35 }
```

Layout Managers

GridLayout Manager (Continued)

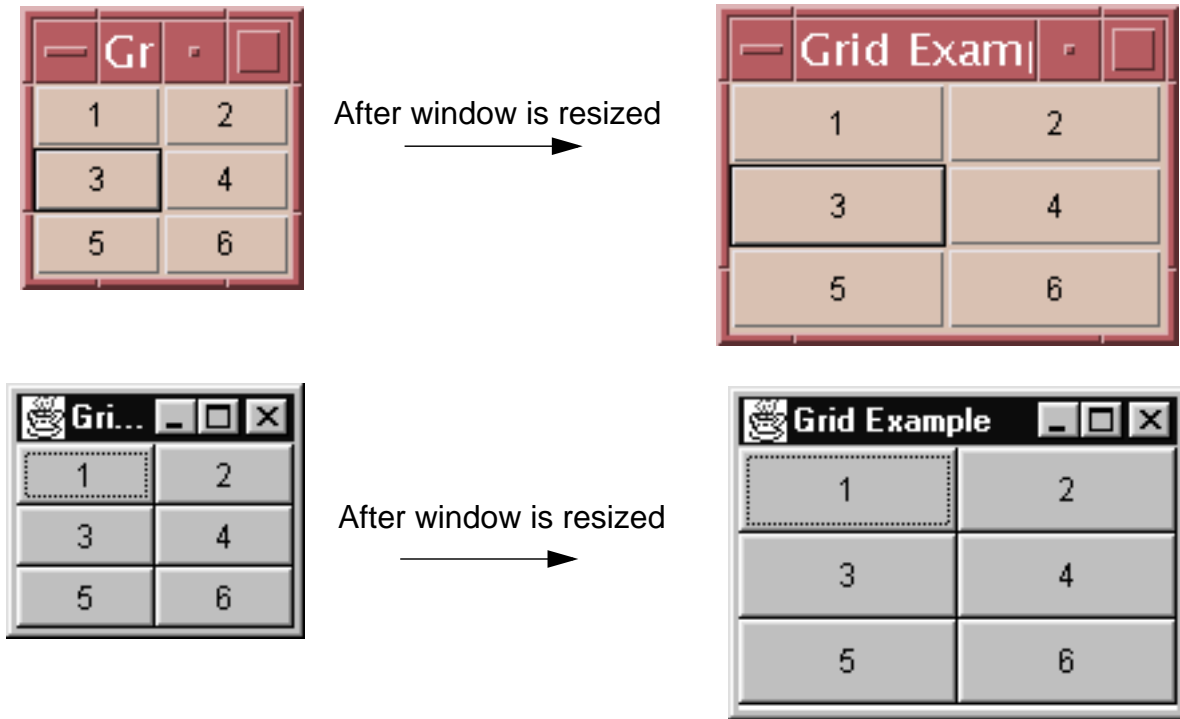


Figure 10-8 Example of GridLayout

Layout Managers

CardLayout Manager

The CardLayout manager enables you to treat the interface as a series of cards, only one of which can be viewed at any time. You can use the add method to add cards to a CardLayout. The add method takes a String as an argument and identifies the Panel in the program. The CardLayout manager's show method switches to a new card on request. The example for this topic demonstrates a single Frame that shows five different Panels with each mouse click. A mouse click in one Panel switches the view to the other Panel, and so on.

Note – This example requires a knowledge of event handling, which is covered in Module 11, "GUI Event Handling."

```
1 import java.awt.*;
2 import java.awt.event.*;
3
4 public class CardExample implements MouseListener {
5     private Panel p1, p2, p3, p4, p5;
6     private Label lb1, lb2, lb3, lb4, lb5;
7
8     // Declare a CardLayout object to call its methods.
9     private CardLayout myCard;
10    private Frame f;
11
12    public CardExample() {
13        f = new Frame ("Card Test");
14        myCard = new CardLayout();
15
16        // Create the panels that I want to use as cards.
17        p1 = new Panel();
18        p2 = new Panel();
19        p3 = new Panel();
20        p4 = new Panel();
21        p5 = new Panel();
22
23        // Create a label to attach to each panel
24        lb1 = new Label("This is the first Panel");
25        lb2 = new Label("This is the second Panel");
26        lb3 = new Label("This is the third Panel");
27        lb4 = new Label("This is the fourth Panel");
28        lb5 = new Label("This is the fifth Panel");
29    }
```

Layout Managers

CardLayout Manager (Continued)

```
30
31 public void launchFrame() {
32     f.setLayout(myCard);
33
34     // change the color of each panel, so they are
35     // easily distinguishable
36     p1.setBackground(Color.yellow);
37     p1.add(lb1);
38     p2.setBackground(Color.green);
39     p2.add(lb2);
40     p3.setBackground(Color.magenta);
41     p3.add(lb3);
42     p4.setBackground(Color.white);
43     p4.add(lb4);
44     p5.setBackground(Color.cyan);
45     p5.add(lb5);
46
47     // Set up the event handling here.
48     p1.addMouseListener(this);
49     p2.addMouseListener(this);
50     p3.addMouseListener(this);
51     p4.addMouseListener(this);
52     p5.addMouseListener(this);
53
54     // Add each panel to my CardLayout
55     f.add(p1, "First");
56     f.add(p2, "Second");
57     f.add(p3, "Third");
58     f.add(p4, "Fourth");
59     f.add(p5, "Fifth");
60
61     // Display the first panel.
62     myCard.show(f, "First");
63
64     f.setSize(200,200);
65     f.setVisible(true);
66 }
67
```

Layout Managers

CardLayout Manager (Continued)

```
68 public void mousePressed(MouseEvent e) {
69     myCard.next(f);
70 }
71
72 public void mouseReleased(MouseEvent e) { }
73 public void mouseClicked(MouseEvent e) { }
74 public void mouseEntered(MouseEvent e) { }
75 public void mouseExited(MouseEvent e) { }
76
77 public static void main (String args[]) {
78     CardExample ct = new CardExample();
79     ct.launchFrame();
80 }
81 }
```

This code creates the following:

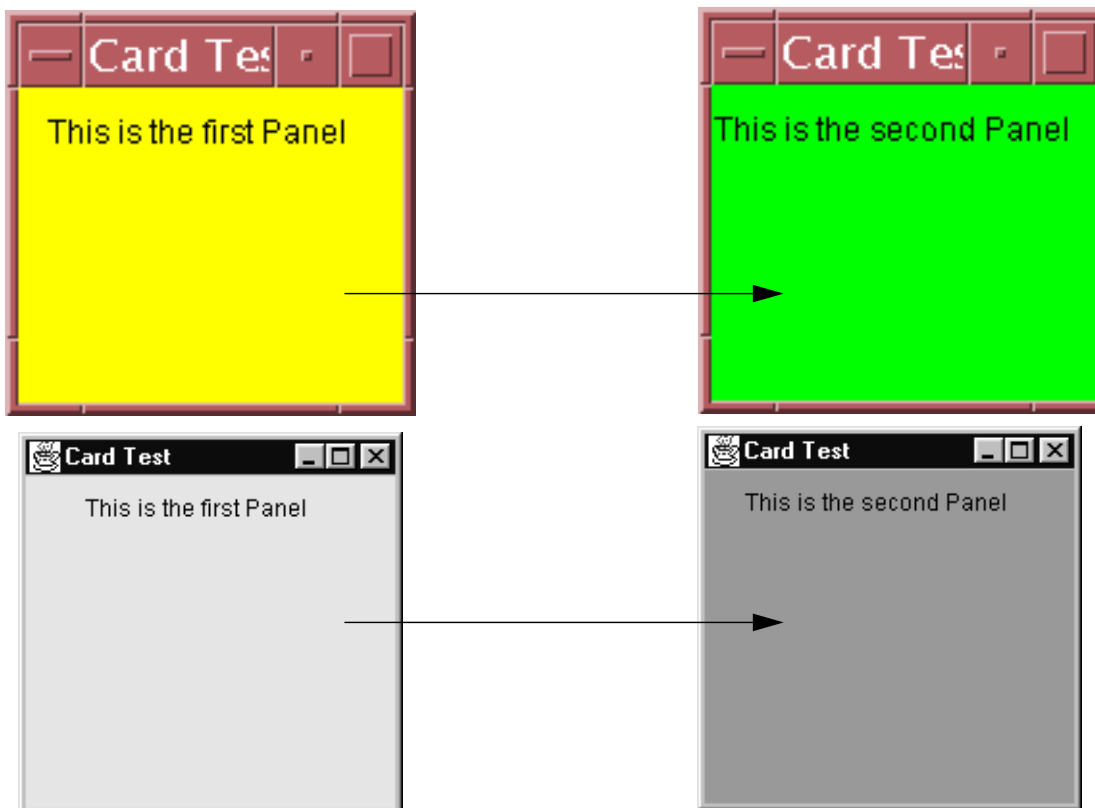


Figure 10-9 Example of CardLayout



GridBagLayout Manager

- Complex layout facilities can be placed in a grid
- A single component can take its preferred size
- A component can extend over more than one cell

Layout Managers

GridBagLayout Manager

In addition to the `FlowLayout`, `BorderLayout`, `GridLayout`, and `CardLayout` managers, the core `java.awt` also provides `GridBagLayout` manager.

The `GridBagLayout` manager provides complex layout facilities based on a grid, but allows single components to take their preferred sizes within a cell rather than fill the whole cell. The `GridBagLayout` manager also allows a single component to extend over more than one cell.

- ✓ `GridBagLayout` **is complex to understand and manipulate. The `GridBagLayout` example from the API is covered in Appendix D, "Using the `GridBagLayout`." If you want to cover it as a module, you should do so now.**

Creating Panels and Complex Layouts

The following program uses a `Panel` to allow two buttons to be placed in the `NORTH` region of a border layout. This kind of nesting is fundamental to complex layouts. The `Panel` is treated just like another component as far as the `Frame` is concerned.

```
1 import java.awt.*;
2
3 public class ComplexLayoutExample {
4     private Frame f;
5     private Panel p;
6     private Button bw, bc;
7     private Button bfile, bhelp;
8
9     public ComplexLayoutExample() {
10        f = new Frame("GUI example 3");
11        bw = new Button("West");
12        bc = new Button("Work space region");
13        bfile = new Button("File");
14        bhelp = new Button("Help");
15    }
16
17    public void launchFrame() {
18        // Add bw and bc buttons in the frame border
19        f.add(bw, BorderLayout.WEST);
20        f.add(bc, BorderLayout.CENTER);
21        // Create panel for the buttons in the north border
22        p = new Panel();
23        p.add(bfile);
24        p.add(bhelp);
25        f.add(p, BorderLayout.NORTH);
26        // Pack the frame and make it visible
27        f.pack();
28        f.setVisible(true);
29    }
30
31    public static void main(String args[]) {
32        ComplexLayoutExample gui = new ComplexLayoutExample();
33        gui.launchFrame();
34    }
35 }
```

Creating Panels and Complex Layouts

When the example is run, the resulting display looks like the following:



Figure 10-10 Combining Layout Managers

If the window is resized, it looks like the following:

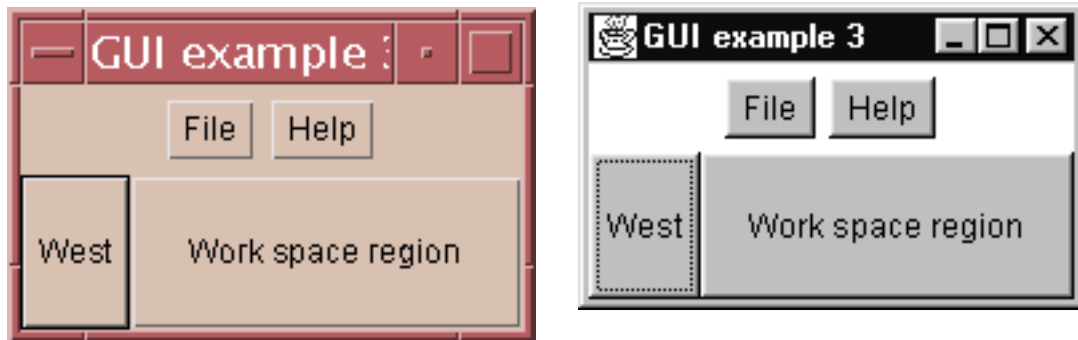
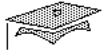


Figure 10-11 Resized Combination Layouts

The NORTH region of the BorderLayout is now holding two buttons. In fact, it holds only the single Panel but that panel contains the two buttons.

The size and position of the Panel is determined by the BorderLayout manager, and the preferred size of a Panel is determined from the preferred size of the components in that Panel. The size and position of the buttons in the Panel are controlled by the FlowLayout that is associated with the Panel by default.



Drawing in AWT

- You can draw in any `Panel` (although AWT provides the `Canvas` class just for this purpose)
- Typically, you would create a subclass of `Panel` and override the `paint` method
- The `paint` method is called every time the component is shown (for example, if another window was overlapping the component and then removed)
- Every component has a `Graphics` object
- The `Graphics` class implements many drawing methods

Drawing in AWT

Every AWT component has a `paint` method that draws the specified component. It is possible for you to draw directly to the screen using the `Graphics` object of a given component. Usually, you would create a subclass of `Panel` that would override the `paint` method.

The `paint` method is called every time the panel needs to be redrawn. The circumstances of when `paint` is called is determined by the AWT thread. For example, `paint` is called when the panel is first visible, when an overlapping window is removed, and when the panel is part of a frame that has been minimized and then restored. The programmer can also force AWT to repaint the panel by calling the `repaint` method.

Drawing in AWT

The paint method takes a single parameter: a Graphics object. The Graphics class includes many methods for drawing various shapes: arcs, ovals (circles), polygons, rectangles, rounded edge rectangles, 3D rectangles, lines, polylines, and strings. The first six of these shapes can be filled with a color.

Figure 10-12 shows a panel with the various shapes drawn on it with the names of the draw method underneath the image.

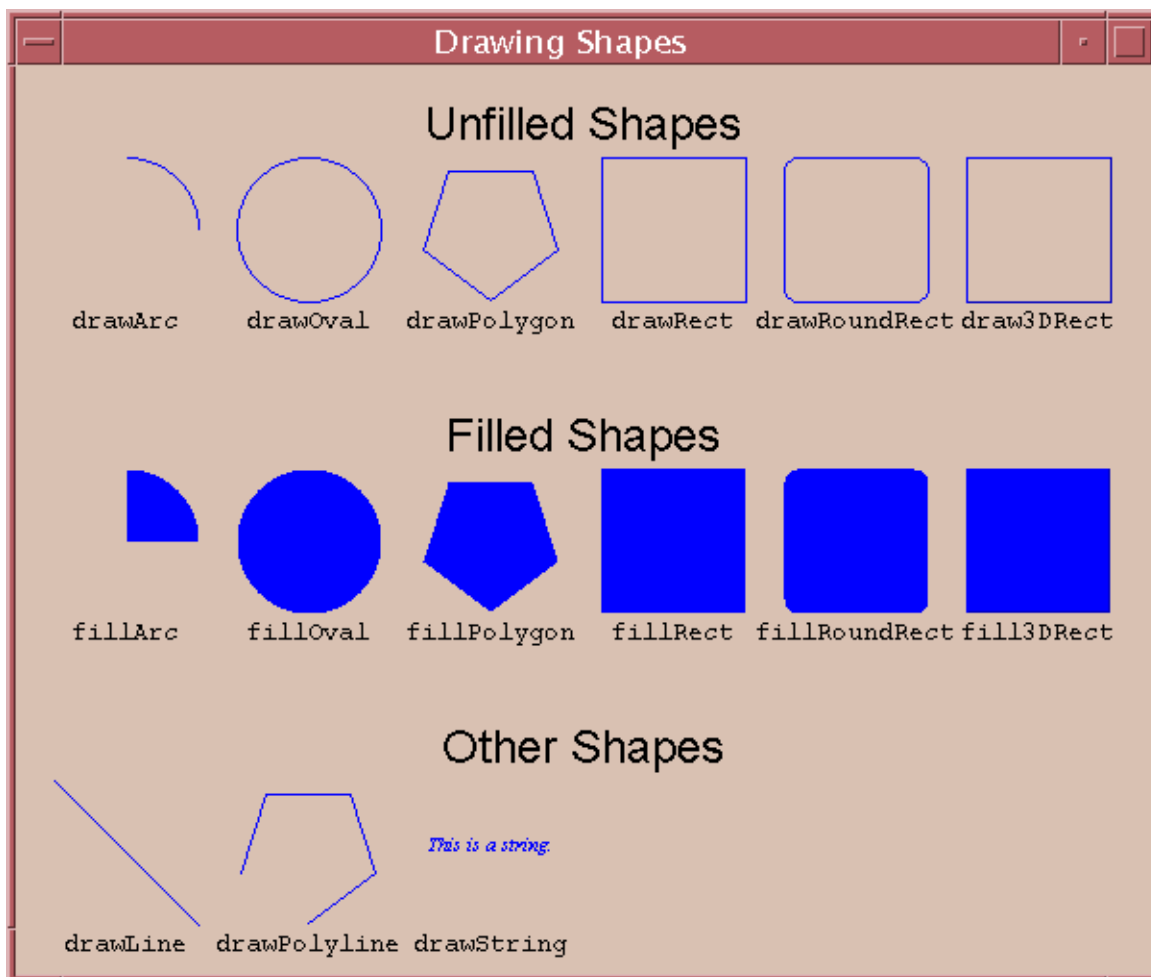


Figure 10-12 Various Shapes Drawn by the Graphics Object

Exercise: Building Java GUIs



Exercise objective – In this lab you will develop two graphical user interfaces.

Preparation

To successfully complete this lab, you must understand the purpose of a graphical user interface and know how to create one using layout managers.

Tasks

In a Web browser view the `lab_files.html` page that is at the top-level of the SL275 directory on your computer. There will be a summary of each exercise and a link to a page that gives a detailed explanation of the exercise.

Go to the SL275 directory on your computer and change to the directory for this module (`mod10`). A listing of this directory will show two subdirectories: one for each of the exercises below.

Exercise 1: Create the ChatClient GUI (Level 1 Lab)

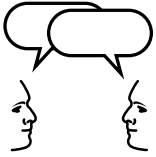
In this exercise you will create a GUI for a "chat room" application. You will use a complex layout to properly position several GUI components in a frame.

Exercise 2: Create the Calculator GUI (Level 2 Lab)

In this exercise you will create a GUI for a "calculator" application. You will use a grid layout to position the digit and operator buttons in a frame.

Exercise: Building Java GUIs

Exercise Summary



Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercises.

✓ *If you do not have time to spend on discussion, just highlight the key concepts students should have learned from the lab exercise.*

- Experiences

✓ *Ask students what their overall experiences with this exercise have been. You might want to go over any trouble spots or especially confusing areas at this time.*

- Interpretations

✓ *Ask students to interpret what they observed during any aspects of this exercise.*

- Conclusions

✓ *Have students articulate any conclusions they have reached as a result of this exercise experience.*

- Applications

✓ *Explore with the students how they might apply what they learned in this exercise to situations at their workplace.*

Check Your Progress

Before continuing on to the next module, check to be sure that you can:

- Describe the AWT package and its components
- Define the terms *containers*, *components*, and *layout managers*, and describe how they work together to build a GUI
- Use layout managers
- Use the `FlowLayout`, `BorderLayout`, `GridLayout`, and `CardLayout` managers to achieve a desired dynamic layout
- Add components to a container
- Use the `Frame` and `Panel` containers appropriately
- Describe how complex layouts with nested containers work
- In a Java program, identify the following:
 - ▼ Containers
 - ▼ The associated layout managers
 - ▼ The layout hierarchy of all components

Think Beyond

You now know how to display a GUI on the computer screen. What do you need to make the GUI useful?

Objectives

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

- Define events and event handling
- Write code to handle events that occur in a GUI
- Describe the concept of adapter classes, including how and when to use them
- Determine the user action that originated the event from the event object details
- Identify the appropriate interface for a variety of event types
- Create the appropriate event handler methods for a variety of event types
- Understand the use of inner classes and anonymous classes in event handling

This module covers the event-based GUI user input mechanism.


Relevance

- ✓ **Present the following questions to stimulate the students and get them thinking about the issues and topics presented in this module. They are not expected to know the answers to all of these questions. Hold discussions where students have input; otherwise, if no one can propose answers, begin the lecture for this module.**



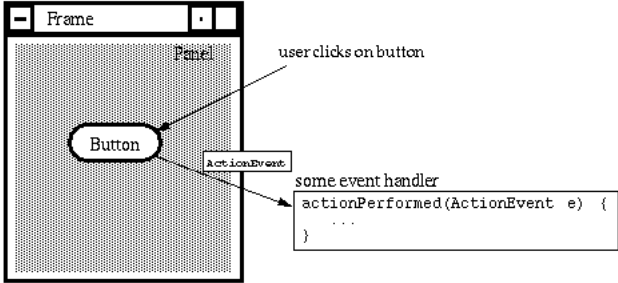
Discussion – The following questions are relevant to the material presented in this module:

- What parts of a GUI are required to make it useful?
 - How does a graphical program handle a mouse click or any other type of user interaction?
- ✓ **Part of a GUI is receiving interactive user input. This input can be used, for example, to fill out a text form on the screen, register a mouse click on a map, or shut down a program. This module describes the event-driven mechanism provided by the AWT for receiving interactive user input.**

 Sun Educational Services

What Is an Event?

- Events – Objects that describe what happened
- Event sources – The generator of an event
- Event handlers – A method that receives an event object, deciphers it, and processes the user's interaction



```
some event handler
actionPerformed(ActionEvent e) {
    ...
}
```

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.

What Is an Event?

Event Sources

An *event source* is the generator of an event. For example, a mouse click on a `Button` component generates an `ActionEvent` with the button as the source. The `ActionEvent` instance is an object that contains information about the events that just took place. It contains:

- `getActionCommand()` – Returns the command name associated with the action
- `getModifiers()` – Returns any modifiers held during the action

Event Handlers

An *event handler* is a method that receives an event object, deciphers it, and processes the user's interaction.

Java 2 SDK Event Model

Delegation Model

The delegation event model came into existence with JDK 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.

Note – The event model of JDK 1.0 is a "hierarchy model." This model has been "out of favor" for many years and we will not discuss it here. For information about this event model, see Appendix B, "JDK 1.0 GUI Event Model." Do not use the old model and the new model together.

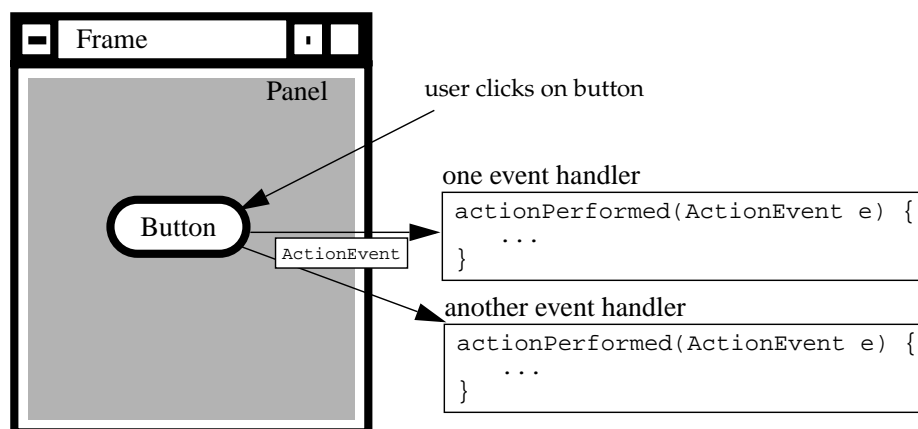


Figure 11-1 Delegation Event Model

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.

Java 2 SDK Event Model

Delegation Model (Continued)

For example, the following is the code for a simple Frame with a single Button on it:

```
1 import java.awt.*;
2
3 public class TestButton {
4     private Frame f;
5     private Button b;
6
7     public TestButton() {
8         Frame f = new Frame("Test");
9         Button b = new Button("Press Me!");
10        b.setActionCommand("ButtonPressed");
11    }
12
13    public void launchFrame() {
14        b.addActionListener(new ButtonHandler());
15        f.add(b, BorderLayout.CENTER);
16        f.pack();
17        f.setVisible(true);
18    }
19
20    public static void main(String args[]) {
21        TestButton guiApp = new TestButton();
22        guiApp.launchFrame();
23    }
24 }
```

The ButtonHandler class is the handler class to which the event is delegated.

```
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 2 SDK Event Model

Delegation Model (Continued)

This example has the following characteristics:

- The `Button` class has an `addActionListener(ActionListener)` method.
- The `ActionListener` interface defines a single method, `actionPerformed`, which receives an `ActionEvent`.
- When a `Button` object is created, it 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 `Button` object is clicked on with the mouse, 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 10, we set the action command for this button to be "ButtonPressed."



Delegation Model

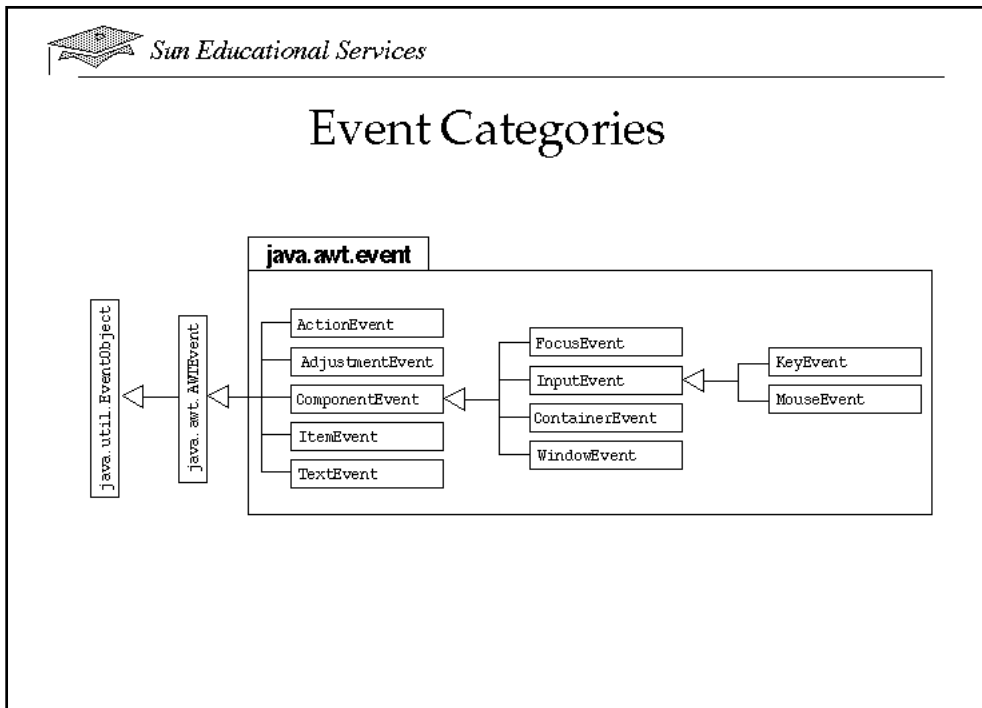
- Client objects (handlers) register with a GUI component they wish to observe
- GUI components only trigger the handlers for the type of event that has occurred
 - ▼ Most components can trigger more than one type of event
- Distributes the work among multiple classes

Java 2 SDK Event Model

Delegation Model (Continued)

- Events are not accidentally handled. The objects that wish 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 received a message that the event occurred.
- The delegation model is good for the distribution of work among classes.

Events do not have to be related to AWT components. This event model provides support for JavaBeans.



GUI Behavior

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.

For each category of events, there is an interface that 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 11-1 lists the categories, giving the interface name for each category and the methods demanded. The method names are mnemonic, indicating the source or conditions that cause the method to be called.

GUI Behavior

Event Categories (Continued)

Table 11-1 Method Categories and Interfaces

Category	Interface Name	Methods
Action	ActionListener	actionPerformed(ActionEvent)
Item	ItemListener	itemStateChanged(ItemEvent)
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)

GUI Behavior

Event Categories (Continued)

Table 11-1 Method Categories and Interfaces

Category	Interface Name	Methods
Window	WindowListener	windowClosing(WindowEvent) windowOpened(WindowEvent) windowIconified(WindowEvent) windowDeiconified(WindowEvent) windowClosed(WindowEvent) windowActivated(WindowEvent) windowDeactivated(WindowEvent)
Container	ContainerListener	componentAdded(ContainerEvent) componentRemoved(ContainerEvent)
Text	TextListener	textValueChanged(TextEvent)

GUI Behavior

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 and the key that was pressed is available in the event that it generated. In the first example on event handling, there was a separate class named `ButtonHandler` that handled events. In the following example, events are handled within the class named `TwoListener`.

GUI Behavior

Complex Example (Continued)

The following example shows the code for the `TwoListener` class:

```
1 import java.awt.*;
2 import java.awt.event.*;
3
4 public class TwoListener
5     implements MouseMotionListener,
6                 MouseListener {
7     private Frame f;
8     private TextField tf;
9
10    public TwoListener() {
11        f = new Frame("Two listeners example");
12        tf = new TextField(30);
13    }
14
15    public void launchFrame() {
16        Label label = new Label("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    }
}
```

GUI Behavior

Complex Example (Continued)

```
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 in this example are discussed in the following sections.

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.

GUI Behavior

Complex Example (Continued)

Listening to Multiple Sources

If you issue the following method calls in lines 21 and 22:

```
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; its class need only implement 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

- Multiple listeners cause unrelated parts of a program to react to the same event
- The handlers of all registered listeners are called when the event occurs

GUI Behavior

Multiple Listeners

The AWT event listening framework allows multiple listeners to be attached to the same component. In general, if you want to write a program that performs multiple actions based on a single event, code that behavior into 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 being added to an existing program.

The listener mechanism allows you to call an `addXxxListener()` method as many times as is 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.

GUI Behavior

Multiple Listeners (Continued)

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 one call the others directly.

✓ *This is called an event multiplexer.*



Event Adapters

- The listener classes that you define can extend adapter classes and override only the methods that you need
- Example:

```
1 import java.awt.*;
2 import java.awt.event.*;
3
4 public class MouseClickHandler extends MouseAdapter {
5
6     // We just need the mouseClicked handler, so we use
7     // the an adapter to avoid having to write all the
8     // event handler methods
9
10    public void mouseClicked(MouseEvent e) {
11        // Do stuff with the mouse click...
12    }
13 }
```

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)`

Event Adapters

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. For example:

```
1 import java.awt.*;
2 import java.awt.event.*;
3
4 public class MouseClickHandler extends MouseAdapter {
5
6     // We just need the mouseClicked handler, so we use
7     // an adapter to avoid having to write all the
8     // event handler methods
9
10    public void mouseClicked(MouseEvent e) {
11        // Do stuff with the mouse click...
12    }
13 }
```

Note – This is a class, not an interface. This means you can extend only one other class. Because listeners are interfaces, you can implement multiple ones.

Note – Be careful when overriding methods. Remember that a declaration, such as `public void mouseClicked(MouseEvent e)` is legal, and really a new method, not an overriding one because `mouseClicked` is misspelled. Such errors are hard to catch if the event handler produces no external effect.

Event Handling Using Inner Classes

You can implement event handlers as inner class (see lines 19 and 26-32). This allows access to the private data of the outer class (line 30).

For example:

```

1 import java.awt.*;
2 import java.awt.event.*;
3
4 public class TestInner {
5     private Frame f;
6     private TextField tf;
7
8     public TestInner() {
9         f = new Frame("Inner classes example");
10        tf = new TextField(30);
11    }
12
13    public void launchFrame() {
14        Label label = new Label("Click and drag the mouse");
15        // Add components to the frame
16        f.add(label, BorderLayout.NORTH);
17        f.add(tf, BorderLayout.SOUTH);
18        // Add a listener that uses an Inner class
19        f.addMouseListener(new MyMouseMotionListener());
20        f.addMouseListener(new MouseClickHandler());
21        // Size the frame and make it visible
22        f.setSize(300, 200);
23        f.setVisible(true);
24    }
25
26    class MyMouseMotionListener extends MouseMotionAdapter {
27        public void mouseDragged(MouseEvent e) {
28            String s = "Mouse dragging: X = " + e.getX()
29                + " Y = " + e.getY();
30            tf.setText(s);
31        }
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 once. Anonymous inner classes are generally used in conjunction with AWT event handling. For example:

```
1 import java.awt.*;
2 import java.awt.event.*;
3
4 public class TestAnonymous {
5     private Frame f;
6     private TextField tf;
7
8     public TestAnonymous() {
9         f = new Frame("Anonymous classes example");
10        tf = new TextField(30);
11    }
12
13    public void launchFrame() {
14        Label label = new Label("Click and drag the mouse");
15        // Add components to the frame
16        f.add(label, BorderLayout.NORTH);
17        f.add(tf, BorderLayout.SOUTH);
18        // Add a listener that uses an anonymous class
19        f.addMouseMotionListener(new MouseMotionAdapter() {
20            public void mouseDragged(MouseEvent e) {
21                String s = "Mouse dragging: X = " + e.getX()
22                    + " Y = " + e.getY();
23                tf.setText(s);
24            }
25        }); // <- note the closing parenthesis
26        f.addMouseListener(new MouseClickHandler());
27        // Size the frame and make it visible
28        f.setSize(300, 200);
29        f.setVisible(true);
30    }
31
32    public static void main(String args[]) {
33        TestAnonymous obj = new TestAnonymous();
34        obj.launchFrame();
35    }
36 }
```

Note – The compilation of an anonymous class generates a file, such as `TestAnonymous$1.class`.

Exercise: Working With Events



Exercise objective – You will write, compile, and run the revised ChatClient GUI and Calculator GUI codes to include event handlers.

Preparation

In order to successfully complete this lab, you must have a clear understanding of how the event model works.

Tasks

In a Web browser view the `lab_files.html` page that is at the top-level of the SL275 directory on your computer. There will be a summary of each exercise and a link to a page that gives a detailed explanation of the exercise.

Go to the SL275 directory on your computer and change to the directory for this module (`mod11`). A listing of this directory will show two subdirectories: one for each of the exercises below.

Exercise 1: Create a ChatClient GUI, Part II (Level 1)

In this exercise, you will implement the basic event handlers for the "chat room" GUI.

Exercise 2: Create a Calculator GUI, Part II (Level 3)

In this exercise, you will implement the basic event handlers for the "calculator" GUI.

Exercise: Working With Events

Exercise Summary



Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercises.

✓ *If you do not have time to spend on discussion, just highlight the key concepts students should have learned from the lab exercise.*

- Experiences

✓ *Ask students what their overall experiences with this exercise have been. You might want to go over any trouble spots or especially confusing areas at this time.*

- Interpretations

✓ *Ask students to interpret what they observed during any aspects of this exercise.*

- Conclusions

✓ *Have students articulate any conclusions they have reached as a result of this exercise experience.*

- Applications

✓ *Explore with the students how they might apply what they learned in this exercise to situations at their workplace.*

Check Your Progress

Before continuing on to the next module, check to be sure that you can:

- Define events and event handling
- Write code to handle events that occur in a GUI
- Describe the concept of adapter classes, including how and when to use them
- Determine the user action that originated the event from the event object details
- Identify the appropriate interface for a variety of event types
- Create the appropriate event handler methods for a variety of event types
- Understand the use of inner classes and anonymous classes in event handling

Think Beyond

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?

Objectives

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

- Differentiate between a standalone application and an applet
- Write an HTML tag to call a Java applet
- Describe the class hierarchy of the applet and AWT classes
- Create the `HelloWorld.java` applet
- List the major methods of an applet
- Describe and use the painting model of AWT
- Use applet methods to read images and files from URLs
- Handle various mouse events within the applet
- Pass parameters to an applet from an HTML file using the `<param>` tag

This module discusses the support for applets by the Java 2 SDK, and describes how applets differ from applications in terms of program form, operating context, and how they are started.

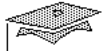
Relevance

- ✓ **Present the following question to stimulate the students and get them thinking about the issues and topics presented in this module. They are not expected to know the answer to this question. Hold discussions where students have input; otherwise, if no one can propose answers, begin the lecture for this module.**



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

- What advantages do applets provide over standalone applications?
- ✓ **It is clear from the Java platform's graphics and network security features that the Web is a significant niche for Java programs. Until now, only standalone Java software programs have been discussed. This module discusses what you need to know to run Java software programs using a Web browser, allowing access over the Web.**
 - ✓ **There are significant differences between application and applet operating contexts; this module discusses the differences between the contexts, and shows how to code applets to operate in their context.**
 - ✓ **Applets interact with their environment differently than applications. This module discusses how applets are started, and describes enough HTML to start an applet from a Web page.**
 - ✓ **The "hooks" or places (that is, methods) by which the environment runs an applet differ from those for applications (which have a "main" method). This module explores these "hook" methods, and when and how they are called by their environment. Additionally, the `Component` `paint` mechanism that updates the screen when its content is changed or otherwise needs to be redisplayed is described.**



What Is an Applet?

A Java class that can be:

- Embedded within an HTML page and downloaded and executed by a Web browser
- Loaded using the browser as follows:
 1. Browser loads URL.
 2. Browser loads HTML document.
 3. Browser loads applet classes.
 4. Browser runs applet.

What Is an Applet?

An *applet* is a Java class that you can embed in an HTML page, and is downloaded and executed by a Web browser. It is a specific type of Java technology container. It differs from an application in the way it is executed. An application is started when its main method is called. The lifecycle of an applet is more complex. This module examines how to run an applet, how to load an applet into the browser, and how to write an applet.

Loading an Applet

An applet runs in the environment of a Web browser, so it is not started directly by typing a command. You must create an HTML file that tells the browser what to load and how to run it. You then “point” the browser at the URL that specifies that HTML file. (The format of the HTML file is discussed later in this module.)

What Is an Applet?

Loading an Applet (Continued)

Figure 12-1 illustrates the steps involved in running an applet.

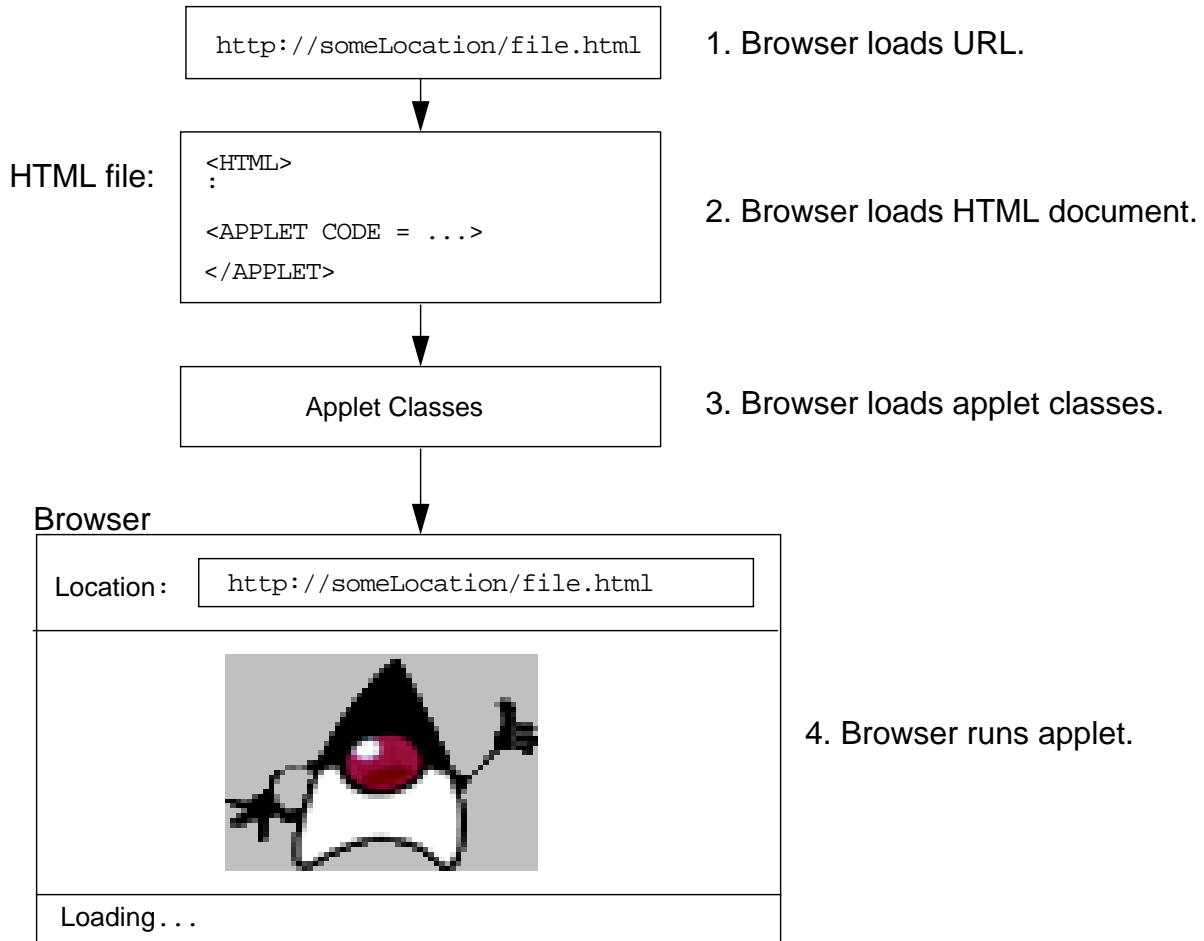


Figure 12-1 Running an Applet



Applet Security Restrictions

- Most browsers prevent the following:
 - ▼ Runtime execution of another program
 - ▼ File I/O
 - ▼ Calls to any native methods
 - ▼ Attempts to open a socket to any system except the host that provided the applet

What Is an Applet?

Applet Security Restrictions

Applets are pieces of code that represent an inherently dangerous prospect because they are loaded over a network. What if someone writes a malicious class that reads your personal files and sends them over the Internet?

The depth to which security is controlled is implemented at the browser level. Most browsers (including Netscape Navigator) prevent the following by default:

- Runtime execution of another program
- File I/O
- Calls to any native methods
- Attempts to open a socket to any system except the host that provided the applet

What Is an Applet?

Applet Security Restrictions (Continued)

These restrictions prevent an applet from violating the privacy of, or damaging a remote system, by restricting the applet's access to the files of that system. Preventing the execution of another program and disallowing calls to native methods restrict the applet from starting code that runs unchecked by the JVM. The restriction on sockets prevents communication with another program, which is stored on a non-trusted host.

Java 2 SDK provides a means of specifying a particular "protection domain," or security environment, in which a particular applet is to be run. A remote system checks the originating URL and signature of the applet it downloads against a local file containing entries mapping special applets to special protection domains. This enables special applets coming from particular places to run with special privileges.

Writing an Applet

To write an applet, you must create a class using the following form:

```
import java.applet.*;

public class HelloWorld extends Applet {
```

The applet's class must be public and its name must match the name of the file it is in; in this case, `HelloWorld.java`. The class must be a subclass of the class `java.applet.Applet`.

Applet Class Hierarchy

The `java.applet.Applet` class is actually a subclass of `java.awt.Panel`. The hierarchy of the applet and AWT classes is as follows:

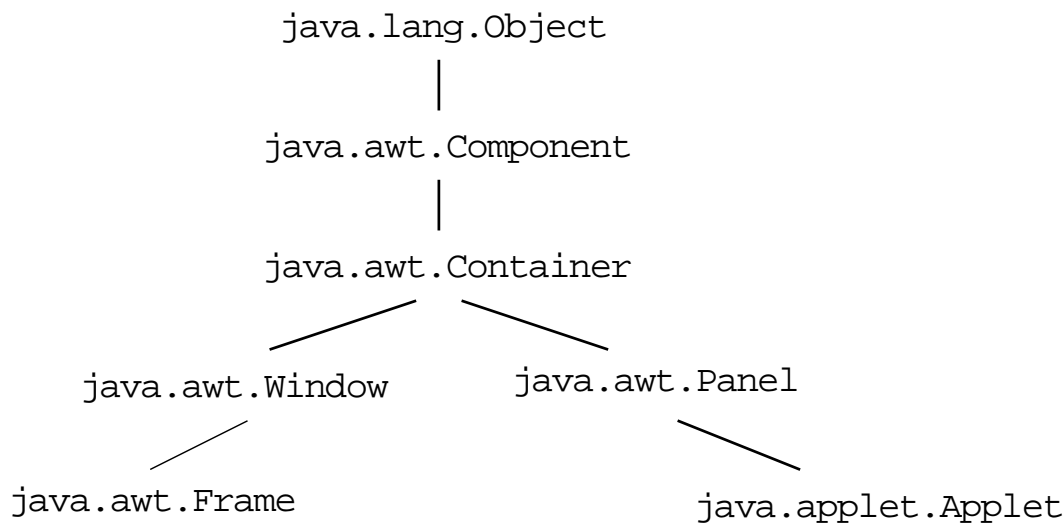
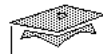


Figure 12-2 Applet and AWT Class Hierarchies

This hierarchy shows that you can use an applet directly as the starting point for an AWT layout. Because an applet is a `Panel`, it has a `FlowLayout` manager by default. The methods of the `Component`, `Container`, and `Panel` classes are inherited by the `Applet` class.



Key Applet Methods

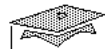
- `init()`
- `start()`
- `stop()`
- `destroy()`
- `paint()`

Writing an Applet

Key Applet Methods

In an application, the program is entered when the main method is called. However, in an applet, this is not the case. After the constructor completes its task, the browser calls `init` to perform basic initialization of the applet. After `init` completes its task, the browser calls the method `start`. `start` is described more closely later in this module; however, in general, it is called when the applet becomes visible.

Both the `init` and `start` methods run to completion before the applet becomes “live,” and because of this they cannot be used to program ongoing behavior into an applet. In fact, unlike the main method in a simple application, there is no method that is executed continuously throughout the “life” of the applet. You will see later how to do this using threads. Additional methods you write for your applet subclass can include `stop`, `destroy`, and `paint`.



The Applet Life Cycle

- `init()`
 - ▼ Called when the applet is created
 - ▼ Can be used to initialize data values
- `start()`
 - ▼ Called when the applet becomes visible
- `stop()`
 - ▼ Called when the applet becomes invisible

Applet Methods and the Applet Life Cycle

The applet lifecycle is more complex than what has been discussed so far. There are three major methods that relate to its lifecycle: `init`, `start`, and `stop`.

`init`

This member function is called at the time the applet is created and loaded into a browser capable of supporting Java technology (such as the `appletviewer`). The applet can use this method to initialize data values. The `init` method runs to completion before `start` is called.

```
public void init() {  
    // set up GUI  
}
```

Applet Methods and the Applet Life Cycle

start

Once the `init` method is completed, the `start` method executes, which causes the applet to become “live.” It also runs whenever the applet becomes visible, such as when the browser is restored after being iconized or when the browser returns to the page containing the applet after moving to another URL. This method is typically used to start threads or an animation or to play sounds.

```
public void start() {  
    musicClip.play();  
}
```

stop

The `stop` method is called when the applet becomes invisible. This happens when the browser is iconified or it follows a link to another URL. The applet uses this method to stop any functionality that should not occupy the CPU when the applet is not on the current browser page.

```
public void stop() {  
    musicClip.stop();  
}
```

The `start` and `stop` methods effectively form a pair. Typically `start` activates a behavior in an applet and `stop` deactivates the behavior.



Applet Display

- Applets are graphical in nature
- The browser environment calls the `paint()` method

```
1 import java.awt.*;
2 import java.applet.*;
3
4 public class HelloWorld extends Applet {
5     private paintCount;
6     public void init() {
7         paintCount = 0;
8     }
9     public void paint(Graphics g){
10        g.drawString("Hello World!", 25, 25);
11        paintCount++;
12        g.drawString("Number of times paint called: "
13                    + paintCount, 25, 50);
14    }
15 }
```

Applet Display

Applets are essentially graphical in nature, so although you can issue `System.out.println()` calls, you do not normally do so. Instead, you create your display in a graphical environment.

You can draw on an applet's panel by creating a `paint` method. The browser environment calls the `paint` method whenever the applet's display needs refreshing. For example, when the browser window is displayed after being minimized or iconified.

Write your `paint` method so that it works properly whenever it is called. Exposure happens asynchronously and is driven by the environment, not the program.

- ✓ **The `paint` method is not limited to use in applets. The `Canvas` and `Frame` classes use `paint` as well.**

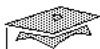
Applet Display

The paint Method and the Graphics Object

The `paint` method takes an argument that is an instance of the `java.awt.Graphics` class. The argument is always the graphics context of the panel that makes up the applet (remember, `Applet` is a subclass of `Panel`). You can use this context to draw or write into your applet. The following is an example of an applet that uses a `paint` method to write text:

```
1 import java.awt.*;
2 import java.applet.*;
3
4 public class HelloWorld extends Applet {
5     private int paintCount;
6     public void init() {
7         paintCount = 0;
8     }
9     public void paint(Graphics g){
10        g.drawString("Hello World", 25, 25);
11        ++paintCount;
12        g.drawString("Number of times paint called: "
13                    + paintCount, 25, 50);
14    }
15 }
```

Note – The numeric arguments to the `drawString` method are the `x` and `y` pixel coordinates for the start of the text. `(0,0)` represents the upper left corner. These coordinates refer to the baseline of the font, so writing at `y` coordinate zero results in the bulk of your text being off the top of the display, only the descenders, such as the tail of the letter *p* are visible.



AWT Painting

- `paint(Graphics g)` – Called when the component is "exposed". This is where the programmer implements the painting algorithm.
- `repaint()` – You call this method to ask the AWT thread to repaint the component.
- `update(Graphics g)` – This method is called by the AWT thread when a repaint has been requested.

AWT Painting

In addition to the basic lifecycle methods, an applet has important methods related to its display. These methods are declared and documented for the AWT Component class. You must adhere to the correct model for display handling using the AWT.

Update the display using a separate thread that is referred to as the *AWT thread*. This thread can be called upon to handle two situations that relate to updating the display.

The first of these conditions is exposure; either when the display is first exposed, or where part of the display has been damaged and must be replaced. Display damage can occur at any time, and your program must be able to update the display at any time.

The second condition is when the program redraws the display with new contents. This redrawing might require that you first remove the old image.

AWT Painting

The paint Method

Exposure handling occurs automatically and results in a call to the `paint` method. A facility of the `Graphics` class, called a clip rectangle, is used to optimize the `paint` method so that updates are not made over the entire area of the graphics unless necessary. Rather, these updates are restricted to the region that has been damaged. Override the `paint` method to control what is painted on your applet.

```
public void paint(Graphics g) {...}
```

The repaint Method

A call to the `repaint` method notifies the system that you want to change the display.

The update Method

The `repaint` method actually causes the AWT thread to call another method, `update`. The `update` method usually clears the current display and calls `paint`. You can modify the `update` method, for example, to reduce flicker by calling `paint` without clearing the display.

AWT Painting

Method Interaction

Figure 12-3 shows how the paint, update, and repaint methods are related.

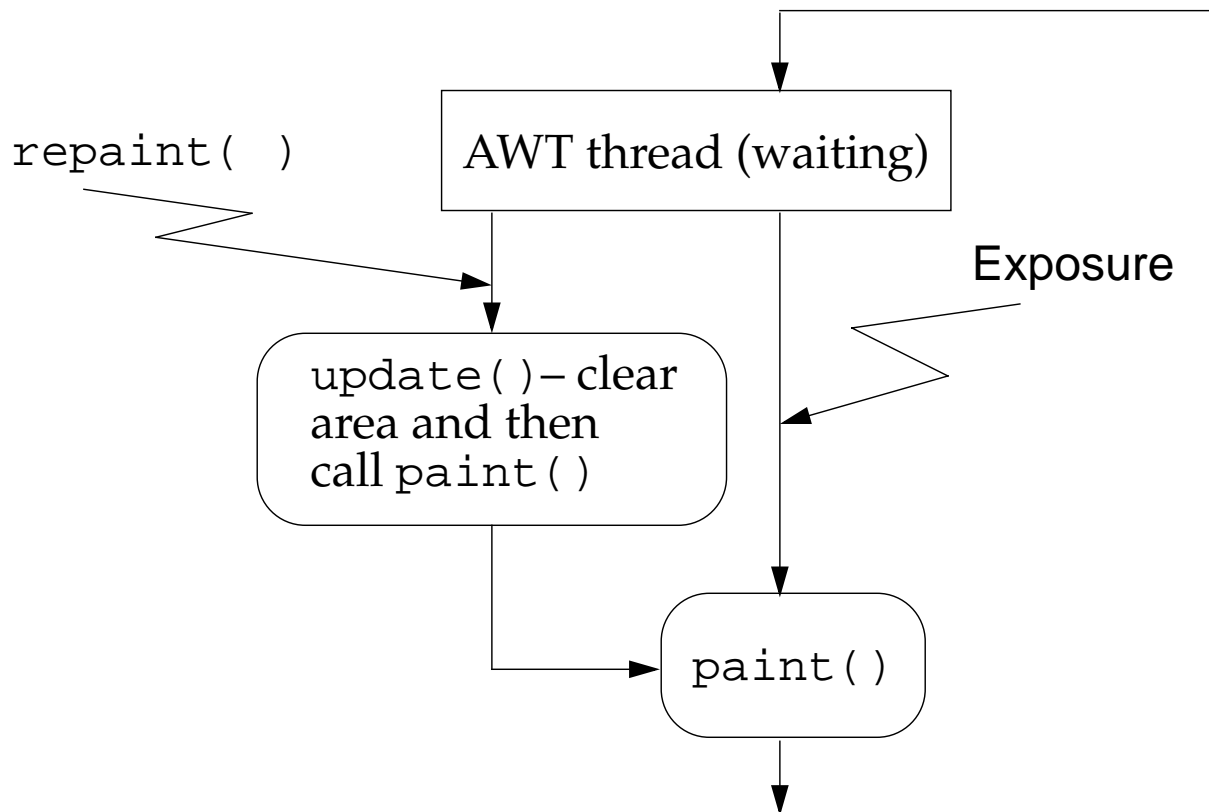


Figure 12-3 Method Relationships



Applet Display Strategies

- Maintain a model of the display
- Use `paint()` to render the display based only on the model
- Update the model and call `repaint()` to change the display

AWT Painting

Applet Display Strategies

The applet model requires that you adopt a specific strategy for maintaining your display. You must do the following:

- Maintain a model of the display. This model defines how to re-render the display. Instructions on how to do this are embodied in the `paint` method.
- Have the `paint` method *render* the display based *only* on the contents of the model. This allows `paint` to regenerate the display consistently whenever it is called, and handle exposure correctly.

AWT Painting

Applet Display Strategies (Continued)

- Have the program change the display by updating the model and then calling the `repaint` method so that the `update` method (and ultimately the `paint` method) gets called by the AWT thread.

Note – A single AWT thread handles all component painting and the distribution of input events. Keep `paint` and `update` methods simple to avoid stalling the AWT thread. In extreme cases, you will need the help of other threads to achieve this. Thread programming is the subject of Module 14, "Threads."

An Example Paint Model

Suppose you want to display the string “Hello World!” in the applet panel wherever the user clicks. A simple implementation of this requirement would be to paint the string “Hello World!” every time the user clicks the mouse in the applet panel. This can be accomplished by keeping track of the *click point* through the `lastClick` data attribute. For example:

```
1 // <APPLET CODE="PaintModel1.class" WIDTH=200 HEIGHT=200></APPLET>
2
3 import java.applet.*;
4 import java.awt.event.*;
5 import java.awt.*;
6
7 public class PaintModel1 extends Applet {
8     // The paint model: the last click Point
9     private Point lastClick = null;
10
11     public void init() {
12         addMouseListener(new MyModelRecorder());
13     }
14
15     public void paint(Graphics g) {
16         if ( lastClick != null ) {
17             g.drawString("Hello World!", lastClick.x, lastClick.y);
18         }
19     }
20
21     private class MyModelRecorder extends MouseAdapter {
22         public void mousePressed(MouseEvent e) {
23             lastClick = e.getPoint();
24             repaint();
25         }
26     }
27 }
```

Unfortunately, the AWT thread repaints the applet using the `update` method which clears the screen before calling our `paint` method.

- ✓ ***There are three implementations of `PaintModel` that demonstrate how to build a reasonable paint model and how `repaint/update/paint` work together. This first model assumes that `paint` will be called without the screen being cleared by `update`.***

An Example Paint Model

A possible solution to the update problem is to override the update method to not clear the screen before calling paint.

```
1 // <APPLET CODE="PaintModel2.class" WIDTH=200 HEIGHT=200></APPLET>
2
3 import java.applet.*;
4 import java.awt.event.*;
5 import java.awt.*;
6
7 public class PaintModel2 extends Applet {
8     // The paint model: the last click Point
9     private Point lastClick = null;
10
11     public void init() {
12         addMouseListener(new MyModelRecorder());
13     }
14
15     public void update(Graphics g) {
16         paint(g);
17     }
18
19     public void paint(Graphics g) {
20         if ( lastClick != null ) {
21             g.drawString("Hello World!", lastClick.x, lastClick.y);
22         }
23     }
24
25     private class MyModelRecorder extends MouseAdapter {
26         public void mousePressed(MouseEvent e) {
27             lastClick = e.getPoint();
28             repaint();
29         }
30     }
31 }
```

Unfortunately, if the applet is obscured and then exposed, the pixels that were lost are not redrawn (except for the last point).

- ✓ **For this model, you want to demonstrate that the user can click several times and “Hello World!” appears at each location (as desired). However, ask the students what would happen if a window were to cover up and then expose some of the applet? The pixels are lost (except that the last point is redrawn).**

An Example Paint Model

Our last model uses a list of `Point` objects to keep track of every user click location. The mouse pressed event handler adds each click point to the `mouseClicks` list and the paint method draws “Hello World!” at each click point by iterating over the list.

```
1 // <APPLET CODE="PaintModel3.class" WIDTH=200 HEIGHT=200></APPLET>
2
3 import java.applet.*;
4 import java.awt.event.*;
5 import java.awt.*;
6 import java.util.List;
7 import java.util.ArrayList;
8
9 public class PaintModel3 extends Applet {
10 // The paint model: a list of click Points
11 private List mouseClicks = new ArrayList(5);
12
13 public void init() {
14     addMouseListener(new MyModelRecorder());
15 }
16
17 public void update(Graphics g) {
18     paint(g);
19 }
20
21 public void paint(Graphics g) {
22     for(int x = 0; x < mouseClicks.size(); x++) {
23         Point p = (Point) mouseClicks.get(x);
24         g.drawString("Hello World!", p.x, p.y);
25     }
26 }
27
28 private class MyModelRecorder extends MouseAdapter {
29     public void mousePressed(MouseEvent e) {
30         mouseClicks.add(e.getPoint());
31         repaint();
32     }
33 }
34 }
```

Note – While it is not imperative to override the `update` method, we have done so to reduce screen flicker.



What Is the appletviewer?

A Java application that:

- Enables you to run applets without using a Web browser
- Loads the HTML file supplied as an argument
- `appletviewer HelloWorld.html`
- Needs at least the following HTML code:

```
1 <applet code="HelloWorld.class" width=300 height=300>  
2 </applet>
```

What Is the appletviewer?

An applet is usually run inside a web browser, such as HotJava™ or the Netscape Navigator, which is capable of running Java software programs. To simplify and speed up development, the Java 2 SDK comes with a tool designed only to view applets, not HTML pages. This tool is the `appletviewer`.

The `appletviewer` is a Java application that enables you to run applets without using a Web browser. It resembles a *minimum browser*.

The `appletviewer` reads the HTML file specified by the URL on the command line. This file must contain the instructions for loading and running one or more applets. The `appletviewer` ignores all other HTML code. It does not display normal HTML or embedded applets in a text page.

- ✓ **Explain that the `appletviewer` has a subset of the functionality of a browser and is designed so that applets created now will work with minimal changes later.**

Starting Applets With the appletviewer

The `appletviewer` posts a frame-like space onto the screen, instantiates an instance of the applet, and posts that applet instance to the Frame.

The `appletviewer` takes, as a command-line argument, a URL to an HTML file containing an applet reference. This applet reference is an HTML tag that specifies the code that the `appletviewer` loads; for example:

```
<applet code=HelloWorld.class width=100 height=100>  
</applet>
```

✓ ***There can be other tags between `applet` and `/applet`; these are covered later.***

The general format of this tag is the same as any other HTML, using the `<` and `>` symbols to delimit the instructions. All the parts shown here are required. You must have both `<applet ...>` and `</applet>`. The `<applet ...>` part specifies a code entry and a width and height.

Note – You should treat applets as being of fixed size and use the size specified in the `<applet>` tag.

Starting Applets With the appletviewer

Synopsis

The appletviewer takes a URL to an HTML file containing the `<applet>` tag as a command-line argument.

```
appletviewer [-debug] URLs ...
```

The only valid option to the appletviewer is the `-debug` flag which starts the applet in the Java debugger, `jdb`. Compile your Java code with the `-g` option to see the source code in the debugger.

Example

The following appletviewer command starts the appletviewer:

```
appletviewer HelloWorld.html
```

This creates and displays the small windows in Figure 12-4.

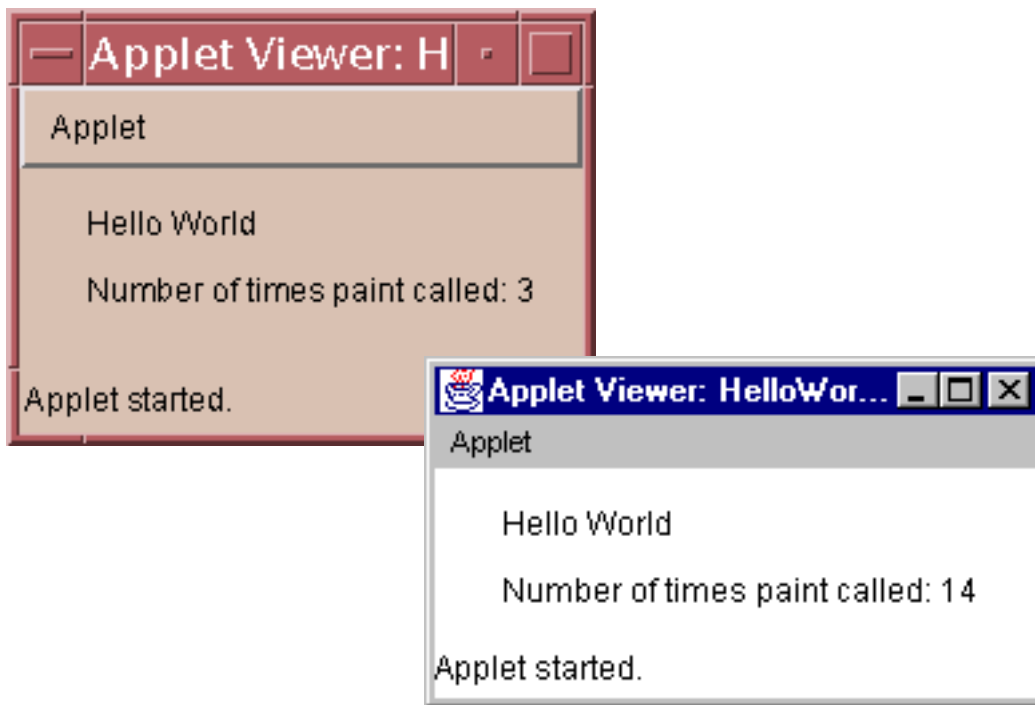


Figure 12-4 HelloWorld Applets

The applet Tag

Syntax

The complete syntax for the applet tag is:

```
<applet
  [archive=archiveList]
  code=appletFile.class
  width=pixels height=pixels
  [codebase=codebaseURL]
  [alt=alternateText]
  [name=appletInstanceName]
  [align=alignment]
  [vspace=pixels] [hspace=pixels]
>
[<param name=appletAttribute1 value=value>]
[<param name=appletAttribute2 value=value>]
. . .
[alternateHTML]
</applet>
```

where

- `archive` = *archiveList* – This optional attribute describes one or more archives containing classes and other resources that are “preloaded.” The classes are loaded using an instance of an `AppletClassLoader` with the given codebase. The archives in *archiveList* are separated by a comma (,).
- `code` = *appletFile.class* – This *required* attribute gives the name of the file that contains the compiled `Applet` subclass. This can also be in the format *package.appletFile.class*.

Note – This file is relative to the base URL of the HTML file from which you are loading. It *cannot* include a path name. To change the base URL of the applet, use the `<codebase>` tag.

- `width` = *pixels* `height` = *pixels* – These *required* attributes give the initial width and height (in pixels) of the applet display area, not including any Windows or Dialogs that the applet displays.

The applet Tag

Description

- `codebase` = *codebaseURL* – This optional attribute specifies the base URL of the applet—the directory that contains the applet's code. If this attribute is not specified, then the document's URL is used.
- `alt` = *alternateText* – This optional attribute specifies which text to display if the browser can read the applet tag but cannot run Java applets.
- `name` = *appletInstanceName* – This optional attribute specifies a name for the applet instance, which makes it possible for applets on the same page to find (and communicate with) each other.
- `align` = *alignment* – This optional attribute specifies the alignment of the applet. The possible values of this attribute are the same as those for the `IMG` tag in basic HTML: `left`, `right`, `top`, `texttop`, `middle`, `absmiddle`, `baseline`, `bottom`, and `absbottom`.
- `vspace` = *pixels* `hspace` = *pixels* – These optional attributes specify the number of pixels above and below the applet (`vspace`) and on each side of the applet (`hspace`). They are treated the same way as the `IMG` tag's `vspace` and `hspace` attributes.
- `<param name = appletAttribute1 value = value>` – This tag provides an applet with a value specified “from the outside,” serving the same functional purpose as command-line arguments serve a Java application. Applets access their attributes with the `getParameter` method, which is covered in more detail later in this module.
- Browsers that are not capable of running Java programs display any regular HTML included between your `<applet>` and `</applet>` tags; browsers capable of supporting Java technology ignore the HTML code between these two tags.



Additional Applet Features

- `getDocumentBase()` – Returns a URL object that describes the directory of the current browser page
- `getCodeBase()` – Returns a URL object that describes the source directory of the applet class
- `getImage(URL base, String target)` and `getAudioClip(URL base, String target)` – Use the URL object as a starting point

Additional Applet Features

A number of additional features are available in an applet.

All Java software programs have access to network features using the classes in the `java.net` package that is examined in Module 15. Applets additionally have methods that allow them to determine information about the browser environment in which they have been launched.

The class `java.net.URL` describes URLs and can be used to connect to them. Two methods in the `Applet` class determine the value of significant URLs:

- `getDocumentBase` returns a URL object that describes the directory of the current browser page (where the HTML file with applet tags resides).
- `getCodeBase` returns a URL object that describes the source directory of the applet class file itself. Often this is the same as the HTML file directory, but this is not always the case.

Additional Applet Features

Using the URL as a starting point, you can put sounds and images into your applet.

- `getImage(URL base, String target)` fetches an image from the file named by `target` located at the URL specified by `base`. The returned value is an instance of the class `Image`.
- `getAudioClip(URL base, String target)` fetches a sound from the file named by `target` located at the URL specified by `base`. The returned value is an instance of the class `AudioClip`.

Note – The `String target` in `getImage(URL, String)` and `getAudioClip(URL, String)` methods can include a relative directory path from the URL. However, relative path names up the directory hierarchy might not be allowed on some systems.

A Simple Image Test

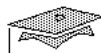
The following applet retrieves the image file `graphics/surferDuke.gif` relative to the directory path returned by the `getDocumentBase` method and displays it.

```
1 // Applet which shows an image of Duke in surfing mode
2
3 import java.awt.*;
4 import java.applet.Applet;
5
6 public class HwImage extends Applet {
7     Image duke;
8
9     public void init() {
10         duke = getImage(getDocumentBase(),
11             "graphics/surferDuke.gif");
12     }
13
14     public void paint(Graphics g) {
15         g.drawImage(duke, 25, 25, this);
16     }
17 }
```

The arguments to the `drawImage` method are:

- The `Image` object to be drawn.
- The `x` coordinate for the drawing.
- The `y` coordinate for the drawing.
- The *image observer*. An image observer is an interface that is notified if the image's status changes (such as what happens during loading). The `Applet` class supports the `ImageObserver` interface.

An image that is loaded by `getImage` changes over time after the call is first issued. This is because the loading is done in the background. Each time more of the image is loaded, the `paint` method is called again. This call to the `paint` method happens because the applet was registered as an observer when it passed itself as the fourth argument to `drawImage`.



Sun Educational Services

Audio Clips

Playing a clip:

```
play(URL soundDirectory, String soundFile);  
play(URL soundURL);
```

Example:

```
play(getDocumentBase(), "bark.au");
```

Audio Clips

The Java programming language also has methods to play audio clips. These methods are in the `java.applet.AudioClip` class. You will need the appropriate hardware for your computer to play audio clips.

Playing a Clip

The easiest way to listen to an audio clip is through an Applet play method:

```
play(URL soundDirectory, String soundFile);
```

or, more simply:

```
play(URL soundURL);
```

For example,

```
play(getDocumentBase(), "bark.au");
```

plays `bark.au`, which exists in the same directory as the HTML file.

- ✓ **In most instances, you cannot use a relative path name that goes up the path hierarchy; for example, `../bark.au`.**

A Simple Audio Test

The following applet prints the message *Audio Test* in the appletviewer and then plays the audio file, *cuckoo.au* in the sounds directory:

```
1 // Applet which plays a sound on every mouse click
2
3 import java.awt.Graphics;
4 import java.awt.event.MouseAdapter;
5 import java.awt.event.MouseEvent;
6 import java.applet.Applet;
7
8 public class HwAudio extends Applet {
9     public void init() {
10         addMouseListener(new MouseAdapter() {
11             public void mouseClicked(MouseEvent event) {
12                 play(getCodeBase(), "sounds/cuckoo.au");
13             }
14         });
15     }
16     public void paint(Graphics g) {
17         g.drawString("Audio Test", 25, 25);
18     }
19 }
```

- ✓ **The applet `play` method creates an `AudioClip` (named `clip`) using the URL you give it and then it calls `clip.play()`.**
- ✓ **Audio clips support only primitive audio formats and simple operations. The Java Media Framework (JMF) supports a richer set of formats and more operations (including mixing of sound sources).**



Looping an Audio Clip

- Loading an Audio Clip
- Playing an Audio Clip
- Stopping an Audio Clip

Looping an Audio Clip

You can load audio clips like images; that is, you can load them and play them later.

Loading an Audio Clip

To load an audio clip, use the `getAudioClip` method from the `java.applet.Applet` class.

```
AudioClip sound;  
sound = getAudioClip(getDocumentBase(), "bark.au");
```

Once a clip is loaded, use one of the three methods associated with it: `play`, `loop`, or `stop`.

Looping an Audio Clip

Playing an Audio Clip

Use the `play` method in the `java.applet.AudioClip` interface to play the loaded audio clip once.

```
sound.play();
```

To start the clip playing and have it loop (automatically repeat), use the `loop` method in `java.applet.AudioClip`.

```
sound.loop();
```

Stopping an Audio Clip

To stop a running clip, use the `stop` method in `java.applet.AudioClip`.

```
sound.stop();
```

A Simple Audio Looping Test

The following example automatically loops through a loaded audio clip:

```
1 // Applet which continuously repeats a sound
2
3 import java.awt.Graphics;
4 import java.applet.*;
5
6 public class HwLoop extends Applet {
7     AudioClip sound;
8
9     public void init() {
10        sound = getAudioClip(getCodeBase(), "sounds/cuckoo.au");
11    }
12
13    public void paint(Graphics g) {
14        g.drawString("Audio Test", 25, 25);
15    }
16
17    public void start() {
18        sound.loop();
19    }
20
21    public void stop() {
22        sound.stop();
23    }
24 }
```

Note – Java 2 SDK supports a sound engine that provides playback for Musical Instrument Digital Interface (MIDI) files and the full range of .wav, aiff, and .au files. It uses the method `newAudioClip(URL url)`. This method retrieves an audio clip from the given URL. The parameter URL points to the audio clip. You can replace the `getAudioClip` method in line 13 with this method. The `newAudioClip` method does not need a `String` as the second parameter. Only the URL parameter should be passed.

Mouse Input

One of the most useful features that the Java programming language supports is direct interactivity. A Java applet, like an application, can pay attention to the mouse and react to mouse events. The following is a quick review of mouse support, to help you understand the next example.

In Module 9, you learned that the Java 2 SDK event model supports an event type for each type of interactivity. Mouse events are received by classes that implement the `MouseListener` interface, which receives events for:

- `mouseClicked` – The mouse has been clicked (mouse button pressed and then released in one motion)
- `mouseEntered` – The mouse cursor enters a component
- `mouseExited` – The mouse cursor leaves a component
- `mousePressed` – The mouse button is pressed down
- `mouseReleased` – The mouse button is later released

✓ ***Presses and releases are always received, but clicks are the result of a single “click” and might not be received if the user holds the mouse button down.***

A Simple Mouse Test

The following program displays the location of the mouse click within the applet.

```
1 // This applet is HelloWorld extended to watch for mouse
2 // input. "Hello World!" is reprinted at the location of
3 // the mouse press.
4
5 import java.awt.Graphics;
6 import java.awt.event.*;
7 import java.applet.Applet;
8
9 public class HwMouse extends Applet {
10 // "paint model data"
11 private int mouseX = 25;
12 private int mouseY = 25;
13
14 // Register an anonymous mouse events handler.
15 public void init() {
16     addMouseListener(new MouseHandler());
17 }
18
19 public void paint(Graphics g) {
20     g.drawString("Hello World!", mouseX, mouseY);
21 }
22
23 private class MouseHandler extends MouseAdapter {
24     public void mousePressed(MouseEvent evt) {
25         // record the position of the mouse
26         // in the "paint model data"
27         mouseX = evt.getX();
28         mouseY = evt.getY();
29         // inform AWT to repaint the applet
30         repaint();
31     }
32 }
33 }
```

Reading Parameters

In an HTML file, a `<param>` tag in an `<applet>` context can pass configuration information to the applet. For example:

```
1 <html>
2 <applet code="Parameters.class" width=200 height=200>
3   <param name=speed value="12">
4   <param name=distance value="500m">
5 </applet>
6 </html>
```

Inside the applet, you can use the method `getParameter()` to read the following values.

```
1 // Parameter test applet. To see a change in "speed",
2 // you must supply it as a <param> tag in the HTML file
3 // which calls this applet.
4
5 import java.applet.Applet;
6 import java.awt.Graphics;
7
8 public class Parameters extends Applet {
9   private String toDisplay;
10  private int speed;
11
12  public void init() {
13    String pv;
14    pv = getParameter("speed");
15    if (pv == null){
16      speed = 10;
17    } else {
18      speed = Integer.parseInt (pv);
19    }
20    toDisplay = "Speed given: " + speed;
21  }
22
23  public void paint(Graphics g) {
24    g.drawString(toDisplay, 25, 25);
25  }
26 }
```

Reading Parameters

The method `getParameter` searches for a match of the name and returns the associated value as a `String`.

If the parameter name cannot be found in any `<param>` tag inside the `<applet></applet>` pair, then `getParameter` returns `null`. A production program should handle this gracefully.

The parameter type is always a `String`. If you need this in other forms you must convert it; for example, read an `int` parameter.

```
int speed = Integer.parseInt(getParameter ("speed"));
```

Parameter names, because of the nature of HTML, are not case sensitive; however, it is good style to make them entirely uppercase or lowercase. Enclose parameter value strings in double quotes if they include embedded spaces. Value strings are case sensitive; their capitalization is maintained regardless of whether you use double quotes.

Exercise: Creating Applets



Exercise objective – In this lab you will become familiar with applet programming in particular the `paint` method used for screen update and refresh.

Preparation

In order to successfully complete this lab, you must be able to display an applet with a browser.

Tasks

In a Web browser view the `lab_files.html` page that is at the top-level of the SL275 directory on your computer. There will be a summary of each exercise and a link to a page that gives a detailed explanation of the exercise.

Go to the SL275 directory on your computer and change to the directory for this module (`mod12`). A listing of this directory will show three subdirectories: one for each of the exercises below.

Exercise 1: Write an Applet (Level 1)

In this exercise you will modify an existing applet, compile it, and view it using the `appletviewer` command.

Exercise 2: Create Concentric Squares (Level 2)

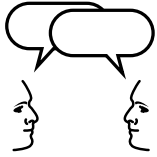
In this exercise you will create an applet that uses graphics methods to draw concentric squares.

Exercise 3: Create a Java Rollover Applet (Level 3)

In this exercise you will create an applet that uses images and audio clips.

Exercise: Creating Applets

Exercise Summary



Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercises.

✓ *If you do not have time to spend on discussion, just highlight the key concepts students should have learned from the lab exercise.*

- Experiences

✓ *Ask students what their overall experiences with this exercise have been. You might want to go over any trouble spots or especially confusing areas at this time.*

- Interpretations

✓ *Ask students to interpret what they observed during any aspects of this exercise.*

- Conclusions

✓ *Have students articulate any conclusions they have reached as a result of this exercise experience.*

- Applications

✓ *Explore with the students how they might apply what they learned in this exercise to situations at their workplace.*

Check Your Progress

Before continuing on to the next module, check to be sure that you can

- Differentiate between a standalone application and an applet
- Write an HTML tag to call a Java applet
- Describe the class hierarchy of the applet and AWT classes
- Create the `HelloWorld.java` applet
- List the major methods of an applet
- Describe and use the painting model of AWT
- Use applet methods to read images and files from URLs
- Handle various mouse events within the applet
- Pass parameters to an applet from an HTML file using the `<param>` tag

Think Beyond

How can you use applets on your company's Web page to improve the overall presentation?

Objectives

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

- Identify the key AWT components and the events that they trigger
- 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
- Use the Java printing mechanism
- Understand how to construct a GUI class that can be used within a Frame or within an Applet

This module covers general topics about constructing applications and applets using GUI presentation.

Relevance

- ✓ **Present the following questions to stimulate the students and get them thinking about the issues and topics presented in this module. They are not expected to know the answers to all of these questions. Hold discussions where students have input; otherwise, if no one can propose answers, begin the lecture for this module.**



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?
- ✓ **There are many GUI components included in the AWT package. Menus are a little special. Frames can contain a menu-bar; a menu-bar can contain zero or more menus; and a menu can contain zero or more menu items (including sub-menus).**

AWT Components

In Module 10, "Building Java GUIs," you were introduced to only one component (Button) and several containers (Panel and Frame). In this section, we will briefly introduce you to the gamut of AWT components. For more information on the look and feel of these components read Appendix C, "The AWT Component Library."

Table 13-1 AWT Component Descriptions

Component Type	Description
Button	A named rectangular box used for receiving mouse clicks.
Canvas	A panel used for drawing.
Checkbox	A component allowing the user to select an item.
CheckboxMenuItem	A checkbox within a menu.
Choice	A pull-down static list of items.
Component	The parent of all AWT components, except menu components.
Container	The parent of all AWT containers.
Dialog	The base class of all modal dialog boxes.
Frame	The base class of all GUI windows with window manager controls.
Label	A text string component.
List	A component that contains a dynamic set of items.
Menu	An element under the menu bar, which contains a set of menu items.
MenuItem	An item within a menu.
Panel	A basic container class used most often to create complex layouts.
Scrollbar	A component which allows a user to "select from a range of values."
ScrollPane	A container class which implements automatic horizontal and/or vertical scrolling for a single child component.
TextArea	A component that allows the user to enter a block of text.
TextField	A component that allows the user to enter a single line of text.
Window	The base class of all GUI windows, with no window manager controls.

AWT Components

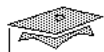
Component Events

Table 13-2 shows the basic AWT components and the event listeners that can be associated with that type of component

Table 13-2 Components and Their Listeners.

Component Type	Act	Adj	Cmp	Cnt	Foc	Itm	Key	Mou	MM	Text	Win
Button	✓		✓		✓		✓	✓	✓		
Canvas			✓		✓		✓	✓	✓		
Checkbox			✓		✓	✓	✓	✓	✓		
CheckboxMenuItem						✓					
Choice			✓		✓	✓	✓	✓	✓		
Component			✓		✓		✓	✓	✓		
Container			✓	✓	✓		✓	✓	✓		
Dialog			✓	✓	✓		✓	✓	✓		✓
Frame			✓	✓	✓		✓	✓	✓		✓
Label			✓		✓		✓	✓	✓		
List	✓		✓		✓	✓	✓	✓	✓		
MenuItem	✓										
Panel			✓	✓	✓		✓	✓	✓		
Scrollbar		✓	✓		✓		✓	✓	✓		
ScrollPane			✓	✓	✓		✓	✓	✓		
TextArea			✓		✓		✓	✓	✓	✓	
TextField	✓		✓		✓		✓	✓	✓	✓	
Window			✓	✓	✓		✓	✓	✓		✓

Act - ActionListener, **Adj** - AdjustmentListener,
Cmp - ComponentListener, **Cnt** - ContainerListener,
Foc - FocusListener, **Itm** - ItemListener, **Key** - KeyListener,
Mou - MouseListener, **MM** - MouseMotionListener,
Text - TextListener, **Win** - WindowListener



How to Create a Menu

1. Create a `MenuBar` object and set it into a menu container such as a `Frame`.
2. Create one or more `Menu` objects and add them to the menu bar object.
3. Create one or more `MenuItem` objects and add them to the menu object.

How to Create a Menu

A Menu is different from other components because you cannot add a Menu to ordinary containers and have them laid out by the layout manager. You can add menus only to a *menu container*. You can start a menu “tree” by putting a menu bar in a `Frame`, using the `setMenuBar()` method. From that point, you can add menus to the menu bar and menus or menu items to the menus.

Pop-up menus are an exception because they appear as floating windows and, therefore, do not require layout.

The Help Menu

Using the menu bar, you can designate one menu to be the Help menu. You do this using the method `setHelpMenu(Menu)`. You must add the menu to be treated as the Help menu to the menu bar; it is then treated in the same way as the Help menu for the local platform. For X/Motif-type systems, this involves flushing the menu entry to the right end of the menu bar.

Creating a MenuBar

A `MenuBar` component is a horizontal menu. You can only add it to a `Frame` object, and it forms the root of all menu trees. A `Frame` displays one `MenuBar` at a time. However, you can change the `MenuBar` based on the state of the program so that different menus appear at various points. For example:

```
1 Frame f = new Frame("MenuBar");
2 MenuBar mb = new MenuBar();
3 f.setMenuBar(mb);
```



Figure 13-1 MenuBar Component

The `MenuBar` does not support listeners. As part of the normal menu behavior, anticipated events that occur in the region of a menu bar are processed automatically.

Creating a Menu

The Menu component provides a basic pull-down menu. You add it either to a MenuBar or to another Menu. For example:

```
1  Frame f = new Frame("Menu");
2  MenuBar mb = new MenuBar();
3  Menu m1 = new Menu("File");
4  Menu m2 = new Menu("Edit");
5  Menu m3 = new Menu("Help");
6  mb.add(m1);
7  mb.add(m2);
8  mb.setHelpMenu(m3);
9  f.setMenuBar(mb);
```

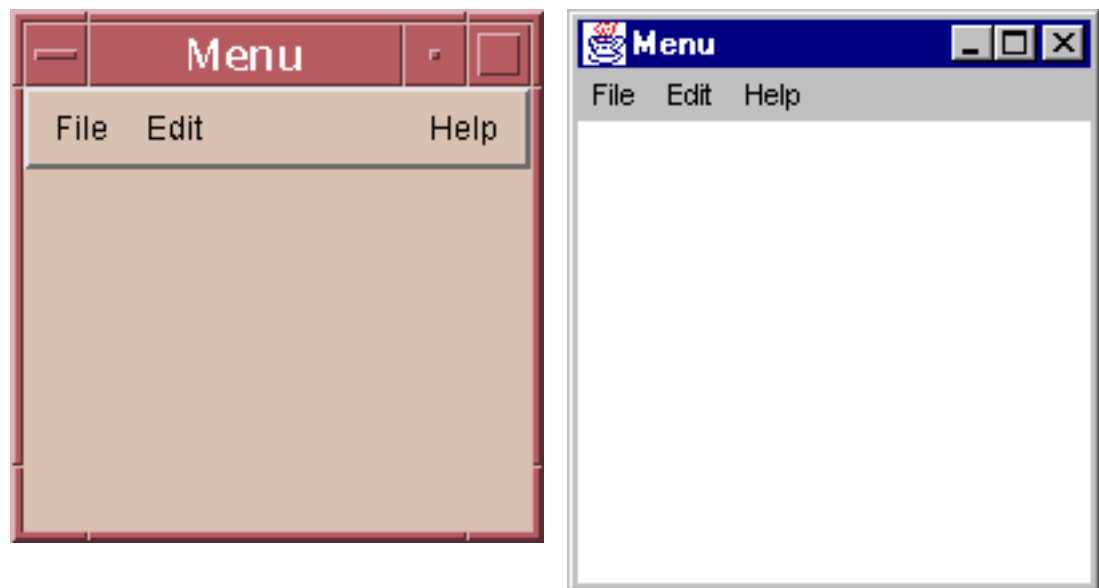


Figure 13-2 Menu Component

Note – The menus shown here are empty, which accounts for the appearance of the File menu.

You *can* add an ActionListener to a Menu object, but this would be unusual. Normally, you use menus to display and control menu items, which are discussed next.

Creating a MenuItem

MenuItem components are the text *leaf* nodes of a menu tree. They are added to a menu to complete it. For example:

```

1 MenuItem mi1 = new MenuItem("New");
2 MenuItem mi2 = new MenuItem("Save");
3 MenuItem mi3 = new MenuItem("Load");
4 MenuItem mi4 = new MenuItem("Quit");
5 mi1.addActionListener(this);
6 mi2.addActionListener(this);
7 mi3.addActionListener(this);
8 mi4.addActionListener(this);
9 m1.add(mi1);
10 m1.add(mi2);
11 m1.add(mi3);
12 m1.addSeparator();
13 m1.add(mi4);

```

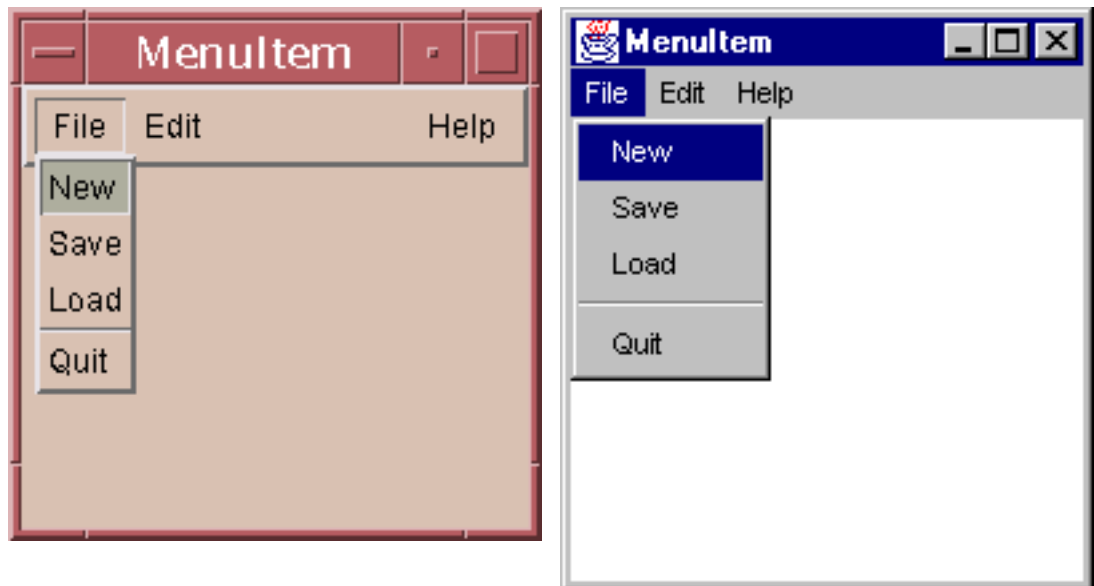


Figure 13-3 MenuItem Component

Usually you add an `ActionListener` to a `MenuItem` object to provide behavior for the menus.

Creating a CheckboxMenuItem

CheckboxMenuItem is a checkable menu item, so you can have selections (*on* or *off* choices) listed in menus. For example:

```
1   MenuBar mb = new MenuBar();
2   Menu m1 = new Menu("File");
3   Menu m2 = new Menu("Edit");
4   Menu m3 = new Menu("Help");
5   mb.add(m1);
6   mb.add(m2);
7   mb.setHelpMenu(m3);
8   f.setMenuBar(mb);
9   .....
10  MenuItem mi2 = new MenuItem("Save");
11  mi2.addActionListener(this);
12  m1.add(mi2);
13  .....
14  CheckboxMenuItem mi5 = new CheckboxMenuItem("Persistent");
15  mi5.addItemListener(this);
16  m1.add(mi5);
```

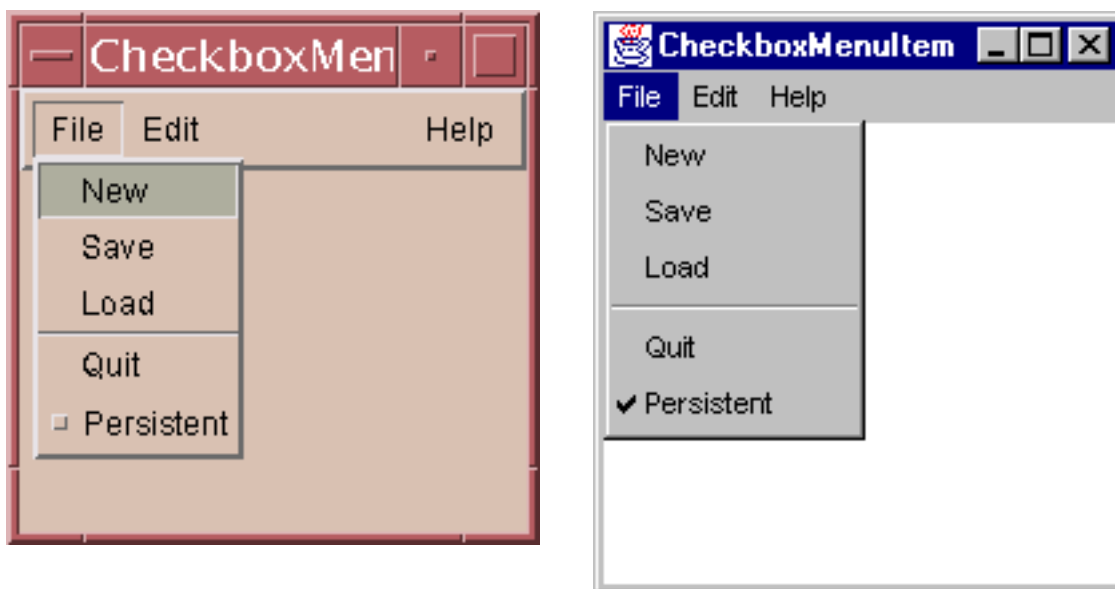


Figure 13-4 CheckboxMenuItem Component

You should monitor the CheckboxMenuItem using the ItemListener interface. The `itemStateChanged` method is called when the checkbox state is modified.



Controlling Visual Aspects

- Colors:
 - ▼ `setForeground()`
 - ▼ `setBackground()`

- Example:

```
int r = 255;  
Color c = new Color(r, 0, 0);
```

Controlling Visual Aspects

You can control the colors used for the foreground and the background of AWT components.

- ✓ **Some platforms do not allow the colors to be changed on specific components. For example, Microsoft Windows does not allow you to alter a button's color.**

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.

Controlling Visual Aspects

Colors (Continued)

In addition, you can create a specific color, such as the following:

```
int r = 255;  
int g = 255;  
int b = 0;  
Color c = new Color(r, g, b);
```

Such a constructor creates a color based on the specified intensities (in a range of 0 to 255 for each) of red, green, and blue.



Controlling Visual Aspects

- Fonts:
 - ▼ You can use the `setFont()` method to specify the font used for displaying text
 - ▼ `Dialog`, `DialogInput`, `Serif`, and `SansSerif` are valid font names

- Example:

```
Font font = new Font("TimesRoman", Font.PLAIN, 14);
```

- Use the `GraphicsEnvironment` class to retrieve the set of all available fonts

```
GraphicsEnvironment ge =  
    GraphicsEnvironment.getLocalGraphicsEnvironment();  
Font[] fonts = ge.getAllFonts();
```

Controlling Visual Aspects

Fonts

You can specify the font used for displaying text in a component by using the method `setFont()`. The argument to this method should be an instance of the `java.awt.Font` class.

No constants are defined for fonts, but you can create a font by specifying the name of the font, the style, and the point size.

```
Font f = new Font("TimesRoman", Font.PLAIN, 14);
```

Standard font names include the following:

- `Font.Dialog`
- `Font.DialogInput`
- `Font.Serif`
- `Font.SansSerif`
- `Font.Monospaced`

Controlling Visual Aspects

Fonts (Continued)

You can determine a full list of fonts by using the following code:

```
GraphicsEnvironment ge =  
    GraphicsEnvironment.getLocalGraphicsEnvironment();  
Font[] fonts = ge.getAllFonts();
```

The following lists the font-style constants, which are actually `int` values:

- `Font.BOLD`
- `Font.ITALIC`
- `Font.PLAIN`
- `Font.BOLD + Font.ITALIC`

You should specify point sizes using an `int` value.

- ✓ ***Java 2D has expanded the available fonts substantially. Java 2D is discussed briefly in Appendix E, "Java Foundation Classes."***



Controlling Visual Aspects

- The `Toolkit` class is an abstract superclass of all actual implementations of the Abstract Window Toolkit
- Subclasses of `Toolkit` are used to bind the various components to particular native toolkit implementations
- Useful methods:

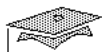
```
getDefaultToolkit  
getImage(String filename)  
getScreenResolution  
getScreenSize  
getPrintJob
```

Controlling Visual Aspects

The Toolkit Class

The `Toolkit` class is an abstract superclass of all platform-specific implementation of the AWT. Subclasses of this class are used to bind the various AWT components to particular native platform implementations. `Toolkit` also supplies a set of useful methods:

- `getDefaultToolkit` – This static method returns the current `Toolkit` object
- `getImage(String filename)` – This method loads an image from a file
- `getScreenResolution` – This method returns the number of "pixels per inch"
- `getScreenSize` – This method returns a `Dimension` object that hold the width and height of the screen in pixels
- `getPrintJob` – This method returns a unique print job object



Printing

- The follow code fragment prints a Frame:

```
1 Frame f = new Frame("Print test");
2 Toolkit toolkit = frame.getToolkit();
3 PrintJob job = toolkit.getPrintJob(frame, "Test Printing", null);
4 Graphics g = job.getGraphics();
5 frame.printComponents(g);
6 g.dispose();
7 job.end();
```

1. Obtain graphics object (line 4).
2. Draw on the graphics object (line 5).
3. Send the graphics object to printer (line 6).
4. End the print job (line 7).

Printing

As of JDK1.1, printing is handled in a fashion that closely parallels screen display. A special kind of `java.awt.Graphics` object is obtained so that any draw instructions sent to that graphic are destined for the printer.

The printing system allows the use of local printer control conventions. When users start a print operation, they see a printer selection dialog box. They choose options, such as paper size, print quality, and which printer to use. For example:

```
1 Frame f = new Frame("Print test");
2 Toolkit toolkit = frame.getToolkit();
3 PrintJob job = toolkit.getPrintJob(frame, "Test Printing", null);
4 Graphics g = job.getGraphics();
5 frame.printComponents(g);
6 g.dispose();
7 job.end();
```

These lines create a `Graphics` object that is “connected” to the printer chosen by the user.

Printing

Obtain the Graphics object from the print job object:

```
g = job.getGraphics();
```

You can use any of the Graphics class's drawing methods to write to the printer. Alternatively, as shown here, you can ask a component to draw itself onto the graphic.

```
f.printComponents(g);
```

The `print()` method asks a component to draw itself in this way, but it only relates to the component for which it has been called. In the case of a container, you can use the `printComponents()` method to draw the container and all of its components on the printer.

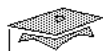
After the page of output is created, use the `dispose()` method to submit that page to the printer. This also frees up any operating system resources that may have been allocated to the print job.

✓ ***This seems strange, but that's what the `PrintJob.getGraphics` method in the API docs says.***

```
g.dispose();
```

When you have completed the job, call the `end()` method on the print job object. This indicates that the print job is complete and allows the printer spooling system to run the job and release the printer for other jobs.

```
job.end();
```



Sun Educational Services

Dual-Purpose Code

- You can write GUI code that can be used as a stand-alone application or an applet
- GUI code can be written as a panel independent of being embedded in a frame or applet

Dual-Purpose Code

With a little bit of effort you can write a GUI class (constituting a complete "application panel") that can be used in two contexts: as a stand-alone application within a frame and as an applet within a browser.

The trick is to encapsulate all of the "application presentation" within a class that implements the top-level GUI layout. This GUI object acts as a *mediator* between all of the GUI components within its content panel. You then create two classes that use that GUI object: one class for the stand-alone application, which launches a frame, and one class that extends `Applet`, which constructs the GUI object in the `init` method and embeds the GUI content panel within the applet's panel.

- ✓ ***There are many ways of creating a GUI that is reusable in the context of a `Frame` or `Applet`. This technique uses a class that implements a GUI on a `Panel` and that panel is then inserted into the frame or applet. This makes much more sense in Swing where both `JFrame` and `JApplet` have a `setContentPane` method.***

Dual-Purpose Code

Example

To demonstrate how this is done we will use a fictitious SalesOrderGUI. Figure 13-5 shows a UML model of this scenario. The SalesOrderApp class implements a stand-alone application that uses a SalesOrderGUI object. Notice that the main and launchFrame methods have been placed in the SalesOrderApp class. The SalesOrderGUI class includes an attribute called contentPanel and a public accessor method called getContentPanel. It is within this method that the GUI is constructed. The SalesOrderApplet class uses a SalesOrderGUI object in the init method.

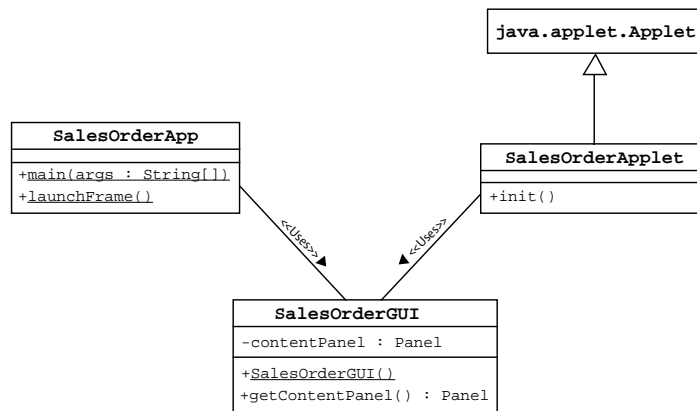


Figure 13-5 The UML Model of a Dual-Purpose GUI Scenario

Dual-Purpose Code

Example (Continued)

The following code fragments show the source code of the SalesOrderGUI class.

```
import java.awt.*;
import java.awt.event.*;

public class SalesOrderGUI {
    // declaration of GUI components
    ...
    private Panel contentPanel = null;

    public SalesOrderGUI() {
        // initialize GUI components
        ...
    }

    public Panel getContentPanel() {
        // return the panel if it has already been created
        if ( contentPanel != null ) {
            return contentPanel;
        }
        contentPanel = new Panel();

        // construction and layout of GUI components
        ...

        // Set up event handling
        ...

        return contentPanel
    }
    // Event handler inner class declarations
    ...
}
```

The key feature of this code is that the complete GUI is contained in the `contentPanel` attribute. This is what is used by the application and applet classes. An instance of `SalesOrderGUI` acts as mediator between its components (such as text fields and buttons) through the event handlers. This mediation is achieved without any help or hindrance from the context that the GUI is running in: stand-alone or applet.

Dual-Purpose Code

Example (Continued)

The following is the complete source code of the `SalesOrderApp` class.

```
1 import java.awt.Frame;
2 import java.awt.BorderLayout;
3 import java.awt.event.WindowAdapter;
4 import java.awt.event.WindowEvent;
5
6 public class SalesOrderApp {
7
8     private static void launchFrame() {
9         SalesOrderGUI salesGUI = new SalesOrderGUI();
10        Frame f = new Frame("SalesOrder");
11
12        f.addWindowListener(new WindowAdapter() {
13            public void windowClosing(WindowEvent event) {
14                System.exit(0);
15            }
16        });
17        f.setSize(200, 200);
18        f.add(salesGUI.getContentPanel(), BorderLayout.CENTER);
19        f.setVisible (true);
20    }
21
22    public static void main(String args[]) {
23        launchFrame();
24    }
25 }
```

`SalesOrderApp` class never needs to be instantiated: All of its methods are static. The main method calls the `launchFrame` method. The `launchFrame` method does all the work. It creates a `SalesOrderGUI` object and a frame. On line 18, it retrieves the content panel of the `SalesOrderGUI` object and places it in the "center" border of the frame. It then makes the frame visible which starts the application running.

Dual-Purpose Code

Example (Continued)

The following is the complete source code of the `SalesOrderApplet` class.

```
1 import java.applet.Applet;
2 import java.awt.BorderLayout;
3
4 public class SalesOrderApplet extends Applet {
5     public void init() {
6         SalesOrderGUI salesGUI = new SalesOrderGUI();
7         setLayout(new BorderLayout());
8         add(salesGUI.getContentPanel(), BorderLayout.CENTER);
9     }
10 }
```

The `SalesOrderApplet` class is instantiated by the appletviewer or browser. When the `init` method is called a `SalesOrderGUI` object is created. The content panel of the `SalesOrderGUI` object is added to the applet's "center" border.

This example was particularly easy for the `SalesOrderApplet` class code. Other cases may need to have special code to start and stop the GUI mediator.



Discussion of Dual-Purpose Code

- Does your business wish to present the same application presentation (the GUI) within the company's intranet as well as over the internet?
- Security issues hinder applet functionality
- Firewall security hinders applet <-> server communication mechanisms (such as raw sockets)
- There are different methods for loading images
- There is some redundant code in the `XxxApp` and `XxxApplet` classes for initializing the GUI and the model

Discussion of Dual-Purpose Code

There are several issues that you should consider in determining whether to construct your GUI classes for dual purposes. The most important issue is whether or not your business wants to present the same look and feel (and functionality) over the internet. Typically a company will want two different sales order forms: one for internal use and one for customers on the Web.

Another issue is Java security. Applets run in a sandbox and do not have access to the same features as a full application does, such as file I/O and network calls to machines other than the server.

- ✓ ***Java 2 security features may allow applets to override the browser's basic security measures.***

Another issue is communication between the client (as an applet) and the application or database server. Typically such connections use non-HTTP protocols, which might not be available behind the client browser's firewall.



Sun Educational Services

Swing

- Swing is a second-generation GUI toolkit
- It builds on top of AWT, but supplants the components with "light-weight" versions
- There are several more components: JTable, JTree, and JComboBox

Swing

AWT is the predominate GUI toolkit for Java technology development. However, as of Java 2 SDK there is another option: Swing. Swing (as part of the Java Foundation Classes) is a second-generation GUI toolkit that is included in Java 2 SDK as a standard extension. Swing has many improvements over AWT. We will only cover a few of these here.

Swing builds on top of AWT (using Color and Font, and so on) but it implements its component classes without using the platform-dependent peers that are used in AWT components. This makes Swing components "light-weight." Swing also adds a variety of new components including a table and tree component.

- ✓ **You may want to have students run the Swing Demo:**

```
java -cp . SwingSet  
java -jar SwingSet.jar
```

- ✓ **This would be a good time to tell students about the Sun course SL320: "GUI Construction in Java Foundation Classes" which focuses on Swing and other JFC APIs.**

Exercise: Building GUI-Based Applications



Exercise objective – You will write, compile, and run the revised ChatClient GUI which includes menus. You will also rewrite the GUI code to be used both as a stand-alone application and as an applet.

Preparation

In order to successfully complete this lab, you must have a clear understanding of constructing a menu and of dual-purpose code construction.

Tasks

In a Web browser view the `lab_files.html` page that is at the top-level of the SL275 directory on your computer. There will be a summary of each exercise and a link to a page that gives a detailed explanation of the exercise.

Go to the SL275 directory on your computer and change to the directory for this module (`mod13`). A listing of this directory will show two subdirectories: one for each of the exercises below.

Exercise 1: Finish the ChatClient GUI (Level 1 Lab)

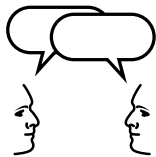
In this exercise you will finish the GUI for a "chat room" application. You will add menus to it and use a dialog box.

Exercise 2: Create Dual Code for the Calculator GUI (Level 2 Lab)

In this exercise you will convert the Calculator GUI to be dual-purpose: existing both as a stand-alone application and as an applet.

Exercise: Working With Events

Exercise Summary



Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercises.

- ✓ *If you do not have time to spend on discussion, just highlight the key concepts students should have learned from the lab exercise.*

- Experiences

- ✓ *Ask students what their overall experiences with this exercise have been. You might want to go over any trouble spots or especially confusing areas at this time.*

- Interpretations

- ✓ *Ask students to interpret what they observed during any aspects of this exercise.*

- Conclusions

- ✓ *Have students articulate any conclusions they have reached as a result of this exercise experience.*

- Applications

- ✓ *Explore with the students how they might apply what they learned in this exercise to situations at their workplace.*

Check Your Progress

Before continuing on to the next module, check to be sure that you can

- Identify the key AWT components and the events that they trigger
- 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
- Use the Java printing mechanism
- Understand how to construct a GUI class that can be used within a Frame or within an Applet

Think Beyond

What problems occur when your GUI code must wait for the application logic to perform its job?

What are the limitation of AWT?

Objectives

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

- Define a thread
- Create separate threads in a Java software 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
- Explain why `suspend`, `resume`, and `stop` methods have been deprecated in JDK 1.2

This module covers multithreading, which allows a program to do multiple tasks at the same time.

Relevance

- ✓ **Present the following question to stimulate the students and get them thinking about the issues and topics presented in this module. They are not expected to know the answer to this question. Hold discussions where students have input; otherwise, if no one can propose answers, begin the lecture for this module.**



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

- How do you get programs to perform multiple tasks concurrently?
-
- ✓ **There are times when a program needs to do multiple tasks concurrently. For example, a Web browser displaying a sports Web page runs a Java applet spinning a football in the corner, but leaves the scrollbar active so the Web page can be traversed at the same time. The Web browser is multithreaded, allowing this flexibility; without multithreading, the browser would be totally preoccupied with spinning the football and you would never be able to scroll the Web page. This module discusses how to set up multiple threads in a single program, so that you can do multiple tasks concurrently.**
 - ✓ **Multithreading is useful even in programs running on a single-processor computer. In this scenario, thread two runs while thread one is blocked waiting for I/O. The I/O can take place in parallel with the CPU running the other thread. In a single-threaded case, thread two would have to wait until thread one finished.**
 - ✓ **Data corruption can result if two independent threads access the same data at the same time. Threads running independently yet sharing common data need to be coordinated. Furthermore, the timing of when one thread runs might depend on the other thread, as in a producer/consumer problem. This module discusses the ramifications of sharing data between threads, and discusses how to coordinate multiple threads sharing the same data.**



Threads

- What are threads?
 - ▼ Virtual CPU

Threads

What Are Threads?

A simplistic view of a computer is that it has a CPU that performs computations, read-only memory (ROM) which contains the program that the CPU executes, and random-access memory (RAM) which holds the data on which the program operates. In this view, there is only one job being performed. A more complete view of most modern computer systems allows for the possibility of performing more than one job at the same time.

You do not need to be concerned with how this 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` allows 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.



Three Parts of a Thread

- CPU
- Code
- Data

Threads in Java Programming

Three Parts of a Thread

A thread or *execution context* is composed of three main parts:

- A virtual CPU
- The code the CPU is executing
- The data on which the code works

Threads in Java Programming

Three Parts of a Thread (Continued)

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 14-1.

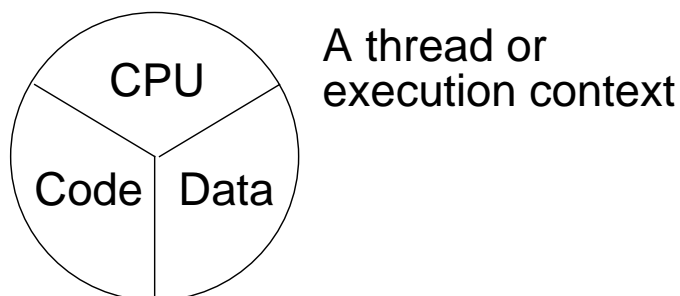


Figure 14-1 A Thread

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 may or may not 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.

Threads in Java Programming

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).



Creating the Thread

- Multithreaded programming:
 - ▼ Multiple threads from the same `Runnable` instance
 - ▼ Threads share the same data and code
- Example:

```
Thread t1 = new Thread(r);  
Thread t2 = new Thread(r);
```

Threads in Java Programming

Creating the Thread (Continued)

A multithreaded programming environment allows 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.

Threads in Java Programming

Creating the Thread (Continued)

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.

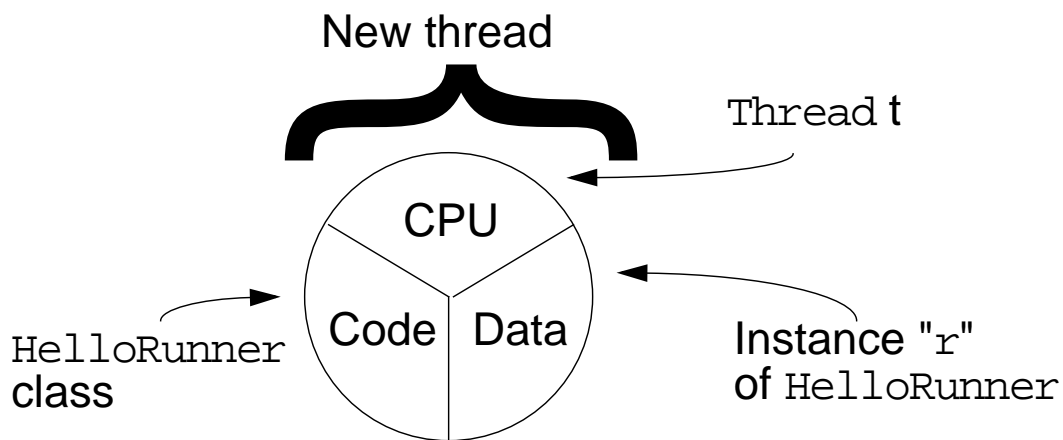
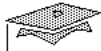


Figure 14-2 Thread Creation



Sun Educational Services

Starting the Thread

- Using the `start` method
- Placing the thread in runnable state

Threads in Java Programming

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 after line 4 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.

Threads in Java Programming

Thread Scheduling

In Java technology, threads are usually *preemptive*, but not necessarily timesliced (the process of giving each thread an equal amount of CPU time). It is a common mistake to believe that "preemptive" is a fancy word for "does timeslicing."

- ✓ ***For the runtime on a Solaris Operating Environment platform, Java technology does not preempt threads of the same priority. However, the runtime on Microsoft Windows platforms uses timeslicing, so it preempts threads of the same priority and even threads of higher priority. Preemption is not guaranteed; however, most JVM implementations result in behavior that appears to be strictly preemptive. Across JVM implementations, there is no absolute guarantee of preemption or timeslicing. The only guarantees lie in the coder's use of `wait` and `sleep`.***

The model of a preemptive scheduler is that many threads might be runnable, but only one thread is actually running. This thread continues to run until it ceases to be runnable or another thread of higher priority becomes runnable. In the latter case, the lower priority thread is *preempted* 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, deliberately asking the thread to pause 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.

- ✓ ***The last sentence is worded loosely because: (1) In most JVM implementations, priorities seem to work in a preemptive manner, although there is no guarantee that priorities have any meaning at all; and (2) Microsoft Window's nice values affect thread behavior so that it is possible that a Java priority 4 thread might be running, in spite of the fact that a runnable Java priority 5 thread is waiting for the CPU.***
- ✓ ***In reality, many JVMs implement pools as queues, but this is not guaranteed behavior.***

Threads in Java Programming

Thread Scheduling (Continued)

A Thread object can exist in several different states throughout its lifetime. Figure 14-3 illustrates this idea.

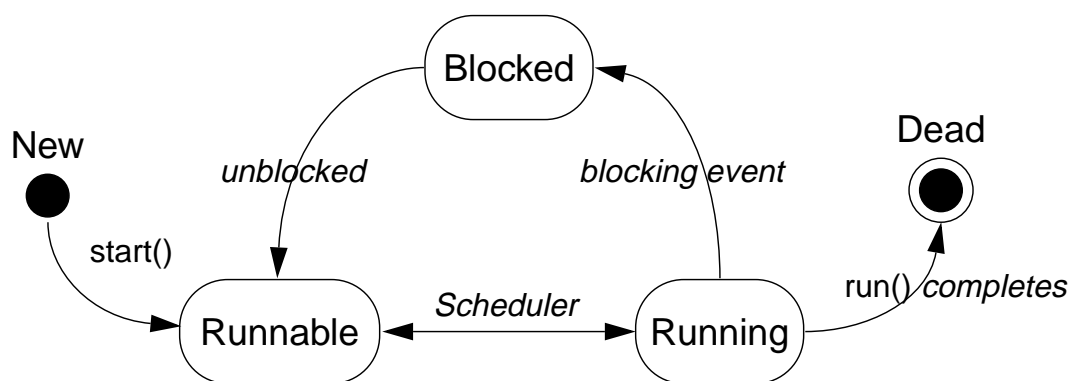


Figure 14-3 Basic Thread States Transition Diagram

- ✓ **With JDK1.2, the `suspend`, `resume`, and `stop` methods have been deprecated. `suspend` is deadlock prone and `stop` is unsafe in terms of data protection.**

Although the thread becomes runnable, it does not necessarily start running immediately. Only one action at a time is performed on a machine with one CPU. The following section describes how the CPU is allocated when more than one thread is runnable.

Threads in Java Programming

Thread Scheduling (Continued)

Given that Java threads are not necessarily timesliced, 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.

```
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 }
```

✓ **You can interrupt a thread with a `Thread.interrupt()` method.**

This code example shows how the `try` and `catch` block is used. `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`.

`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.

Basic Control of Threads

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             ...
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 }
```

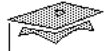
✓ **The method `stop` of the `Thread` class has been deprecated in JDK 1.2.**

Basic Control of Threads

Terminating a Thread (Continued)

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(
8             "Thread " + Thread.currentThread().getName() + " completed");
9     }
10 }
```



Basic Control of Threads

- Testing threads:
 - ▼ `isAlive()`
- Thread priority:
 - ▼ `getPriority()`
 - ▼ `setPriority()`
- Putting threads on hold:
 - ▼ `Thread.sleep()`
 - ▼ `join()`
 - ▼ `Thread.yield()`

Basic Control of Threads

Testing a Thread

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
```

Basic Control of Threads

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 (a) the thread "waking up" is of a higher priority, or (b) the running thread blocks for some other reason.

- ✓ ***The methods `suspend` and `resume` of the `Thread` class have been deprecated in JDK 1.2. These methods, plus the `stop` method, have been deprecated for essentially the same reason: They do not work well within the Java technology's model of object synchronization.***

Basic Control of Threads

Putting Threads on Hold (Continued)

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 }
```

You can also call the join method with a timeout value in milliseconds. For example:

```
void join(long timeout);
```

where the join method either suspends the current thread for *timeout* milliseconds or until the thread it calls on terminates.

Basic Control of Threads

Putting Threads on Hold (Continued)

The Thread.yield() Method

Use the method `Thread.yield()` to give other threads of the same priority a chance to execute. If other threads at the same priority are runnable, `yield` places the calling thread into the runnable pool and allows another thread to run. If no other threads are runnable at the same priority, `yield` does nothing.

A `sleep` call gives threads of lower priority a chance to execute. The `yield` method gives only threads of the same priority a chance to execute.

- ✓ ***This is not necessarily true for a timesliced operating system. A timesliced operating system gives threads at a lower priority a chance to execute, and because a higher priority thread might exist and not be executing, a timesliced operating system can also give a higher priority thread a chance to execute as a result of the `yield` method being called.***

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 (running) {
4             // do lots of interesting stuff
5             try {
6                 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

- Implementing `Runnable`:
 - ▼ Better object-oriented design
 - ▼ Single inheritance
 - ▼ Consistency
- Extending `Thread`:
 - ▼ Simpler code

Other Ways to Create Threads

Selecting a Way to Create Threads

Given a choice of approaches, how can you decide between them? Each approach has its advantages that are described in this next section.

Other Ways to Create Threads

Selecting a Way to Create Threads (Continued)

Advantages of Implementing Runnable

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 are changing or extending 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 allows only single inheritance, you cannot extend any other class, such as Applet, if you have already extended Thread. 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.

Advantage of Extending Thread

The advantage of extending Thread is that when a run method is embodied in a class that extends the Thread class. This makes for slightly more simple code.

Note – While both techniques are possible, you should consider very carefully why you would extend Thread. Do so only while you are changing or extending the behavior of a thread, not just implementing a run method.

Exercise: Using Basic Threads



Exercise objective – In this lab you will become familiar with the concepts of multithreading by writing a simple multithreaded program.

Preparation

To successfully complete this lab, you must understand how to create, run and terminate a thread.

Tasks

In a Web browser view the `lab_files.html` page that is at the top-level of the SL275 directory on your computer. There will be a summary of each exercise and a link to a page that gives a detailed explanation of the exercise.

Go to the SL275 directory on your computer and change to the directory for this module (`mod14`). A listing of this directory will show three subdirectories. This one will be in the directory called `exercisel`.

Exercise 1: Use Basic Threads (Level 1)

In this exercise you will have hands-on experience in the creation and running of threads.

Using synchronized in Java Technology

✓ **Use synchronized in your code to prevent data corruption.**

This section describes the use of the `synchronized` keyword. It provides the Java programming language with a mechanism that allows 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 is used to generate this information.

Using synchronized in Java Technology

The Problem (Continued)

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 preempted. At this point, the data model represented in the object is inconsistent.

```
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
```

Using synchronized in Java Technology


The Problem (Continued)

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.

Note – One approach would be to prevent thread a from being switched out until it had completed the critical section of code. This approach is common in low-level machine programming but is generally inappropriate in multi-user systems.

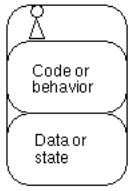
Note – Another approach, and the one on which Java technology works, is to provide a mechanism to treat the data *delicately*. This approach allows a thread atomic access to data regardless of whether that thread gets switched out in the middle of performing that access.


Sun Educational Services

The Object Lock Flag

- Every object has a flag that can be thought of as a "lock flag"
- `synchronized` allows interaction with the lock flag

Object this



Thread before `synchronized(this)`

```

public void push(char c) {
    synchronized (this) {
        data[idx] = c;
        idx++;
    }
}
                    
```

Using `synchronized` in Java Technology

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 allows 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++;
        }
    }
    ...
}
    
```

Using synchronized in Java Technology

The Object Lock Flag (Continued)

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 14-4)

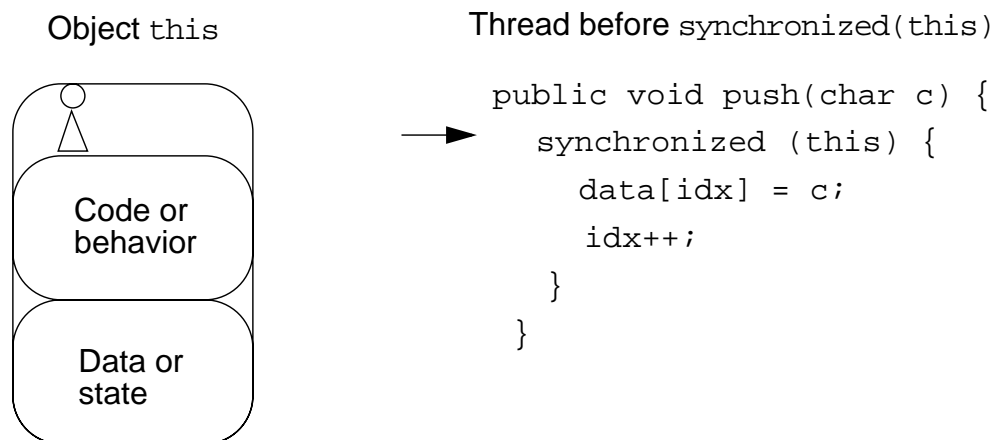


Figure 14-4 Using the synchronized Statement Before a Thread

Using synchronized in Java Technology

The Object Lock Flag (Continued)

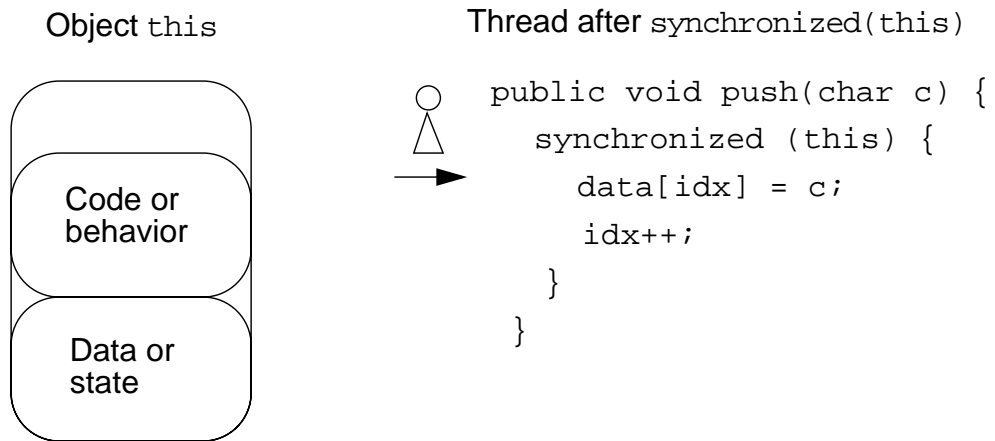


Figure 14-5 Using the synchronized Statement After a Thread

You should realize that this has not protected 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.

Using synchronized in Java Technology

The Object Lock Flag (Continued)

Figure 14-6 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.

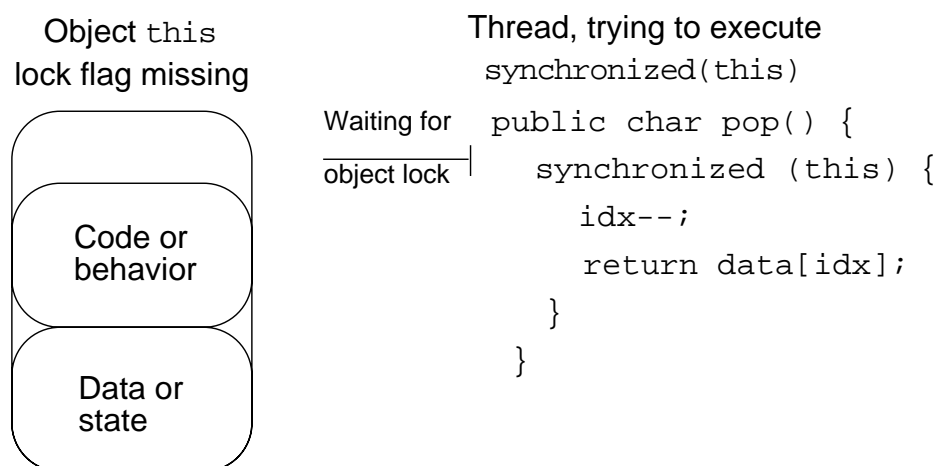


Figure 14-6 Thread Trying to Execute `synchronized`

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

- Released when the thread passes the end of the `synchronized` code block
- Automatically released when a break or exception is thrown by the `synchronized` code block

Using `synchronized` in Java Technology

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 or break 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 correctly released 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.



Sun Educational Services

synchronized – Putting It Together

- *All* access to delicate data should be synchronized
- Delicate data protected by synchronized should be private

Using synchronized in Java Technology

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*. Consider the accessibility of the data items that form the delicate parts of the object. If these are not marked as *private*, they can be accessed from code outside the class definition; therefore, other programmers must not omit the protections that are required.

Using synchronized in Java Technology

synchronized – Putting It Together (Continued)

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) {
        :
        :
    }
}
```

```
public synchronized void push(char c) {
    :
    :
}
```

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 allows users of the method to know, from javadoc-generated documentation, that synchronization is taking place. This can be important when designing against deadlock (which is discussed in the next 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.

Using synchronized in Java Technology

Thread States

Synchronization is a special thread state. Figure 14-7 illustrates the new state transition diagram for a thread.

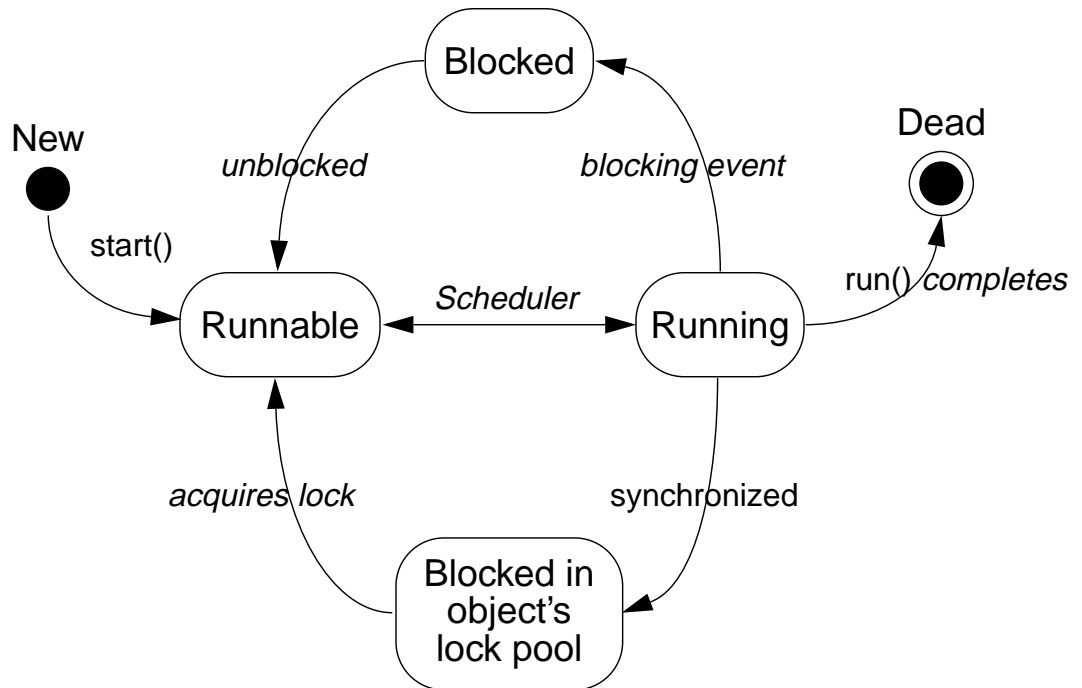


Figure 14-7 Thread States Transition Diagram With Synchronization



Deadlock

- Is two threads, each waiting for a lock from the other
- Is not detected or avoided
- Can be avoided by:
 - ▼ Deciding on the order to obtain locks
 - ▼ Adhering to this order throughout
 - ▼ Releasing locks in reverse order

Using synchronized in Java Technology

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

- Scenario:
 - ▼ Consider yourself and a cab driver as two threads
- The problem:
 - ▼ How to determine when you are at your destination
- The solution:
 - ▼ You notify the cabbie of your destination and relax
 - ▼ Cabbie drives and notifies you upon arrival at your destination

Thread Interaction – wait and notify

- ✓ **Use wait and notify to communicate between threads. If multiple threads are waiting for the same rendezvous object, then they must all be waiting for the same condition.**

Different threads are created specifically to perform unrelated tasks. However, sometimes the jobs they perform are actually 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 cabbie wants to take on a passenger to make a fare. So, each of you has a task.

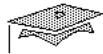
Thread Interaction – wait and notify

The Problem

You expect to get into a cab and rest comfortably until the cabbie notifies you that you have arrived at your destination. It would be annoying, for both you and the cabbie, to ask every 2 seconds, “are we there yet?” Between fares, the cabbie wants to sleep in the cab until a passenger needs to be driven somewhere. The cabbie 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 cabbie require some way of communicating your needs to each other. While you are busy walking down the street toward the cab stand, the cabbie is sleeping peacefully in the cab. When you notify the cabbie that you want a ride, the cabbie wakes up and begins driving, and you get to relax. Once you have arrived at your destination, the cabbie notifies you to get out of the cab and go to work. The cabbie now gets to wait and nap again until the next fare comes along.



Thread Interaction

- wait and notify
- The pools:
 - ▼ Wait pool
 - ▼ Lock pool

Thread Interaction

wait *and* notify

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 cabbie waiting in the cab translates to the “cabbie” 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.

Thread Interaction

wait and notify (Continued)

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` automatically releases that object's lock flag. You can invoke different `wait` methods.

```
wait or wait(long timeout)
```

- ✓ **If a thread that calls `wait` happens to be holding more than one object's lock flag, only the lock flag for the object associated with the `wait` call is released.**

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 allows the thread to continue running where it left off when it called `wait`.

In many systems that implement the `wait/notify` mechanism, the thread that wakes up is the one that has been waiting the longest. However, Java technology does not guarantee this.

A `notify` call can be issued 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 Interaction

Thread States

The "wait pool" is also a special thread state. Figure 14-8 illustrates the final state transition diagram for a thread.

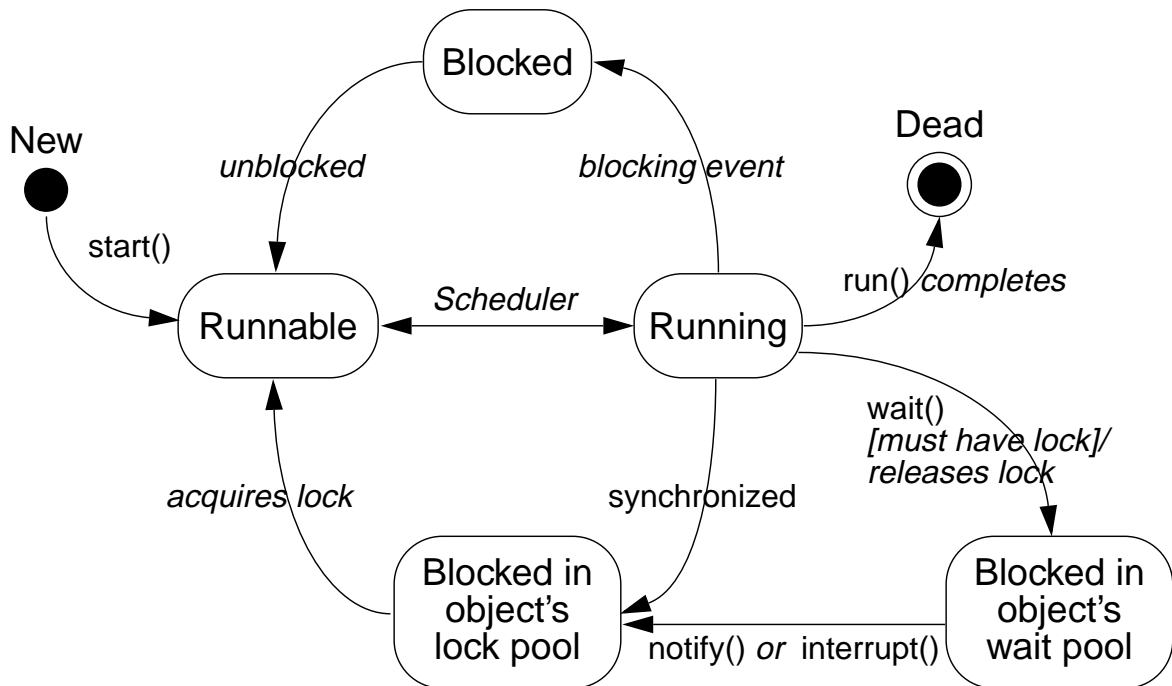


Figure 14-8 Thread States Transition Diagram With wait/notify



Monitor Model for Synchronization

- Leave shared data in a consistent state
- Ensure programs cannot deadlock
- Do not put threads expecting different notifications in the same wait pool

Thread Interaction

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 following code 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();
```

Putting It Together

Producer

The producer thread runs the following method:

```
1  public void run() {
2      char c;
3
4      for (int i = 0; i < 200; i++) {
5          c = (char)(Math.random() * 26 + 'A');
6          theStack.push(c);
7          System.out.println("Producer" + num + ": " + c);
8          try {
9              Thread.sleep((int)(Math.random() * 300));
10         } catch (InterruptedException e) {
11             // ignore it
12         }
13     }
14 }
```

This example generates 200 random uppercase characters and pushes them onto the stack with a random delay of 0 to 300 milliseconds between each push. Each pushed character is reported on the console, along with an identifier for which producer thread is executing.

Putting It Together

Consumer

The consumer thread runs the following method:

```
1 public void run() {
2     char c;
3     for (int i = 0; i < 200; i++) {
4         c = theStack.pop();
5         System.out.println("Consumer" + num + ": " + c);
6
7         try {
8             Thread.sleep((int)(Math.random() * 300));
9         } catch (InterruptedException e) { }
10
11     }
12 }
```

This example collects 200 characters from the stack, with a random delay of 0 to 300 milliseconds between each attempt. Each popped 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.

- ✓ ***Designing code that also communicates information about the stack being full is more complex. It requires multiple rendezvous objects and the use of semaphores. Time constraints prevent discussing such a complex example here.***
- ✓ ***You might remember the producer/consumer example from the older versions of this course. This code worked because the test harness used only one producer thread and one consumer thread. The `notify` calls were made on the single stack object, for different reasons and whether the stack was full or empty. This setup is inherently flawed. As soon as multiple producer and consumer threads are introduced, deadlock occurs.***

Putting It Together

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 buffer = new ArrayList(400);  
  
    public synchronized char pop() {  
    }  
  
    public synchronized void push(char c) {  
    }  
}
```

There are no constructors. It is considered good style to include a constructor, but it has been omitted here for brevity.

Putting It Together

SyncStack Class (Continued)

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.

The pop method is as follows:

```
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 = ((Character)buffer.remove(buffer.size()-1)).charValue();
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. It must wait if its wait is interrupted and the stack is still empty.

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`.

Putting It Together

SyncStack Class (Continued)

The `push` method uses `this.notify()` to release a thread from the stack object's wait pool. Once 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 modifications are made to the stack's shared data. This is important, because the data *must* be in a consistent state before the object's lock is released and a thread's execution changes the stack's data.

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 is similar; 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.

The `push` method is as follows:

```
public synchronized void push(char c) {
    this.notify();
    Character charObj = new Character(c);
    buffer.addElement(charObj);
}
```

Putting It Together

SyncStack Class (Continued)

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 actually gets 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.

SyncStack Example

Complete Code

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*.

The following is an example of the `SyncTest.java` application code:

```
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 constT1 = new Thread (c1);
19        constT1.start();
20
21        Consumer c2 = new Consumer(stack);
22        Thread constT2 = new Thread (c2);
23        constT2.start();
24    }
25 }
```


SyncStack Example

Complete Code (Continued)

The following is an example of the `Producer.java` application code:

```
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        for (int i = 0; i < 200; i++) {
16            c = (char)(Math.random() * 26 +'A');
17            theStack.push(c);
18            System.out.println("Producer" +num+ ": " +c);
19            try {
20                Thread.sleep((int)(Math.random() * 300));
21            } catch (InterruptedException e) {
22                // ignore it
23            }
24        }
25    }
26 }
```

SyncStack Example

Complete Code (Continued)

The following is an example of the `Consumer.java` application code:

```
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
23        }
24    }
25 }
```

SyncStack Example

Complete Code (Continued)

The following is an example of the SyncStack.java application code:

```
1 package mod13;
2
3 import java.util.*;
4
5 public class SyncStack {
6     private List buffer = new ArrayList(400);
7
8     public synchronized char pop() {
9         char c;
10        while (buffer.size() == 0) {
11            try {
12                this.wait();
13            } catch (InterruptedException e) {
14                // ignore it...
15            }
16        }
17        c = ((Character)buffer.remove(buffer.size()-1)).
18            charValue();
19        return c;
20    }
21
22    public synchronized void push(char c) {
23        this.notify();
24        Character charObj = new Character(c);
25        buffer.addElement(charObj);
26    }
27 }
```

SyncStack Example

Complete Code (Continued)

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
```



The suspend and resume Methods

- Have been deprecated in JDK 1.2
- Should be replaced with `wait` and `notify`

Thread Control in Java 2 SDK

The suspend and resume Methods

The methods `suspend` and `resume` have been deprecated as of JDK 1.2. The `resume` method's sole purpose is to unsuspend threads, so without `suspend`, there is no longer a need for `resume`. `suspend` is inherently dangerous from a design perspective for two reasons: It is deadlock-prone, and it allows one thread to have direct control over another thread's code execution.

Assume that you have two threads, `threadA` and `threadB`. While executing its code, `threadB` obtains the lock on an object and then continues with its task. `threadA`'s executing code calls `threadB.suspend()`, which causes `threadB` to stop executing its code.

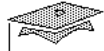
Thread Control in Java 2 SDK

The suspend and resume Methods (Continued)

Deadlock can occur because `threadB.suspend()` does not release the lock that *threadB* is holding. If the thread that called `threadB.suspend()` requires the lock that *threadB* is holding, the two threads are deadlocked.

Assume *threadA* calls `threadB.suspend()`. If *threadA* is in control when *threadB* is suspended, *threadB* never gets the opportunity to manipulate the shared data into a steady state. Only *threadB* should determine when to stop executing its own code.

Rather than use `suspend` and `resume`, control your threads using `wait` and `notify` mechanisms on a rendezvous object. This forces the thread to determine when to “suspend” itself by executing a `wait` call. This causes the rendezvous object’s lock to be released automatically, and gives the thread an opportunity to stabilize any data before calling `wait`.



The stop Method

- Releases the lock before it terminates
- Can leave shared data in an inconsistent state
- Should be replaced with `wait` and `notify`

Thread Control in Java 2 SDK

The stop Method

The situation with the `stop` method is similar, but with different consequences. When a thread that holds an object lock is stopped, it releases the lock that it holds before it terminates. This avoids the deadlock problem discussed previously, but it introduces another problem.

In the previous example, if the thread is stopped after the character has been added to the stack but before the index value is incremented, you have an inconsistent stack structure when the lock is released.

There are always certain critical operations that must be executed atomically, and stopping a thread that is executing one of those operations defeats the automation of the operation.

Thread Control in Java 2 SDK

The stop Method (Continued)

Another issue about stopping threads involves general thread design strategies. You can create a thread to do one particular job and live for the life of the entire program. In other words, you do not design your program so that it creates and disposes of threads arbitrarily or creates endless numbers of dialogs or socket endpoints. Each thread takes system resources that are not infinitely available. This does not imply that a thread must execute continuously, it means that you should use the proper, safe mechanisms of `wait` and `notify` for thread control.

Proper Thread Control

Now that you know how to design your threads to behave well and communicate using `wait` and `notify` calls, and not to rely on `suspend` or `stop`, examine the following code. The `run` method shown ensures that shared data are in a consistent state before execution is paused or terminated.

- ✓ ***It has come to my attention that the following code may not in fact work as advertised. I did not have the time to investigate these issues. Consider this a fair warning that you might want to steer away from this topic.***

Thread Control in Java 2 SDK

Proper Thread Control (Continued)

```
1 public class ControlledThread extends Thread {
2     static final int SUSP = 1;
3     static final int STOP = 2;
4     static final int RUN = 0;
5     private int state = RUN;
6
7     public synchronized void setState(int s) {
8         state = s;
9         if ( s == RUN ) {
10            notify();
11        }
12    }
13
14    public synchronized boolean checkState() {
15        while ( state == SUSP ) {
16            try {
17                wait();
18            } catch (InterruptedException e) {
19                // ignore
20            }
21        }
22        if ( state == STOP ) {
23            return false;
24        }
25        return true;
26    }
27
28    public void run() {
29        while ( true ) {
30            doSomething();
31
32            // Be sure shared data is in consistent state in
33            // case the thread is waited or marked for exiting
34            // from run()
35            if ( !checkState() ) {
36                break;
37            }
38        }
39    }
40 }
```

Thread Control in Java 2 SDK

Proper Thread Control (Continued)

A thread that wants to suspend, resume, or stop a controlled thread calls that thread's `setState` method with the appropriate value, and when the thread determines that it is safe to do so, it suspends (by using the `wait` method) or stops (by exiting from the `run` method) itself.

A more detailed discussion of this problem is beyond the scope of this module.

Exercise: Using Multithreaded Programming



Exercise objective – In this lab you will become familiar with the concepts of multithreading by writing two multithreaded programs, one of which is an applet.

Preparation

To successfully complete this lab, you must understand the concepts of multithreading as presented in this module.

Tasks

In a Web browser view the `lab_files.html` page that is at the top-level of the SL275 directory on your computer. There will be a summary of each exercise and a link to a page that gives a detailed explanation of the exercise.

Go to the SL275 directory on your computer and change to the directory for this module (`mod14`). A listing of this directory will show three subdirectories. These two will be found in the directories called `exercise2` and `exercise3`.

Exercise 2: Use Threads in Applet Animation (Level 2)

In this exercise you will use threads to control the animation within an applet.

Exercise 3: Use Advanced Thread Control (Level 3)

In this exercise you will use `wait/notify` and proper thread control to create a printer manager. You will create several threads that generate print jobs and a single print manager thread to process those jobs.

Exercise: Using Multithreaded Programming

Exercise Summary



Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercises.

✓ *If you do not have time to spend on discussion, just highlight the key concepts students should have learned from the lab exercise.*

- Experiences

✓ *Ask students what their overall experiences with this exercise have been. You might want to go over any trouble spots or especially confusing areas at this time.*

- Interpretations

✓ *Ask students to interpret what they observed during any aspects of this exercise.*

- Conclusions

✓ *Have students articulate any conclusions they have reached as a result of this exercise experience.*

- Applications

✓ *Explores with the students how they might apply what they learned in this exercise to situations at their workplace.*

Check Your Progress

Before continuing on to the next module, check to be sure that you can

- Define a thread
- Create separate threads in a Java software 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
- Explain why `suspend`, `resume`, and `stop` methods have been deprecated in JDK 1.2

Think Beyond

Do you have applications that could benefit from being multithreaded?

Objectives

At the end of this module, you should be able to:

- Describe the main features of the `java.io` package
- Construct node and processing streams, and use them appropriately
- Distinguish readers and writers from streams, and select appropriately between them
- Use the `Serialization` interface to encode the state of an object

This module examines how the Java programming language uses streams to handle byte, character, and object I/O. It also discusses node (source and sink) streams as well as processing streams.

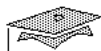
Relevance

- ✓ **Present the following questions to stimulate the students and get them thinking about the issues and topics presented in this module. They are not expected to know the answer to these questions. Hold discussions where students have input; otherwise, if no one can propose answers, begin the lecture for this module.**



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

- What mechanisms are in place within the Java programming language to read and write from sources (or sinks) other than files?
 - How are international character sets supported in I/O operations?
 - What are the possible sources and sinks of character and byte streams?
- ✓ **Input can come from more places than just a GUI, and output can go to more places than just the screen or console. Often a program needs to save its state or some data to a file, or communicate over the network to a server. This module describes the framework of the Java platform's stream I/O mechanism. The framework's flexibility is demonstrated by showing how it is easily adapted to various I/O scenarios. This module shows how whole objects are saved and restored, allowing them to be instantiated without being reinitialized. Other handy file-related classes are also covered.**



I/O Fundamentals

- A *stream* can be thought of as a flow of data from a source to a sink
- A *source* stream initiates the flow of data, also called an input stream
- A *sink* stream terminates the flow of data, also called an output stream
- Sources and sinks are both *node streams*
- Types of node streams are: files, memory, and pipes between threads or processes

I/O Fundamentals

A *stream* is a flow of data from a *source* to a *sink*. Typically, your program will be one end of that stream and some other node (for example, a file) will be 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 15-1 Fundamental Stream Classes

	byte streams	character streams
source streams	InputStream	Reader
sink streams	OutputStream	Writer



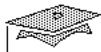
Data Within Streams

- Java technology supports two types of streams: character and byte
- Input and output of character data is handled by readers and writers
- Input and output of byte data is handled by input streams and output streams
 - ▼ Normally, the term stream refers to a byte stream
 - ▼ The terms reader and writer refer to character streams

I/O Fundamentals

Java technology supports two types of data in streams: raw bytes or Unicode characters. Normally, the term "stream" is used to refer 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.



InputStream Methods

- The three basic read methods:

```
int read()
int read(byte[] buffer)
int read(byte[] buffer, int offset, int length)
```

- The other methods:

```
void close()
int available()
skip(long n)
boolean markSupported()
void mark(int readlimit)
void reset()
```

Byte Streams

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 `-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 subrange in the target array that needs to be filled.

Note – For efficiency, always read data in the largest practical block.

Byte Streams

InputStream *Methods (Continued)*

```
void close()
```

When you have finished with a stream, close it. If you have a “stack” of streams, using filter streams, close the stream at the top of the stack. This operation also closes the lower streams.

```
int available()
```

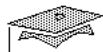
This method reports the number of bytes that are immediately available to be read from the stream. An actual read operation following this call might return more bytes.

```
skip(long n)
```

This method discards the specified number of bytes from the stream.

```
boolean markSupported()  
void mark(int readlimit)  
void reset()
```

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 is used to indicate 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.



OutputStream Methods

- The three basic `write` methods:

```
void write(int c)
void write(byte[] buffer)
void write(byte[] buffer, int offset, int length)
```

- The other methods:

```
void close()
void flush()
```

Byte Streams

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 allows you to force writes.



Reader Methods

- The three basic read methods:

```
int read()
int read(char[] cbuf)
int read(char[] cbuf, int offset, int length)
```

- The other methods:

```
void close()
boolean ready()
skip(long n)
boolean markSupported()
void mark(int readAheadLimit)
void reset()
```

Character Streams

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 `-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 subrange in the target array that needs to be filled.

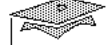
Note – Like for input streams, use the largest practical block for efficiency.

Character Streams

Reader Methods (Continued)

```
void close()  
boolean ready()  
skip(long n)  
boolean markSupported()  
void mark(int readAheadLimit)  
void reset()
```

These methods are analogous to the input stream versions.



Writer Methods

- The three basic write methods:

```
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)
```

- The other methods:

```
void close()
void flush()
```

Character Streams

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)
```

Just like output streams, writers include the close and flush methods.

```
void close()
void flush()
```


Node Streams

In Java 2 SDK there are three fundamental types of nodes: files, memory (such as arrays or `String` objects), and pipes (a channel from one process or thread 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 will require dealing with native function calls to a device driver and this is non-portable.

Table 15-2 Types of Node Streams

Type	Character Streams	Byte Streams
File	<code>FileReader</code> <code>FileWriter</code>	<code>FileInputStream</code> <code>FileOutputStream</code>
Memory: Array	<code>CharArrayReader</code> <code>CharArrayWriter</code>	<code>ByteArrayInputStream</code> <code>ByteArrayOutputStream</code>
Memory: String	<code>StringReader</code> <code>StringWriter</code>	
Pipe	<code>PipedReader</code> <code>PipedWriter</code>	<code>PipedInputStream</code> <code>PipedOutputStream</code>

A Reader/Writer Example

The following example 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.

```
1 import java.io.*;
2
3 public class TestNodeStreams {
4     public static void main(String[] args) {
5         try {
6             FileReader input = new FileReader(args[0]);
7             FileWriter output = new FileWriter(args[1]);
8             char[] buffer = new char[128];
9             int charsRead;
10
11             // read the first buffer
12             charsRead = input.read(buffer);
13
14             while ( charsRead != -1 ) {
15                 // write the buffer out to the output file
16                 output.write(buffer, 0, charsRead);
17
18                 // read the next buffer
19                 charsRead = input.read(buffer);
20             }
21
22             input.close();
23             output.close();
24         } catch (IOException e) {
25             e.printStackTrace();
26         }
27     }
28 }
```

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 ability 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.

A Buffered Reader/Writer Example

This program performs the same function as the previous program.

```
1 import java.io.*;
2
3 public class TestBufferedStreams {
4     public static void main(String[] args) {
5         try {
6             FileReader    input    = new FileReader(args[0]);
7             BufferedReader bufInput = new BufferedReader(input);
8             FileWriter    output   = new FileWriter(args[1]);
9             BufferedWriter bufOutput = new BufferedWriter(output);
10            String line;
11
12            // read the first line
13            line = bufInput.readLine();
14
15            while ( line != null ) {
16                // write the line out to the output file
17                bufOutput.write(line, 0, line.length());
18                bufOutput.newLine();
19
20                // read the next line
21                line = bufInput.readLine();
22            }
23
24            bufInput.close();
25            bufOutput.close();
26        } catch (IOException e) {
27            e.printStackTrace();
28        }
29    }
30 }
```

The basic 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 13 and 21). This 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 in-turn 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 15-1 demonstrates an example input stream; in this case, a file stream is "buffered" for efficiency and then converted into data (Java primitives) items.

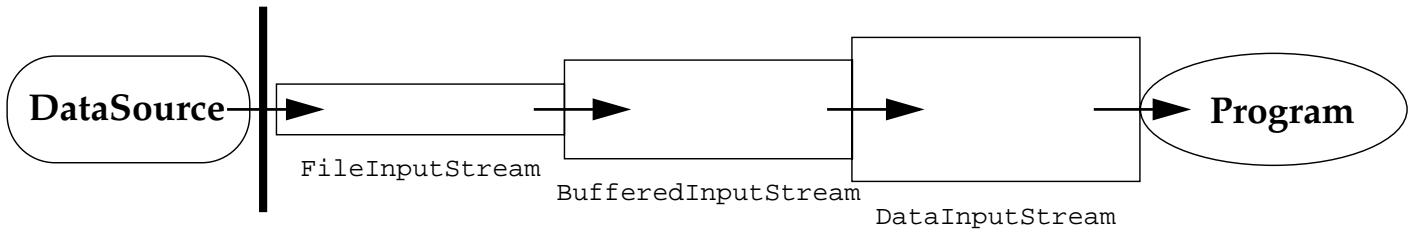


Figure 15-1 An Input Stream Chain Example

Figure 15-2 demonstrates an example output stream; in this case, data is written, then buffered, and finally written to a file.

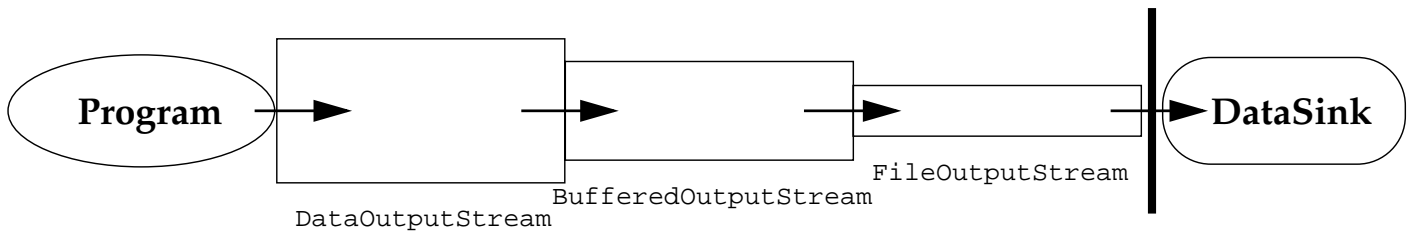


Figure 15-2 An Output Stream Chain Example

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 allows you to convert the raw data into a more usable form for your application. It is very easy to create new processing streams. This is discussed in the next section.

Table 15-3 Types of Processing Streams

Type	Character Streams	Byte Streams
Buffering	BufferedReader BufferedWriter	BufferedInputStream BufferedOutputStream
Filtering	<i>FilterReader</i> <i>FilterWriter</i>	<i>FilterInputStream</i> <i>FilterOutputStream</i>
Converting between bytes and character	InputStreamReader OutputStreamWriter	
Object serialization		ObjectInputStream ObjectOutputStream
Data conversion		DataInputStream DataOutputStream
Counting	LineNumberReader	LineNumberInputStream
Peeking ahead	PushbackReader	PushbackInputStream
Printing	PrintWriter	PrintStream

Note – *FilterXxx* are abstract classes and can not be used directly. You may subclass them to implement your own processing streams.



Processing Streams as Decorators

- A Decorator is a design pattern that allows one object (the decorator) to wrap around another object
- The `FilterXxx` classes provide a base class for you to extend to provide your own processing of an input or output stream
- For example, you could write a pair of classes, `RecordInputStream` and `RecordOutputStream`, that reads and writes database records to a stream
- A program could then decorate a file input stream with a record input stream to read database records

Processing Streams as Decorators

A Decorator is a design pattern. A decorator is an object that wraps around another object. Some method calls to the first object are passed directly to the enclosed object (this is called delegation), whereas other method calls are handled by the first object. This is used to a great extent in the Java technology I/O package.

For example, a buffered reader can be used to decorate a file reader. The `FileReader` class only implements low-level read methods, such as `read(char[] buffer)`. But the `BufferedReader` class implements the more high-level read method `readLine`, which returns a `String` object for each text line in the file. If all you need is to read a line at a time, the buffered reader object handles all of the low-level buffering for you.

Processing Streams as Decorators

By extending the `FilterXxx` abstract classes you build your own processing streams. Suppose that you built your own database using files that store raw data (ints, floats, and so on) with one "record" per line in the file. You could build a pair of classes: `RecordInputStream` and `RecordOutputStream` to handle the I/O operations at a high-level; that is, reading and writing records.

Figure 15-3 shows a UML diagram of the I/O stream classes that you would write to implement this functionality. Notice that the constructor takes a "data stream" as a parameter. This is the object decorated by the "record stream" object.

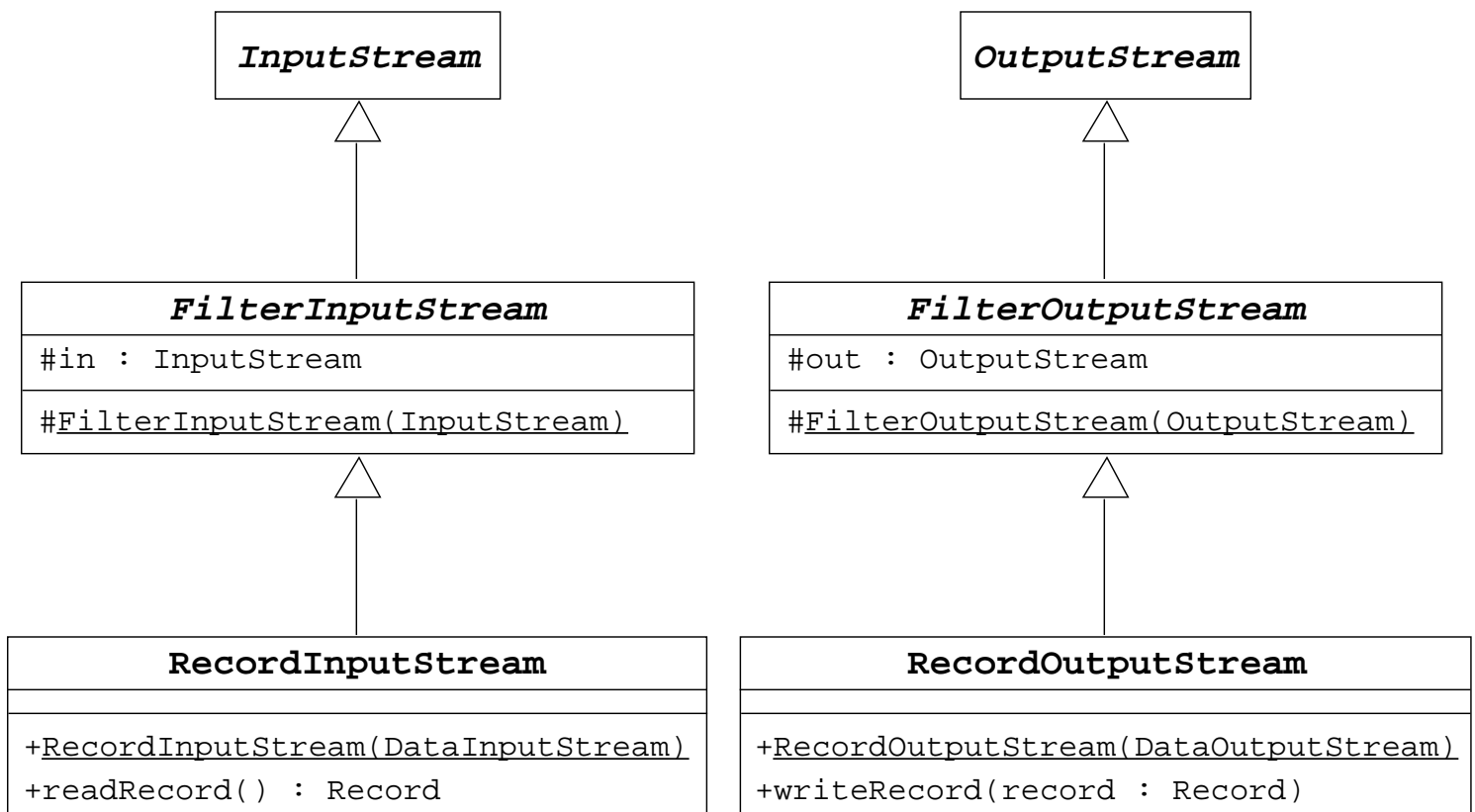


Figure 15-3 I/O Streams for Dealing With Database Record Objects

Basic Byte Stream Classes

Figure 15-4 and Figure 15-5 illustrate the hierarchy of input and output byte stream classes in the `java.io` package. Some of the more common ones are described in the following sections.

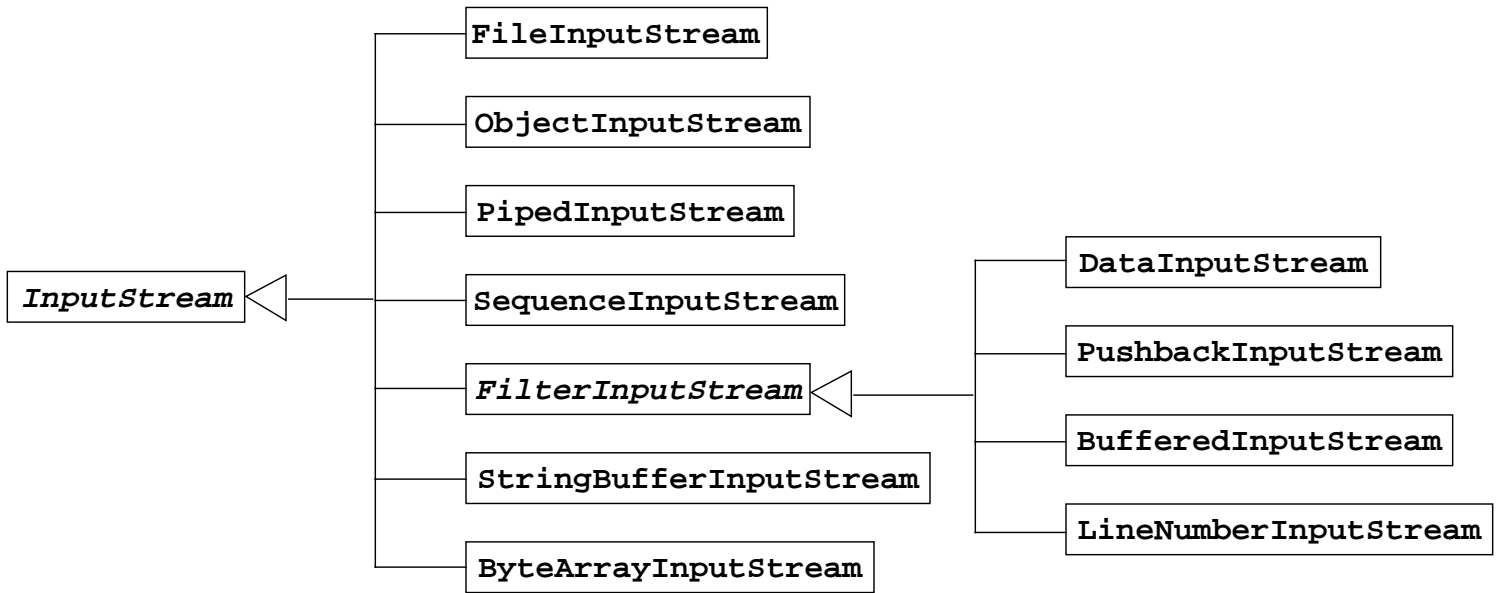


Figure 15-4 The Complete `InputStream` Class Hierarchy

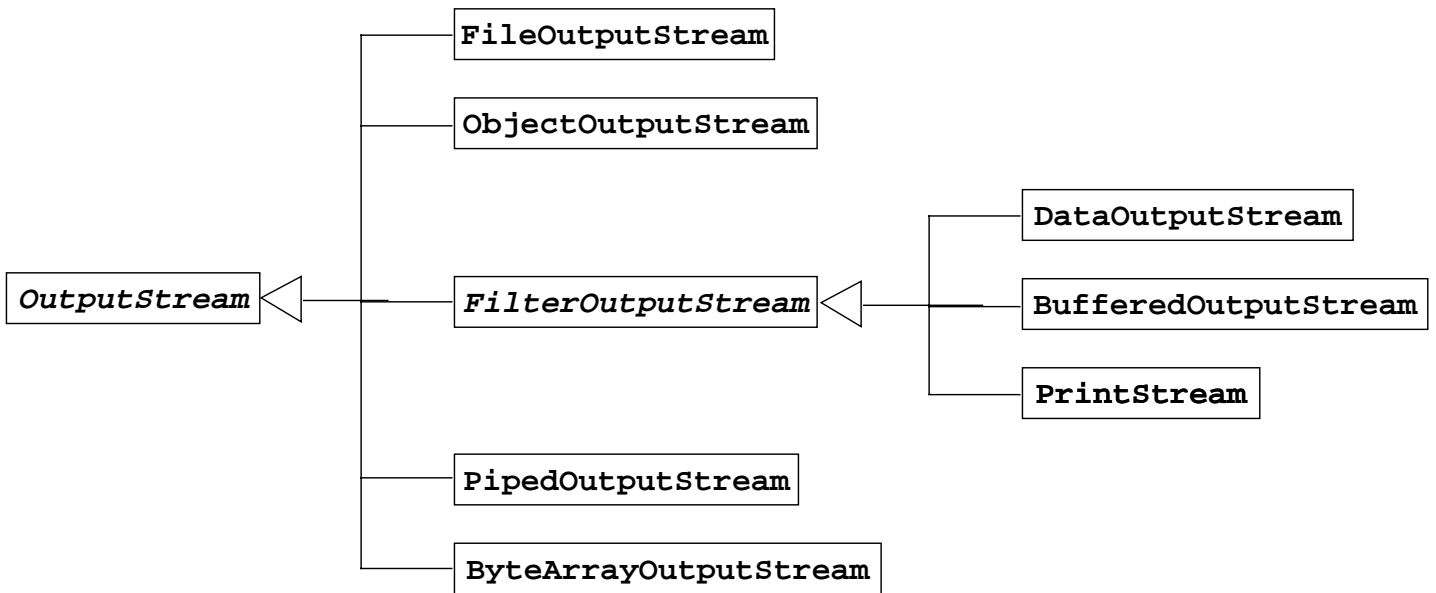


Figure 15-5 The Complete `OutputStream` Class Hierarchy

Basic Byte Stream Classes

FileInputStream *and* FileOutputStream

These classes are node streams and, as the name suggests, they use disk files. The constructors for these classes allow 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");
```

BufferedInputStream *and* BufferedOutputStream

These are filter streams that should be used to increase the efficiency of I/O operations.

PipedInputStream *and* PipedOutputStream

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.

Basic Byte Stream Classes

DataInputStream *and* DataOutputStream

These filter streams allow reading and writing of Java primitive types and some special formats using streams. A number of methods are provided for the different primitives. For example:

DataInputStream *Methods*

```
byte readByte()  
long readLong()  
double readDouble()
```

DataOutputStream *Methods*

```
void writeByte(byte)  
void writeLong(long)  
void writeDouble(double)
```

Notice that the methods of DataInputStream are paired with the methods of DataOutputStream.

These streams have methods for reading and writing strings, but these methods should not be used. They have been deprecated and replaced by readers and writers that are discussed later.

Basic Character Stream Classes

Figure 15-6 and Figure 15-7 illustrate the hierarchy of Reader and Writer character stream classes in the `java.io` package. Some of the more common ones are described in the following sections.

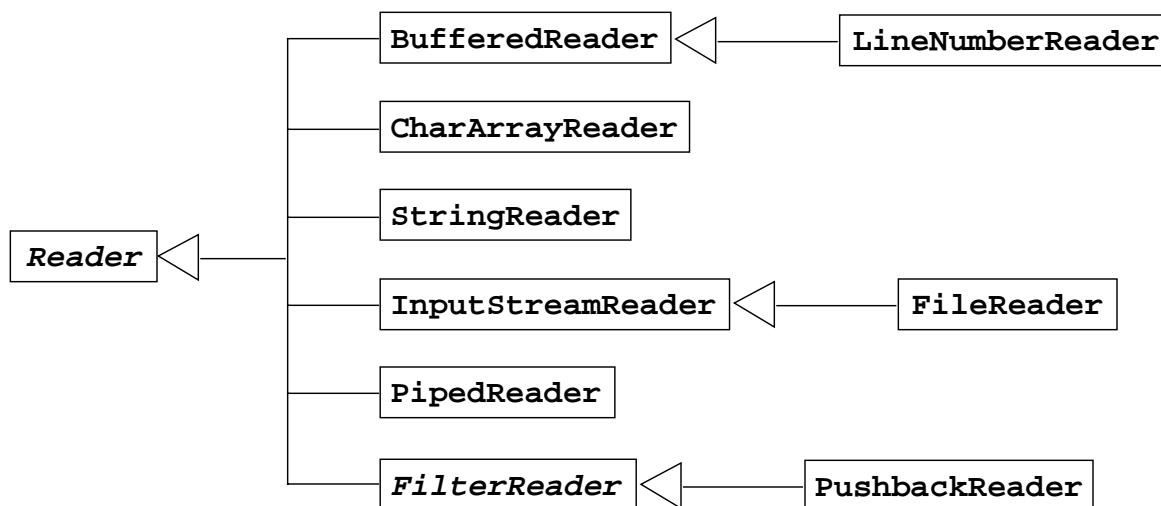


Figure 15-6 The Complete Reader Class Hierarchy

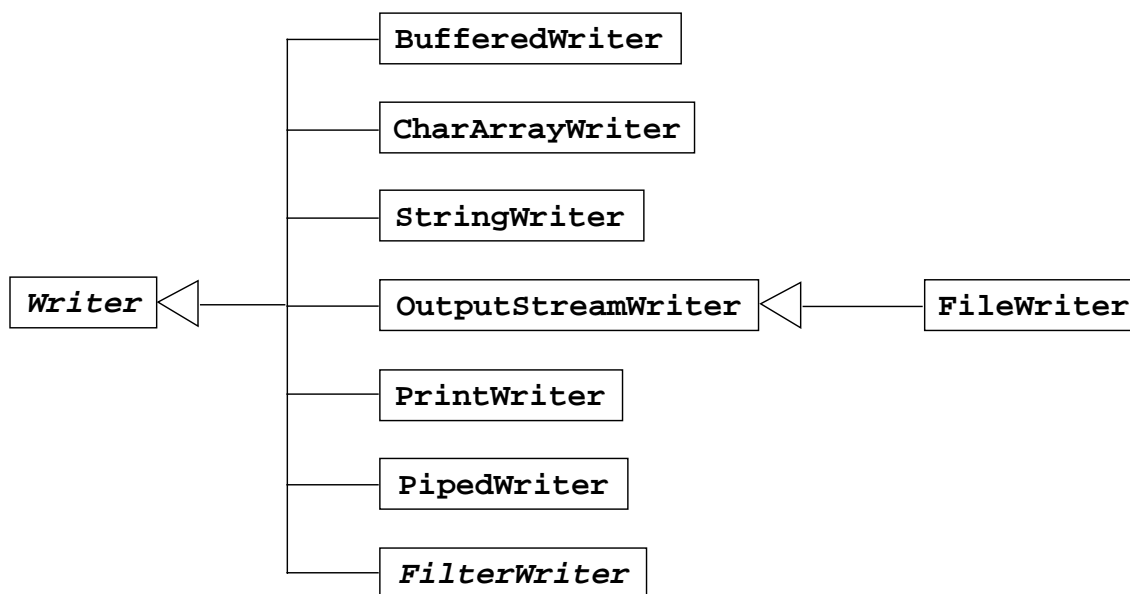


Figure 15-7 The Complete Writer Class Hierarchy

Basic Character Stream Classes

InputStreamReader *and* OutputStreamWriter

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. You can find a list of the supported encoding forms in the documentation found at jdk1.2/docs/guide/internet/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:

```
ir = new InputStreamReader(System.in, "ISO8859_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.

Basic Character Stream Classes

FileReader *and* FileWriter

These classes are node streams that are the Unicode character analogues of the `FileInputStream` and `FileOutputStream` classes.

BufferedReader *and* BufferedWriter

These are filter character streams that should be used to increase the efficiency of I/O operations.

StringReader *and* StringWriter

These are node character streams that "read" from or "write" to Java technology `String` objects.

Suppose you wrote a set of report classes whose methods accept a `Writer` parameter (the destination of the report text). Since the method makes calls against a generic interface, the program can choose to pass in a `FileWriter` object or a `StringWriter` object; the method code doesn't care. You would 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.

PipedReader *and* PipedWriter

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.



URL Input Streams

```
1 java.net.URL imageSource;
2
3 try {
4     imageSource = new URL("http://mysite.com/~info");
5 } catch (MalformedURLException e) {
6     // ignore
7 }
8
9 images[0] = getImage(imageSource, "Duke/T1.gif");
```

URL Input Streams

In addition to basic file access, Java technology provides you with the ability to use uniform resource locators (URLs) as a means of accessing objects across a network. You implicitly use a URL object when accessing sounds and images using the `getDocumentBase()` method for applets.

```
1 String imageFile = new String ("images/Duke/T1.gif");
2 images[0] = getImage(getDocumentBase(), imageFile);
```

However, you can provide a direct URL as follows:

```
1 java.net.URL imageSource;
2
3 try {
4     imageSource = new URL("http://mysite.com/~info");
5 } catch (MalformedURLException e) {}
6 images[0] = getImage(imageSource, "Duke/T1.gif");
```

✓ **This example assumes the host `mysite.com` is running an `httpd` daemon that can handle the request.**

URL Input Streams

Opening an Input Stream

You can open an input stream off of an appropriate URL object by storing a data file in or below the document base directory.

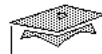
```
1 InputStream is = null;
2 String fileName = new String("Data/data.1-96");
3 byte buffer[] = new byte[24];
4
5 try {
6     // new URL throws a MalformedURLException
7     URL fileLocation = new URL(getDocumentBase(), fileName);
8
9     // URL.openStream() throws an IOException
10    is = fileLocation.openStream();
11 } catch (Exception e) {
12     // ignore
13 }
```

Now you can use it to read information, just as with a `FileInputStream` object:

```
14 try {
15    is.read(buffer, 0, buffer.length);
16 } catch (IOException e1) {
17     // ignore
18 }
```



Caution – Remember that most users have their browser security set to prevent applets from accessing files.



Creating a Random Access File

- With the file name:

```
myRAFFile = new RandomAccessFile(  
    String name, String mode);
```

- With a File object:

```
myRAFFile = new RandomAccessFile(  
    File file, String mode);
```

Random Access Files

Often you will find that you want to read data within a file without reading the file from beginning to end. You might want to access a text file as a database, in which case you move around reading one record, then another, and then another—each in different parts of the file. The Java programming language provides a `RandomAccessFile` class for handling this type of input or output.

Creating a Random Access File

You have two options for opening a random access file.

- With the file name:

```
myRAFFile = new  
    RandomAccessFile(String name, String mode);
```

Random Access Files

Creating a Random Access File (Continued)

- With a File object:

```
myRAFile = new RandomAccessFile(  
    File file, String mode);
```

The mode argument determines whether you have read-only (*r*) or read/write (*rw*) access to this file.

For example, you open a database file for updating:

```
RandomAccessFile myRAFile;  
myRAFile = new RandomAccessFile("stock.dbf", "rw");
```



Random Access Files

- `long getFilePointer()`
- `void seek(long pos)`
- `long length()`

Random Access Files

Accessing Information

`RandomAccessFile` objects expect to read and write information in the same manner as data input and data output objects. You have access to all of the `read()` and `write()` operations found in the `DataInputStream` and `DataOutputStream` classes.

The Java programming language provides several methods to help you move around inside the file. Use the following method to obtain the current location of the file pointer:

```
long getFilePointer();
```

Random Access Files

Accessing Information (Continued)

Use the following method to set the file pointer to the specified absolute position:

```
void seek(long pos);
```

The position is given as a byte-offset from the beginning of the file. Position 0 marks the beginning of the file.

Use the following method to obtain the length of the file:

```
long length();
```

The position at `length()` marks the end of the file.

Appending Information

Use random access files to accomplish an appending mode for file output:

```
myRAFile = new RandomAccessFile("java.log", "rw");  
myRAFile.seek(myRAFile.length());  
// Any subsequent writes will be appended to the file
```



Serialization

- Only the object's data are serialized
- Data marked with the `transient` keyword are not serialized

```
1 public class MyClass implements Serializable {
2     public Thread myThread;
3     private String customerID;
4     private int total;
5 }
```

```
1 public class MyClass implements Serializable {
2     public transient Thread myThread;
3     private transient String customerID;
4     private int total;
5 }
```

- Serialization is used to store the state of an object to a file; storing the state of an object is called *persistence*

Serialization

A new feature of the Java programming language since JDK 1.1 is the addition of the `java.io.Serializable` interface. This required additional changes to the JVM to support the ability to read or write a Java technology object to a stream.

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 `java.io.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. Objects from classes that do not implement `Serializable` cannot save or restore their state.

✓ **The implementation of a class that implements `Externalizable` is beyond the scope of this discussion. Also excluded is how you define `readObject` and `writeObject` in a `Serializable` class's implementation.**

Serialization

Object Graphs

When an object is serialized, only the data of the object are preserved; methods and constructors are not part of the serialized stream. When a data variable is an object, the data members of that object are also serialized if that object's class is serializable. The tree, or structure of an object's data, including these sub-objects, constitutes the object *graph*.

Some object classes are not serializable because the data they represent are constantly changing; for example, `java.io.FileInputStream` and `java.lang.Thread`. 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.

```
public class MyClass implements Serializable {
    public transient Thread myThread;
    private String customerID;
    private int total;
}
```

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 UTF (file system safe universal character set transformation format) characters. The `transient` keyword prevents the data from being serialized.

```
public class MyClass implements Serializable {
    public transient Thread myThread;
    private transient String customerID;
    private int total;
}
```

Writing and Reading an Object Stream

Writing

Writing and reading an object to a file stream is a simple process. The following code fragment sends an instance of a `java.util.Date` object to a file:

```
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 }
```

Writing and Reading an Object Stream

Reading

Reading the object is as simple as writing it, 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.

```
1 import java.io.*;
2 import java.util.Date;
3
4 public class UnSerializeDate {
5
6     UnSerializeDate () {
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
20        System.out.println(
21            "Unserialized Date object from date.ser");
22        System.out.println("Date: "+d);
23    }
24
25    public static void main (String args[]) {
26        new UnSerializeDate();
27    }
28 }
```

Exercise: Getting Acquainted With I/O



Exercise objective – In this lab you will become familiar with stream I/O by writing programs which perform I/O operations.

Preparation

You should understand the basic concepts of a database and the basic concepts of writing data to a stream.

Tasks

In a Web browser view the `lab_files.html` page that is at the top-level of the SL275 directory on your computer. There will be a summary of each exercise and a link to a page that gives a detailed explanation of the exercise.

Go to the SL275 directory on your computer and change to the directory for this module (`mod15`). A listing of this directory will show three subdirectories: one for each of the exercises below.

Exercise 1: Implement Object Serialization (Level 1)

In this exercise you will read and write a serialized object from and to a file.

Exercise 2: Implement the Record Processing Streams (Level 2)

In this exercise you will write the `RecordInputStream` and `RecordOutputStream` filter stream classes based on a particular `Record` class.

Exercise 3: Create a Simple Database Program (Level 3)

In this exercise you will use random access files to implement a simple database program.

Exercise: Getting Acquainted With I/O

Exercise Summary



Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercises.

✓ *If you do not have time to spend on discussion, just highlight the key concepts students should have learned from the lab exercise.*

- Experiences

✓ *Ask students what their overall experiences with this exercise have been. You might want to go over any trouble spots or especially confusing areas at this time.*

- Interpretations

✓ *Ask students to interpret what they observed during any aspects of this exercise.*

- Conclusions

✓ *Have students articulate any conclusions they have reached as a result of this exercise experience.*

- Applications

✓ *Explore with the students how they might apply what they learned in this exercise to situations at their workplace.*

Check Your Progress

Before continuing on to the next module, check to be sure that you can

- Describe the main features of the `java.io` package
- Construct node and processing streams, and use them appropriately
- Distinguish readers and writers from streams, and select appropriately between them
- Use the `Serialization` interface to encode the state of an object

Think Beyond

Do you have applications that could benefit from creating specialized stream and character filters?

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 implementing TCP/IP clients and servers

This module discusses Java 2 SDK support for sockets and socket programming. Socket programming is used to communicate with other programs running on computers on the same network.

Relevance

- ✓ **Present the following question to stimulate the students and get them thinking about the issues and topics presented in this module. They are not expected to know the answer to this question. Hold discussions where students have input; otherwise, if no one can propose answers, begin the lecture for this module.**



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?

- ✓ **Networked servers are a convenient way of providing services to any client on the same network that needs them. Sockets provide a means of network communication to make this possible. The Java 2 SDK provides an interface to sockets through classes in the `java.net` package. This module examines two kinds of socket programming offered by the JDK, from both a server and client perspective.**



Networking

- Sockets:
 - ▼ Sockets hold two streams
- Setting up the connection:
 - ▼ Set up is similar to a telephone system

Networking

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 name socket has been reused in other programming models, including Java technology.

When processes communicate over a network, Java technology uses its 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.

Once the network connection has been set up, using the streams associated with that connection is similar to using any other stream.

Networking

Setting up the Connection

To set up the connection, one machine must be running a program that is waiting for a connection, and the other machine must try to reach the first. This is similar to a telephone system; one party must make the call, while the other party must be waiting by the telephone when that call is made.

Transmission Control Protocol/Internet Protocol, or *TCP/IP*, is the first type of connection protocol that is presented in this module.

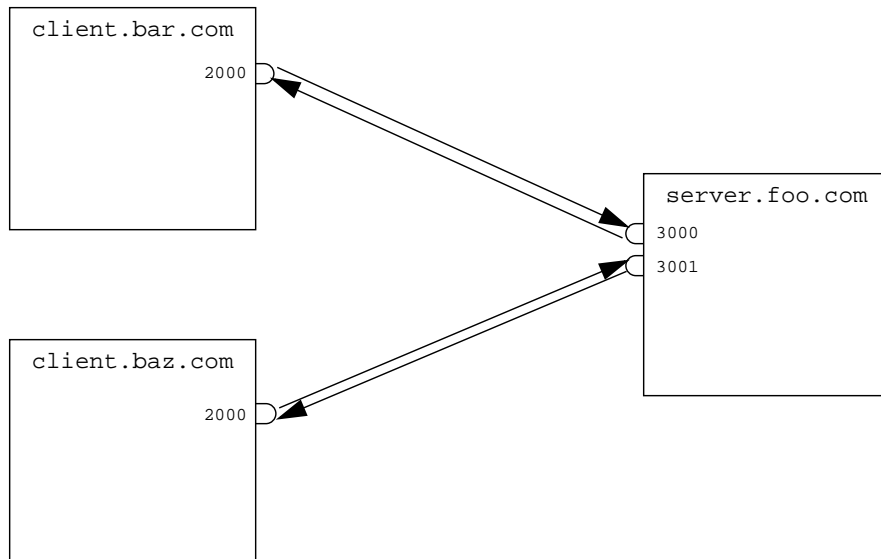
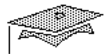


Figure 16-1 Diagram of Network Connections



Sun Educational Services

Networking With Java Technology

- Addressing the connection:
 - ▼ Address or name of remote machine
 - ▼ Port number to identify purpose
- Port numbers:
 - ▼ Range from 0 to 65535

Networking With Java Technology

Addressing the Connection

When making a telephone call, you need to know the telephone number to dial. When making 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 that you can think of as a telephone extension number. Once you have connected 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.

Networking With Java Technology

Port Numbers

Port numbers in TCP/IP systems are 16-bit numbers and range from 0 to 65535. In practice, port numbers below 1024 are reserved for predefined services, and you should not use them unless you want to communicate with one of those services (such as `telnet`, Simple Mail Transport Protocol (SMTP) mail, `ftp`, and so on).

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, no communication will occur.

Networking With Java Technology

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.

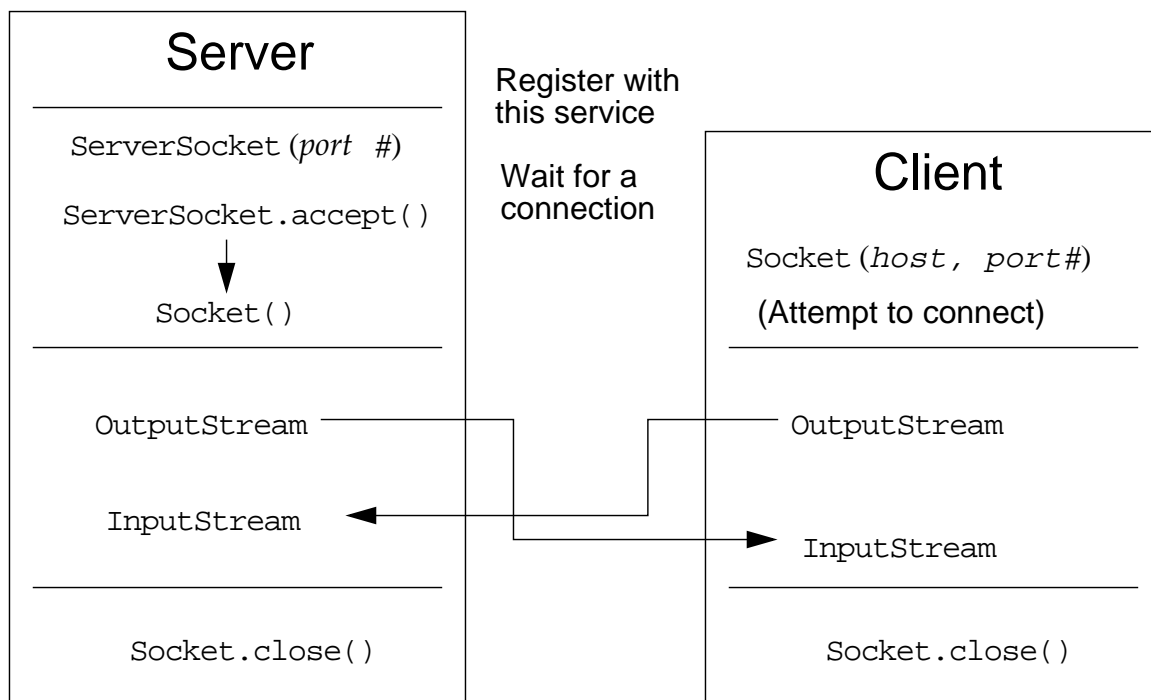


Figure 16-2 TCP/IP Socket Connections

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;
7
8         // Register your service on port 5432
9         try {
10            s = new ServerSocket(5432);
11        } catch (IOException e) {
12            // ignore
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 slout = s1.getOutputStream();
23                DataOutputStream dos = new DataOutputStream(slout);
24
25                // Send your string!
26                dos.writeUTF("Hello Net World!");
27
28                // Close the connection, but not the server socket
29                dos.close();
30                s1.close();
31            } catch (IOException e) {
32                // ignore
33            }
34        }
35    }
36 }
```

✓ **If you use `writeUTF`, you must use `readUTF` on the other end of the connection.**

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 has been done by the `Socket` class. The client attaches to the server presented on the previous page and 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             dis.close();
21             s1.close();
22         } catch (ConnectException connExc) {
23             System.err.println("Could not connect to the server.");
24         } catch (IOException e) {
25             // ignore
26         }
27     }
28 }
```

- ✓ **Currently, the Microsoft Windows `Socket` `close` method does not correctly send an EOF (end of file) message to the connected socket. A temporary fix is to send an ending close message such as "bye."**

Note – UTF stands for UCS Transformation Format. UCS stands for Universal Character Set. UTF is a string format that is platform-independent. If you exchange strings across a network, you should always use UTF.

Exercise: Using Socket Programming



Exercise objective – Gain experience using sockets by implementing a client and server which communicate using sockets.

Preparation

To successfully complete this lab, you must have a clear understanding of HTML and the network.

Tasks

In a Web browser view the `lab_files.html` page that is at the top-level of the SL275 directory on your computer. There will be a summary of each exercise and a link to a page that gives a detailed explanation of the exercise.

Go to the SL275 directory on your computer and change to the directory for this module (`mod16`). A listing of this directory will show two subdirectories: `exercise1` and `exercise2`.

Exercise 1: Finish the ChatClient (Level 2)

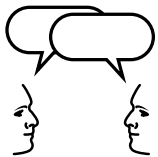
In this exercise you will finish the "chat room" client program. Your client will connect to a "chat server" so that you may chat with other students in the class.

Exercise 2: Create a File Server (Level 3)

In this exercise you will have hands-on experience in the creation of a simple file server and client.

Exercise: Using Socket Programming

Exercise Summary



Discussion – Take a few minutes to discuss what experiences, issues, or discoveries you had during the lab exercises.

- ✓ *If you do not have time to spend on discussion, just highlight the key concepts students should have learned from the lab exercise.*
 - Experiences
- ✓ *Ask students what their overall experiences with this exercise have been. You might want to go over any trouble spots or especially confusing areas at this time.*
 - Interpretations
- ✓ *Ask students to interpret what they observed during any aspects of this exercise.*
 - Conclusions
- ✓ *Have students articulate any conclusions they have reached as a result of this exercise experience.*
 - Applications
- ✓ *Explore with the students how they might apply what they learned in this exercise to situations at their workplace.*

Check Your Progress

Before continuing, check to be sure that you can:

- Develop code to set up the network connection
- Understand the TCP/IP protocol
- Use `ServerSocket` and `Socket` classes for implementing TCP/IP clients and servers

Think Beyond

How can you create a distributed object system using object serialization and these network protocols? Have you heard of Remote Method Invocation (RMI)?

There are several advanced Java platform topics, many of which are addressed in other Sun Educational Services courses. Be sure and check out the JavaSoft™ Web site (<http://www.javasoft.com>) as well.

Elements of Advanced Java Programming

A 

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

Introduction to Two- 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 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.

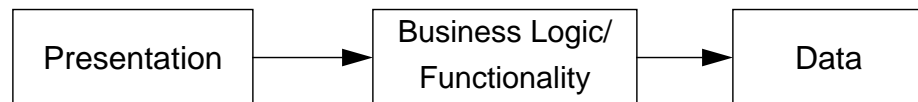
A major performance penalty is paid in two-tier client-server. The client software ends up larger and more complex because most of the logic is handled there. The use of server side logic is limited to database operations. The client here is referred to as *thick client*.

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.

The 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



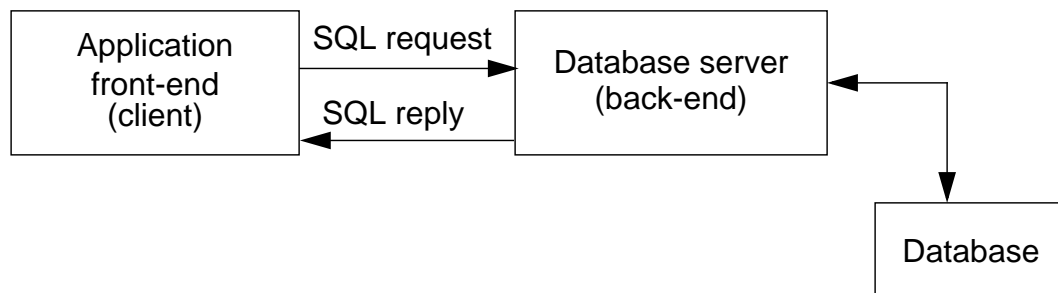
The three components or tiers of a three-tier client-server environment are *presentation*, *business logic* or *functionality*, and *data*. They are separated such that the software for any one of the tiers can be replaced by a different implementation without affecting the other tiers. For example, 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 Frontend

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.

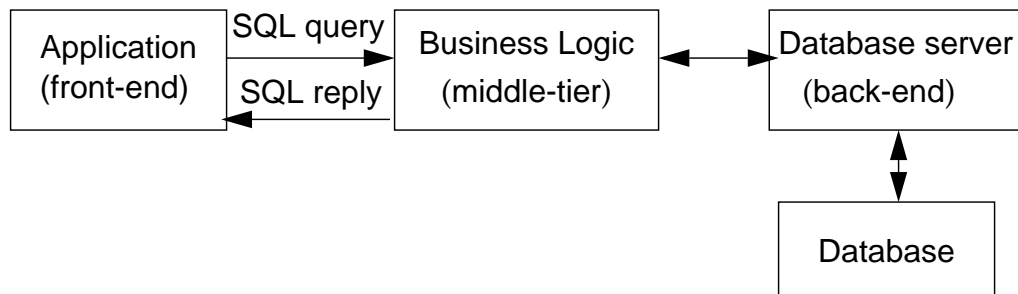
In a two-tier model, the database resides on a database server. The client executes a front-end application that opens a socket for communication over the network. The socket provides a communication path between the client application and the back-end server. In the following illustration, client programs send SQL database query requests to the database server. The server returns the results to the client, which formats the results for presentation.



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.

A Database Frontend

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.



Introduction to the JDBC API

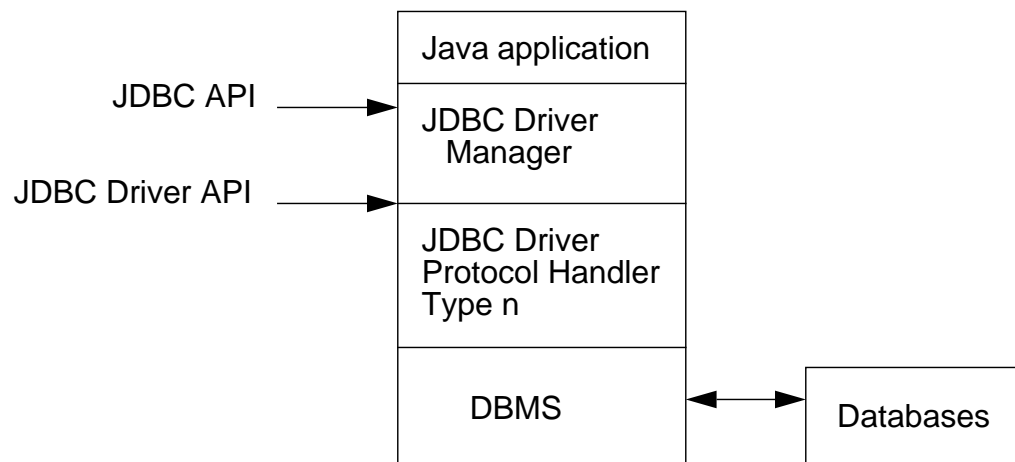
The ability to create robust, platform-independent applications and Web-based applets prompted developers to create front-end connectivity solutions. JavaSoft worked with database and database-tool vendors to create a database management system (DBMS)-independent mechanism that would enable developers 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.

- ✓ **Choosing ODBC was a pragmatic choice because it is a widely accepted and implemented standard for SQL database access. Virtually all databases support ODBC.**

As shown in the following figure, 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.



Introduction to the JDBC API

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 details 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.

Application
JDBC Driver Manager
JDBC-ODBC Bridge
ODBC Driver Manager
ODBC Driver Libraries

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.

✓ **For advanced coverage in distributed programming, recommend the SL-301 course.**

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

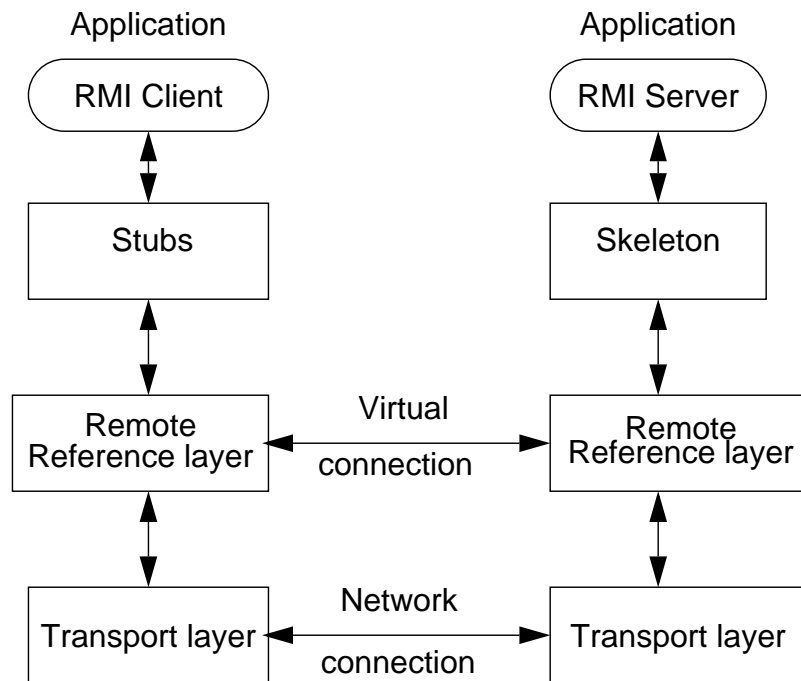
When an RMI client binds a remote object through the registry, it receives a local reference to the remote instantiated object through its interface and communicates with the object through that reference. Local references to the same remote object can exist on multiple clients; any variables and methods contained within the remote object are shared.

The applet begins by importing the appropriate RMI packages and creating a reference to the remote object. Once the applet establishes this link, it can call the remote object's methods as if they were locally available to the applet.

RMI

RMI Architecture

The RMI architecture provides three layers: Transport layers, Remote Reference, and Stubs/Skeleton.



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.

RMI

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:

- Define interfaces for remote classes
- Create and compile implementation classes for the remote classes
- Create stub and skeleton classes using the `rmic` command
- Create and compile the server application
- Start the RMI Registry and the server application
- Create and compile a client program to access the remote objects
- 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 details refer to the following URL:

<http://www.omg.org>

You can write CORBA objects in any programming language, including C and C++. These objects can also exist on any platform, including the Solaris Operating Environment, Microsoft Windows, openVMS, Digital UNIX, HP-UX, and many others. This means a Java application running on a Microsoft Windows platform can dynamically load and use C++ objects stored on the Internet by using a UNIX Web server.

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 resource 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 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 Web 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 Interface Definition Language (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 was designed to be a secure solution. This means that building RMI applications is simple, and all remote objects have the same features as local objects. You can also combine the best of JDBC and RMI for a multi-tier solution.

CORBA benefits from the fact that it is a language-independent solution, which adds significant complexity to the development cycle and precludes garbage collection features. For a database application developer, CORBA provides the ultimate flexibility in a heterogeneous environment. The server could be developed in C or C++, and the client could be a Java applet.

The JavaBeans Component Model

JavaBeans 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 Java Bean 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.

The JavaBeans Component Model

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 1.1. The event model used in Java 2 SDK is the same event model that was introduced in JDK 1.1. There are three parts to this communication: event object, event listener, and event source.

JavaBeans communicate 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)`.

The JavaBeans Component Model

Bean Architecture (Continued)

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.

The JavaBeans Component Model

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 Java Beans 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-1).

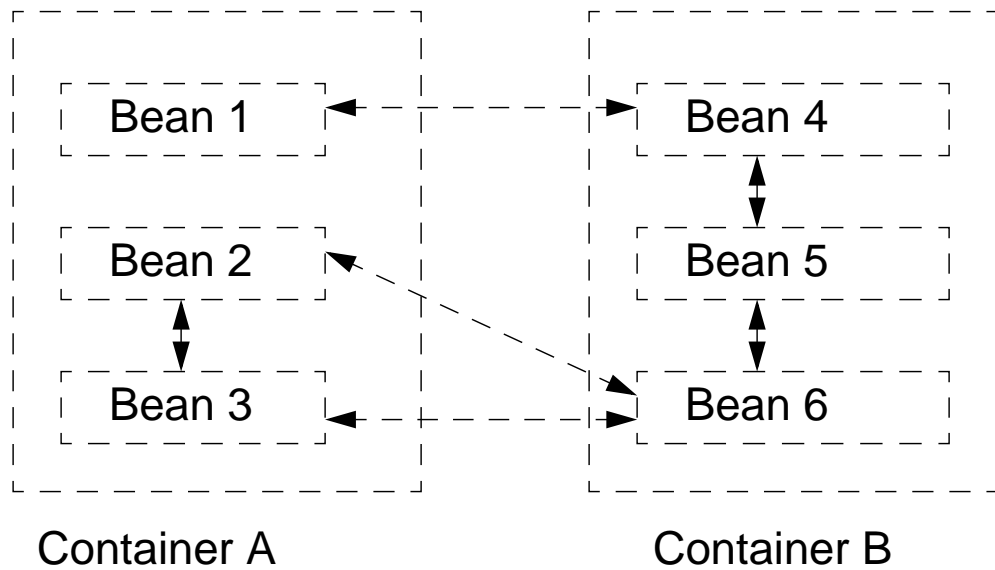


Figure A-1 A Sample Bean Interaction

The JavaBeans Component Model

A Sample Bean Interaction

In Figure A-1, 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- 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 allows Java 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 Java Beans API.

✓ **Recommend the SL-291 course.**

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/animatorm.jar"
  width=460 height=160>
  <param name=foo value="bar">
</applet>
```

Check Your Progress

At the end of this appendix, check to be sure that you can:

- 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

JDK 1.0 GUI Event Model

B 

This appendix provides an overview of the event handling under JDK 1.0.x and JDK 1.1 and provides a table that maps 1.0.x events and corresponding methods to their 1.1 counterparts.

Additional Resources



Additional Resources – The contents of this appendix were obtained from the Web page “How to Convert Programs to the 1.1 AWT API.” [Online]. Available:
<http://www.javasoft.com/products/JDK/1.1/docs/guide/awt/HowToUpgrade.html>.

Event Handling

Event Handling Before JDK 1.1

Before JDK 1.1, the `Component` `handleEvent` method (along with the methods it called, such as the `action` method) was the center of event handling. Only `Component` objects could handle events, and the component that handled an event had to be either the component in which the event occurred or a component above it in the component containment hierarchy.

Event Handling in JDK 1.1

In JDK 1.1, event handling is no longer restricted to objects in the component containment hierarchy, and the `handleEvent` method is no longer the center of event handling. Instead, objects of any type can register as event listeners. Event listeners receive notification only about the types of events in which they have registered their interest. You do not have to create a `Component` subclass to handle events.

When upgrading to the JDK 1.1 release, the easiest way to convert event-handling code is to leave it in the same class, and make it a listener for that type of event.

Another possibility is to centralize event-handling code in one or more non-component listeners (adaptors or filters). This approach lets you separate the GUI of your program from implementation details. It requires that you modify your existing code so that the listeners can get whatever state information they require from the components. This approach can be worthwhile if you are trying to keep your program's architecture clean.

JDK 1.0 Event Model Compared to Java 2 SDK Event Model

JDK 1.1 introduced significant changes in the way that events are received and processed. This section compares the previous event model (JDK 1.0) to the current event model (JDK 1.1 and Java 2 SDK).

JDK 1.0 uses a hierarchical event model, while JDK 1.1 and beyond use a delegation event model.

Hierarchical Model (JDK 1.0)

The hierarchical event model is based on containment. Events are sent to the component first, but then propagate up the containment hierarchy. Events that are not handled by the component *automatically* continue to propagate to the component's container.

✓ **These are containers, not the `Container` class. They extend the `Container` class.**

For example, in Figure B-1, a mouse click on the `Button` object (contained by a `Panel` on a `Frame`) sends an action event to the `Button` first. If it is not handled by the `Button`, the event is then sent to the `Panel`, and if it is not handled there, the event is sent to the `Frame`.

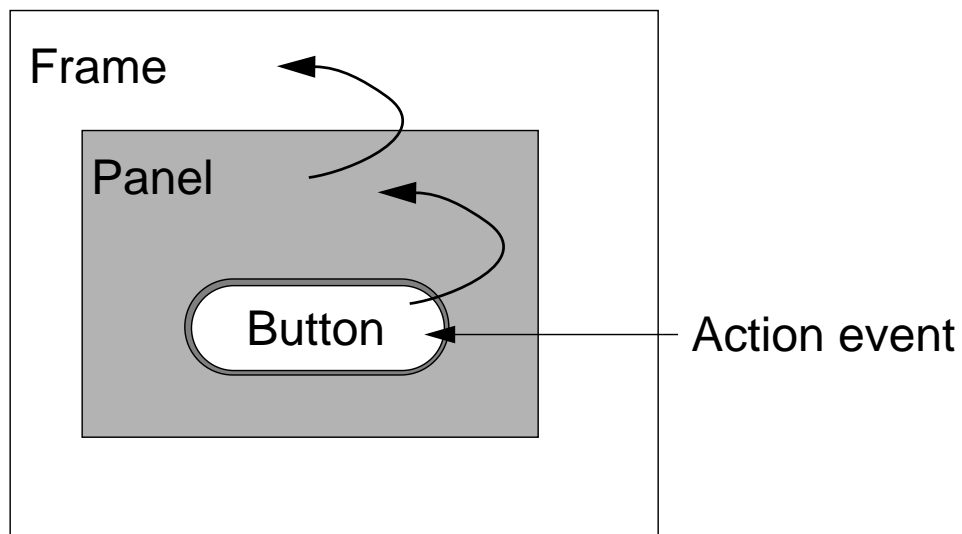


Figure B-1 Hierarchical Event Model

JDK 1.0 Event Model Compared to Java 2 SDK Event Model

Hierarchical Model (JDK 1.0) (Continued)

There is an obvious advantage to this model:

- It is simple and uses features of an object-oriented programming environment; after all, Java software components extend from the `java.awt.Component` class, which defines `handleEvent()`. To customize event handling, you override `handleEvent()`.

However, there are some disadvantages:

- The event can be handled only by the component from where it originates or by one of the containers that contains it. This restriction violates one of the fundamental principles of object-oriented programming: Functionality should reside in the most appropriate class. Often the most appropriate class for handling an event is not a member of the originating component's containment hierarchy.
 - A large number of CPU cycles are wasted on unrelated events. Any event unrelated or unimportant to the program would traverse the containment hierarchy before eventually being discarded. There is no simple way to filter events.
- ✓ ***You can throw a new event in JDK 1.0, but it requires knowledge of the superclass hierarchy and specialized methods like `postEvent`.***
- To handle events, you must either subclass the component that receives the event or create a `handleEvent()` method at the base container.
- ✓ ***This involves quite a bit of software.***

Converting 1.0 Event Handling to 1.1

The biggest part of converting most 1.0 AWT-using programs to the 1.1 API is converting the event-handling code. The process can be straightforward, once you figure out which events a program handles and which components generate the events. Searching for “Event” in a source file lets you find the event-handling code.

Note – While you are looking at the code you should note whether any classes exist solely for the purpose of handling events; you might be able to eliminate such classes.

You can use Table B-1 in the conversion process to help map 1.0 events and methods to their 1.1 counterparts.

- The first column lists each 1.0 event type, along with the name of the method (if any) that is specific to the event.

Where no method is listed, the event is always handled by the `handleEvent` method.

- The second column lists the 1.0 components that can generate the event type.
- The third column lists the listener interface that helps you handle the 1.1 equivalents of the listed events.
- The fourth column lists the methods in each listener interface.

Table B-1 Event Conversion Table

1.0.x		1.1	
Event/Method	Generated by	Interface	Methods
ACTION_EVENT/action	Button List MenuItem TextField	ActionListener	actionPerformed(ActionEvent)
	Checkbox CheckboxMenuItem Choice	ItemListener	itemStateChanged(ItemEvent)
WINDOW_DESTROY WINDOW_EXPOSE WINDOW_ICONIFY WINDOW_DEICONIFY	Dialog Frame	WindowListener	windowClosing(WindowEvent) windowOpened(WindowEvent) windowIconified(WindowEvent) windowDeiconified(WindowEvent) windowClosed(WindowEvent) ^a windowActivated(WindowEvent) ^a windowDeactivated(WindowEvent) ^a
WINDOW_MOVED	Dialog Frame	ComponentListener	componentMoved(ComponentEvent) ComponentHidden(ComponentEvent) ^a componentResized(ComponentEvent) ^a componentShown(ComponentEvent) ^a

Table B-1 Event Conversion Table (Continued)

1.0.x		1.1	
Event/Method	Generated by	Interface	Methods
SCROLL_LINE_UP SCROLL_LINE_DOWN SCROLL_PAGE_UP SCROLL_PAGE_DOWN SCROLL_ABSOLUTE SCROLL_BEGIN SCROLL_END	Scrollbar	AdjustmentListener (Or use the new ScrollPane class)	adjustmentValueChanged(AdjustmentEvent)
LIST_SELECT LIST_DESELECT	Checkbox CheckboxMenuItem Choice List	ItemListener	itemStateChanged(ItemEvent)
MOUSE_DRAG/mouseDrag MOUSE_MOVE/mouseMove	Canvas Dialog Frame Panel Window	MouseMotionListener	mouseDragged(MouseEvent) mouseMoved(MouseEvent)
MOUSE_DOWN/mouseDown MOUSE_UP/mouseUp MOUSE_ENTER/mouseEnter MOUSE_EXIT/mouseExit	Canvas Dialog Frame Panel Window	MouseListener	mousePressed(MouseEvent) mouseReleased(MouseEvent) mouseEntered(MouseEvent) mouseExited(MouseEvent) mouseClicked(MouseEvent) ^a

Table B-1 Event Conversion Table (Continued)

1.0.x		1.1	
Event/Method	Generated by	Interface	Methods
KEY_PRESS/keyDown KEY_RELEASE/keyUp KEY_ACTION/keyDown KEY_ACTION_RELEASE/keyUp	Component	KeyListener	keyPressed (KeyEvent) keyReleased (KeyEvent) keyTyped (KeyEvent) ^a
GOT_FOCUS/gotFocus LOST_FOCUS/lostFocus	Component	FocusListener	focusGained (FocusEvent) focusLost (FocusEvent)
<i>No 1.0 equivalent</i>		ContainerListener	componentAdded (ContainerEvent) componentRemoved (ContainerEvent)
<i>No 1.0 equivalent</i>		TextListener	textValueChanged (TextEvent)

a. No 1.0 equivalent

Making a Component a Listener

Use the following general steps to convert a 1.0 component into a 1.1 listener:

1. Change the source file so that it imports the `java.awt.event` package:

```
import java.awt.event.*
```

2. Use Table B-1 to determine which components generate each event type.

For example, if you are converting event code that is in an action method, you should look for `Button`, `List`, `MenuItem`, `TextField`, `Checkbox`, `CheckboxMenuItem`, and `Choice` objects.

3. Change the class declaration so that the class implements the appropriate listener interfaces, as indicated in Table B-1.

For example, if you are trying to handle an action event generated by a `Button`, Table B-1 informs you that you must implement the `ActionListener` interface.

```
public class MyClass extends SomeComponent  
    implements ActionListener {
```

4. Determine where the components that generate the events are created. Just after the code that creates each one, register this as the appropriate type of listener. For example:

```
newComponentObject.addActionListener(this);
```

Making a Component a Listener

5. Create empty implementations of all the methods in the listener interfaces your class must implement. Copy the event-handling code into the appropriate methods.

For example, `ActionListener` has just one method, `actionPerformed`. An easier way to create the new method and copy the event-handling code to it is to change the signature of an action method from:

```
public boolean action(Event event, Object arg) {  
to  
public void actionPerformed(ActionEvent event) {
```

6. Modify the event-handling code as follows:
 - a. Delete all `return` statements.
 - b. Change references from `event.target` to `event.getSource()`.
 - c. Delete any code that unnecessarily tests for the component from which the event came. (Now that events are forwarded only if the generating component has a listener, you do not have to worry about receiving events from an unwanted component.)
 - d. Perform any other modifications required to make the program compile cleanly and execute correctly.

Features of the AWT

The AWT provides a wide variety of standard features. This appendix introduces many of the components that are available to you, and outlines any particular anomalies about which you might need to know. You should be aware of the full set of GUI components, so that you can choose the appropriate ones when building your own interfaces.

Button

You have already become familiar with the Button component. It provides a basic “push to activate” user interface component. It can be constructed with a text label that informs the user of its use.

```
1 f = new Frame("Sample Button");
2 b = new Button("Sample");
3 b.addActionListener(this);
4 f.add(b);
```



Figure C-1 Button Component

The `actionPerformed()` method of any class implementing the `ActionListener` interface, which is registered as a listener, is called when the button is “pressed” by a mouse click.

```
1 public void actionPerformed( ActionEvent ae) {
2     System.out.println("Button press received.");
3     System.out.println("Button's action command is: " +
4         ae.getActionCommand());
5 }
```

The `getActionCommand()` method of the `ActionEvent` that is issued when the button is pressed returns the label string by default. The action command or label is changed by using the button’s `setActionCommand()` method.

```
1 b = new Button("Sample");
2 b.setActionCommand("Action Command Was Here!");
3 b.addActionListener(this);
4 f.add(b);
```

Note – You can find the complete source for `SampleButton` and `ActionCommandButton` in the `examples` directory.

Checkbox

The Checkbox component provides a simple “on/off” input device with a text label beside it.

```
1  f = new Frame("Sample Checkbox");
2  one = new Checkbox("One", true);
3  two = new Checkbox("Two", false);
4  three = new Checkbox("Three", false);
5
6  one.addItemListener(this);
7  two.addItemListener(this);
8  three.addItemListener(this);
9
10 f.setLayout(new FlowLayout());
11 f.add(one);
12 f.add(two);
13 f.add(three);
```



Figure C-2 Checkbox Component

Selection or deselection of a checkbox is sent to the `ItemListener` interface. The `ItemEvent` that is passed contains the method `getStateChange()`, which returns `ItemEvent.DESELECTED` or `ItemEvent.SELECTED`, as appropriate. The method `getItem()` returns the affected checkbox as a `String` object that represents its label.

```
1  public void itemStateChanged(ItemEvent ev) {
2      String state = "deselected";
3      if (ev.getStateChange() == ItemEvent.SELECTED) {
4          state = "selected";
5      }
6      System.out.println (ev.getItem() + " " + state);
7  }
```

Checkbox Group – Radio Buttons

CheckboxGroup provides the means to group multiple Checkbox items into a mutual exclusion set, so that only one Checkbox in the set has the value `true` at any time. The Checkbox with the value `true` is the currently selected Checkbox. You can create each Checkbox of a group using a constructor that takes an additional argument, a `CheckboxGroup`. It is this `CheckboxGroup` object that connects the Checkbox items together into a set. The appearance of each Checkbox item added to the group is changed to a “radio button.”

```

1  f = new Frame("CheckBoxGroup");
2  cbg = new CheckboxGroup();
3  one = new Checkbox("One", cbg, false);
4  two = new Checkbox("Two", cbg, false);
5  three = new Checkbox("Three", cbg, true);
6
7  f.setLayout(new FlowLayout());
8
9  one.addItemListener(this);
10 two.addItemListener(this);
11 three.addItemListener(this);
12
13 f.add(one);
14 f.add(two);
15 f.add(three);

```



Figure C-3 CheckboxGroup Component

Choice

The Choice component provides a simple “select one from this list” type of input. For example:

```
1 f = new Frame("Sample Choice");
2 choice = new Choice();
3 choice.addItem("First");
4 choice.addItem("Second");
5 choice.addItem("Third");
6 choice.addItemListener(this);
7 f.add(choice, BorderLayout.CENTER);
```



Figure C-4 Choice Component

When you click on the Choice, it displays a list of items that have been added to it. The items added are String objects.



Figure C-5 Choice With Items

The `ItemListener` interface is used to observe changes in the Choice. The details are the same as those for the Checkbox.

Canvas

A Canvas provides a blank (background colored) space. It has a preferred size of zero by zero, unless you explicitly specify a size using `setSize()`. To specify the size, place it in a layout manager that specifies the size.

You can use the space to draw, write text, or receive keyboard or mouse input. "Drawing in AWT" on page 10-39 in Module 10, "Building Java GUIs," shows you how to draw effectively in the AWT.

Generally, a Canvas is used either to provide general drawing space or to provide a working area for a custom component.

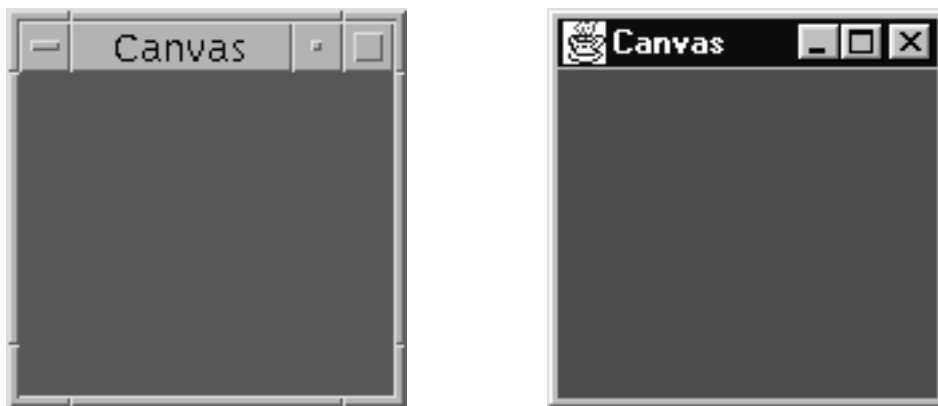


Figure C-6 Canvas Component

The Canvas can “listen” to all the events that are applicable to a general component. In particular, you might want to add `KeyListener`, `MouseMotionListener`, or `MouseListener` objects to it to allow it to respond in some way to user input.

Note – To receive key events in a Canvas, it is necessary to call the `requestFocus` method of the Canvas. If this is not done, it is generally not possible to “direct” the keystrokes to Canvas. Instead, the keystrokes go to another component or are perhaps lost entirely.

✓ **A Canvas is usually extended for use as a drawing component.**

Canvas

The following is an example of a Canvas. This program changes the color of the Canvas each time a key is pressed.

```
1 import java.awt.*;
2 import java.awt.event.*;
3 import java.util.*;
4
5 public class MyCanvas extends Canvas
6     implements KeyListener{
7     private int index;
8     Color colors[] = { Color.red, Color.green, Color.blue };
9
10    public void paint(Graphics g) {
11        g.setColor(colors[ index ]);
12        g.fillRect(0, 0, getSize().width, getSize().height);
13    }
14
15    public void keyTyped(KeyEvent ev) {
16        index++;
17        if ( index == colors.length ) {
18            index = 0;
19        }
20        repaint();
21    }
22
23    // Unused KeyListener methods
24    public void keyPressed(KeyEvent ev) { }
25    public void keyReleased(KeyEvent ev) { }
26
27    public static void main(String args[]) {
28        Frame f = new Frame("Canvas");
29        MyCanvas mc = new MyCanvas();
30        mc.setSize(150, 150);
31        f.add(mc, BorderLayout.CENTER);
32        mc.requestFocus();
33        mc.addKeyListener(mc);
34        f.pack();
35        f.setVisible(true);
36    }
37 }
```

Label

A `Label` object displays a single line of static text. The program can change the text, but the user cannot. No special borders or other decorations are used to delineate a `Label`.

```
1 Frame f = new Frame("Label");
2 Label lb = new Label("Hello");
3 f.add(lb);
```



Figure C-7 Label

`Label` is not usually expected to handle events, but can do so in the same manner as a `Canvas`. That is, keystrokes can be picked up reliably only by calling `requestFocus()`.

TextField

The TextField is a single line text input device. For example:

```
1 Frame f = new Frame("TextField");
2 TextField tf = new TextField("Single line", 30);
3 tf.addActionListener(this);
4 f.add(tf);
```

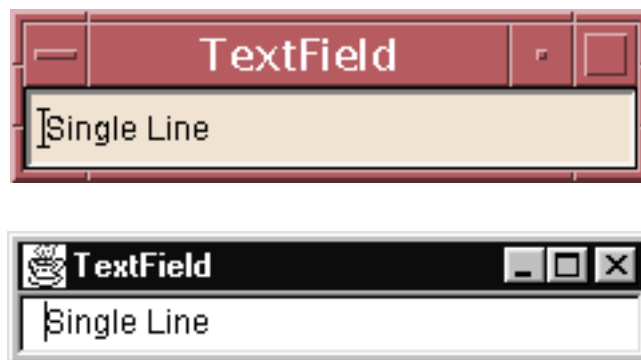


Figure C-8 TextField

Because you can only have one line, an `ActionListener` can be informed, using `actionPerformed()`, when the Enter or Return key is pressed. You can add other component listeners if desired.

Note – The second argument in the `TextField` constructor is used for the number of characters that are visible. There is no limit on the number of characters allowed in the `TextField`. Scrolling occurs when the text overflows.

TextField

You can use the `TextField` application to mask certain keys from input. The following code creates a `TextField` that ignores the typing of digits:

```
1 import java.awt.*;
2 import java.awt.event.*;
3
4 public class SampleTextField {
5     private Frame f;
6     private TextField tf;
7
8     public void go() {
9         f = new Frame("TextField");
10        tf = new TextField("Single Line", 30);
11        tf.addKeyListener( new NameHandler() );
12        f.add(tf, BorderLayout.CENTER);
13        f.pack();
14        f.setVisible(true);
15    }
16
17    class NameHandler extends KeyAdapter {
18        public void keyPressed(KeyEvent e) {
19            char c = e.getKeyChar();
20            if ( Character.isDigit(c)) {
21                e.consume();
22            }
23        }
24    }
25
26    public static void main (String args[]) {
27        SampleTextField txtf = new SampleTextField();
28        txtf.go();
29    }
30 }
```

TextArea

The `TextArea` is a multiple-line, multiple-column text input device. You can set it to read-only, using the method `setEditable(boolean)`. It displays horizontal and vertical scrollbars.

The following example sets up a 4 row X 30 character text area containing "Hello!" initially.

```
1 f = new Frame("TextArea");
2 ta = new TextArea("Hello!", 4, 30);
3 f.add(ta, BorderLayout.CENTER);
```

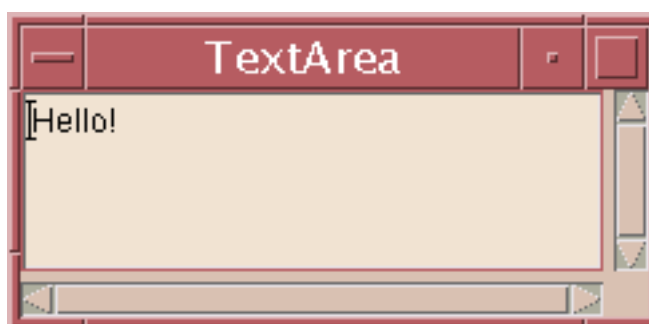


Figure C-9 `TextArea`

The listener you specify with `addTextListener()` receives notification of key strokes in the same way that `TextField` does.

You can add general component listeners to the text area, but because the text is multi-line, pressing the Enter key puts another character into the buffer. If you need to recognize "completion of input," you can put an Apply or Commit button next to the text area to allow the user to indicate this.

Text Components

Both `TextArea` and `TextField` are documented in two parts. If you look up a class called `TextComponent` you find several methods that `TextArea` and `TextField` have in common; for example, `setEditable()`. This is because `TextArea` and `TextField` are both subclasses of `TextComponent`.

You have seen that the constructors for both `TextArea` and `TextField` classes allow you to specify the number of columns for the display. Remember that the size of a displayed component is the responsibility of a layout manager, so these preferences might be ignored. Furthermore, the number of columns is interpreted in terms of the average width of characters in the font that is being used. The number of characters that are actually displayed might vary radically if a proportionally spaced font is used.

`TextField` and `TextArea` components inherit the default behavior for keystrokes from `TextComponent`; that is, the characters are added to an internal character buffer and are displayed on the screen. The character buffer can be retrieved (as a `String`) by the `getText()` method.

List

A `List` presents text options that are displayed in a region that allows several items to be viewed at one time. The `List` is scrollable and supports both single- and multiple-selection modes. For example:

```
1 List lst = new List(4, true);
2 lst.add("Hello");
3 lst.add("there");
4 lst.add("how");
```



Figure C-10 List

The numeric argument to the constructor defines the preferred height of the list in terms of the number of visible rows. As always, a layout manager can override this value. A true boolean argument indicates that the list should allow the user to make multiple selections.



Figure C-11 List With Items Selected

When an item is selected or deselected, AWT sends an instance of `ItemEvent` to the list. When the user double-clicks on an item in a scrolling list, an `ActionEvent` is generated by the list in both single- and multiple-selection modes. Items are selected from the list according to platform conventions. For a UNIX Motif environment, a single click highlights an entry in the list, but you must double click to trigger the action of the list.

Dialog

A `Dialog` component is associated with a `Frame`. It is a free-standing window with some decorations. It differs from a `Frame` in that fewer decorations are provided and you can request a “modal” dialog, which causes it to store all forms of input until it is closed.



Figure C-12 Dialog

`Dialog` is either modeless or modal. A modeless `Dialog` means you can interact with both the `Frame` and the `Dialog` at the same time. A modal `Dialog` blocks input to the remainder of the application, including the `Frame`, until the `Dialog` box is dismissed.

Because `Dialog` subclasses `Window`, its default layout manager is `BorderLayout`.

```

1    d = new Dialog(f, "Dialog", true);
2    d.setLayout(new GridLayout(2,1));
3    dl = new Label("Hello, I'm a Dialog");
4    dbl = new Button("OK");
5    d.add(dl);
6    d.add(dbl);
7    d.pack();

```

The first argument in the `Dialog` constructor designates the owner of the `Dialog` that is being constructed. In the previous example, `f` is the `Frame` that owns the `Dialog`.

Dialog

A Dialog is usually not made visible to the user when it is first created. It is displayed in response to some user interface action, such as the pressing of a button.

```
public void actionPerformed(ActionEvent ev) {  
    d.setVisible(true);  
}
```

Note – Treat a Dialog as a reusable device. That is, do not destroy the individual object when it is dismissed from the display; keep it so it can be used later. The garbage collector can make it too easy to waste memory. Remember, creating and initializing objects takes time and should not be done without some thought.

To hide a Dialog, you must call `setVisible(false)`. You do this by adding a `WindowListener` to it and awaiting a call to the `windowClosing()` method in that listener. This parallels the handling of closing a `Frame`.

- ✓ **Handling window events is more complicated. The listener class can extend `WindowAdapter` or implement `WindowListener`. The `PaintGUI` class in this module's solution directory contains an example of Window handling code.**

FileDialog

FileDialog is an implementation of a file selection device. It has its own free standing window, and allows the user to browse the file system and select a particular file for further operations. For example:

```
1 FileDialog d = new FileDialog(parentFrame, "FileDialog");
2 d.setVisible(true); // block here until OK selected
3 String fname = d.getDirectory() + d.getFile();
```

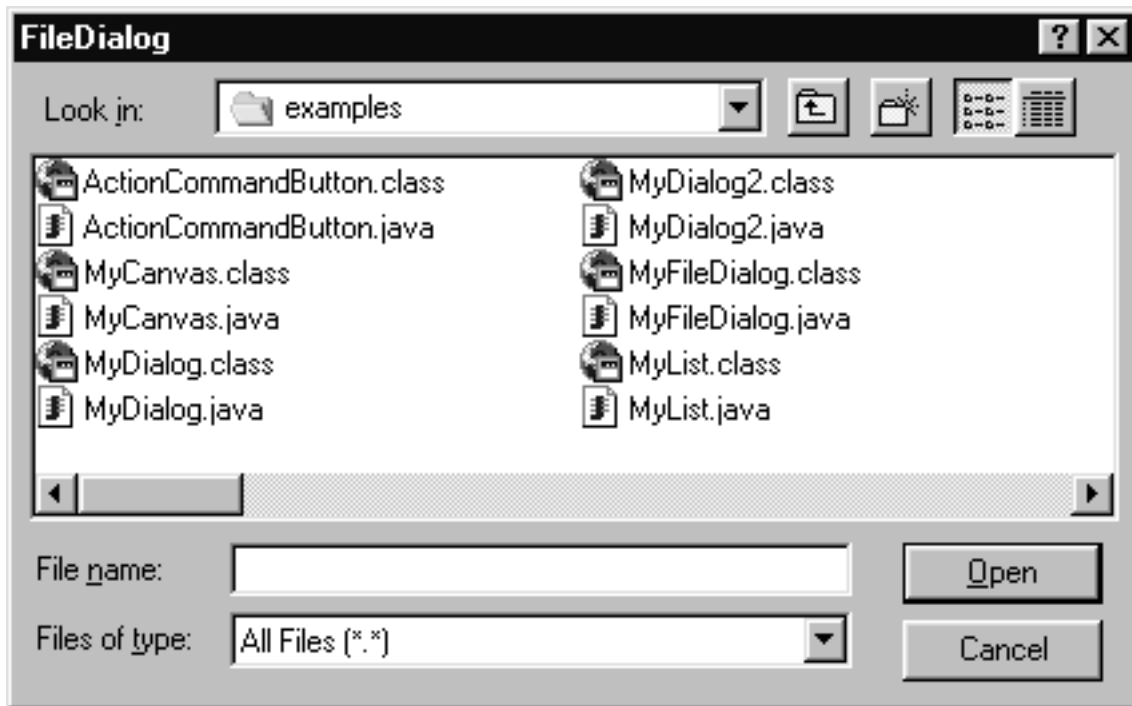


Figure C-13 Microsoft Windows Implementation of FileDialog

In general, it is not necessary to handle events from the FileDialog. The setVisible(true) call blocks events until the user selects OK, at which point the name of the selected file is requested. This information is returned as a String.

✓ **You can use the FileDialog constructor that takes three parameters, and send FileDialog. You must use SAVE as the third parameter to get a "Save" dialog.**

FileDialog

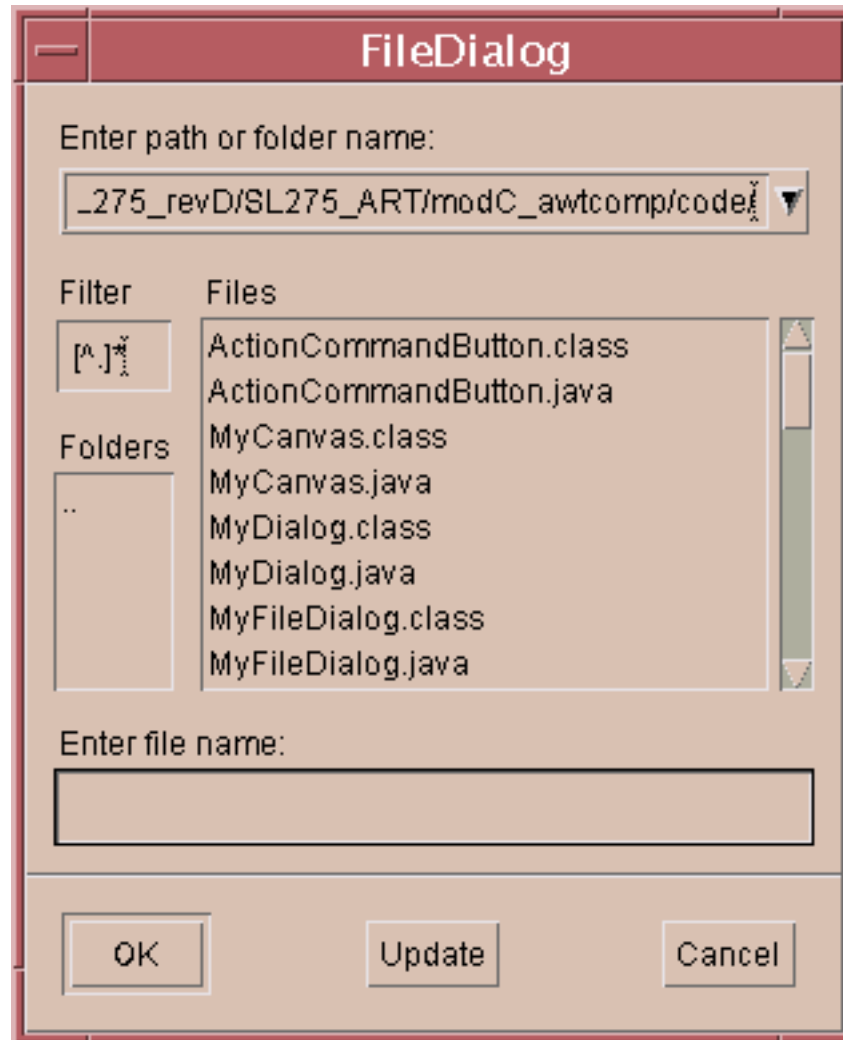


Figure C-14 Solaris Operating Environment Implementation of FileDialog

ScrollPane

ScrollPane provides a general container that cannot be used as a free standing component. It should always be associated with a container (for example, a Frame). It provides a viewport onto a larger region and scrollbars to manipulate that viewport. For example:

```

1  Frame f = new Frame("ScrollPane");
2  Panel p = new Panel();
3  ScrollPane sp = new ScrollPane();
4  p.setLayout(new GridLayout(3, 4));
5  .
6  .
7  .
8  sp.add(p);
9  f.add(sp, BorderLayout.CENTER);
10 f.setSize(100, 100);
11 f.setVisible(true);

```

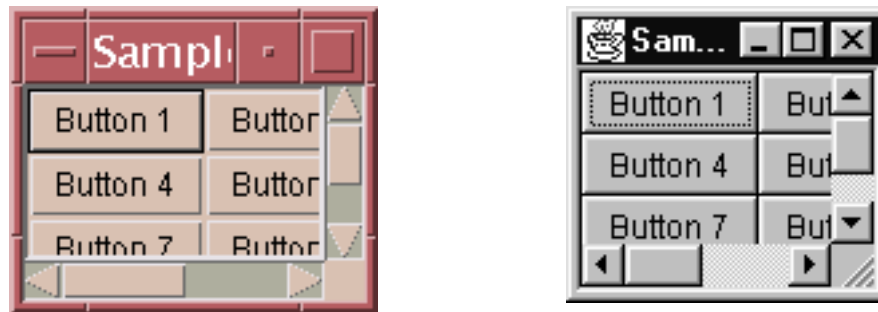


Figure C-15 ScrollPane

The ScrollPane creates and manages the scroll bars and holds a single component. You cannot control the layout manager it uses. Instead, you can add a Panel to the scroll pane, configure the layout manager of that panel, and place your components in that panel.

Generally, you do not handle events on a ScrollPane; events are handled through the components that they contain.

PopupMenu

The `PopupMenu` provides a standalone menu that can be displayed on any component. You can add items or menus to a pop-up menu. For example:

```
1 Frame f = new Frame("PopupMenu");
2 Button b = new Button("Press Me");
3 PopupMenu p = new PopupMenu("Popup");
4 MenuItem s = new MenuItem("Save");
5 MenuItem ld = new MenuItem("Load");
6 b.addActionListener(this);
7 f.add(b, BorderLayout.CENTER);
8 p.add(s);
9 p.add(ld);
10 f.add(p);
```

For the `PopupMenu` to be displayed, you must call the `show()` method. The `show()` method requires a component reference to act as the origin for the x and y coordinates. Usually you would use the trigger component for this, but in this case, the trigger is the `Button b`.

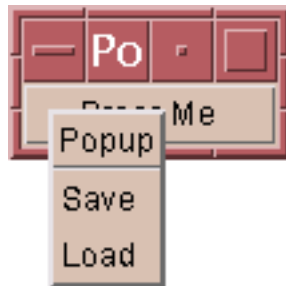


Figure C-16 `PopupMenu`

- ✓ *The Microsoft Windows version of the pop-up menu is not available because it does not work properly in Java 2 SDK. You get a pop-up menu containing two buttons that are labeled Press Me. This could be an opportunity to discuss compatibility. A bug report has been filed.*


Popup Menu

```
1 public void actionPerformed(ActionEvent ev) {  
2  
3     // display popup at (10,10) relative to b  
4     p.show(b, 10, 10);  
5 }
```

Note – You must add the `PopupMenu` to a “parent” component. This is not the same as adding components to containers. In this example, the pop-up menu has been added to the enclosing `Frame`.

✓ *At this point, students might have questions about the restrictions on the containment hierarchy.*

Using the GridBagLayout

D 

This appendix discusses the use of the GridBagLayout in the production of complex user interfaces.



Layout Managers

- Position and size components in a `Container`
- Adhere to a policy
- Make absolute coordinates platform dependent
- Determine limitations of:
 - `FlowLayout`
 - `GridLayout`
 - `BorderLayout`

Layout Managers

GUIs should make extensive use of layout managers, because the alternative, absolute positioning by pixel coordinates, is not platform portable. Issues such as the sizes of fonts and screens ensure that a layout that is correct and based on coordinates is unusable on any other platform.

Layout managers avoid these difficulties by laying out the GUI according to a 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.

This course assumes you know about the basic three layout managers, `FlowLayout`, `GridLayout`, and `BorderLayout`. If you are unsure about any of these, ask your instructor if you can discuss them during a break.

Layout Managers

If you know the basic three layout managers, you also know that they are somewhat limited in their capabilities, and that it can be hard, often involving many nested panels, to produce a layout that is useful in a production program. This appendix looks at the `GridBagLayout`, which is more powerful.



The GridBagLayout

- Divides the region into rows and columns
- Sizes components to fit width, height, both, or neither of their *regions* (one or more contiguous rows and one or more contiguous columns)

The GridBagLayout

The GridBagLayout lays out components using a grid. However, unlike the GridLayout, child components are not necessarily constrained to occupy exactly one entire grid cell, neither are all rows and columns equal in size. Rather, a component can be assigned multiple cells, horizontally, vertically, or both, and can exist within that region.

The GridBagLayout

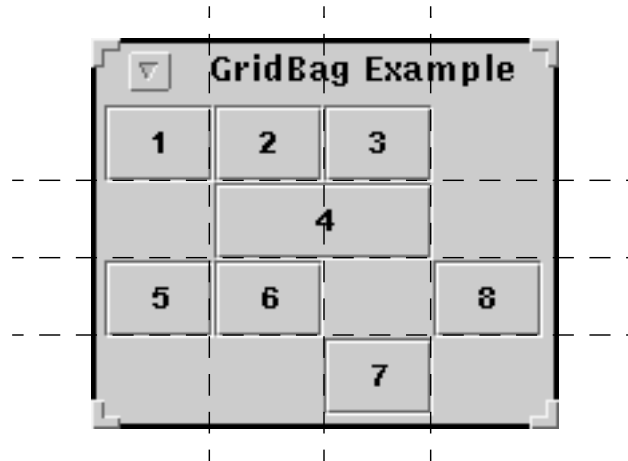


Figure D-1 Sample GridBagLayout With Four Rows and Four Columns

The number of rows and columns in a GridBagLayout is determined by the number of cells that are in use. This contrasts with the GridLayout where (generally) you specify the row and column count at the time the layout is constructed.

The basic height of a row is determined by the largest component in that row. Similarly, the basic width of a column depends on the largest component in it. In Figure D-1, each grid cell is the basic size of a JButton with a single-digit label.

The GridBagLayout

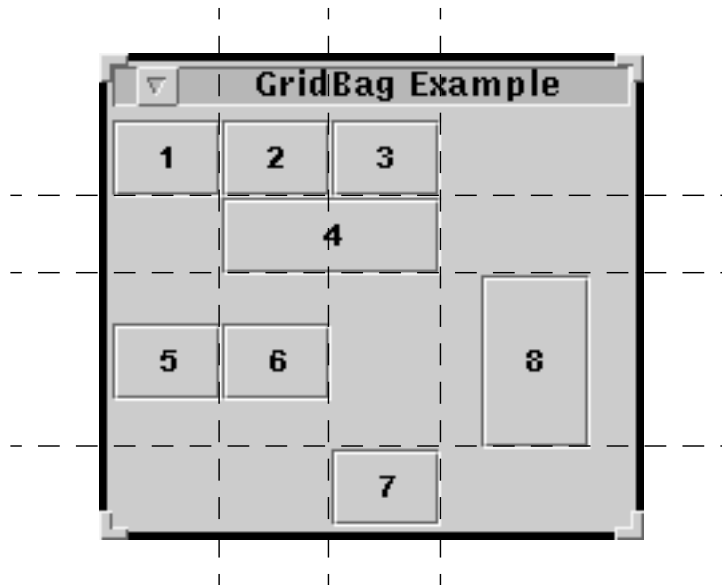


Figure D-2 Sample GridBagLayout Showing Cells Expanded by Weight

When the total space available to the GridBagLayout exceeds that needed for all the basic dimensions, the extra space is shared using a concept called *weight*. In Figure D-2, the weight has been applied to the last column and to the third row (that is, the row and column that includes the button labeled “8”).

A component in a GridBagLayout can occupy multiple consecutive rows, and multiple consecutive columns if desired. The total space allotted to one component is referred to as the component’s *region*. In Figure D-2, the button labeled “4” extends across two columns horizontally.

The size of a component in a GridBagLayout is not necessarily constrained to occupy the entire assigned region. Instead, the component can have its natural size, its natural height with the full width of its region, or its natural width with the full height of its region. Of course, it can also be constrained to fill the region. This property is known as the *fill* of a component. In Figure D-2, the buttons labeled “5” and “6” do not fill the vertical space available to them; similarly, the button labeled “8” fills vertically, but not horizontally.

The GridBagLayout

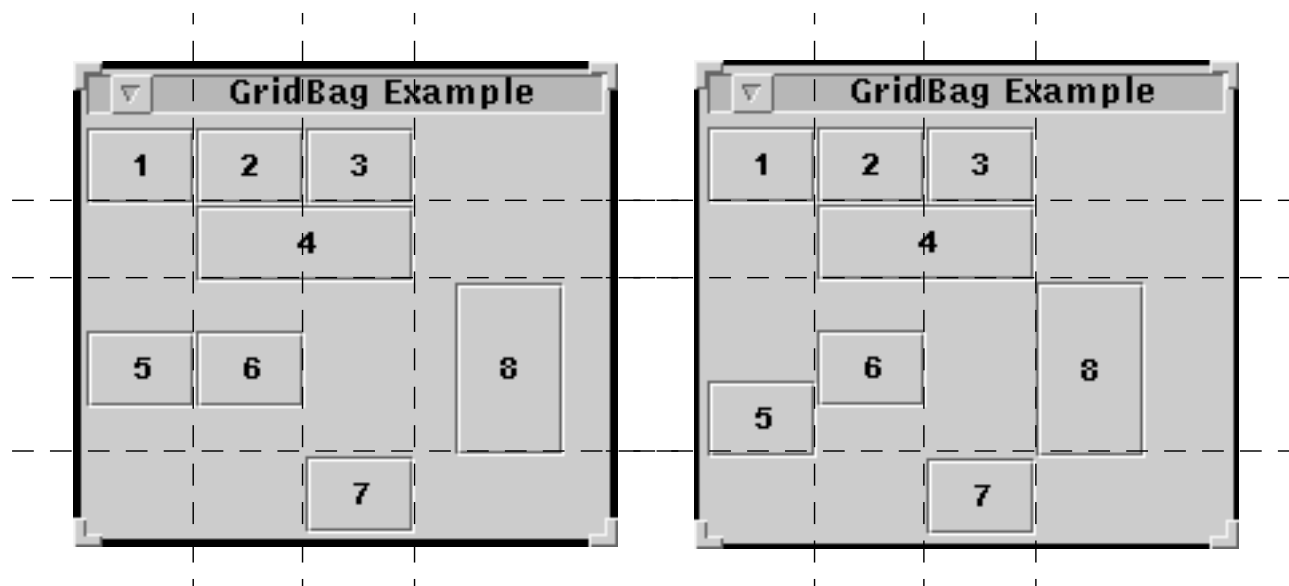


Figure D-3 Sample GridBagLayout Showing the Effect of Anchor

When a component does not fill the entire region allocated to it, its position within that region can be controlled using a concept called *anchor*. Anchor takes one of nine values. Eight of these values are compass points, NORTH, SOUTHWEST, and so on. The ninth is CENTER. If a component has its natural size and an anchor of NORTHWEST, then it will be positioned at the top left of its allocated region.

✓ *In some systems (XMotif for example) the effect of anchor is referred to as gravity.*

In Figure D-3, the two examples have differing anchor settings. Specifically, the button labeled “5” has a CENTER anchor in the left-hand example, but a SOUTH anchor in the right-hand example. The button labeled “8” has a CENTER anchor in the left-hand example, but a WEST anchor in the right-hand example.

Clearly, there is some interaction between anchor and the fill of a component. If the fill specifies that the component occupies the entire region allotted to it, then anchor has no significance. If the fill value specifies that a component occupies the allocated region entirely in the horizontal direction, then the only anchor values that are useful are NORTH, CENTER, and SOUTH



The GridBagConstraints Class

- For each component, specify:
 - Top left corner of cell with `gridx` and `gridy`
 - Cell size with `gridwidth` and `gridheight`
 - Capacity with `fill`
 - `anchor`
- For each row and column, specify:
 - Capacity with `weightx` and `weighty`

The GridBagConstraints Class

You have seen the principles by which the `GridBagLayout` makes positioning decisions, but not how those preferences are supplied to it. This is done using an object of the class `GridBagConstraints`. Each time you add a `Component` to a `Container` that has a `GridBagLayout`, you provide an instance of `GridBagConstraints` that contains the values needed to describe the layout of that `Component`.

The most significant fields of the `GridBagConstraints` object are:

- `gridx` and `gridy` – These integer fields are used to specify the row and column numbers at the top left of the component's region. They are effectively the component's coordinates.
- `gridwidth` and `gridheight` – These integer fields describe the number of columns and rows, respectively, over which the component's region extends.
- `fill` – This field indicates how the component is sized within its region. Values for this field are constants in the `GridBagConstraints` class. The four symbolic values are: `NONE`, `HORIZONTAL`, `VERTICAL`, and `BOTH`.

The GridBagConstraints Class

- **anchor** – This field indicates the anchor applied to the component. Values are constants in the `GridBagConstraints` class. The nine symbolic values are: `NORTH`, `SOUTH`, `EAST`, `WEST`, `NORTHEAST`, `NORTHWEST`, `SOUTHEAST`, `SOUTHWEST`, and `CENTER`.
- **weightx** and **weighty** – These fields are somewhat unusual in that they apply to the column and row to which the component is being added, not the component itself. The weight values are used to distribute “spare” space when the layout has more screen area available to it than it needs. The actual values of weight are significant only in a relative sense. That is, it does not matter if a particular value is 5 or 0.5. What matters is the proportion of the weight allocated to the sum of all weights allocated.

Note – Avoid setting weights on the same row or column for more than one component. Doing so will confuse anyone reading the program.



Designing with GridBagLayout

- Sketch all components
- Sketch all components on resized container
- Identify all gridlines and rowhence / column counts
- Identify stretchy rows / columns and allocate weights
- Identify starting row / column for each component
- Identify width / height for each component
- Identify `fill` for each component
- Identify `anchor` for each component
- Define row / column weights for each component

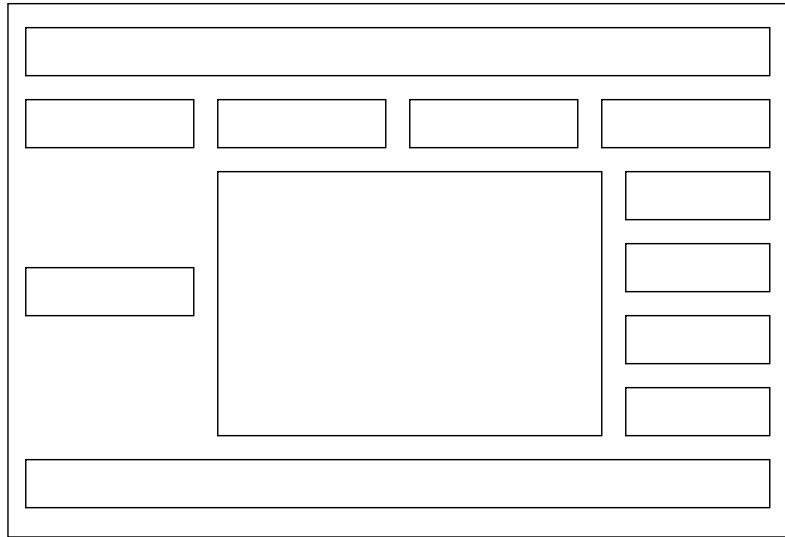
Designing With GridBagLayout

Design Steps

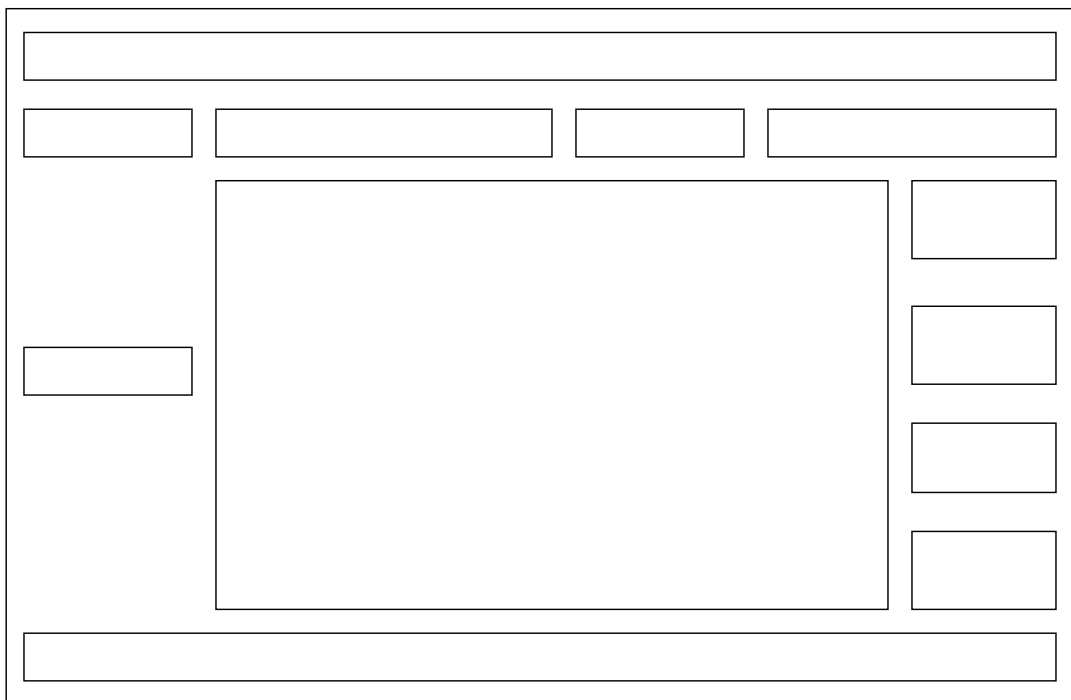
When designing with a GridBagLayout:

1. Sketch the components as you want them to appear.
2. Make another sketch with the window enlarged, and plan how you want the extra space to be allocated.

Designing With GridBagLayout

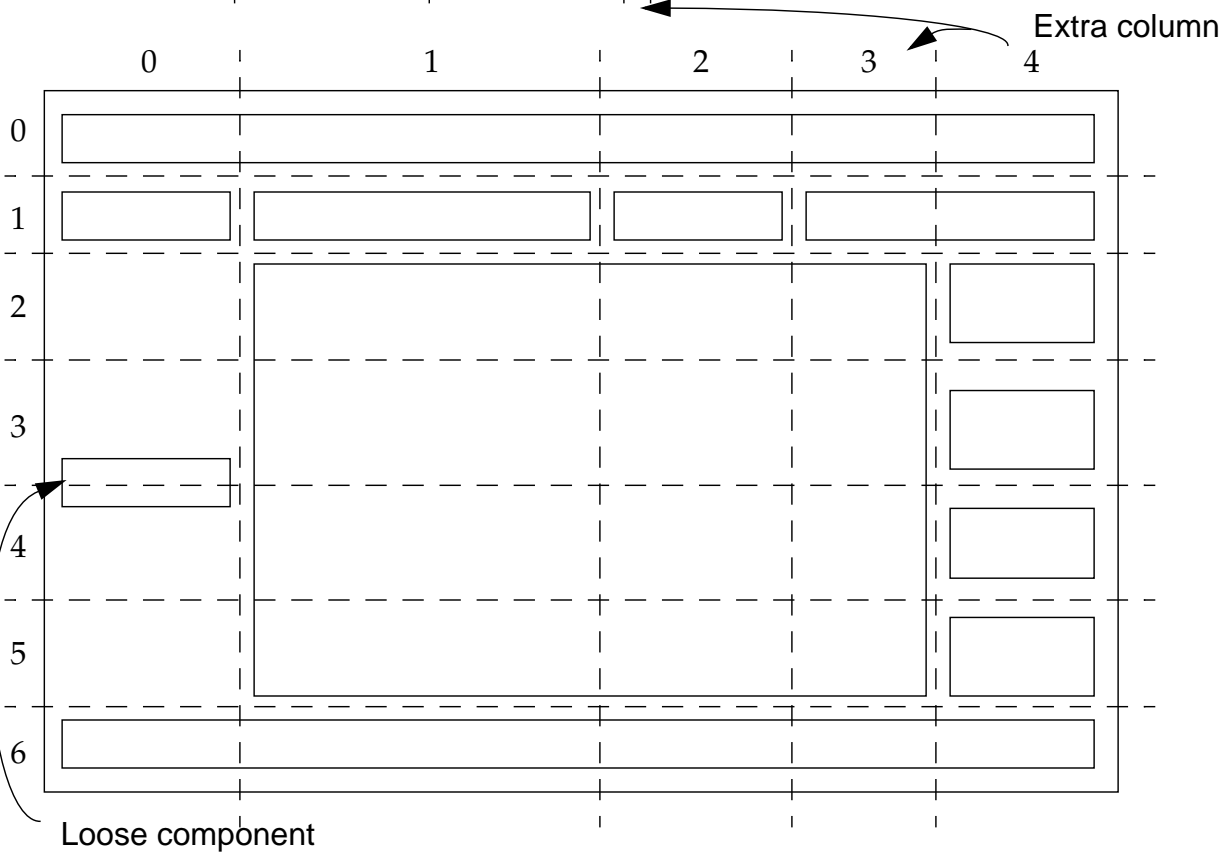
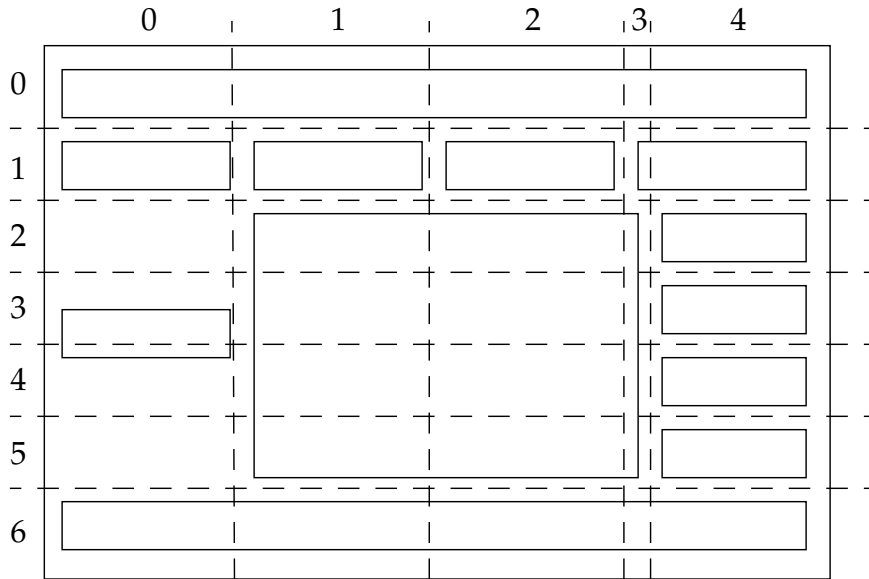


Basic, unexpanded layout proposal



Basic, expanded layout proposal

- Identify the gridlines based on the edges of components in your pictures. Be particularly careful if your diagram shows two component edges in nearly the same alignment—did you mean them to be aligned? When you have identified the gridlines on one drawing, do this again on the second sketch.



Designing With GridBagLayout

4. Decide how you want to allocate the extra space. In some cases, it might be easiest to do this in terms of percentages. Once you have determined your percentages, you can use them as `weightx` and `weighty` values directly (even if they do not finally add up to 100).

The expanded version brings out the existence of an extra column, which is not really noticeable until the display is expanded. You would be unlikely to recognize this column's existence in the unexpanded diagram.

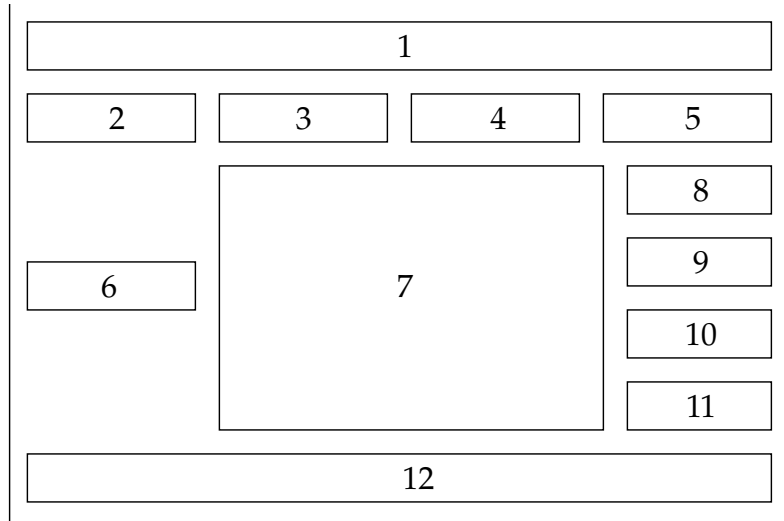
The "loose component" in column 0, third row down, does not match any of the grid cell boundaries. Rather, it appears to overlap rows 4 and 5. The component actually is located in a region that extends over rows 3 through 6 inclusive, and is vertically centered in that region.

Columns 0, 2, and 4 do not change size, but columns 1 and 3 do. It is not entirely clear how the space is shared, but a reasonable working guess is that new space is allocated equally between them.

Rows 0, 1, and 6 do not change size, but rows 2 through 5 all stretch equally.

Designing With GridBagLayout

- Now that you have designed the underlying grid, you can start to position each component over that grid. Start by identifying the top left row and column for each component region; this gives you the `gridy` and `gridx` values for each.

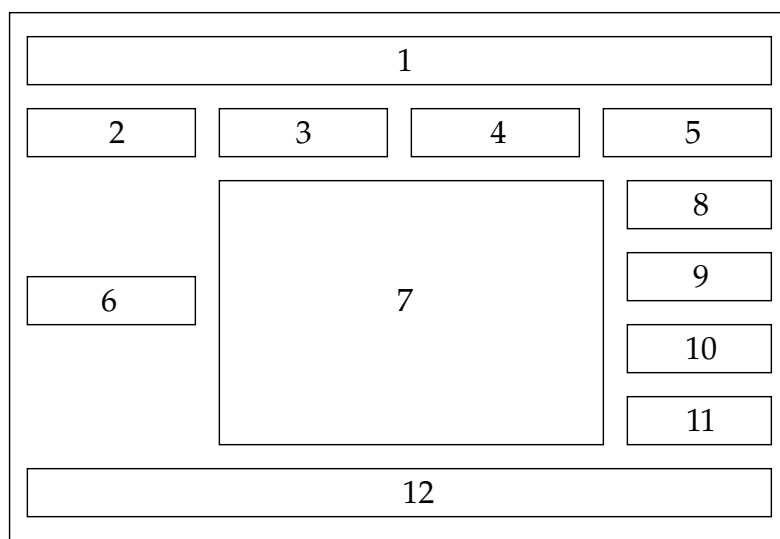


- Determine the width and height of the region in terms of columns and rows; these are the `gridwidth` and `gridheight` values.

Component	gridx	gridy	gridwidth	gridheight
1	0	0	5	1
2	0	1	1	1
3	1	1	1	1
4	2	1	1	1
5	3	1	2	1
6	0	2	1	4
7	1	2	3	4
8	4	2	1	1
9	4	3	1	1
10	4	4	1	1
11	4	5	1	1
12	0	6	5	1

Designing With GridBagLayout

- For each component, consider how it occupies the region allocated to it. If it fills the region entirely, it has a `fill` value of `BOTH`. If it fills the region from side to side but not vertically, then its `fill` value is `HORIZONTAL`. If it fills its region vertically but not horizontally, then its `fill` value is `VERTICAL`. If it does not fill the region in either direction, then its `fill` value is `NONE`.



The `fill` value should be `BOTH` for all components that take the full size of their available regions. This is important even if the region does not stretch. For example, the cells occupied by components 8, 9, 10, and 11 do not stretch horizontally, so you might think that a horizontal component of `fill` was unnecessary. However, if you specify only a `fill` of `VERTICAL`, you will find that the components are given their preferred sizes, and because their labels are shorter, components 8 and 9 are slightly smaller than components 10 and 11.

So, in this example, the only component that is not set to fill `BOTH` is component 6. This should have a `fill` value of `HORIZONTAL`, to ensure that it takes up the full width of its region.

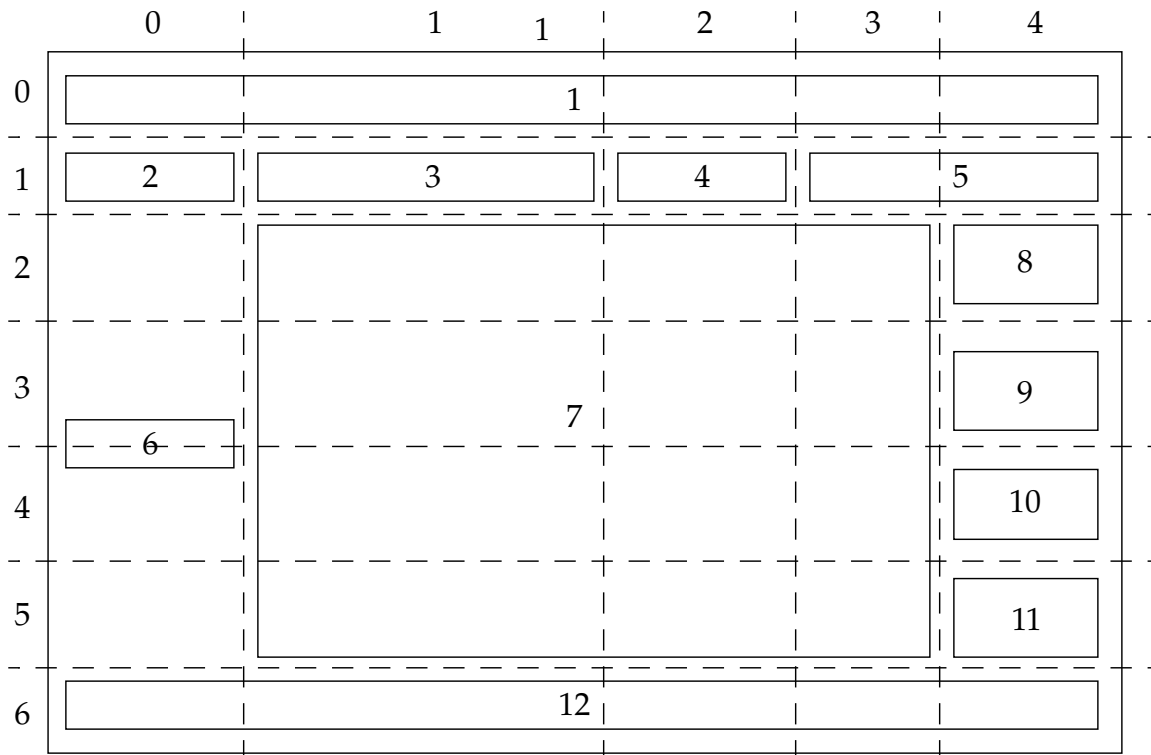
Designing With GridBagLayout

8. For each component, consider how it is positioned within the region allocated to it and hence the anchor value for the component. If a component has a `fill` value of `BOTH` then the anchor value is irrelevant. Components with `HORIZONTAL` `fill` should have an anchor of `NORTH`, `CENTER`, or `SOUTH`. Components with `VERTICAL` `fill` should be anchored `WEST`, `CENTER`, or `EAST`. Components with a `fill` of `NONE`, can have an anchor of any of the nine values.

Anchor values are significant only when a component's region is larger than the component itself. In this example, this applies only to component 6. Here the component must be centered vertically, although it fills the available width. In consequence, any of the anchor values `EAST`, `WEST`, or `CENTER` would result in the required behavior, but `CENTER` is probably the most reasonable because it most directly expresses the required result.

Designing With GridBagLayout

9. Add the components and allocate the weights to the rows and columns. Choose one component for each row and one component for each column for the weight values. These components should occupy only one column if they are providing `weightx`, and one row if they are providing `weighty`. If possible, use components on the top row to specify `weightx` and components in the left column to specify `weighty`.



Designing With GridBagLayout

Example

To allocate weights to the rows and columns, identify one component in each column that needs to stretch horizontally, and one component in each row that stretches vertically. These components should occupy only a single cell along the axis of stretch, and be near the edges of the layout. This improves the consistency and readability of your code.

For this example, the stretch is in columns 1 and 3, and rows 2 through 5.

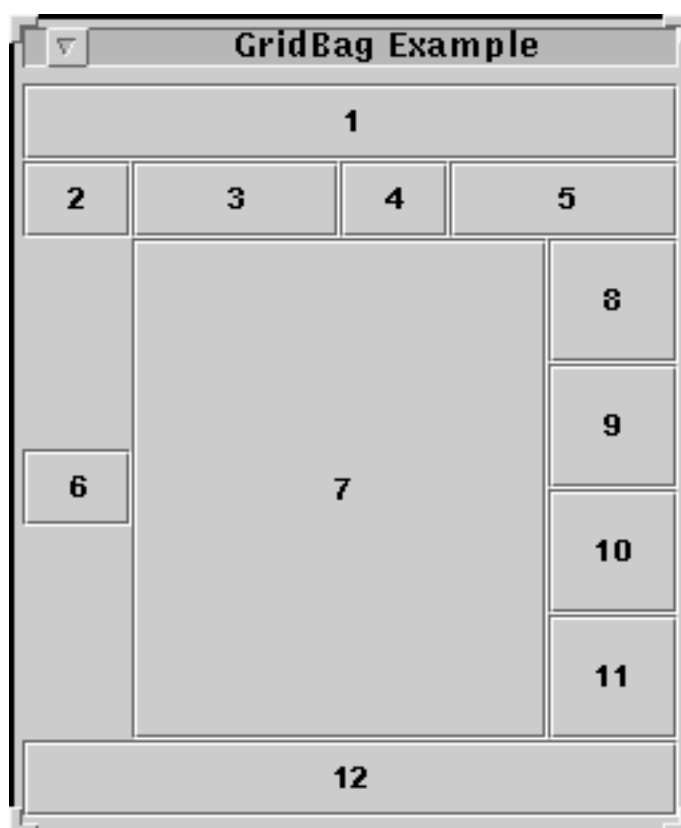
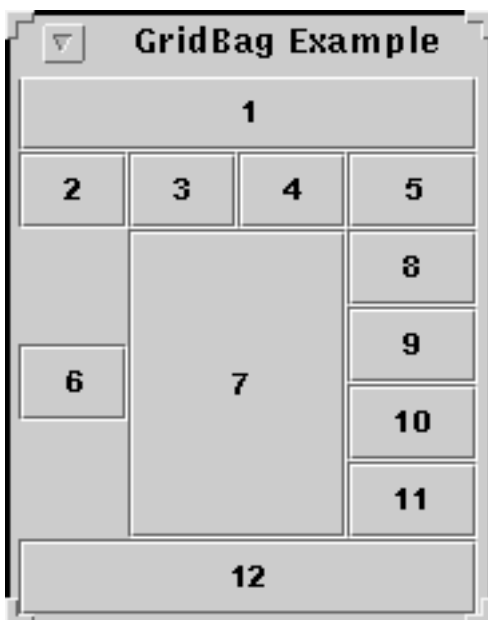
Components 8, 9, 10, and 11 are suitable to apply the vertical weight values for rows 2 through 5, and component 3 is appropriate to apply the horizontal weight for column 1. However, there is no obvious component with which a horizontal weight can be applied to column 3.

One way to approach this is to add a dummy component to the cell at row 2, column 3. This component must have zero by zero size so that it does not obscure component 8. A Canvas is suitable for this because its preferred size is zero by zero, unless explicitly set otherwise. Once added into row 3 column 4, it remains at zero size provided it has a fill value of NONE.

You can use this approach to simplify any layout. Create an extra row and an extra column along the bottom and right hand edges of your layout. Populate these cells with zero-sized canvases and use them to apply weights.

Designing With GridBagLayout

Example (Continued)



Once you have applied the `GridBagConstraints` values to components added to a `GridBagLayout`, the desired behavior is achieved. The screen shots shown here are derived from the implementation program listed on the next page.

Designing With GridBagLayout

Example (Continued)

The following is the main part of the program for this example, with the values used in the GridBagConstraints.

```
1 import java.awt.*;
2 import javax.swing.*;
3
4 public class ExampleGB {
5     public static void main(String args[]) {
6         JFrame f = new JFrame("GridBag Example");
7         Container c = f.getContentPane();
8         c.setLayout(new GridBagLayout());
9         GridBagConstraints.add(c, new Canvas(), 3, 2, 1, 1, 1, 0,
10            GridBagConstraints.NONE, GridBagConstraints.CENTER);
11         GridBagConstraints.add(c, new JButton("1"), 0, 0, 5, 1, 0, 0,
12            GridBagConstraints.HORIZONTAL, GridBagConstraints.CENTER);
13         GridBagConstraints.add(c, new JButton("2"), 0, 1, 1, 1, 0, 0,
14            GridBagConstraints.BOTH, GridBagConstraints.CENTER);
15         GridBagConstraints.add(c, new JButton("3"), 1, 1, 1, 1, 1, 0,
16            GridBagConstraints.HORIZONTAL, GridBagConstraints.CENTER);
17         GridBagConstraints.add(c, new JButton("4"), 2, 1, 1, 1, 0, 0,
18            GridBagConstraints.BOTH, GridBagConstraints.CENTER);
19         GridBagConstraints.add(c, new JButton("5"), 3, 1, 2, 1, 0, 0,
20            GridBagConstraints.HORIZONTAL, GridBagConstraints.CENTER);
21         GridBagConstraints.add(c, new JButton("6"), 0, 2, 1, 4, 0, 0,
22            GridBagConstraints.HORIZONTAL, GridBagConstraints.CENTER);
23         GridBagConstraints.add(c, new JButton("7"), 1, 2, 3, 4, 0, 0,
24            GridBagConstraints.BOTH, GridBagConstraints.CENTER);
25         GridBagConstraints.add(c, new JButton("8"), 4, 2, 1, 1, 0, 1,
26            GridBagConstraints.BOTH, GridBagConstraints.CENTER);
27         GridBagConstraints.add(c, new JButton("9"), 4, 3, 1, 1, 0, 1,
28            GridBagConstraints.BOTH, GridBagConstraints.CENTER);
29         GridBagConstraints.add(c, new JButton("10"), 4, 4, 1, 1, 0, 1,
30            GridBagConstraints.BOTH, GridBagConstraints.CENTER);
31         GridBagConstraints.add(c, new JButton("11"), 4, 5, 1, 1, 0, 1,
32            GridBagConstraints.BOTH, GridBagConstraints.CENTER);
33         GridBagConstraints.add(c, new JButton("12"), 0, 6, 5, 1, 0, 0,
34            GridBagConstraints.HORIZONTAL, GridBagConstraints.CENTER);
35         f.pack();
36         f.setVisible(true);
37     }
```

Designing With GridBagLayout

Example (Continued)

Supporting the code on the previous page is this inner class. It provides the add method that simplifies setting up the GridBagConstraints values.

```
38 static class GridBagAdder {
39     // OK to reuse this as we overwrite all elements every time
40     // Note that this is not threadsafe however!
41     static GridBagConstraints cons = new GridBagConstraints();
42     public static void add(Container cont,Component comp,int x, int y,
43         int width,int height,int weightx,int weighty,
44         int fill,int anchor) {
45
46         cons.gridx = x;
47         cons.gridy = y;
48         cons.gridwidth = width;
49         cons.gridheight = height;
50         cons.weightx = weightx;
51         cons.weighty = weighty;
52         cons.fill = fill;
53         cons.anchor = anchor;
54         cont.add(comp, cons);
55     }
56 }
57 }
```



RELATIVE and REMAINDER

- Shorthand for position, size, or both
- For `gridx/gridy`:
`RELATIVE` => extends to the next position
- For `gridwidth/gridheight`:
`RELATIVE` => extends to last one
- For `gridwidth/gridheight`:
`REMAINDER` => extends to last one
- Careful use of these helps maintenance, but it:
 - Makes adding order significant
 - Might decrease readability of code

RELATIVE *and* REMAINDER

Where a layout involves a large number of components in a fairly simple layout, it can be quite time consuming to set up the `gridx` and `gridy` values for each one. This situation is aggravated when maintenance is needed; for example, to insert one new component.

To help with this situation, you can use the value `RELATIVE` to indicate that a component should be positioned just to the right, or just underneath, the one previously added.

In addition, you can use `RELATIVE` as a value in the `gridwidth` and `gridheight` fields, to make the component extend over all rows below, or all columns to the right, of the one to which the component is added, *except the last row or column*.

If you set the value `REMAINDER` in a `gridwidth` or `gridheight` field, the component extends to the very last row or column.

RELATIVE and REMAINDER

Careful use of these shorthand features can make code easier to write and shorter, which can make it easier to read. However, in some situations, the layout is dependent on the order of adding components and actually makes the code more difficult to read.

Objectives

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

- Identify the key features of Java Foundation Classes
- Describe the key features of the `javax.swing` package
- Identify Swing components
- Define *containers* and *components*, and explain how they work together to build a Swing GUI
- Write, compile, and run a basic Swing application
- Use top-level containers, such as `JFrame` and `JApplet` effectively

Java 2 SDK offers the Java Foundation Classes (JFC), part of which is Swing. Swing is a set of components (written in 100% Pure Java™ for platform independence), layered on top of the AWT. This appendix introduces JFC, and the implementation of Swing GUIs.

Additional Resources



Additional resources – the following reference can provide additional details on the topics discussed in this appendix:

- *The Java Tutorial*, an online tutorial from Sun Microsystems, available from:
<http://java.sun.com/docs/books/tutorial>

Swing Introduction

The *Java Foundation Classes* (JFC) are a comprehensive set of GUI components and services that dramatically simplify the development and deployment of robust Java applications.

JFC, an integral part of Java 2 SDK, is primarily composed of five APIs: AWT, Java 2D, Accessibility, Drag and Drop, and Swing. It provides a full set of application development packages to assist the developer in designing complex interactive applications.

The AWT components, as discussed in Appendix C, "The AWT Component Library," provide a variety of GUI tools for a wide class of Java applications.

Java 2D is a graphics API designed to provide Java applications with an advanced set of classes for two-dimensional (2D) graphics and imaging. The Java 2D API extends the capabilities of the `java.awt` and the `java.awt.image` packages and provides a rich set of paint styles, mechanisms for defining complex shapes, methods, and classes for fine-tuning the rendering process. An extended font set is included in this API.

The *Accessibility* API provides an advanced set of tools to assist in developing applications that use non-traditional means for input and output. It provides an interface for assistive technologies, such as screen readers, screen magnifiers, audible text readers (speech processing), and so on.

Drag and Drop technology provides interoperability between Java and native applications to exchange data across Java applications and those applications that do not support Java technology.

This JFC appendix focuses primarily on *Swing*. Swing is designed for forms-based application development. It provides a rich set of components and a framework to specify how to present GUIs that are visually independent of the platform.

Swing Introduction

Swing provides a full set of GUI components written in the Java programming language for portability.

Pluggable Look and Feel

Pluggable look and feel enables developers to build applications that execute on any platform as if it were developed for that specific platform. A program executed in the Microsoft Windows environment behaves as if it were developed for this environment; and the same program executed on the UNIX platform behaves as if it were developed for the UNIX environment.

Developers can create their own custom Swing components, with any kind of look and feel they want 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.

Swing – Introduction

Swing Architecture

Swing provides a more comprehensive set of components than the AWT, introducing new features and rich capabilities. The Swing APIs are built around a number of APIs that implement various parts of the AWT. This ensures that all of the earlier AWT components can still be used. Most Swing components do not use any of the platform-specific implementations that the AWT does, which gives Swing its customization and pluggable look and feel features.

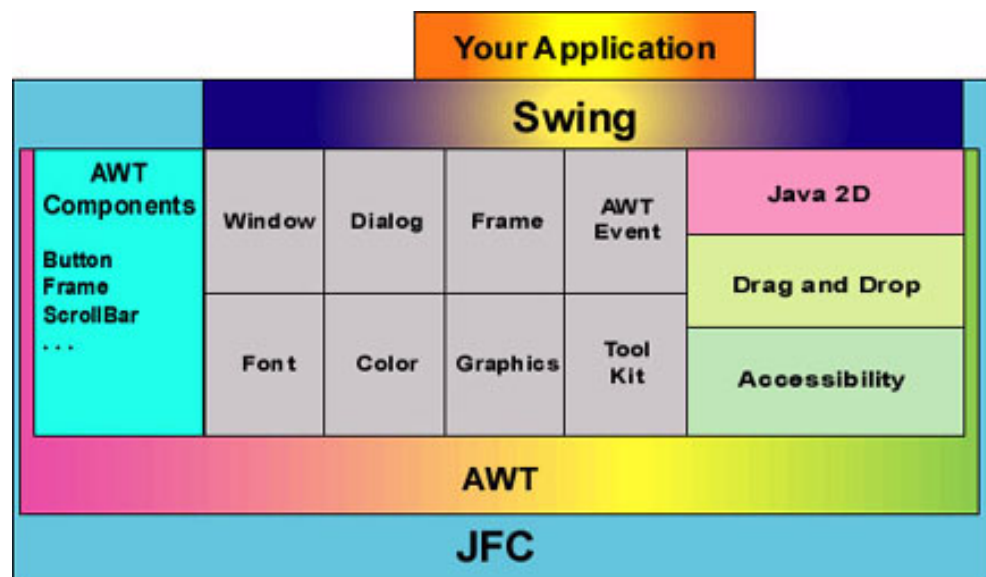


Figure E-1 Java Foundation Classes

Figure E-1 illustrates the interrelationship between various parts of JFC. Java 2D, Drag and Drop, and Accessibility APIs are part of AWT and JFC, but they are not part of Swing. This is because these components use some native code, whereas Swing does not.

Swing is built around a new component called the `JComponent`, which extends from the AWT's `Container` class.

Swing – Introduction

The Swing Hierarchy

Figure E-2 illustrates the Swing component hierarchy.

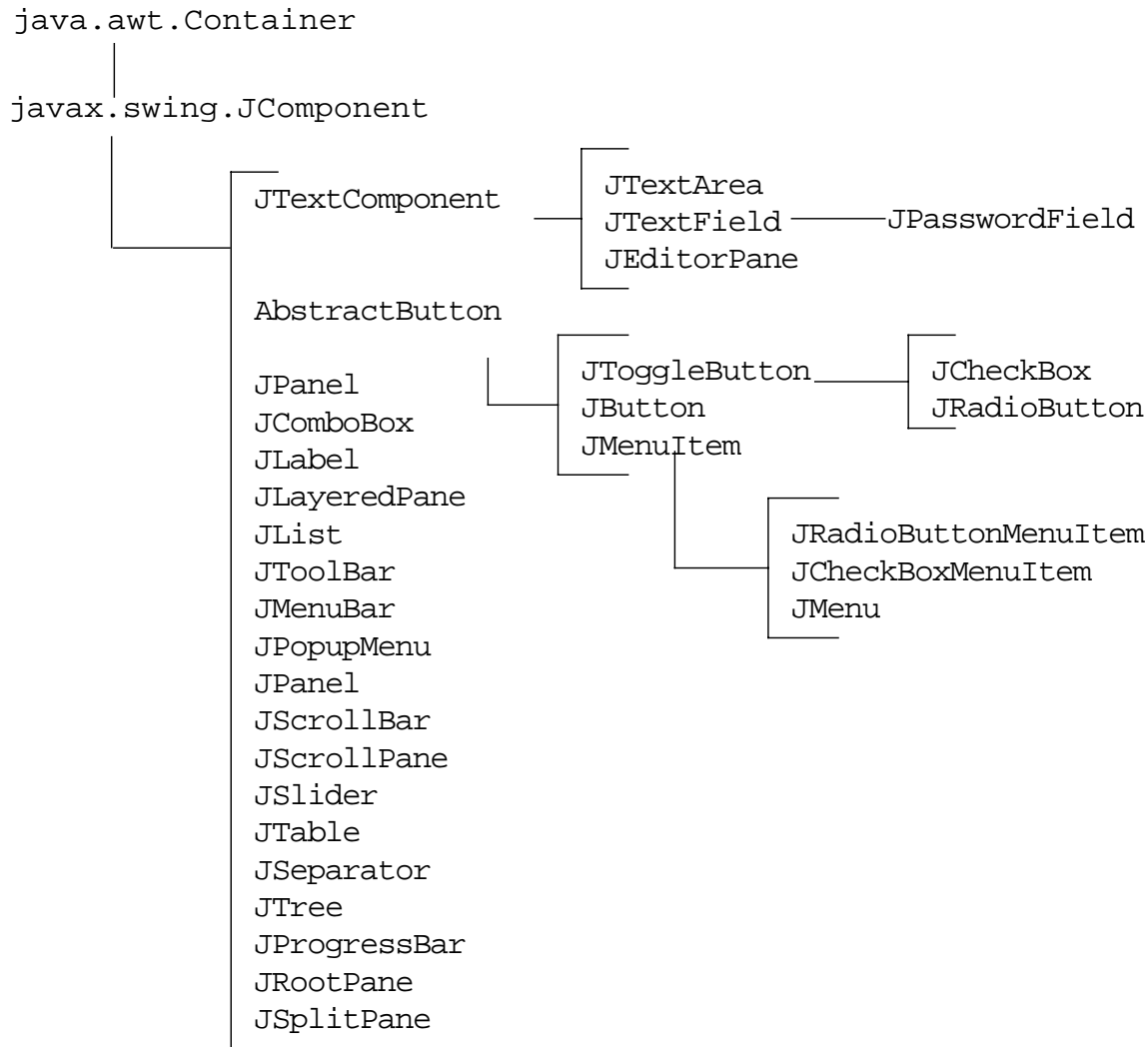


Figure E-2 Swing Component Hierarchy

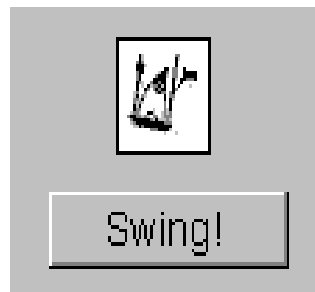
Swing GUIs use two kinds of classes: GUI classes and non-GUI support classes. The GUI classes are visual and descendents of `JComponent`, and are called “J” classes. The non-GUI classes provide services and perform vital functions for GUI classes; however, they do not produce any visual output.

Note – Swing’s event handling classes are examples of non-GUI classes.

Swing – Introduction

Swing Components

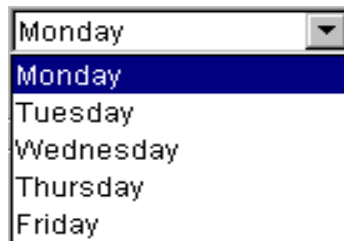
The Swing components primarily provide components for text handling, buttons, labels, lists, panes, combo boxes, scroll bars, scroll panes, menus, tables, and trees. Some of the components appear as follows:



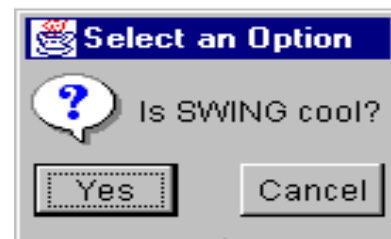
JApplet



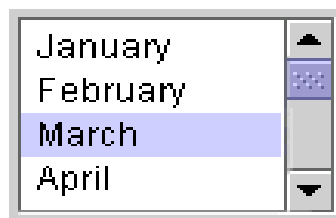
JButton, JToggleButton



JComboBox



JOptionPane



JList



JLabel

Figure E-3 Swing Components

Swing – Introduction

Swing Components (Continued)



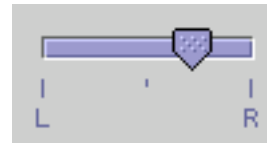
JScrollPane

First Na...	Last Name
Mark	Andrews
Tom	Ball
Alan	Chung
Jeff	Dinkins

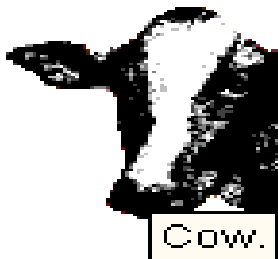
JTable



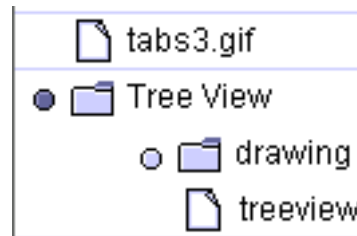
JScrollBar



JSlider



JTooltip



JTree

Figure E-4 More Swing Components

Basic Swing Application

The output of the `HelloSwing` application displays the window as shown in Figure E-5.

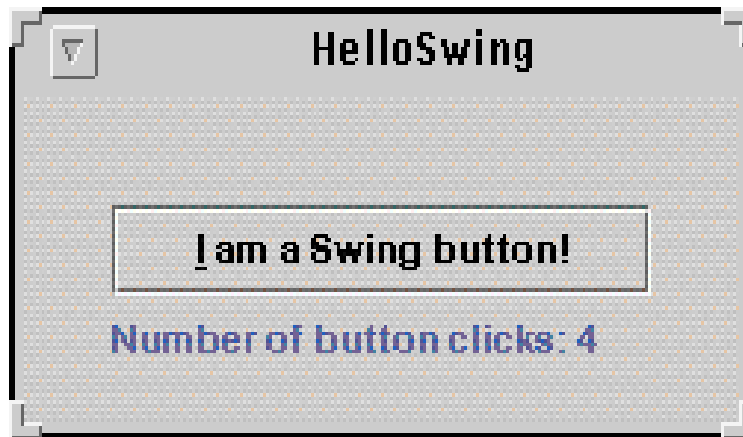


Figure E-5 HelloSwing Application

Each time the user clicks on a button, the label is updated.

Basic Swing Application

HelloSwing

The following is an example of the HelloSwing application.

```
1 import java.awt.*;
2 import java.awt.event.*;
3 import javax.swing.*;
4 import javax.accessibility.*;
5
6 public class HelloSwing implements ActionListener {
7     private JFrame jFrame;
8     private JLabel jLabel;
9     private JPanel jPanel;
10    private JButton jButton;
11    private AccessibleContext accContext;
12
13    private String labelPrefix = "Number of button clicks: ";
14    private int numClicks = 0;
15
16    public void go() {
17        jFrame = new JFrame("HelloSwing");
18        jLabel = new JLabel(labelPrefix + "0");
19
20        jButton = new JButton("I am a Swing button!");
21
22        // Create a shortcut: make ALT-I be equivalent
23        // to pressing mouse over button.
24        jButton.setMnemonic('i');
25
26        jButton.addActionListener(this);
27
28        // Add support for accessibility.
29        accContext = jButton.getAccessibleContext();
30        accContext.setAccessibleDescription(
31            "Pressing this button increments " +
32            "the number of button clicks");
33
34        // Set up pane.
35        // Give it a border around the edges.
36        jPanel = new JPanel();
37        jPanel.setBorder(
38            BorderFactory.createEmptyBorder(30,30,10,30));
```

Basic Swing Application

HelloSwing (*Continued*)

```
39
40 // Arrange for compts to be in a single column.
41 jPanel.setLayout(new GridLayout(0, 1));
42
43 // Put compts in pane, not in JFrame directly.
44 jPanel.add(jButton);
45 jPanel.add(jLabel);
46 JFrame.setContentPane(jPanel);
47
48 // Set up a WindowListener inner class to handle
49 // window's quit button.
50 WindowListener wl = new WindowAdapter() {
51     public void windowClosing(WindowEvent e) {
52         System.exit(0);
53     }
54 };
55
56 JFrame.addWindowListener(wl);
57
58 JFrame.pack();
59 JFrame.setVisible(true);
60 }
61
62 // Button handling.
63 public void actionPerformed(ActionEvent e) {
64     numClicks++;
65     jLabel.setText(labelPrefix + numClicks);
66 }
67
68 public static void main(String[] args) {
69     HelloSwing helloSwing = new HelloSwing();
70     helloSwing.go();
71 }
72 }
73
```

Basic Swing Application

Importing Swing Packages

The line `import javax.swing.*` imports the entire Swing package, which includes the standard Swing components and functionality.

Choosing the Look and Feel

Lines 20–26 of `HelloSwing` formats the application's look and feel. The method `getLookAndFeel()` returns the Windows look and feel on a Microsoft Windows environment. On machines running the Solaris Operating Environment, it returns the Motif look and feel. These lines are not necessary for this example because they are the default value.

Basic Swing Application

Setting up Windows

Swing programs implement their primary windows with `JFrame` objects. The `JFrame` class is a subclass of AWT's `Frame` class. It also adds some features found only in Swing. The following is the `HelloSwing` code that deals with its `JFrame`:

```
public HelloSwing() {
    JFrame jFrame;
    JPanel jPanel;
    .....
    jFrame = new JFrame("HelloSwing");
    jPanel = new JPanel();
    .....
    jFrame.setContentPane(jPanel);
}
```

The code is similar to the code for using a `Frame`. The only difference is that you cannot add components to a `JFrame` directly. Instead, you either add components to the `JFrame`'s content pane, or you provide a new content pane.

A *content pane* is a `Container` that contains all of the frame's visible components except for the menu bar (if there is one). To get a `JFrame`'s content pane, use the `getContentPane()` method. To set its content pane (as shown in the preceding example), use the `setContentPane()` method.

- ✓ **`JFrame` is an extended version of `java.awt.Frame` that adds support for interposing input and painting behavior in front of the frame's children (see `glassPane`), support for special children that are managed by a `LayeredPane` (see `rootPane`), and for Swing `MenuBar`s.**

Basic Swing Application

Setting up Swing Components

The `HelloSwing` program explicitly instantiates four Swing components: a `JFrame`, `JButton`, `JLabel`, and `JPanel`. `HelloSwing` uses the code in lines 33–45 to initialize the `JButton`.

- ✓ **Emphasize the fact that keyboard navigation is a feature of JFC and that it was not present earlier. Also mention that Swing adds automatic window-close handling to `JFrame` so that you do not always have to implement a window listener.**

Line 31 creates the button. Line 35 sets the ALT-I key combination as a shortcut used to simulate a button click. Line 37 registers an event handler for the click. Lines 40–43 describe a button, so that assistive technologies can provide information on the button's functionality.

Lines 47–57 initialize the `JPanel`. They create the `JPanel` object, give it a border, and set its layout manager to one that puts the panel's contents in a single column. Finally, a button and a label are added to the `Panel`. The `Panel` in `HelloSwing` uses an invisible border to put extra padding around it.

Basic Swing Application

Supporting Assistive Technologies

The only code in `HelloSwing.java` that exists solely to support the assistive technologies is the following:

```
accContext = jButton.getAccessibleContext();
accContext.setAccessibleDescription(
    "Pressing this button increments " +
    "the number of button clicks.");
```

Assistive technologies can also use the following set of information:

```
jButton = new JButton("I'm a Swing button!");
jLabel = new JLabel(labelPrefix + "0");
jLabel.setText(labelPrefix + numClicks);
```

Accessibility support is built into `JFrame`, `JButton`, `JLabel`, and all other Swing components. Assistive technologies can get the text of or the text associated with a specific part of a component.

✓ **Emphasize the advantages of using assistive technologies.**

Building a Swing GUI

The Swing package defines two types of components:

- Top-level containers (JFrame, JApplet, JWindow, and JDialog)
- Lightweight components (such as JButton, JPanel, and JMenu)

The top-level containers provide the framework in which the lightweight components exist. Specifically, a top-level Swing container provides an area in which lightweight Swing components can draw themselves. Top-level containers are Swing subclasses of their heavyweight AWT component counterpart. These Swing containers rely on the native code of their AWT superclass to properly interface with the hardware.

Every Swing component should have a top-level Swing container above it in its containment hierarchy. For example, you should implement every applet containing Swing components as a subclass of `JApplet` (which is itself a subclass of `java.applet.Applet`). Similarly, you should implement every main window that contains Swing components with a `JFrame`. Typically, if you are using Swing components, you use only Swing components and Swing containers.

You can add swing components to a content pane that is associated with a top-level container, but you cannot add swing components to the top-level container directly.

Building a Swing GUI

Figure E-6 illustrates the GUI *containment* hierarchy for a typical Swing program that implements a window containing two buttons: a text field and a list.

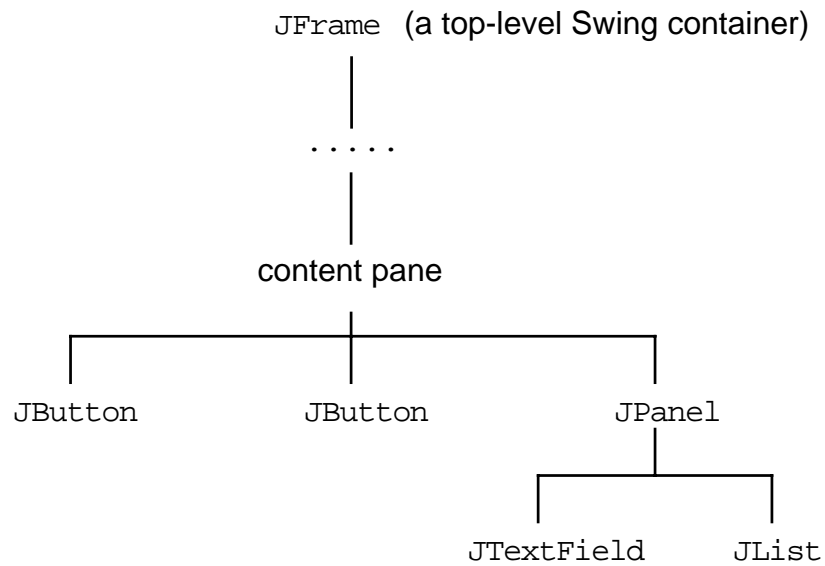


Figure E-6 GUI Containment Hierarchy

Building a Swing GUI

Figure E-7 illustrates another *containment* hierarchy figure for the same GUI, except that the GUI is in an applet running in a browser:

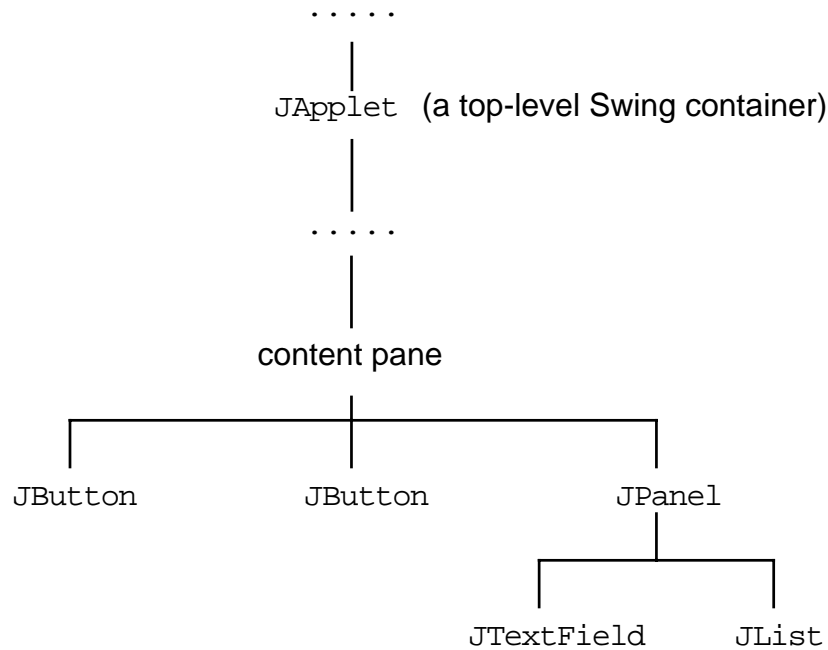


Figure E-7 Containment Hierarchy of an Applet

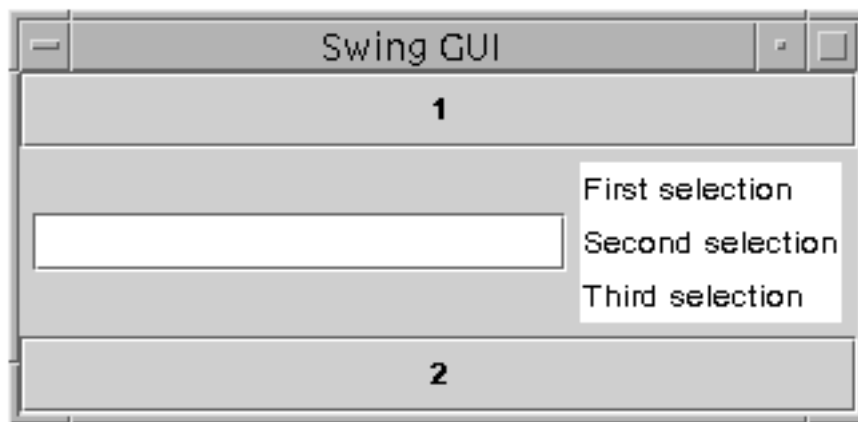
Building a Swing GUI

The following is the code that constructs the GUI hierarchies shown in the preceding figures.

```
1 import java.awt.*;
2 import javax.swing.*;
3
4 public class SwingGUI {
5     private JFrame topLevel;
6     private JPanel jPanel;
7     private JTextField jTextField;
8     private JList jList;
9
10    private JButton b1;
11    private JButton b2;
12    private Container contentPane;
13
14    private Object listData[] = {
15        new String("First selection"),
16        new String("Second selection"),
17        new String("Third selection")
18    };
19
20    public void go() {
21        topLevel = new JFrame("Swing GUI");
22
23        // Set up the JPanel, which contains the text field
24        // and list.
25        jPanel = new JPanel();
26        jTextField = new JTextField(20);
27        jList = new JList(listData);
28
29        contentPane = topLevel.getContentPane();
30        contentPane.setLayout(new BorderLayout());
31
32        b1 = new JButton("1");
33        b2 = new JButton("2");
34        contentPane.add(b1, BorderLayout.NORTH);
35        contentPane.add(b2, BorderLayout.SOUTH);
36
37        jPanel.setLayout(new FlowLayout());
38        jPanel.add(jTextField);
39        jPanel.add(jList);
40        contentPane.add(jPanel, BorderLayout.CENTER);
```

Building a Swing GUI

```
41
42     topLevel.pack();
43     topLevel.setVisible(true);
44 }
45
46 public static void main (String args[]) {
47     SwingGUI swingGUI = new SwingGUI();
48     swingGUI.go();
49 }
50 }
```



- ✓ ***In general, you should avoid using heavyweight components in Swing GUI's (except for the top-level Swing container that hosts the GUI). The most noticeable problem with mixing heavyweight and lightweight components is that when they overlap within a container, the heavyweight component is always drawn on top of (that is, in front of) the lightweight component.***

The JComponent Class

All Swing components are implemented as subclasses of the `JComponent` class, which inherits from the `Container` class. Swing components inherit the following functionality from `JComponent`:

- Borders

Using the `setBorder()` method, you can specify the border that a component displays around its edges. You can specify that a component have extra space around its edges using an `EmptyBorder` instance.

- Double buffering

Double buffering can improve the appearance of a frequently changing component. You do not have to write the double buffering code—Swing provides it for you. By default, Swing components are double buffered.

- Tool tips

By specifying a string with the `setToolTipText()` method, you can provide help to users of a component. When the cursor pauses over the component, the specified string is displayed in a small window that appears near the component.

- Keyboard navigation

Using the `registerKeyboardAction()` method, you can enable the user to use the keyboard, instead of the mouse, to maneuver around the GUI. The combination of character and modifier keys that the user must press to start an action is represented by a `KeyStroke` object.

- Application-wide pluggable look and feel

Each Java application runtime has a `UIManager` object that determines the look and feel of that runtime's Swing components. Subject to security restrictions, you can choose the look and feel used by all Swing components by invoking the `UIManager.setLookAndFeel()` method. Behind the scenes, each `JComponent` object has a corresponding `ComponentUI` object that performs all drawing, event handling, size determination, and so on for that `JComponent`.

Native Methods

You have seen many features and basic functions within the Java programming language. However, you might want to perform tasks with applications written in the Java programming language that you cannot accomplish with the Java programming language alone. In these instances, you can use native methods to link in C programs that handle your specific needs.

This added functionality comes at a price—your applications are no longer easily portable. Other machines sharing your architecture must have a local copy of your compiled C programs. Other machines with different architectures require porting your C programs to those architectures where they can be compiled by a native compiler.

This process can be difficult for complicated programs or impossible for programs that rely on features found in the underlying operating system. However, if you are dealing with a single architecture or you want to add a well-defined adaptable feature, native methods can be the best option to meet your needs.

Native HelloWorld

The first task is to call a native method from a Java program. To do this, create a native method for the HelloWorld program.

- ✓ HelloWorld **requires no arguments and supplies no return values. How to handle arguments is discussed later.**

The following is an overview of the four basic steps required to integrate native methods into your Java programs:

1. Define a Java class with the appropriate native method declarations.
2. Create a header file for use with your C modules. Use the `javah` utility to do this.
3. Write the C modules containing the native methods.
4. Compile the C code into a dynamic loadable library.

- ✓ **JNI, unlike NMI of the past, does not require a stub file to interface to native code.**

Defining Native Methods

Like other methods, you must declare all native methods you plan to use, and they must be defined within a class.

Define your native HelloWorld method as follows:

```
public native void nativeHelloWorld();
```

There are two changes from the other `public void` methods you have written:

- The key word `native` is used as a method modifier
- The body of the method (the actual implementation) is not defined here; it is replaced with a semicolon (`;`)

A Native HelloWorld

Defining Native Methods (Continued)

You must place the native method declaration inside a class definition. The class containing the native method also contains a `static` code block that loads the dynamic library with the implementation of your method. The following is an example of a class definition for the simple `nativeHelloWorld()` method:

```
1    class NativeHello {
2        public native void nativeHelloWorld();
3        static {
4            System.loadLibrary("hello1");
5        }
6    }
```

The Java runtime environment executes the defined `static` code block when the class is loaded. In the example above, the `hello1` library is loaded when the class `NativeHello` is loaded.

Calling Native Methods

Once you have wrapped your native methods into a class, you can create objects of that class to access the native methods, just as you would with regular class methods. The following is a program that creates a new `NativeHello` object and calls your `nativeHelloWorld` method.

```
1    class UseNative {
2        public static void main (String args[])
3        {
4            NativeHello nh = new NativeHello();
5            nh.nativeHelloWorld();
6        }
7    }
```

Use `javac` to compile the `.java` files. The `.class` files are used when creating header files.

A Native HelloWorld

The javah Utility

You can create a C header file with the `jvah` utility, based on the `NativeHello.class` file. You invoke `jvah` as follows:

```
jvah -jni NativeHello
```

The generated file, `NativeHello.h`, provides you with the information you need to write the C program. Here is the file as generated by `jvah` for this example:

```
1  /* DO NOT EDIT THIS FILE - it is machine generated */
2  #include <jni.h>
3  /* Header for class NativeHello */
4
5  #ifndef _Included_NativeHello
6  #define _Included_NativeHello
7  #ifdef __cplusplus
8  extern "C" {
9  #endif
10 /*
11  * Class:      NativeHello
12  * Method:     nativeHelloWorld
13  * Signature:  ()V
14  */
15 JNIEXPORT void JNICALL Java_NativeHello_nativeHelloWorld
16 (JNIEnv *, jobject);
17
18 #ifdef __cplusplus
19 }
20 #endif
21 #endif
22
```

The portion in bold characters gives the signature of the native method yet to be implemented.

A Native HelloWorld

Coding C Functions for Native Methods

At this point, the C program is the only piece of code missing. The C code you write must include the header file above, plus `jni.h`, supplied with the Java 2 SDK in the `$JAVA_HOME/include` directory. (`$JAVA_HOME` refers to the "root" directory of the Java 2 SDK.) Of course, include any other header files necessary for your functions as well.

For each function declared in the header file, you provide the body. For this example, the C file, called `MyNativeHello.c`, looks like the following:

```
1  #include <jni.h>
2  #include "NativeHello.h"
3  #include <stdio.h>
4
5  void Java_NativeHello_nativeHelloWorld
6      (JNIEnv *env, jobject obj) {
7      printf ("Hello from C");
8  }
9
```

Putting It Together

Now that you have all of the pieces, you must tell the system how to assemble them. First, compile your C program. You might have to specify the location of the `include` files.

The following example uses the C compiler in the Solaris Operating Environment (`$JAVA_HOME` represents the directory where the Java 2 SDK is installed):

```
cc -I$JAVA_HOME/include -I$JAVA_HOME/include/solaris -  
G MyNativeHello.c -o libhello1.so
```

The two include directories you must access to compile this code are `$JAVA_HOME/include` and `$JAVA_HOME/include/solaris`. You can specify them on the command line or you can modify your `INCLUDE` environment variable.

This example uses the Microsoft C compiler for Microsoft Windows:

```
C:\> cl MyNativeHello.c -Fehello1.dll -MD -LD javai.lib
```

Once you have created the library file, you can run your native method test program:

```
java UseNative  
Hello Native World!
```

If you get a `java.lang.UnsatisfiedLinkError`, you might need to update the system `LD_LIBRARY_PATH` variable to include the current directory (so JVM can find `libhello1.so`).

Passing Information to a Native Method

The previous sample native method does not handle information accessed from the defining class, nor does it accept any arguments. Both of the following tasks occur regularly in programming.

Passing a Java Primitive as an Argument

Arguments can be supplied to native methods in a Java program, just as they are supplied to other methods. Suppose the following code declaration exists in a file called `NativeHello2.java` for a native method that prints `count` times:

```
1 public native void nativeHelloWorld2(int count);
```

This declaration produces the following entry in the header file `NativeHello2.h`:

```
1 JNIEXPORT void JNICALL Java_NativeHello2_nativeHelloWorld2
2     (JNIEnv *, jobject, jint);
```

Now rewrite your C method to loop for the supplied number of times (which comes in as the method's third argument).

```
1 #include <jni.h>
2 #include "NativeHello2.h"
3 #include <stdio.h>
4
5 JNIEXPORT void JNICALL Java_NativeHello2_nativeHelloWorld2
6     (JNIEnv *env, jobject obj, jint countMax) {
7     int count;
8     for (count = 0; count < countMax; count++) {
9         printf ("Hello from C, count = %d\n",count);
10    }
11 }
```

Passing Information to a Native Method

Accessing a Java Primitive as an Object Data Member

The most common requirement in a native method is access to the class data members. The `jni.h` file (in `$JAVA_HOME/include`) contains several interface functions for use with objects inside a native code module.

For example, consider writing a class that has two `int` variables, one of which is `static`, which are accessed and modified by a native method:

```

1  class NativeHello4 {
2      static int statInt = 2;
3      int instInt = 4;
4      public native int nativeHelloWorld4();
5      static {
6          System.loadLibrary("hello4");
7      }
8  }
```

Inside your C program, you can access these variables using functions in `<jni.h>`.

```

1  #include <jni.h>
2  #include "NativeHello4.h"
3  #include <stdio.h>
4
5  /* The names of the Java object fields to be accessed. */
6  #define STAT_FIELD_NAME "statInt"
7  #define INST_FIELD_NAME "instInt"
8
9  /* This method displays the statInt and instInt fields and
10 returns the product of the two. */
11 jint Java_NativeHello4_nativeHelloWorld4
12     (JNIEnv *env, jobject obj) {
13
14     /* Class object. Used to find all fields and access
15     static ones.
16     jclass class = (*env)->GetObjectClass(env,obj);
17
18     jfieldID fid;    /* A field reference. */
19     jint staticInt; /* A C copy of the static int. */
20     jint instanceInt; /* A C copy of the int. */
21
```

Passing Information to a Native Method

Accessing a Java Primitive as an Object Data Member (Continued)

```
22
23     /* Get reference to the static field. The "class"
24     argument connects the field to a class. The third
25     argument is the field's name, and the last argument is
26     the field's type. See the union jvalue entry in jni.h,
27     then capitalize for the proper primitive value. */
28     fid = (*env)->GetStaticFieldID(env, class,
29         STAT_FIELD_NAME, "I");
30     if (fid == 0)
31         return;
32
33     /* Get that field's data. */
34     staticInt = (*env)->GetStaticIntField(env, class, fid);
35
36     /* Process it, change it... */
37     printf
38         ("In C, doubling original %s value of %d to %d\n",
39         STAT_FIELD_NAME, staticInt, staticInt*2);
40     staticInt *= 2;
41
42     /* ... and store it back into the class object. */
43     (*env)->SetStaticIntField(env, class, fid, staticInt);
44
45
46     /* Now for the nonstatic int, part of the current
47     object. Get the field reference as before... */
48     fid = (*env)->GetFieldID
49         (env, class, INST_FIELD_NAME, "I");
50     if (fid == 0)
51         return;
52
53     /* Get the field. Refer to the object, not the class. */
54     instanceInt = (*env)->GetIntField(env, obj, fid);
55
56     /* Process it, change it... */
57     printf
58         ("In C, tripling original %s value of %d to %d\n",
59         INST_FIELD_NAME, instanceInt, instanceInt*3);
60     instanceInt *= 3;
61
```

Passing Information to a Native Method

Accessing a Java Primitive as an Object Data Member (Continued)

```
62     /* ... and store it back. */
63     (*env)->SetIntField(env, obj, fid, instanceInt);
64
65     /* Return the product of the two. */
66     return (staticInt * instanceInt);
67 }
68
```


Passing Information to a Native Method

Accessing Strings

As you might recall, strings in the Java programming language consist of 16-bit Unicode characters. However, C strings consist of 8-bit American Standard Code for Information Interchange (ASCII) characters. Because strings are common objects which are passed between Java code and native code, several functions have been defined in `jni.h` to help make string manipulation easier. The C datatype for strings in the Java programming language is `jstring`.

✓ **The type `unicode` is a `simple` typedef for an unsigned short.**

Suppose a native method that prints out a string is declared in a Java program as follows:

```
public void nativeHelloWorld3(String printMe);
```

The following C function implements the native code for it. You must call `ReleaseStringUTFChars` to free the memory allocated for the C string when done.

```
1  #include <jni.h>
2  #include "NativeHello3.h"
3  #include <stdio.h>
4
5  void Java_NativeHello3_nativeHelloWorld3 (
6      JNIEnv *env, jobject obj, jstring javaString) {
7
8      const char * CString;
9
10     /* Convert java string to C string. */
11     CString = (*env)->GetStringUTFChars(env, javaString, 0);
12
13     printf ("In C, string is %s\n", CString);
14
15     /* Tell VM to release mem for CString as done w/ it. */
16     (*env)->ReleaseStringUTFChars(env, javaString, CString);
17 }
```

Passing Information to a Native Method

Accessing Strings (Continued)

As a final example, consider accessing strings as object data members. Suppose a class is defined with a `String` field and native method as follows:

```

1  class NativeHello5 {
2      String stringField = "original";
3      public native String nativeHelloWorld5();
4      static {
5          System.loadLibrary("hello5");
6      }
7  }
```

This class is instantiated and the native method called from the following program:

```

1  class UseNative5 {
2      public static void main (String args[]) {
3          String changedString;
4          NativeHello5 nh = new NativeHello5();
5
6          System.out.println ("In Java, nh's string says '"
7              + nh.stringField + "'");
8
9          /* Call native method to print and change string in
10             the current object, and return a third string. */
11             changedString = nh.nativeHelloWorld5();
12
13             System.out.println ("In Java, nh's string says '"
14                 + nh.stringField + "'");
15
16             System.out.println ("Native method returned '" +
17                 changedString + "'");
18
19         }
20     }
```

The following native code extracts the string from the object, prints it out, changes and stores the new version, and then returns another new string in the function's return:

```

1  #include <jni.h>
2  #include "NativeHello5.h"
3  #include <stdio.h>
```

Passing Information to a Native Method

Accessing Strings (Continued)

```
4
5  #define NEW_STRING1 "Revised"
6  #define NEW_STRING2 "Revised again"
7
8  jstring Java_NativeHello5_nativeHelloWorld5
9      (JNIEnv *env, jobject obj) {
10
11     jclass class = (*env)->GetObjectClass(env,obj);
12     jfieldID fid;
13     jstring javaString;
14     const char *CString;
15
16     fid = (*env)->GetFieldID
17         (env, class, "stringField", "Ljava/lang/String;");
18     if (fid == 0)
19         return;
20
21     /* Get the field reference for the java string. */
22     javaString = (*env)->GetObjectField(env, obj, fid);
23
24     /* Retrieve the C string from the object. */
25     CString = (*env)->GetStringUTFChars(env, javaString, 0);
26
27     printf ("In C, changing string from %s to %s\n",
28            CString, NEW_STRING1);
29
30     /* Tell VM to release memory for CString as done w/it. */
31     (*env)->ReleaseStringUTFChars(env, javaString, CString);
32
33     /* Alloc a new Java string to store back in the obj. */
34     javaString = (*env)->NewStringUTF(env, NEW_STRING1);
35
36     /* Store it back. */
37     (*env)->SetObjectField(env, obj, fid, javaString);
38
39     /* Allocate one more new Java string to return. */
40     javaString = (*env)->NewStringUTF(env, NEW_STRING2);
41
42     return (javaString);
43 }
44
```

Summary

The exact process for integrating native methods is:

1. Create a program containing the native method declarations and the static code to load the dynamic library:

```
vi NativeHello.java
```

2. Create a program containing calls to the native methods:

```
vi UseNative.java
```

3. Compile the .java files:

```
javac NativeHello.java UseNative.java
```

4. Create the C header file:

```
javah -jni NativeHello
```

5. Create the C program implementing your native methods:

```
vi MyNativeHello.c
```

6. Compile the dynamic library:

```
cc -I$JAVA_HOME/include -  
I$JAVA_HOME/include/solaris -G MyNativeHello.c -o  
libhello.so
```

7. Set the LD_LIBRARY_PATH variable:

```
setenv LD_LIBRARY_PATH .:$LD_LIBRARY_PATH
```

8. Run the program:

```
java UseNative
```

See the JNI tutorial on the JavaSoft Web site for information about other JNI functionality. You can access the tutorial using a hyperlink from your locally loaded master Java API index.

What Is UML?

The Unified Modeling Language (UML) is a graphical language for modeling software systems. It was created in the early 1990's by three leaders in the object modeling world: Grady Booch, James Rumbaugh, and Ivars Jacobson. Their goal was to unify the three major modeling languages at the time: the Booch method, Object Modeling Technique (OMT), and Object-Oriented Software Engineering (OOSE, best known for the introduction of use-case analysis). UML is now a standard of the Object Management Group (OMG); version 1.1 was adopted by the OMG on 14 November 1997.

UML is a big language. In this course, we will use a small subset of the language that includes: package diagrams, class diagrams, object diagrams, and state diagrams.

Package Diagrams

In UML, packages allow you to arrange your modeling elements into groups. There is not a one-to-one mapping between UML packages and Java packages. However, it can be used to model Java packages. Figure G-1 demonstrates a single package that contains a group of classes in a class diagram.

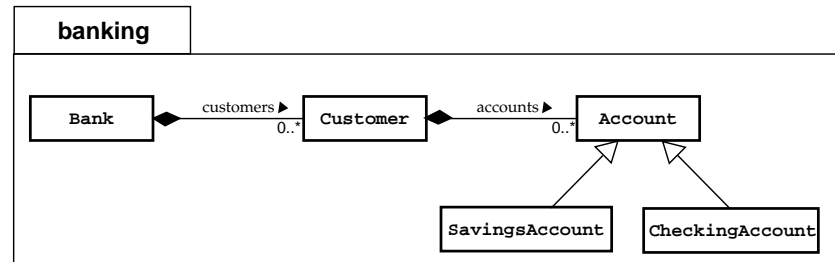


Figure G-1 A Package Containing a Class Diagram

The mapping to Java packages implies that the classes would contain the package declaration of package `banking`. For example, in the file `Customer.java`:

```

package banking;
// import statements
public class Customer {
    // declarations
}
  
```

Package Diagrams

Figure G-2 demonstrates a simple hierarchy of packages. The shipping package contains three sub-packages: GUI, reports, and domain. 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, GUI uses elements in the domain package.

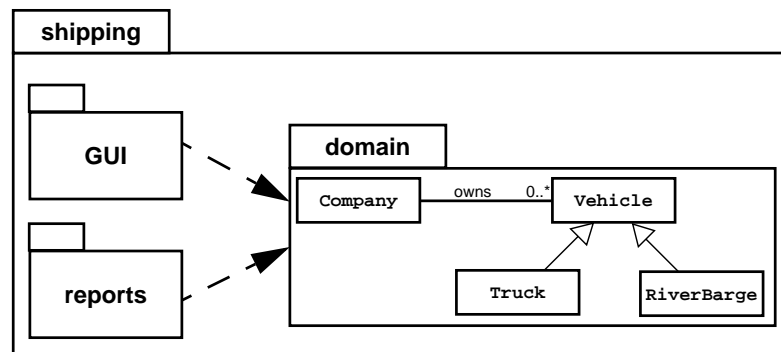


Figure G-2 A Package Containing Sub-Packages

The mapping to Java packages implies that the classes would contain the package declaration of package `shipping.domain`. For example, in the file `Company.java`:

```
package shipping.domain;
// import statements
public class Company {
    // declarations
}
```

Notice that in Figure G-2, 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.

Class Diagrams

A class diagram gives a visual representation of the members of a class and the static relationships between classes.

Class Nodes

Figure G-3 shows several *class nodes*. You do not have to model every aspect of an entity every time that entity is used. A class node can just be the name of the class, as in examples a, b, and c. Example (a) is a concrete class, where no members are modeled. Example (b) is an abstract class (name is in italics). Example (c) is an interface. Example (d) is a concrete class, where members are modeled.

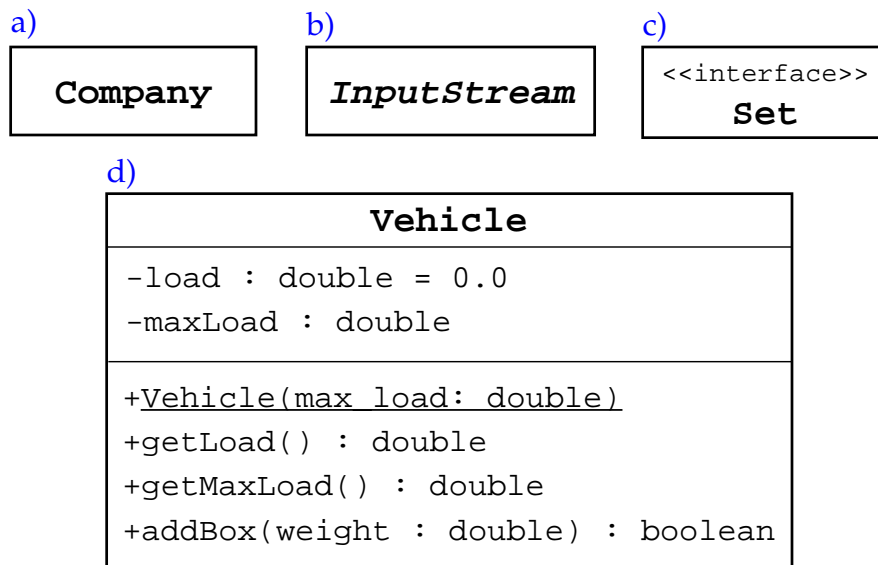


Figure G-3 Several "Class" Declarations

Class Diagrams

Class Nodes (Continued)

A full specified class node has three basic elements: the name of the class in the top, the set of attributes under the first bar, and the set of methods under the second bar.

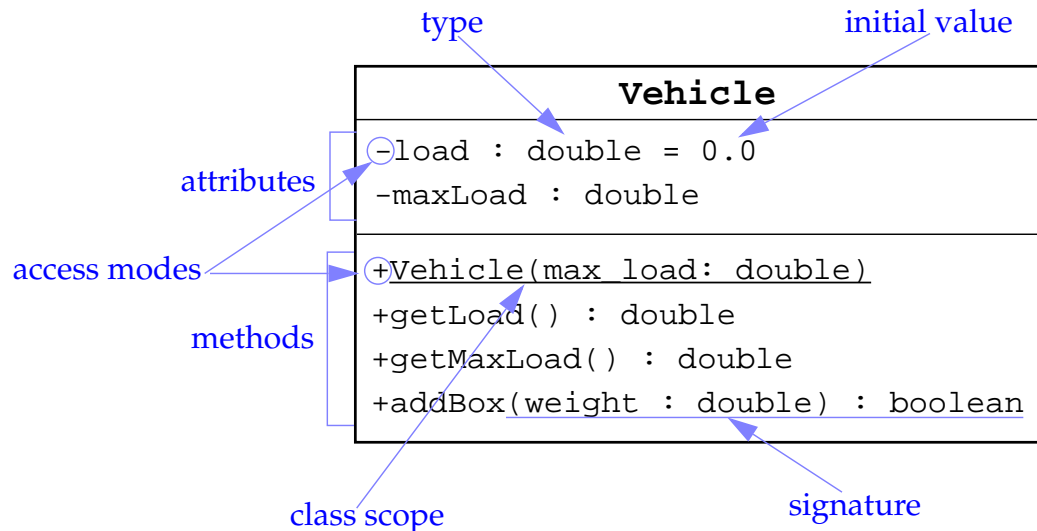


Figure G-4 Elements Of a Class Node

An attribute is specified by four elements: access mode, name, data type, and initial value. A method is specified by three elements: access mode, name, and signature. The signature is specified by the parameter list and the return value; if the return value is not specified, then no value is returned (`void`). An underline beneath either an attribute or method specifies that that member has a class scope (`static`). Constructors are modeled as a method using the name of the class for the method name, with class scope and no return value.

Table 7-1 UML Defined Access Modes and Their Symbol

access mode	symbol
private	-
protected	#
public	+

Class Diagrams

Inheritance and Interface Implementation

Figure G-5 shows class inheritance through the "is-a relationship arrow."

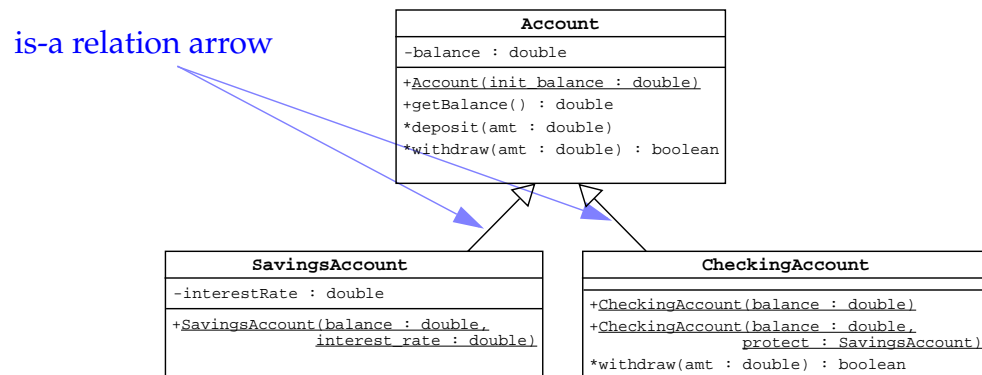


Figure G-5 Class Inheritance Relationship

This is implemented in Java with the extends keyword. For example:

```
public class SavingsAccount extends Account {
    // declarations here
}
```

Figure G-6 shows how to model a class implementing an interface, using the "realization arrow."

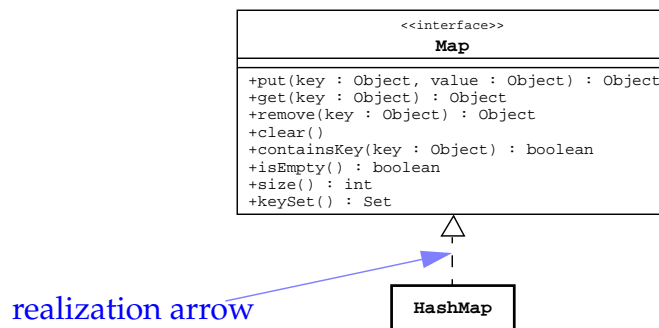


Figure G-6 An Example of a Class Implementing an Interface

```
public class HashSet implements Set {
    // declarations here
}
```

Class Diagrams

Association and Aggregation

Figure G-7 shows several class associations. An *association* is a link between two classes that can be "navigated" from one object to another. For example, a customer has zero or more bank accounts. An *aggregation* (also called a *composite*) is an association in which one object contains a group of parts (the other objects) where the parts cannot exist independent of the "whole" object. For example, the association between Customer and Account is an aggregation because an account object must exist within the context of a customer and accounts are not shared between customers.

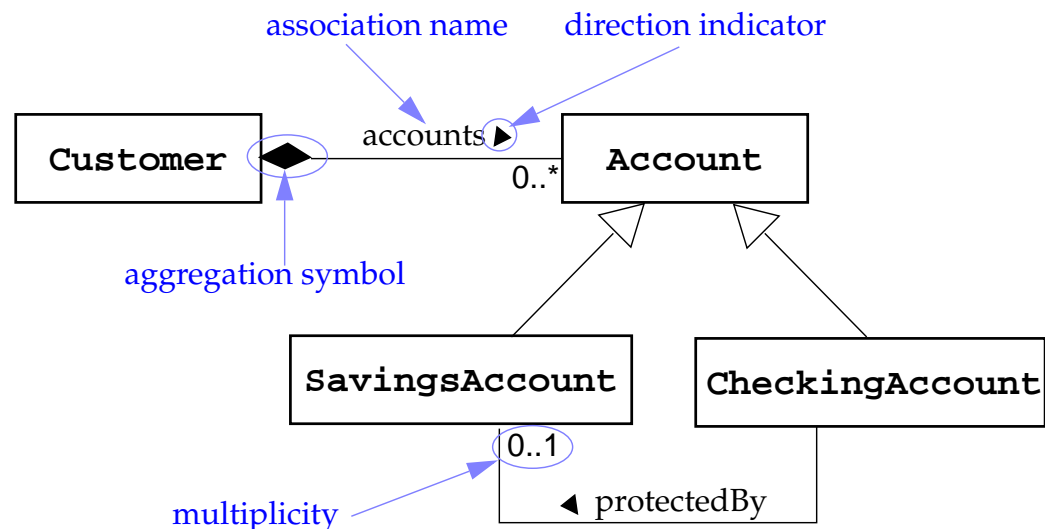


Figure G-7 Class Associations and Aggregations

An association has a direction indicating the path of navigation. An association has *multiplicity* which indicates how many objects on one side of the relation can be associated with objects on the other side of the relation. Multiplicity can be any set of non-negative numbers; including the * symbol, which means any number of elements. The default multiplicity is 1. A range is specified by $n..m$, meaning at least n elements but no more than m elements; therefore, $1..*$ means any number of elements, but at least one.

Class Diagrams

Association and Aggregation (Continued)

Associations are typically represented in Java as an attribute in the class at the tail of the relation (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.

For example, in Figure G-7 the `protectedBy` association might be represented in the `CheckingAccount` class as:

```
public class CheckingAccount {
    private SavingsAccount  protectedBy;
}
```

Also, in Figure G-7 the `accounts` aggregation might be represented in the `Customer` class as:

```
public class Customer {
    private Account[]  accounts = new Account[MAX_ACCOUNT];
}
```

or as:

```
public class Customer {
    private List  accounts = new ArrayList();
}
```

The latter representation is preferable if you do not know the maximum number of accounts ahead of time.

Object Diagrams

An object diagram is used to model a particular example of a set of objects, their relationships, and their relationships to classes.

In Figure G-8, two objects are shown, `c1` and `c2`, with their instance data. They refer to the class node for `Count` and the arrows indicate that the object is an instance of the class `Count`.

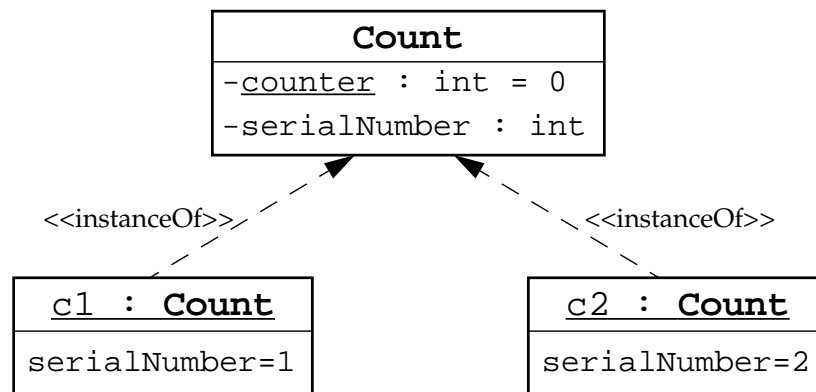


Figure G-8 An Example of an Object Diagram

State Diagrams

A state diagram is used to model the internal states of an object throughout its lifetime in response to events. The definition of an object state is dependent on the object and the level of depth you wish to model.

Figure G-9 shows an example state diagram. Every state 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.

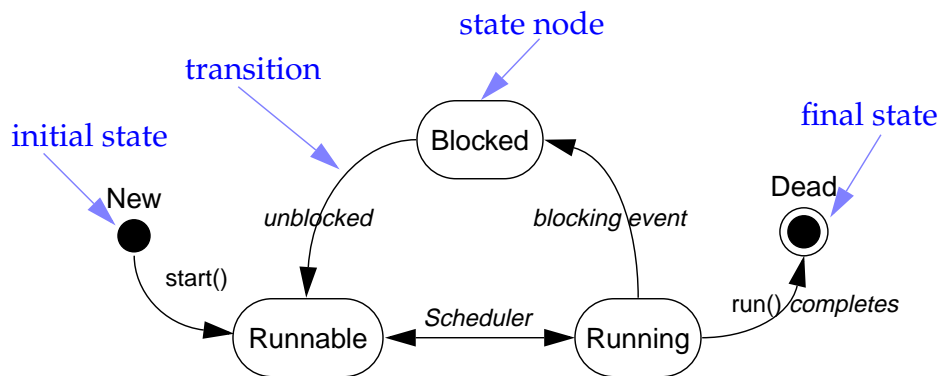


Figure G-9 An Example State Transition Diagram

There is no pre-defined way of implementing a state diagram. This course has used these diagrams only as a demonstration of the behavior of existing objects, such as for threads.

State Diagrams

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 making the state transition
- Target state – The state that is active after the completion of the transition

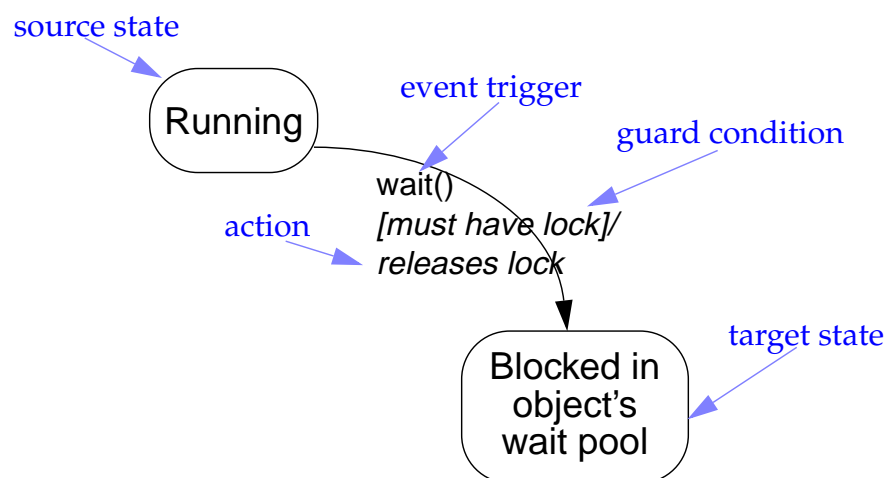


Figure G-10 An Example State Transition

Other UML 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 the UML to allow modelers to design their own semantics to modeling elements (nodes and relationships). Figure G-11 shows the use of a stereotype tag <<interface>> to declare that the class node Set is a Java interface declaration. Stereotype tags can adorn relationships as well as nodes.

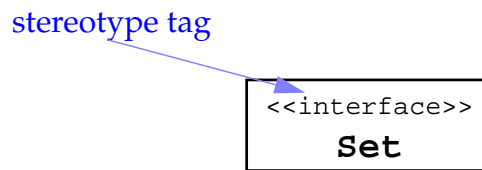


Figure G-11 An Example Stereotype Tag on a Class Node

Diagram Annotation

They also built into the language a method for annotating the diagrams. Figure G-12 shows a simple annotation.

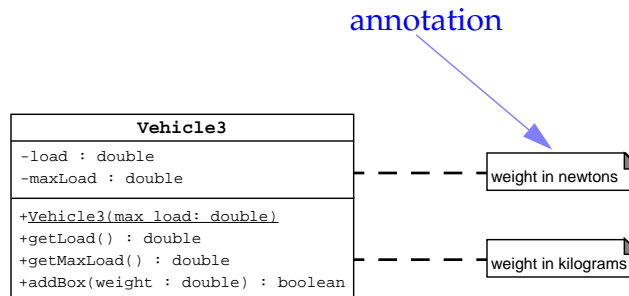


Figure G-12 An Example Annotation

Index

A

applet
 definition 12-3
 tag syntax 12-24
Applet Viewer
 definition 12-21
 synopsis 12-23
ArithmeticException 8-15
ArrayIndexOutOfBoundsException
 on 8-16
arrays 5-3

B

basic Java application Hello
 World 1-8
BorderLayout manager 10-15,
 10-24
break statement 4-31
bytecode verifier 1-28

C

CardLayout manager 10-15
comments in Java 3-3
compile-time errors 1-16
complete applet tag syntax 12-24
container layouts 10-14
containers and components 10-6
continue statement 4-31
converting 1.0 event handling to
 1.1 B-6

D

directory utilities 9-15
do loop 4-29

E

event conversion table B-7
event handling
 before JDK 1.1 B-3
 converting 1.0 to 1.1 B-6
 in JDK 1.1 B-3
 listeners B-10
exceptions 8-4
 example 8-5
 handling 8-7
 importance of 8-6
 throwing 8-23

F

file names 9-14
File object 9-14
file tests 9-15
finally statement 8-9
FlowLayout manager 10-15
for loop 4-25

G

getAudioClip() 12-31
GridBagLayout manager 10-15
GridLayout manager 10-15,
 10-29

I

identifiers 3-7, 3-8
if, else statements 4-20
importance of exceptions 8-6
init() 12-9

J

Java language
 comments 3-3
 compile-time errors 1-16
 compiling programs 1-14
 identifiers 3-7
 keywords 3-9
 layout managers 10-15
 operators 4-7
 running programs 1-15
 runtime errors 1-17
Java networking model 16-7
java.awt package overview 10-5

K

keywords 3-9
 public 1-10
 static 1-10
 void 1-10

L

label statement 4-31
layout managers
 BorderLayout 10-15
 CardLayout 10-15
 FlowLayout 10-15
 GridBagLayout 10-15
 GridLayout 10-15
loop() 12-32
loops
 do 4-29, 4-30
 for 4-25
 while 4-27, 4-28

N

native methods F-1 to F-6, G-1
 accessing objects F-8
 passing arguments F-7
 summary F-14
NegativeArraySizeException
 8-16
NullPointerException 8-15

O

operators 4-7

P

panels 10-12
 creating 10-37, 10-38
play() 12-29
playing audio clips 12-29
public modifier 1-10

R

random access files 15-26
RandomAccessFile class 15-26
runtime errors 1-17

S

sockets 16-3
start() 12-10
statements
 break 4-31
 continue 4-31
 else 4-20
 finally 8-9
 if 4-20
 label 4-31
 switch 4-22
 throw 8-23
static modifier 1-10
stop() 12-10, 12-32
switch statement 4-22
System.out.println() 1-13

T

TCP/IP client example 16-9
TCP/IP server server
 example 16-8
throw statement 8-23
throwing an exception 8-23

U

uniform resource locators
 (URLs) 15-24
URL input streams 15-24
URL object 15-24

V

void modifier 1-10

W

while loop 4-27
windows and frames 10-10

Copyright 2000 Sun Microsystems Inc., 901 San Antonio Road, Palo Alto, California 94303, Etats-Unis. Tous droits réservés.

Ce produit ou document est protégé par un copyright et distribué avec des licences qui en restreignent l'utilisation, la copie, la distribution, et la décompilation. Aucune partie de ce produit ou document ne peut être reproduite sous aucune forme, par quelque moyen que ce soit, sans l'autorisation préalable et écrite de Sun et de ses bailleurs de licence, s'il y en a.

Le logiciel détenu par des tiers, et qui comprend la technologie relative aux polices de caractères, est protégé par un copyright et licencié par des fournisseurs de Sun.

Des parties de ce produit pourront être dérivées du système Berkeley 4.3 BSD licenciés par l'Université de Californie. UNIX est une marque déposée aux Etats-Unis et dans d'autres pays et licenciée exclusivement par X/Open Company Ltd.

Sun, Sun Microsystems, the Sun logo, Solstice, Java, JavaBeans, JavaChip, Java HotSpot, JavaOS, JavaSoft, JDBC, JDK, JVM, OpenWindows, Write Once, Run Anywhere et Solaris sont des marques de fabrique ou des marques déposées de Sun Microsystems, Inc. aux Etats-Unis et dans d'autres pays.

Toutes les marques SPARC sont utilisées sous licence sont des marques de fabrique ou des marques déposées de SPARC International, Inc. aux Etats-Unis et dans d'autres pays.

Les produits portant les marques SPARC sont basés sur une architecture développée par Sun Microsystems, Inc.

Netscape Navigator is a trademark of Netscape Communications Corporation.

UNIX est une marque déposée aux Etats-Unis et dans d'autres pays et licenciée exclusivement par X/Open Company, Ltd.

L'interface d'utilisation graphique OPEN LOOK et Sun™ a été développée par Sun Microsystems, Inc. pour ses utilisateurs et licenciés. Sun reconnaît les efforts de pionniers de Xerox pour la recherche et le développement du concept des interfaces d'utilisation visuelle ou graphique pour l'industrie de l'informatique. Sun détient une licence non exclusive de Xerox sur l'interface d'utilisation graphique Xerox, cette licence couvrant également les licenciés de Sun qui mettent en place l'interface d'utilisation graphique OPEN LOOK et qui en outre se conforment aux licences écrites de Sun.

L'accord du gouvernement américain est requis avant l'exportation du produit.

Le système X Window est un produit de X Consortium, Inc.

LA DOCUMENTATION EST FOURNIE "EN L'ETAT" ET TOUTES AUTRES CONDITIONS, DECLARATIONS ET GARANTIES EXPRESSES OU TACITES SONT FORMELLEMENT EXCLUES, DANS LA MESURE AUTORISEE PAR LA LOI APPLICABLE, Y COMPRIS NOTAMMENT TOUTE GARANTIE IMPLICITE RELATIVE A LA QUALITE MARCHANDE, A L'APTITUDE A UNE UTILISATION PARTICULIERE OU A L'ABSENCE DE CONTREFAÇON.



Please
Recycle



Adobe PostScript

