Java Programming Language Workshop

SL-285

Student Guide

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

Revision B.2, February 2000

Contents

About This Course...xi
Course Overview xii
Course Map.. xiv
Module-by-Module Overview xv
Course Objectives... xvii
Skills Gained by Module... xviii
Guidelines for Module Pacing xix
Topics Not Covered.. xx
How Prepared Are You?.. xxi
Introductions xxiii
How to Use Course Materials xxiv
Course Icons and Typographical Conventions xxvi
Typographical Conventions xxvii

Java Application Design...1-1
Relevance.. 1-2
Software Development Cycle.. 1-3
Analysis 1-4
Design 1-4
Software Development Cycle.. 1-5
Implementation 1-5
Testing 1-5
Revisions 1-6
Developing Java Programs 1-7
Java Program Types...1-8
Class Definition 1-9
Java API Design..1-9
Complete API Declarations 1-10
Coding Conventions...1-11
Ensuring Project Success..1-14
The BrokerTool Program 1-15
Problem Definition...1-15
Legacy System 1-16
What You Need to Do 1-17

Specifications of the Legacy Server 1-18
Defining the BrokerTool Program.......................................1-20
Project Constraints..1-23
Notes 1-24
Exercise: The BrokerTool Program Initial Design..................... 1-25
Preparation..1-25
Tasks 1-25
Notes 1-26
Exercise Summary.. 1-27
Check Your Progress 1-28
Think Beyond 1-29

Managing Database Queries..2-1
Relevance.. 2-2
Relational Database Management Systems.............................. 2-3
Relational Models 2-5
Tables 2-5
Data Independence 2-6
Structured Query Language (SQL) 2-7
Introducing mSQL 2-9
Implementing mSQL 2-11
mSQL Tools...2-12
mSQL Commands.. 2-14
The `SELECT` Statement...2-15
The `INSERT` Statement...2-18
The `DELETE` Statement...2-18
The `UPDATE` Statement...2-18
Exercise: mSQL Database Queries.. 2-20
Preparation..2-20
Tasks 2-20
Exercise Summary.. 2-22
Check Your Progress 2-23
Think Beyond 2-24

Introduction to JDBC 3-1
Relevance.. 3-2
Introducing the JDBC Interface.. 3-3
ANSI SQL-2 Conformance..3-4
JDBC Driver Interface...3-6
`MsqlDriver` – A Real-World JDBC Driver...........................3-8
Connecting Through the JDBC Interface.................................... 3-10
The `java.sql` Package...3-10
The JDBC Flow 3-11
JDBC Example 3-12
Explicitly Creating an Instance of a JDBC Driver 3-14
Registering a Driver...3-17
Specifying a Database...3-18

Opening a Database Connection 3-19
Using Database Resolution..3-20
JDBC Statements 3-21
Submitting a Query 3-21
Receiving Results 3-22
Using the `get`*XXX* Methods 3-24
Working With Prepared Statements 3-25
An Example of Using a Prepared Statement......................3-26
Using the `set`*XXX* Methods..3-27
Creating Callable Statements 3-28
An Example of Using a Callable Statement 3-29
Mapping SQL Data Types Into Java Data Types....................... 3-30
ABCStock – APIs to Access the `StockMarket` Database........... 3-31
The `StockMarket` Database 3-31
The Database Utility Classes 3-31
ABCStock – Support Classes for the `StockMarket` Database .. 3-33
`CustomerRec.java` 3-33
`SharesRec.java`..3-35
`StockRec.java`...3-37
ABCStock – Database API 3-39
Methods..3-40
Exercise: Implementing A Database Class Wrapper 3-45
Preparation...3-45
Tasks 3-45
Notes 3-46
Exercise Summary... 3-47
Check Your Progress 3-48
Think Beyond 3-49

Building GUIs..4-1
Relevance.. 4-2
GUI Design Principles 4-3
Basic Principles...4-3
Subjective Versus Objective..4-4
Assess Your Audience..4-4
Clarify User Tasks..4-5
Keep It Simple 4-6
Maintain Consistency 4-7
Style Is Everything...4-10
Classes and Object-Oriented Design.. 4-12
AWT and Swing... 4-15
The `java.awt` and `javax.swing` Packages 4-17
Building Blocks.. 4-18
Containers 4-18
AWT..4-19
Swing 4-20

Panels 4-21
Components 4-22
Layout Managers 4-24
Wrap-up 4-25
Exercise: Creating the Stock Market GUI 4-26
Preparation 4-26
Tasks 4-26
Exercise Summary 4-30
Check Your Progress 4-31
Think Beyond 4-32

Networking Connections 5-1
Relevance 5-2
Networking With Java Technology 5-3
The Live-Feed Application 5-4
The Live-Feed Application Specifications 5-5
The `TickerTape` Object 5-6
Adding the `TickerTape` Object 5-8
Class Hierarchy 5-8
Instance Variables 5-8
The `TickerTape` Class 5-9
The `MakeFraction` Class 5-11
The `TickerReader` Class 5-12
Exercise: Creating the `TickerReader` 5-14
Preparation 5-14
Tasks 5-14
Exercise Summary 5-16
Check Your Progress 5-17
Think Beyond 5-18

Multiple-Tier Database Design 6-1
Relevance 6-2
The Tiered Database Model 6-3
One-Tier Databases 6-3
Two-Tier Databases 6-4
Three-Tier Databases 6-5
Business Rules 6-7
The BrokerTool Program 6-9
The Three-Tier Database Design 6-10
Protocol Design 6-12
Command and Result Formats 6-13
`Result.java` 6-14
`Command.java` 6-15
Command Implementation 6-17
Exercise: Implementing the Protocol 6-23
Preparation 6-23
Tasks 6-23

Notes 6-25
Exercise Summary 6-26
Check Your Progress 6-27
Think Beyond 6-28

Porting Considerations and Wrap-Up 7-1
Relevance 7-2
Solaris Operating Environment-to-Windows
Porting Issues 7-3
Logical Layout, Not Absolute 7-4
File Access 7-5
Font Availability 7-6
Mouse Buttons 7-7
`MouseTest.java` 7-7
Threads 7-9
Platform-Specific Implementation Bugs 7-10
Making the BrokerTool Program Fully Functional 7-11
Exercise: Integrating Modules 7-12
Preparation 7-12
Tasks 7-12
Exercise Summary 7-13
Check Your Progress 7-14
Think Beyond 7-15

Building GUIs With AWT A-1
The `java.awt` Package A-2
Building Graphical User Interfaces A-4
Containers and Components A-4
Positioning Components A-4
Component Sizing A-4
Frames A-6
Creating a Simple Frame A-6
Running the Program A-7
Panels A-8
Creating Panels A-8
Running the Program A-9
Container Layouts A-10
Layout Managers A-10
A Simple GUI Example A-11
The `main` Method A-11
`new Frame("GUI Example")` A-12
`f.setLayout(new FlowLayout())` A-12
`new Button("Press Me")` A-12
`f.add(b1)` A-12
`f.pack()` A-12
`f.setVisible(true)` A-13
Layout Managers A-14

Flow Layout Manager A-14
Border Layout Manager A-15
Grid Layout Manager A-19
Card Layout Manager A-20
Other Layout Managers A-23
Containers A-24
Frames....................A-24
Panels A-24
Creating Panels and Complex Layouts A-25

Using the `GridBagLayout` Manager B-1
Layout Managers B-2
The `GridBagLayout` Manager B-4
The `GridBagConstraints` Class B-8
Designing With `GridBagLayout` B-10
Design Steps.................... B-10
RELATIVE and REMAINDER B-20

The AWT Event Model C-1
What Is an Event? C-2
Event Sources....................C-2
Event Handlers....................C-2
How Events Are Processed C-2
Hierarchical Model (JDK 1.0) C-3
Delegation Model....................C-5
GUI Behavior C-8
Categories of Events C-8
A More Complex Example C-11
Multiple Listeners C-14
Event Adapters.................... C-15

Swing Foundations....................D-1
Comparing Swing and AWT Components D-2
Naming and Event Model D-2
Selecting Swing or AWT D-3
Converting From AWT to Swing.................... D-4
New Components in Swing.................... D-5
Top-Level Swing Containers D-6
Using a `RootPaneContainer`.................... D-7
The Root, Glass, and Layered Panes D-8
`JFrame` Essentials D-9
Reacting to the System Menu....................D-10
Implementing an `Icon` D-12
The `JLabel` Class D-13
Tool Tips.................... D-14
Buttons in Swing D-15
The `JButton` Class D-16

The `JCheckBox` Class D-17
The `JRadioButton` Class D-18
The `JComboBox` Class D-20
The `JMenu` Class D-21
Additional Features of the `JMenu` Class D-22
Keyboard Accelerators D-22
Menu Positioning D-23

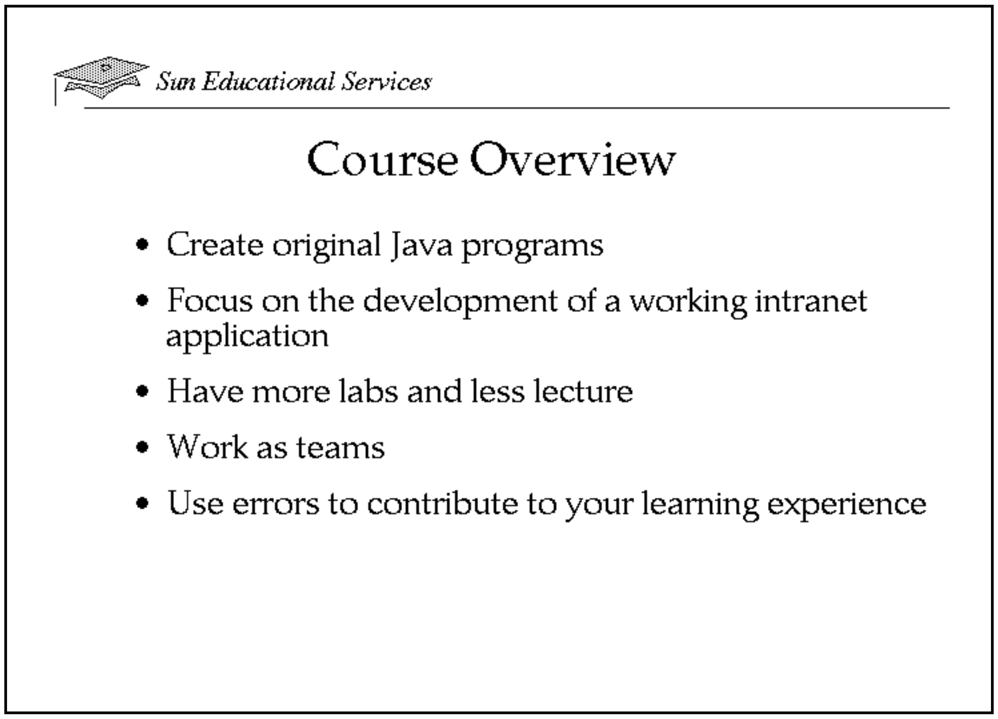
The AWT Component Library E-1
Facilities of the AWT E-2
`TextField` E-11
`List` E-13
`Frame` E-14
`Panel` E-15
`Dialog` E-16
`FileDialog` E-17
`ScrollPane` E-18
Menus E-19
The Help Menu E-19
`MenuBar` E-20
`Menu` E-21
`MenuItem` E-22
`CheckboxMenuItem` E-23
`PopupMenu` E-24
Controlling Visual Aspects E-25
Colors E-25
Fonts E-26
Printing E-27

Object Serialization F-1
Additional Resources F-1
Introduction F-2
Object Serialization: The Old Way F-2
Object Serialization: The New Way F-3
Serialization Architecture F-4
`java.io` Package F-4
`ObjectOutput` Interface F-5
`ObjectInput` Interface F-5
`Serializable` Interface F-6
What Gets Serialized F--6
Object Graphs F-7
Writing and Reading an Object Stream F-7
Serialization Versus Externalization F-8

About This Course

Course Goal

The *Java Programming Language Workshop* course provides you with the knowledge and skills necessary to design a program using Java™ technology and to carry the design through all phases of the software development cycle. This includes developing programs for multiple-tiered applications and understanding issues of porting between the Sun™ Solaris™ Operating Environment and Microsoft Windows environments.



Sun Educational Services

Course Overview

- Create original Java programs
- Focus on the development of a working intranet application
- Have more labs and less lecture
- Work as teams
- Use errors to contribute to your learning experience

Course Overview

During the next five days, you will have an opportunity to learn how to develop Java programs from beginning to end. This course is structured as a workshop because it focuses on the development of a working Java intranet application.

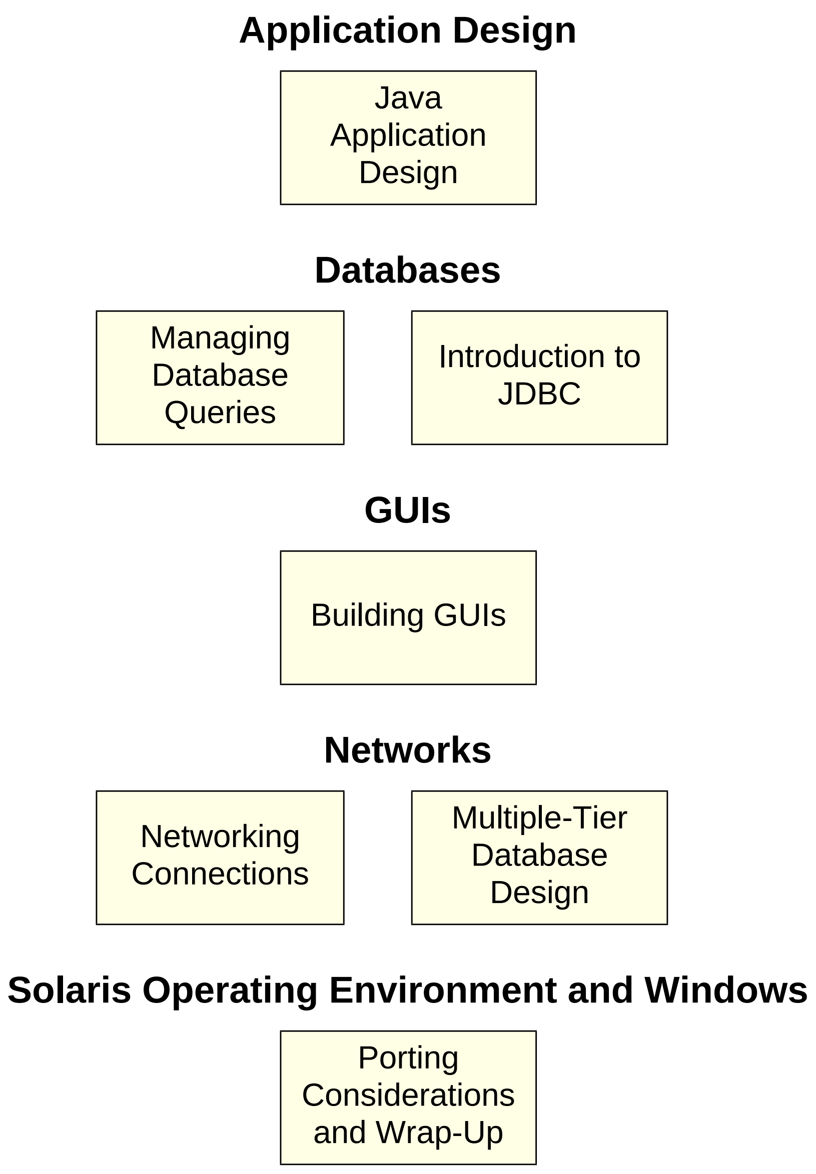
This course might differ from previous courses you have attended; there is less lecture time and more lab exercise time than in a typical five-day course. Consequently, the role of the instructor is to facilitate the lab environment rather than to lecture.

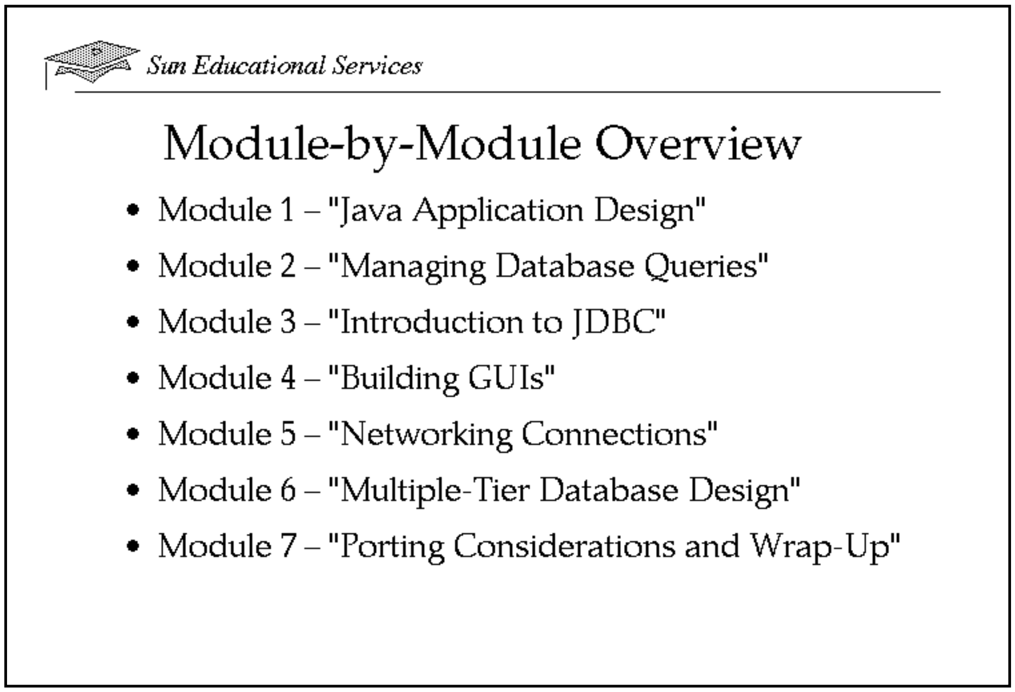
You will be divided into teams by the instructor, who will begin each lab session with information relevant to your development effort. Discussion among members of your team is encouraged. During the labs, divide the tasks equally among your team members. Consider taking on a task that you are less familiar with to promote your own learning experience.

As with any large software development effort, your team might find they have made a design error. These errors can greatly contribute to the learning experience—do not be discouraged, try to solve the problem.

Course Map

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


Application Design
Java Application Design
Databases
Managing Database Queries
Introduction to JDBC
GUIs
Building GUIs
Networks
Networking Connections
Multiple-Tier Database Design
Solaris Operating Environment and Windows
Porting Considerations and Wrap-Up


Sun Educational Services

Module-by-Module Overview

- Module 1 – "Java Application Design"
- Module 2 – "Managing Database Queries"
- Module 3 – "Introduction to JDBC"
- Module 4 – "Building GUIs"
- Module 5 – "Networking Connections"
- Module 6 – "Multiple-Tier Database Design"
- Module 7 – "Porting Considerations and Wrap-Up"

Module-by-Module Overview

This course contains the following modules:

- Module 1 – "Java Application Design"

  This module presents an overview of the phases of software development and highlights object-oriented software development techniques. In this module, you are presented with the problem statement for the Java program you will work on this week.

- Module 2 – "Managing Database Queries"

  This module presents an overview of relational database design and introduces the Structured Query Language (SQL) syntax implemented by the mSQL database system, for which you will be writing code using the Java programming language.

- Module 3 – "Introduction to JDBC"

  This module also introduces the JavaSoft™ Java DataBase Connectivity (JDBC™) Application Programming Interface (API), which you and your team will use to pass SQL commands to the database.

- Module 4 – "Building GUIs"

  This module presents design principles for good graphical user interface (GUI) design. You and your team will create an effective GUI design from the specifications provided in this module.

- Module 5 – "Networking Connections"

  This module presents information related to the live-data feed application that you and your team will integrate into the Java program you are creating.

- Module 6 – "Multiple-Tier Database Design"

  This module presents a multiple-tier database design and describes how the project could be expanded to include a middle tier.

- Module 7 – "Porting Considerations and Wrap-up"

  This module explores how to modify or enhance your team's code for portability across Java platforms.

Course Objectives

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

- Describe relational databases
- Explain the JDBC™ API
- Develop classes to connect Java programs to SQL database systems
- Develop a GUI that uses database classes
- Create classes that make socket connections as well as retrieve and format data
- Describe a multiple-tier design
- Create a multiple-tier database system
- Discuss porting issues between the Sun Solaris Operating Environment and Microsoft Windows environment
- Analyze, design, implement, and test an original commercial intranet application using Java technology

Skills Gained by Module

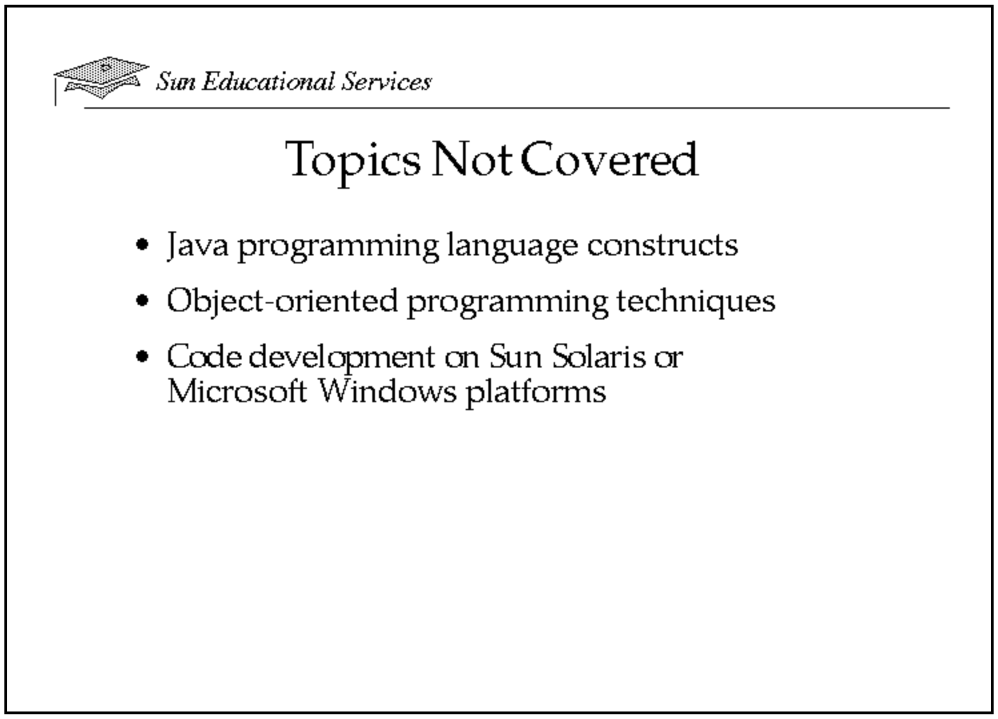
The skills for *Java Programming Language Workshop* are shown in column 1 of the matrix below. The black boxes indicate the main coverage for a topic; the gray boxes indicate the topic is briefly discussed.

	Module						
Skills Gained	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Describe relational databases		■	▨				
Explain the new JDBC API			■				
Develop classes to connect Java programs to SQL database systems			■	■	▨		
Develop a GUI that uses database classes	▨			■			
Create classes that make socket connections as well as retrieve and format data					■		
Describe a multiple-tier design						■	
Create a multiple-tier database design						■	
Discuss porting issues between Solaris Operating Environment and Microsoft Windows							■
Analyze, design, implement, and test an original commercial intranet application using Java technology	■	■	■	■	■	■	

Guidelines for Module Pacing

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

Module	Day 1	Day 2	Day 3	Day 4	Day5
"About This Course"	A.M.				
"Java Application Design"	A.M.				
"Managing Database Queries"	P.M.				
"Introduction to JDBC"		A.M.			
"Building GUIs"		P.M.	A.M.		
"Networking Connections"			P.M.		
"Multiple-Tier Database Design"				A.M./ P.M.	
"Porting Considerations and Wrap-Up"					A.M./ P.M.


Sun Educational Services

Topics Not Covered

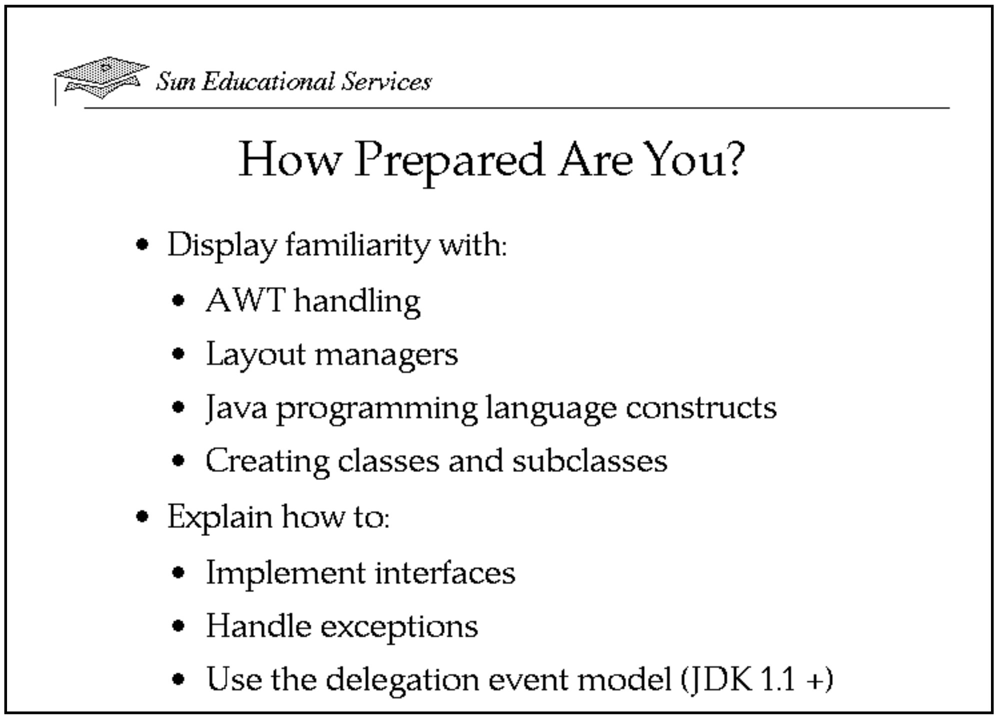
- Java programming language constructs
- Object-oriented programming techniques
- Code development on Sun Solaris or Microsoft Windows platforms

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:

- Java programming language constructs – Covered in SL275: *Java Programming Language*
- Object-oriented programming techniques – Covered in:
  - OO-225: *Object-Oriented Application Analysis and Design for Java Technology (OMT)*
  - OO-226: *Object-Oriented Application Analysis and Design for Java Technology (UML)*
- Code development documentation for the Solaris Operating Environment or Microsoft Windows

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


Sun Educational Services

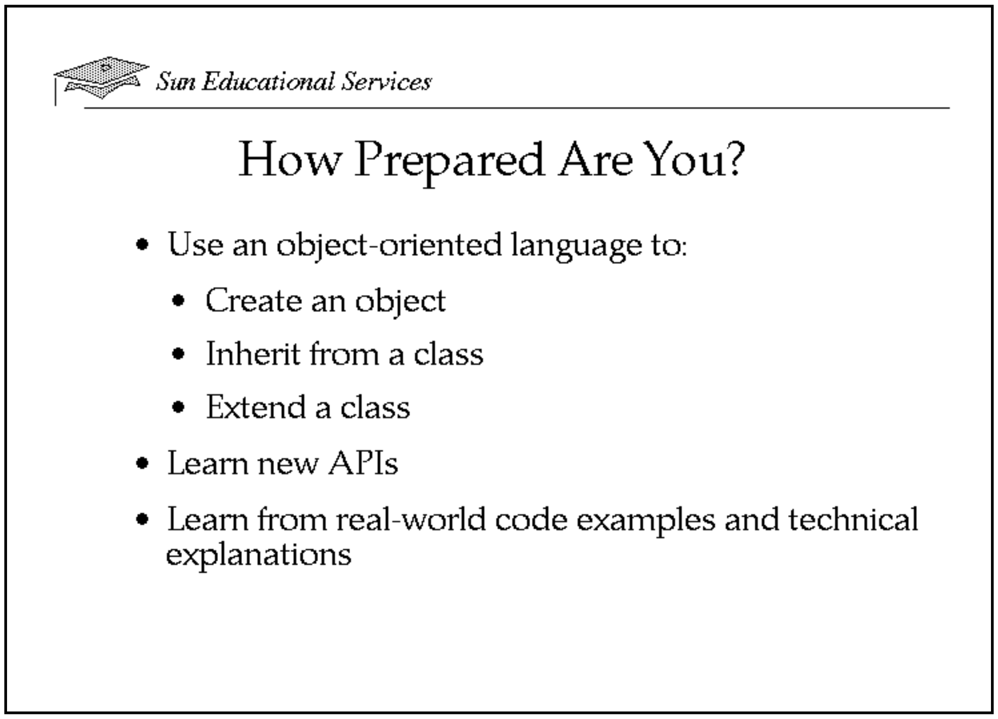
How Prepared Are You?

- Display familiarity with:
  - AWT handling
  - Layout managers
  - Java programming language constructs
  - Creating classes and subclasses
- Explain how to:
  - Implement interfaces
  - Handle exceptions
  - Use the delegation event model (JDK 1.1 +)

How Prepared Are You?

To be sure you are prepared to take this course, you should be:

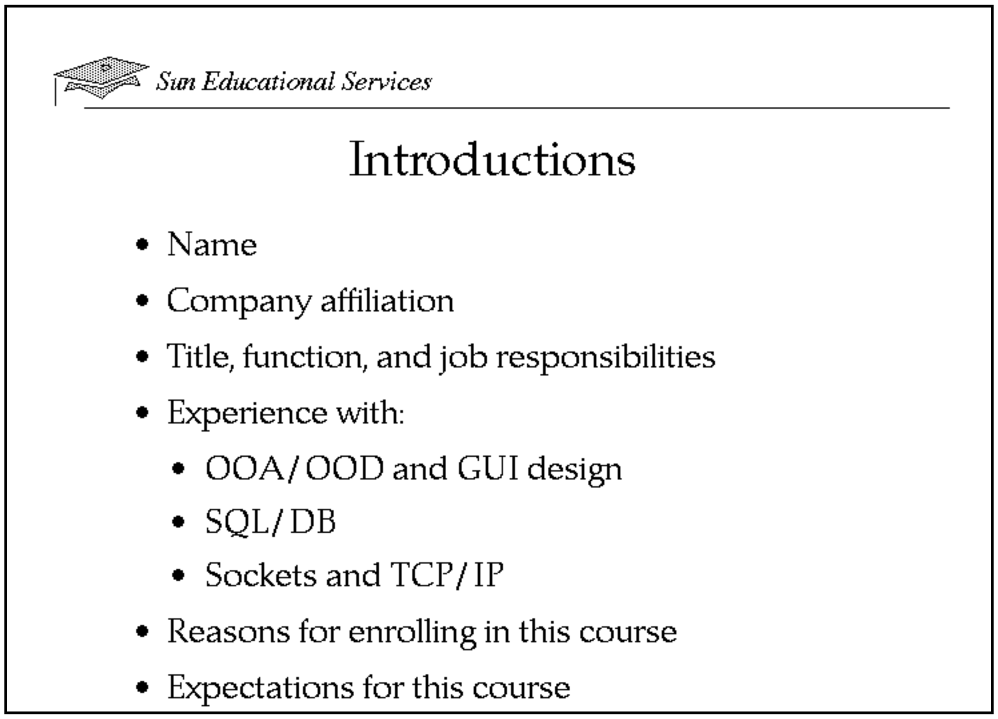
- Familiar with Abstract Window Toolkit (AWT) event handling, layout managers, Java programming language constructs, and creating classes and subclasses, all of which are necessary to develop the code components in this course
- Able to demonstrate how to implement interfaces, handle exceptions, and use the delegation event model


Sun Educational Services

How Prepared Are You?

- Use an object-oriented language to:
  - Create an object
  - Inherit from a class
  - Extend a class
- Learn new APIs
- Learn from real-world code examples and technical explanations

- Able to create programs using an object-oriented language (such as the Java programming language or C++); including creating an object, inheriting from a class, extending a class, and so on
- Comfortable learning the main concepts of the JDBC API presented in this course
- Able to learn from the real-world code examples and technical explanations presented in this course

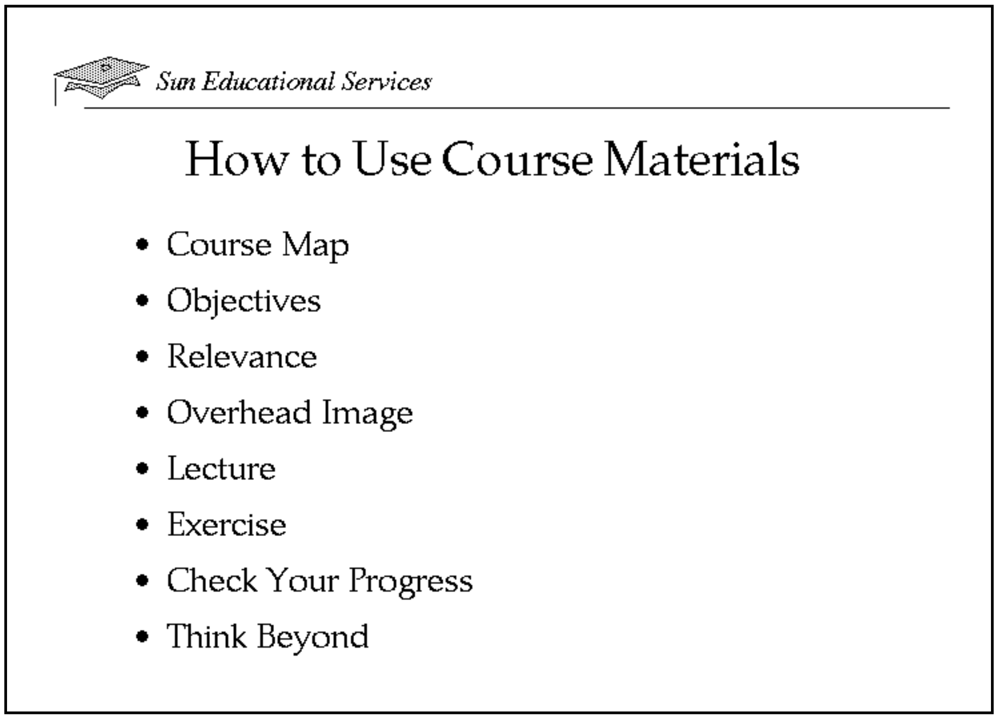

Sun Educational Services

Introductions

- Name
- Company affiliation
- Title, function, and job responsibilities
- Experience with:
  - OOA/OOD and GUI design
  - SQL/DB
  - Sockets and TCP/IP
- 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 the instructor, addressing the items shown on the above overhead.


Sun Educational Services

How to Use Course Materials

- Course Map
- Objectives
- 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** – An overview of the course content appears in the "About This Course" module so you can see how each module fits 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 thinking about how the module content relates to carrying a design through every phase of the software development cycle and to multiple-tier database designs.

- **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.
- **Lecture** – The instructor presents information specific to the topic of the module. This information helps you learn the knowledge and skills necessary to succeed with the exercises.
- **Exercise** – Lab exercises 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.

Demonstration - Indicates a demonstration is recommended at this time.

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.

Additional resources – Indicates additional reference materials are available.

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:

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

`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 Operating Environment:

  ```
  cd server_root/bin
  ```

  On Microsoft Windows:

  ```
  cd server_root\bin
  ```

Java Application Design 1

Objectives

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

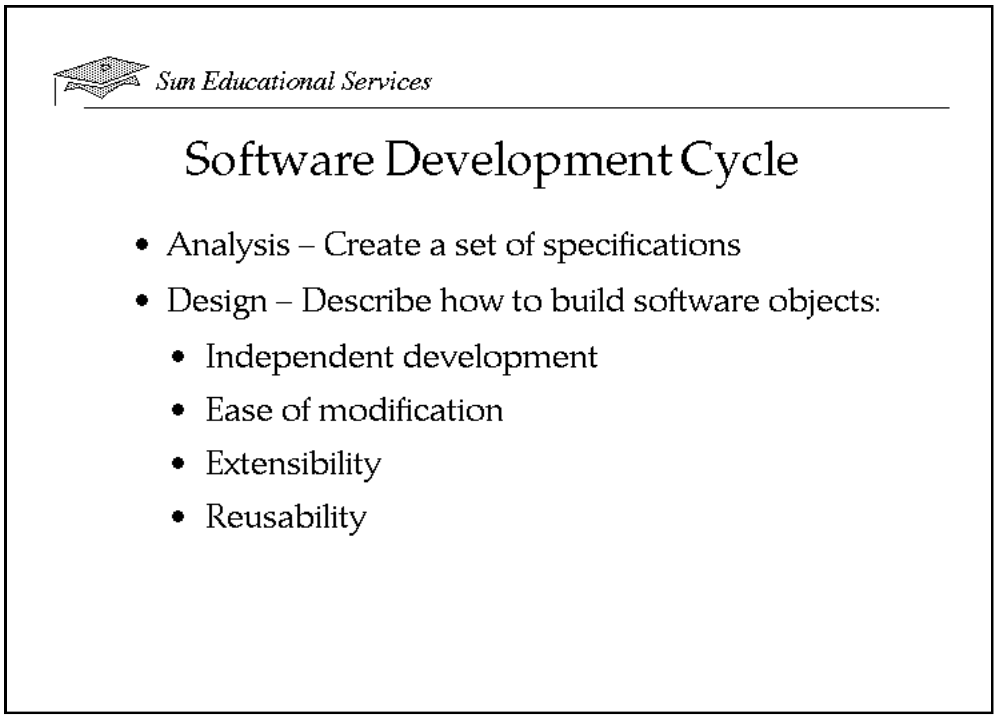
- Explain the goals of each phase of the software development cycle
- Demonstrate an understanding of the Java runtime environment by relating the application components to the appropriate Java program type
- Discuss the basic elements of API design
- Define the BrokerTool program components given the functional requirements of the project

In this module, you will discuss the goals of each phase of the software development cycle and how to apply application components to the Java programming language data types. You will also begin the analysis and design for a program called BrokerTool.

Relevance

Discussion – Analysis and design are critical steps in developing an object-oriented (OO) solution. Designing correct classes up-front can lead to increased performance and successful milestones as the project progresses.

In contrast, a poorly designed project will fail quickly if developers try to piece together ill-conceived software components. How might “lack of planning” be obvious when trying to deploy project-specific packages and classes?


Sun Educational Services

Software Development Cycle

- Analysis – Create a set of specifications
- Design – Describe how to build software objects:
  - Independent development
  - Ease of modification
  - Extensibility
  - Reusability

Software Development Cycle

The software development cycle is an iterative process comprising the following phases:

- Analysis
- Design
- Implementation
- Testing
- Revision

The amount of time spent on the first two phases can greatly reduce the number of iterations during the development cycle. This module focuses on the software analysis and design phase.

The development cycle begins with prototyping iterations and then moves through alpha and beta iterations until a final product is completed. In developing the BrokerTool program, you make a single pass through an initial prototype, and then conclude the course with a discussion of the possible types of revisions that might be desirable.

Analysis

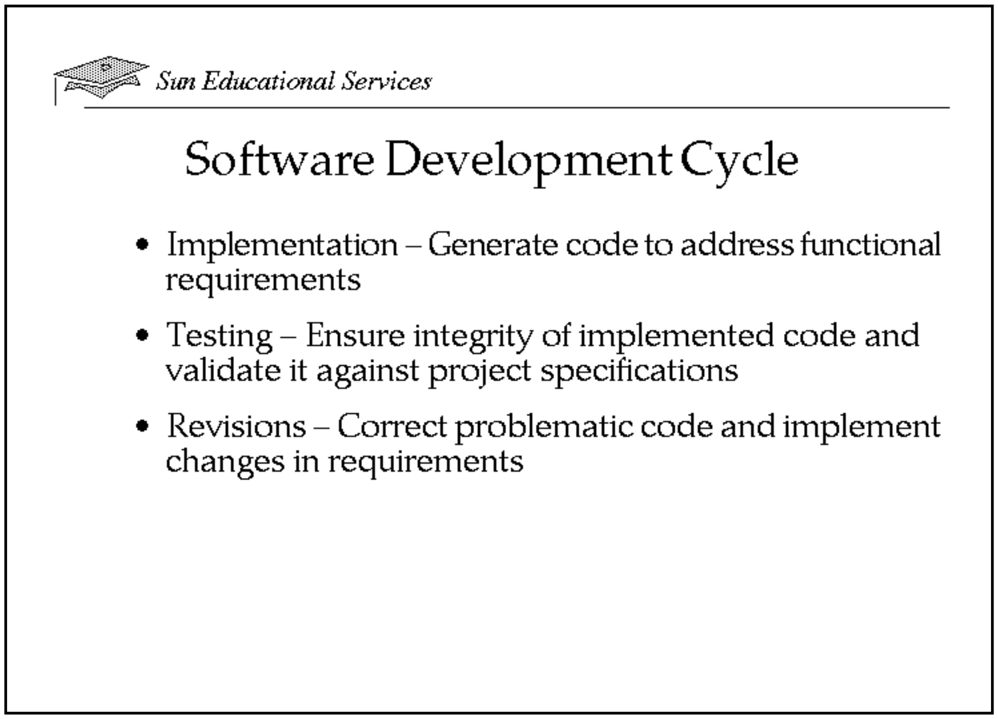
During the analysis phase of OO software development, you transform a minimum target specification and certain project requirements into a complete entity. The specifications for the analysis phase include functional, resource, and performance requirements as well as a description of essential characteristics, behavior, and project constraints.

Design

During the design phase, you generate a description of how to build software objects that behave in accordance with the analysis models and that meet all other system requirements.

The benefits of an object-oriented, modular design are:

- Independent development
- Ease of modification
- Extensibility
- Reusability


Sun Educational Services

Software Development Cycle

- Implementation – Generate code to address functional requirements
- Testing – Ensure integrity of implemented code and validate it against project specifications
- Revisions – Correct problematic code and implement changes in requirements

Software Development Cycle

Implementation

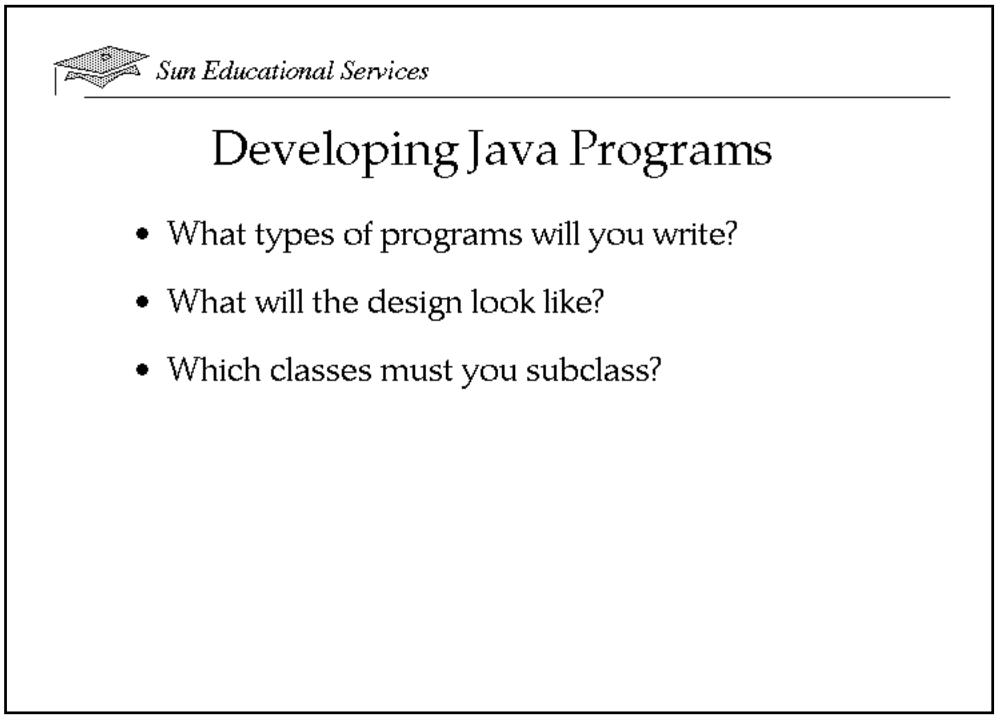
During the implementation phase, you generate code that addresses the functional requirements of the project.

Testing

The testing phase ensures the integrity of the software developed during implementation and validates that the software meets the project specifications identified during the analysis phase. During this phase, you can independently test each module of the application.

Revisions

Revisions are typically the most time-consuming aspect of any software development project. Revisions are necessary to correct any problematic code and to make any modifications in response to changing requirements.



Sun Educational Services

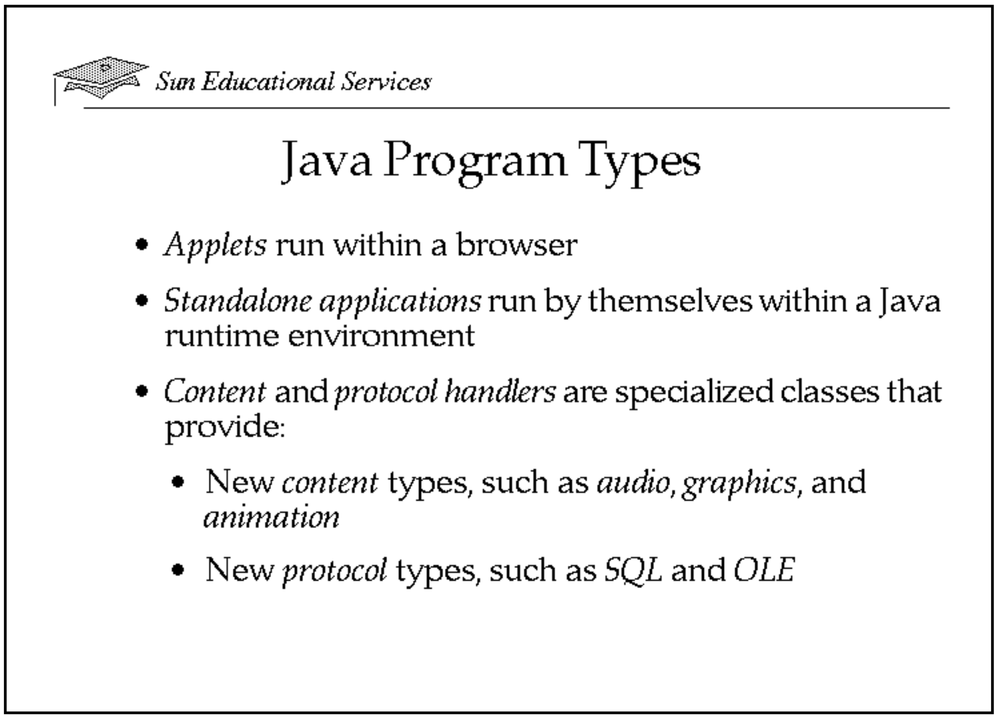
Developing Java Programs

- What types of programs will you write?
- What will the design look like?
- Which classes must you subclass?

Developing Java Programs

Java program development follows the analysis, design, development, test, and revision process. However, you should consider the following:

- What types of program will you write – applets or applications? Is there a need to write either a content or protocol handler?
- What will the design look like?
- Which classes can you use as is, and which classes must you use as subclasses?



Sun Educational Services

Java Program Types

- *Applets* run within a browser
- *Standalone applications* run by themselves within a Java runtime environment
- *Content* and *protocol handlers* are specialized classes that provide:
  - New *content* types, such as *audio*, *graphics*, and *animation*
  - New *protocol* types, such as *SQL* and *OLE*

Java Program Types

The Java platform provides support for several different program types:

- *Applets* run within a browser that contains a Java runtime environment. Applets inherit functionality from an Applet class. Included in this functionality is a graphics object that contains drawing methods for displaying graphic images.

- *Standalone applications* run by themselves within a Java runtime environment. Applications in Java technology must have access to a runtime interpreter implemented for the specific operating system.

- *Content* and *protocol handlers* are specialized Java classes that enhance functionality by providing a means for introducing new content types (audio, graphics, and animations) or new protocol types (SQL, OLE [object linking and embedding] and so on).

Class Definition

After you decide what type of program you will create, you must then define classes, which make up the backbone of your program.

To define a class:

- Create a class name for each application component derived from the project analysis
- Determine whether your class inherits functionality. You use the `extends` keyword to indicate whether your class inherits functionality.

To determine attributes and variables:

- Determine what attributes characterize the class
- Declare class variables of appropriate types

To declare methods:

- Determine if the class needs a constructor method
- Identify methods required to:
    - Create or destroy supporting objects (GUI components, threads, network connections)
    - Set and get class attribute values
- Determine what methods will be required to:
    - Respond to user-initiated events
    - Respond to timing events or error conditions

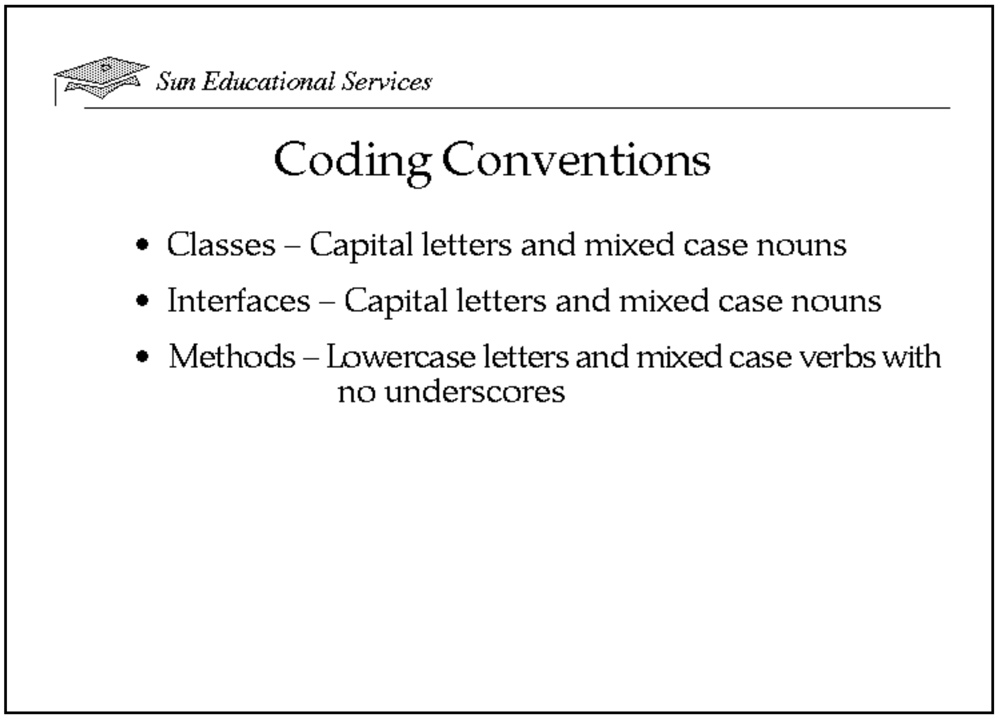
Java API Design

The API of an object-oriented program identifies and describes the classes and all public methods defined within those classes.

Complete API Declarations

Once you have named your methods, you must specify three more characteristics to complete the declaration. You must determine the following:

- The parameters required by the method. If parameters are required, should they be objects or primitive data types?
- The return type of the method, which can be one of the following:
  - boolean (indicating success or failure)
  - Data type (for example, `int`, `String`, and so on)
  - List of elements
  - Nothing (`void`)
- The scope of the method, which can be one of the following:
  - `private`
  - `protected`
  - `public`
  - package/default



Sun Educational Services

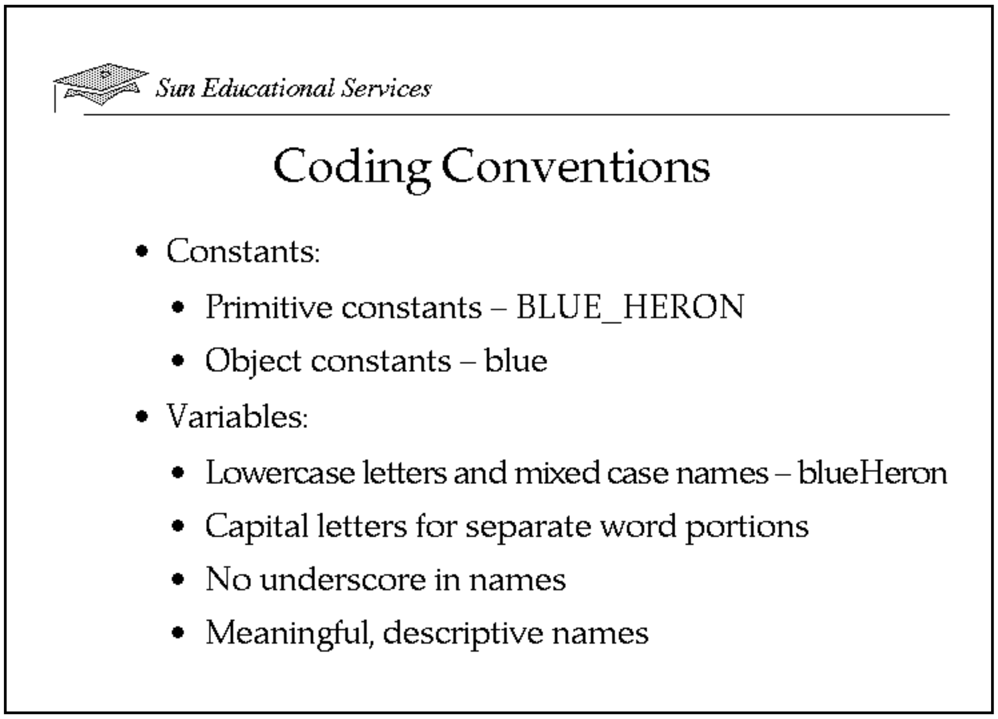
Coding Conventions

- Classes – Capital letters and mixed case nouns
- Interfaces – Capital letters and mixed case nouns
- Methods – Lowercase letters and mixed case verbs with no underscores

Coding Conventions

The coding conventions are:

- Classes – Use nouns for class names. Use mixed case, with the first letter of each word capitalized.
- Interfaces – Capitalize interface names like class names.
- Methods – Use verbs in mixed case with the first letter lowercase for method names. Within a method name, capital letters start each of the separate words. The underscore in method names is usually not used.



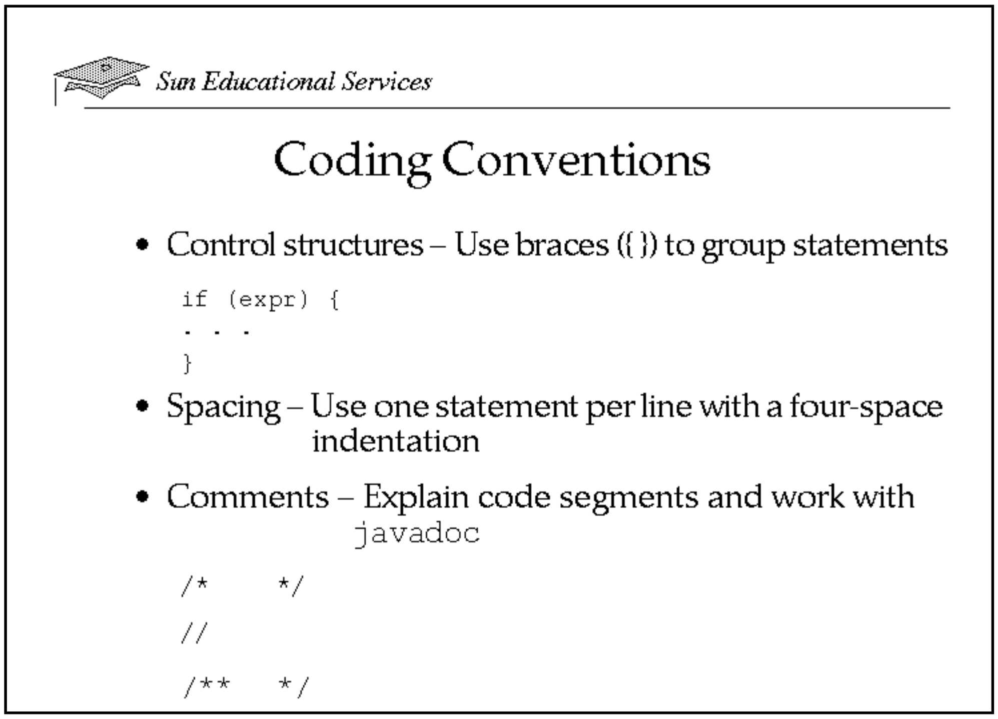
Sun Educational Services

Coding Conventions

- Constants:
  - Primitive constants – BLUE_HERON
  - Object constants – blue
- Variables:
  - Lowercase letters and mixed case names – blueHeron
  - Capital letters for separate word portions
  - No underscore in names
  - Meaningful, descriptive names

- Constants – Use all uppercase for primitive constants with words separated by underscores. Object constants can use mixed case letters.
- Variables – Use mixed case with a lowercase first letter for all instance, class, and global variables. Words are separated by capital letters. The underscore in variable names is usually not used.

Variables should be meaningful and descriptive. The name should indicate to the casual reader the intent of its use. Avoid one-character names except for temporary "throwaway" variables (for example, i, j, and k for a loop control variable that is not used outside the loop).



Sun Educational Services

Coding Conventions

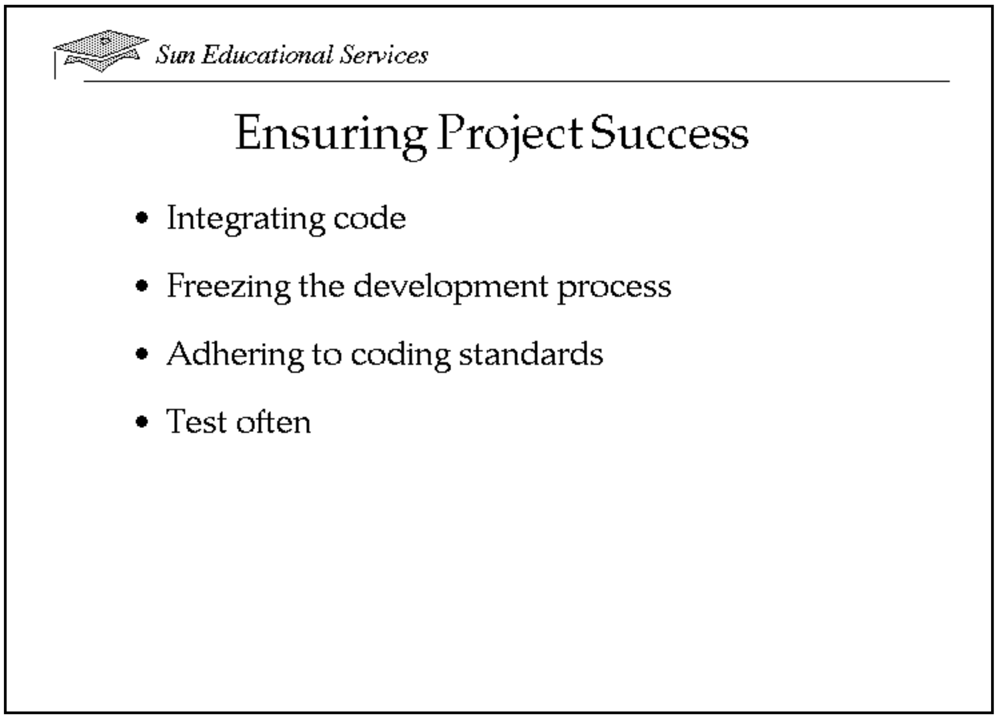
- Control structures – Use braces ({ }) to group statements

```
if (expr) {
. . .
}
```

- Spacing – Use one statement per line with a four-space indentation
- Comments – Explain code segments and work with `javadoc`

```
/*    */

//

/**   */
```

- 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.
- Spacing – Place only a single statement on any line, and use a four-space indentation to make your code more readable.
- Comments – Use comments to explain code segments that are not obvious. Use the `//` comment delimiter for single-line or short commenting, and `/* . . . */ for` large sections of code. Use the `/** . . . */` documenting comment and `javadoc` to provide your future maintenance person with an API.


Sun Educational Services

Ensuring Project Success

- Integrating code
- Freezing the development process
- Adhering to coding standards
- Test often

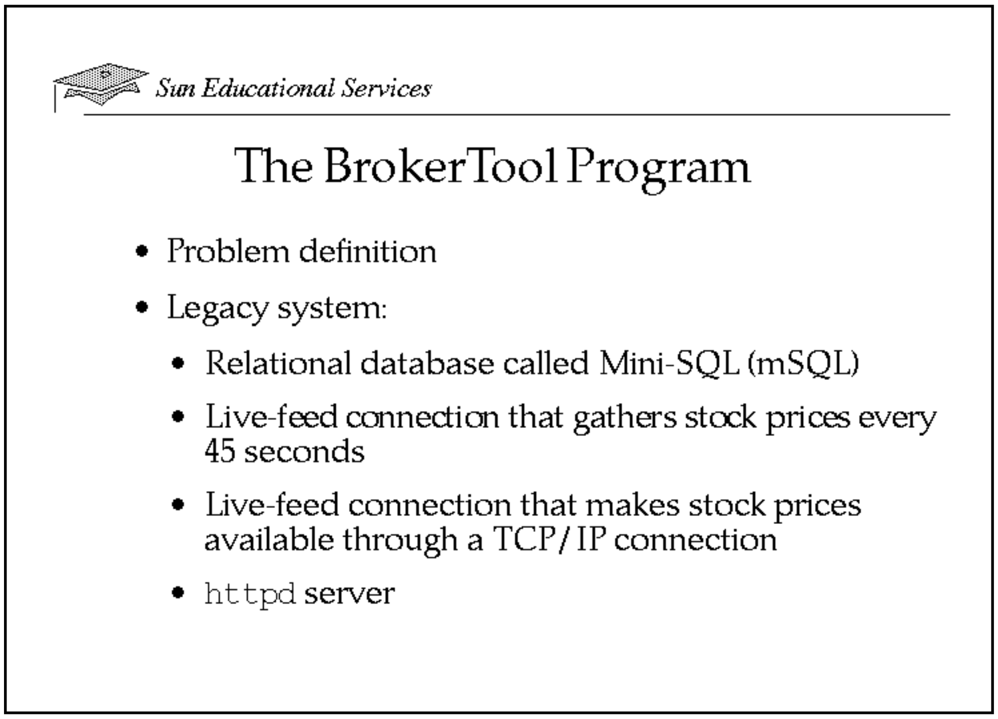
Ensuring Project Success

Integrating code is one of the most common stumbling blocks in development projects that require team effort. When incorporating code from many sources, writing the complete API declarations as part of the design process helps reduce the time spent in the revision cycle.

Freezing the development process periodically enables the team to evaluate whether the system is meeting its design objectives and provides a recovery position if future modifications result in the need to return the code to a "known" state.

Adhering to coding standards improves the ability to read and maintain code. Additionally, it enables team members to review each other's code, without having to learn new conventions.

Test often. Try to test every module you write. What seems time consuming at first will make life easier in the end.



Sun Educational Services

The BrokerTool Program

- Problem definition
- Legacy system:
  - Relational database called Mini-SQL (mSQL)
  - Live-feed connection that gathers stock prices every 45 seconds
  - Live-feed connection that makes stock prices available through a TCP/IP connection
  - `httpd` server

The BrokerTool Program

To adhere to a single program specification, the basic elements of the BrokerTool program have been defined for you. Further details of the BrokerTool API, such as GUI design and implementation, database query, and network communication, are described in subsequent modules.

Problem Definition

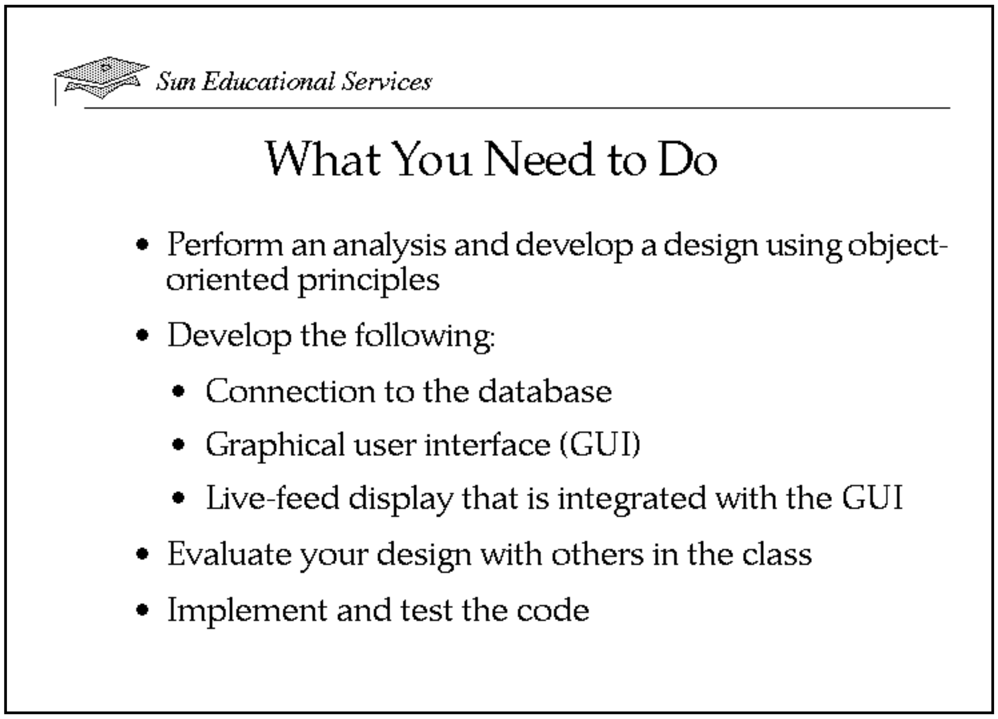
Some time ago, a consultant was hired to develop an application for the ABC Stock Trading Company. Unfortunately, this person won the local state lottery and promptly gave notice. The last known communication from this person was a postcard from the Cayman Islands.

For the duration of the week, assume that you are a group of consultants hired by the ABC Stock Trading Company. Your only internal resource is the MIS (management information systems) manager, who has gathered you here today to complete the development of the BrokerTool.

Legacy System

The consultant was in the midst of developing an application that would serve ABC's new intranet (an internal and secure local area network [LAN]). The following pieces of the application have been completed:

- A relational database created on a small but powerful database program called Mini-SQL (mSQL)
- A live-feed connection that gathers stock prices every 45 seconds and makes the current prices available through a Transmission Control Protocol/Internet Protocol (TCP/IP) connection
- An `httpd` server installation



Sun Educational Services

What You Need to Do

- Perform an analysis and develop a design using object-oriented principles
- Develop the following:
  - Connection to the database
  - Graphical user interface (GUI)
  - Live-feed display that is integrated with the GUI
- Evaluate your design with others in the class
- Implement and test the code

What You Need to Do

The MIS director recently attended the JavaOne^{SM} conference and was intrigued by the possibility of writing a single program that would run on several hundred client machines, ranging from Macintosh computers and Intel Pentium-based PCs (running Microsoft Windows) to Sun workstations running the Solaris Operating Environment.

This week, you will divide into teams and complete the following:

- Analyze the problem and develop a design using object-oriented principles
- Develop the following;
  - ▼ Connection to the database
  - ▼ GUI
  - ▼ Live-feed display that is integrated with the GUI
- Evaluate your design with others in the class
- Implement and test your code

Specifications of the Legacy Server

The MIS director has given you the following information:

- The database, live feed, and `httpd` all run on a single server.
- The database is mSQL, a small but powerful SQL database.
    - The database is designed with a specific schema that you cannot change.
    - The database can receive string SQL commands and return string responses.
    - You access the database through TCP/IP port 1112.
- The live feed is a custom-built Java application created by the previous consultant. It has the following characteristics:
    - Has data that changes every 45 seconds
    - Updates the database at each 45-second interval
    - Is accessed through TCP/IP port 5432
    - Accepts client connections
- The `httpd` is a web server that does the following:
    - Uses the standard port 80.
    - Stores hypertext markup language (HTML) files in a directory shared by the server.

The Legacy Database

The existing database, `StockMarket`, contains the following schema:

Customer Table

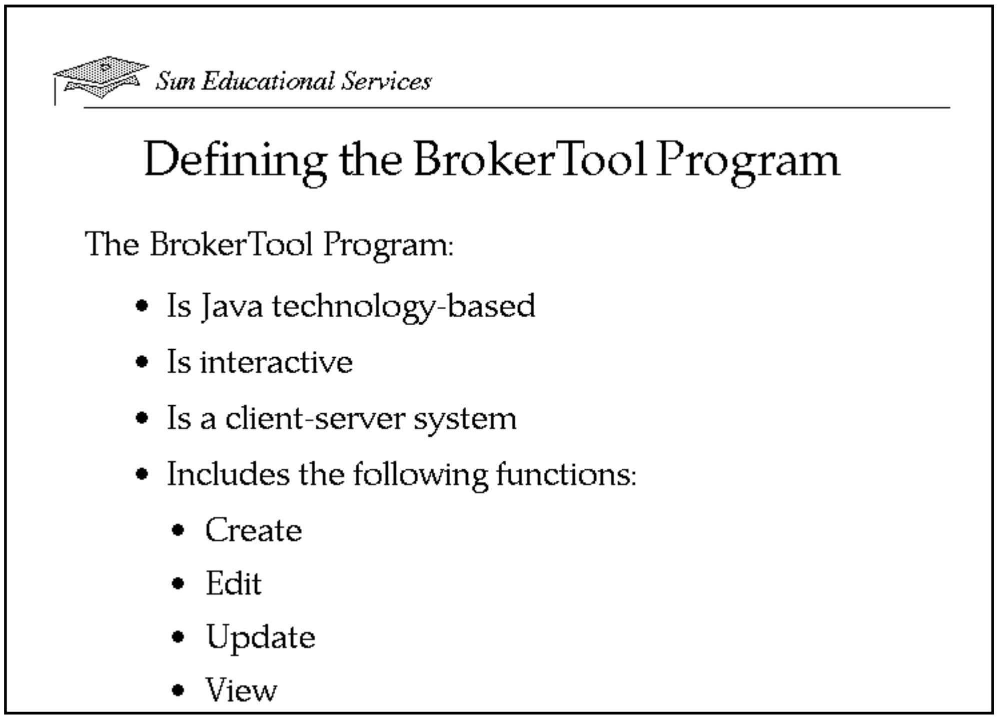
Field Name	Type	Comment
`ssn`	char (15)	
`cust_name`	char (40)	
`address`	char (100)	
`idx1`	index	unique

Shares Table

Field Name	Type	Comment
`ssn`	char (15)	Not null
`symbol`	char (8)	Not null
`quantity`	int (4)	

Stock Table

Field Name	Type	Comment
`symbol`	char (8)	
`price`	real	
`idx2`	index	unique



Sun Educational Services

Defining the BrokerTool Program

The BrokerTool Program:

- Is Java technology-based
- Is interactive
- Is a client-server system
- Includes the following functions:
  - Create
  - Edit
  - Update
  - View

Defining the BrokerTool Program

The BrokerTool program that you have been assigned to develop will be a Java technology-based, interactive, client-server system for creating, editing, updating, and viewing customer and stock information that is contained in the legacy database.

Functional Requirements

Your program must enable the end-user to:

- Buy and sell stocks for customers in the database and update the database and GUI appropriately
- Add and remove customers from the database
- Modify the customer's name or address, but not a social security number
- View the current price for any stock in the database
- Read the contents of the live-feed data stream from the server, and see the contents as a scrolling ticker tape in the final GUI

Note – Use the APIs provided so that, at the end of the week, any of your classes can be swapped with any of the other groups' classes and integrated seamlessly.

Consider any other functions you think the end-users might need to perform their job.

Figure 1-1 provides a visual overview of the BrokerTool program.

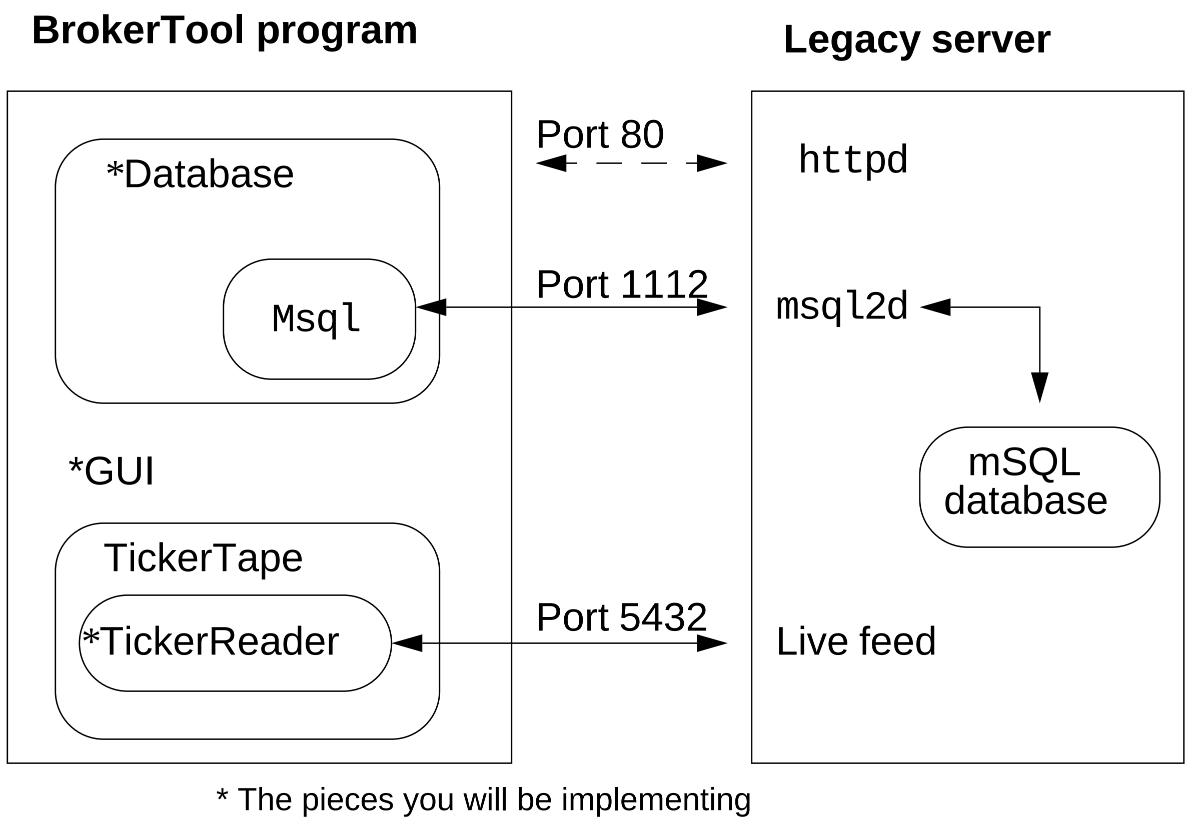
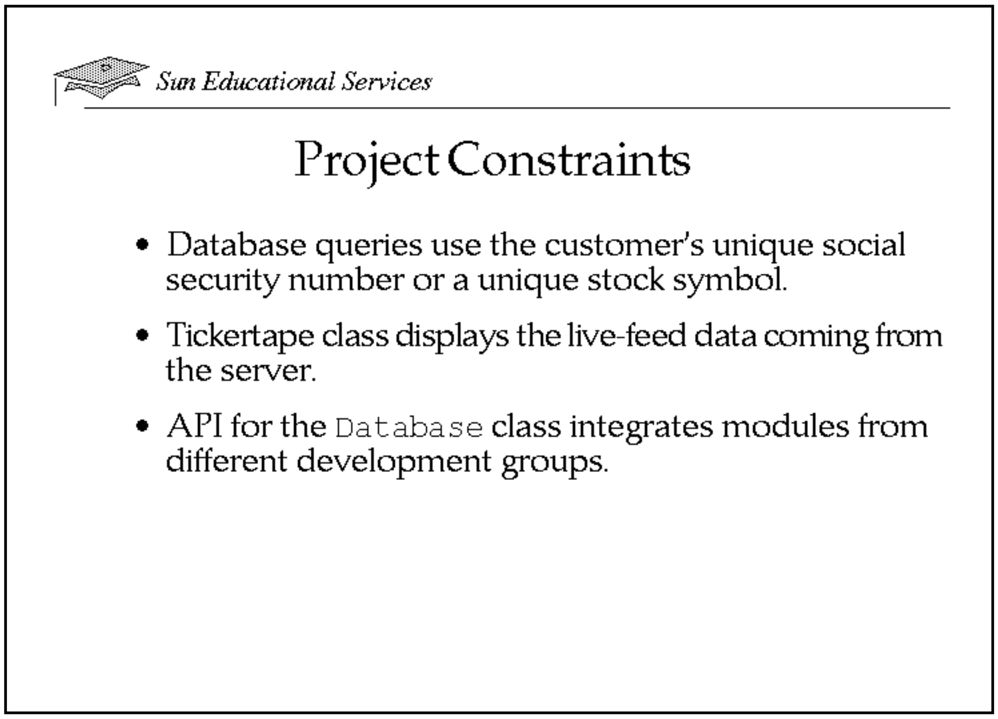


Figure 1-1 BrokerTool Program Overview

Resources

The following resources are provided:

- A server machine, installed with the legacy database, and live-data feed networked to multiple clients.
- Multiple client graphic workstations installed with the Java 2 Platform.
- Appletviewer or a browser, such as Netscape Navigator™ or HotJava™.



Sun Educational Services

Project Constraints

- Database queries use the customer's unique social security number or a unique stock symbol.
- Tickertape class displays the live-feed data coming from the server.
- API for the `Database` class integrates modules from different development groups.

Project Constraints

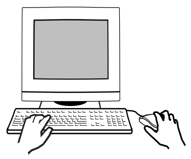
Given the requirements for the project, the first step is to define those requirements in terms of the resources and technology available. Specifically, you must apply constraints to limit the functionality of the project to the given technology, resource, and time limitations.

Given the time limitations:

- Database queries should use the customer's unique social security number or a unique stock symbol.
- A ticker-tape class will be provided for integration into the final GUI and to display the live-feed data coming from the server.
- An API for the `Database` class that integrates modules from different development groups will be provided.

Notes

Exercise: The BrokerTool Program Initial Design



Exercise objective – Given the instructions for defining the BrokerTool project, create a preliminary design.

Preparation

You must have a basic knowledge of the Java programming language and object-oriented programming. You should also know basic GUI design and how to use databases.

Tasks

Complete the following steps:

1. Form groups of three to five students.
2. Elect a group leader. This person guides the development effort and represents your group.
3. As a group, perform a simple analysis of what you know, and create a preliminary design. Consider the following design issues:
    - Applet or application – Which will you write?
    - Define object – What kinds of things need objects?
    - Define classes – How will you map objects?
    - GUI capabilities – What will the GUI enable the user to do?
    - Information flow – How will objects communicate with each other?
4. Save any questions you have for the discussion.

Notes

Exercise Summary

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

- Experiences
- Interpretations
- Conclusions
- Applications

Check Your Progress

Before continuing on to the next module, check that you are able to accomplish or answer the following:

- ❑ Explain the goals of each phase of the software development cycle
- ❑ Demonstrate an understanding of the Java runtime environment by relating the application components to the appropriate Java program type
- ❑ Discuss the basic elements of API design
- ❑ Define the BrokerTool program components given the functional requirements of the project

Think Beyond

Many legacy systems are not written in the Java programming language.

How does a programmer that uses the Java programming language address the integration of legacy software so that new components take advantage of the legacy software's functionality?

Managing Database Queries 2

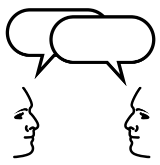
Objectives

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

- Describe relational database design
- Explain Codd's first three rules of relational database design
- Construct mSQL queries
- Create a connection to an mSQL database
- Extract information from an mSQL database

This module describes how to connect to basic relational databases, such as mSQL, and also describes how to extract and change information.

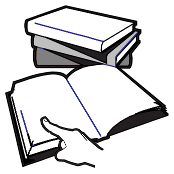
Relevance



Discussion – Relational databases are prevalent in computing today. Being able to integrate software solutions with databases allows for persistent storage of just about any type of information.

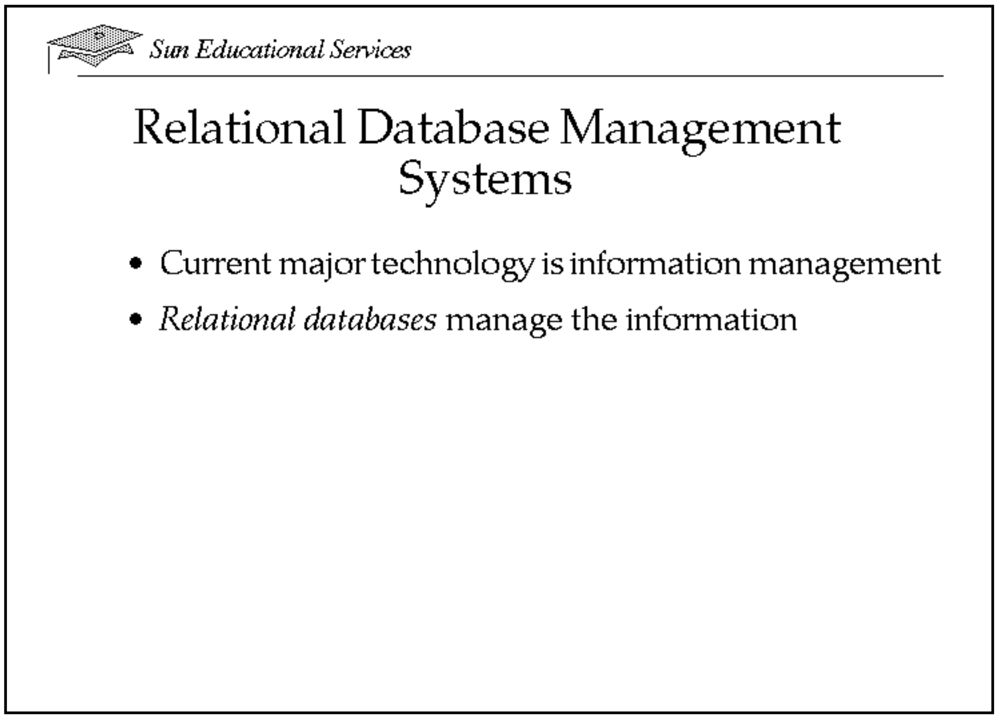
What are some examples of the types of information that a software project should make persistent (survive beyond the application's execution)?

Additional Resources



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

- Emerson, Darnovsky, and Bowman. 1989. *The Practical SQL Handbook,* Addison-Wesley.
- Hughes. 1997. *Mini SQL 2.0 User Guide*. Available: `http://www.Hughes.com.au.` There is also an online version of the *Mini SQL 2.0 User Guide* included with the course materials: `SL285_LF/labfiles/msql_documentation/manual-html/manual.html`.



Sun Educational Services

Relational Database Management Systems

- Current major technology is information management
- *Relational databases* manage the information

Relational Database Management Systems

The major technology of our age is not the computer, but information management. Once the quantity of information exceeds our natural capacity for memory, it is necessary to store the information somewhere and store the data in a way that makes it easy to retrieve and store additional information.

The solution is the *relational database*. Dr. E. F. Codd is credited as the inventor of the relational model, which defines how a relational system must operate. Codd's Rules is a comprehensive, 12-part test that describes the criteria that must be met for a system to be classified as a relational database management system (RDBMS)[1].

1. Source: *The Practical SQL Handbook*, Emerson, Darnovsky and Bowman, Addison-Wesley Publishing, 1989.

To be considered fully relational, an RDBMS must:

- Represent all of the information in the database as tables
- Keep the logical representation of the data independent of its physical storage characteristics
- Use a high-level language for structuring, querying, and changing the data
- Support relational operations (such as selection and joining) and set operations (such as intersection and difference)
- Support views, which allow alternative methods for looking at data in tables
- Provide a method for differentiating between unknown values and nulls, zeros, and blanks
- Support mechanisms for integrity, authorization, transactions, and recovery

Relational Models

Tables

The first rule of an RDBMS (according to Codd) is that data are represented in tables. Tables have *rows* (horizontally) and *columns* (vertically) and can represent any type of data. Table 2-1 illustrates a table containing names, addresses, and phone numbers.

Table 2-1 Address Table

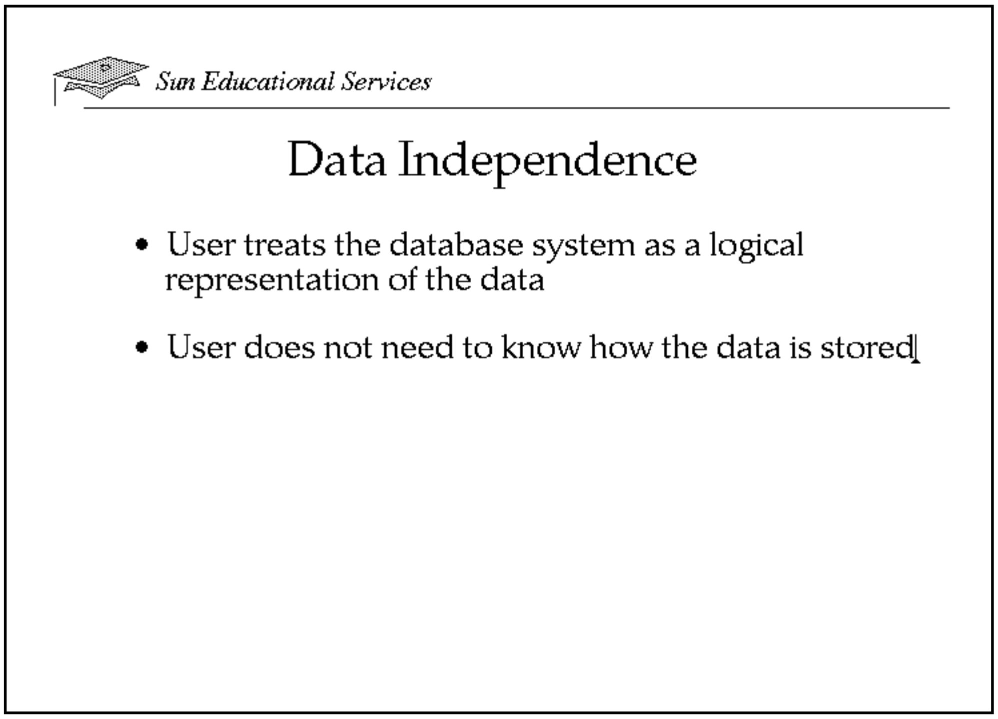
Name	Address	Phone Number
Brown, John	101 High Street	(408) 555-2024
Callaway, Tim	1334 East Main	(515) 555-1200
Devroe, Anthony	1 Park Place	(212) 555-0908
Nikolai, Peter	108 High Plain Rd	(508) 555-2701

Tables identify the column names and, generally, give indices to rows. Rows are also referred to as *records*, and the columns are referred to as *fields*. One of the fields can be identified as a *primary key*, which identifies the record as unique.

A set of tables forms a database and enables you to create *relations* between data. Table 2-2 illustrates a table containing social security numbers.

Table 2-2 Social Security Table

Name	Social Security Number	Age
Brown, John	999-01-1313	23
Callaway, Tim	999-25-8976	34
Devroe, Anthony	999-98-0123	55
Nikolai, Peter	999-02-1009	2



Sun Educational Services

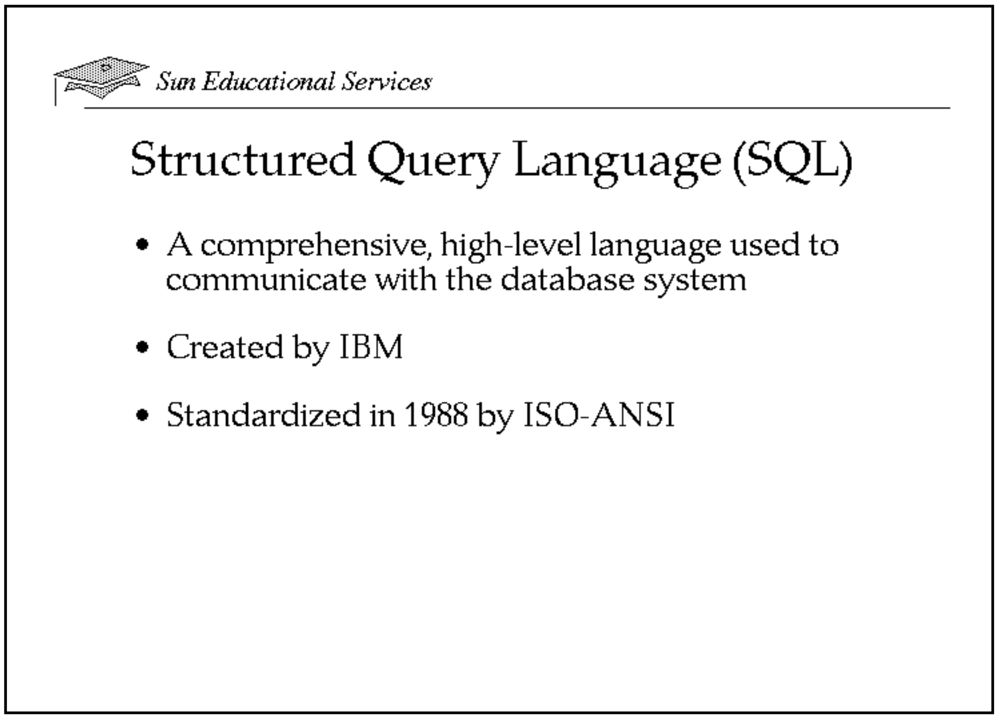
Data Independence

- User treats the database system as a logical representation of the data
- User does not need to know how the data is stored

Data Independence

The second of Codd's rules specifies that data must be stored logically and physically and that the two are maintained independent of each other. The underlying principle here is that the user does not need to know how the data are stored. The user treats the database system as a logical representation of the data.

Therefore, the underlying database storage mechanisms (hard disk, memory, and so on) are irrelevant to the user. In fact, as long as the tables are stored using the same logical representation, it should be possible to completely replace one physical database system with another.



Sun Educational Services

Structured Query Language (SQL)

- A comprehensive, high-level language used to communicate with the database system
- Created by IBM
- Standardized in 1988 by ISO-ANSI

Structured Query Language (SQL)

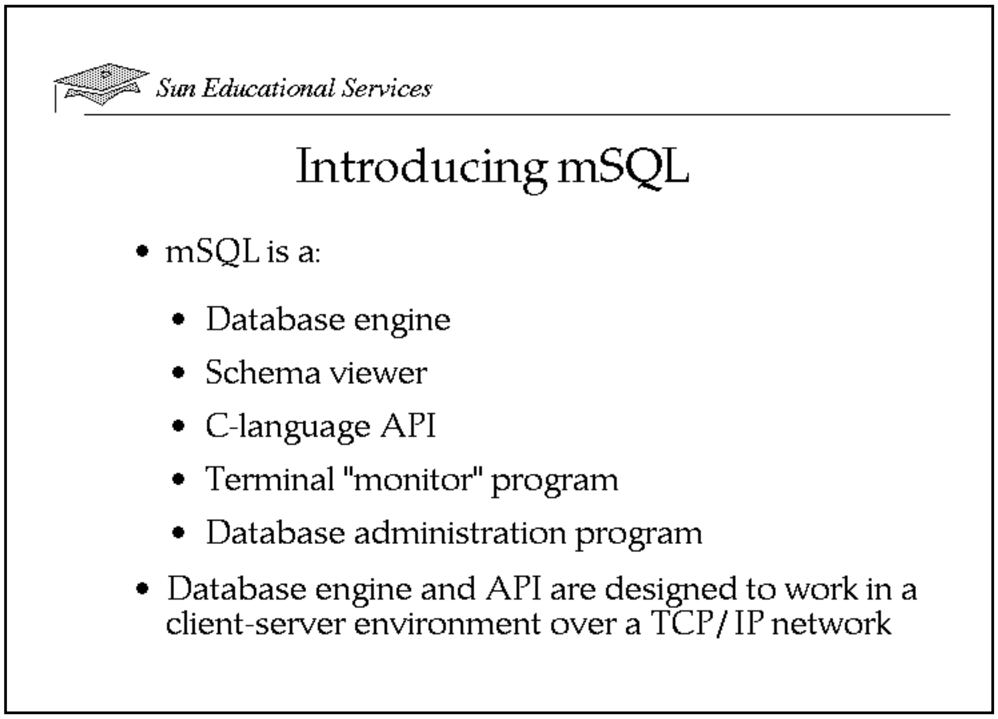
Codd's third rule states that there is a comprehensive, high-level language used to communicate with the database system. In the world of commercial database systems, SQL is the language of choice. Its name is officially pronounced "ess-cue-ell," but many people refer to the language as "sequel." You decide which one you want to use.

IBM created SQL between 1970–1980 and released its first SQL-based product, SQL/DS, in 1981. Several other vendors followed suit during the 1980s and, although the number of database companies has dwindled since then, the majority provide some variant of SQL in their product offerings.

SQL is lacking a true “standard” like the C and C++ programming languages that became popular quickly. Many of the early SQL adopters identified enhancements to the proposed 1988 ISO-ANSI (International Organization for Standardization – American National Standards Institute) standard, so it is difficult sometimes to know what is and what is not part of the true SQL specification.

This course avoids the feature richness of Sybase and Oracle databases and concentrates on core SQL. As it is, the SQL language enables you to manipulate data, define tables, and administer databases.

Rather than spend a lot of time looking at SQL query structure, the next few pages describe the database you will use in the lab exercises.



Sun Educational Services

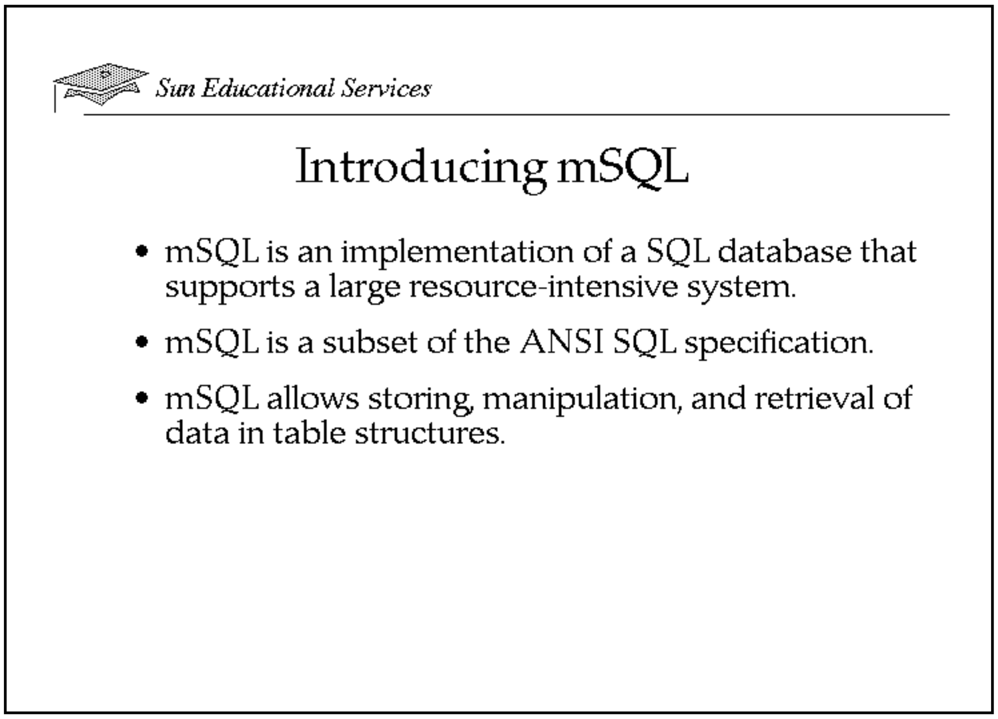
Introducing mSQL

- mSQL is a:
  - Database engine
  - Schema viewer
  - C-language API
  - Terminal "monitor" program
  - Database administration program
- Database engine and API are designed to work in a client-server environment over a TCP/IP network

Introducing mSQL

This course uses a SQL database called mini-SQL, or mSQL for short. mSQL is a licensed product created by David Hughes[2]. The mSQL package includes the database engine, a terminal "monitor" program, a database administration program, a schema viewer, and a C-language API. The API and the database engine have been designed to work in a client-server environment over a TCP/IP network.

2. Mini SQL is provided with the courtesy of Hughes Technologies Pty Ltd, Australia. Further information and an evaluation copy of Mini SQL can be found on the Hughes Technologies Web server at `http://www.Hughes.com.au`.


Sun Educational Services

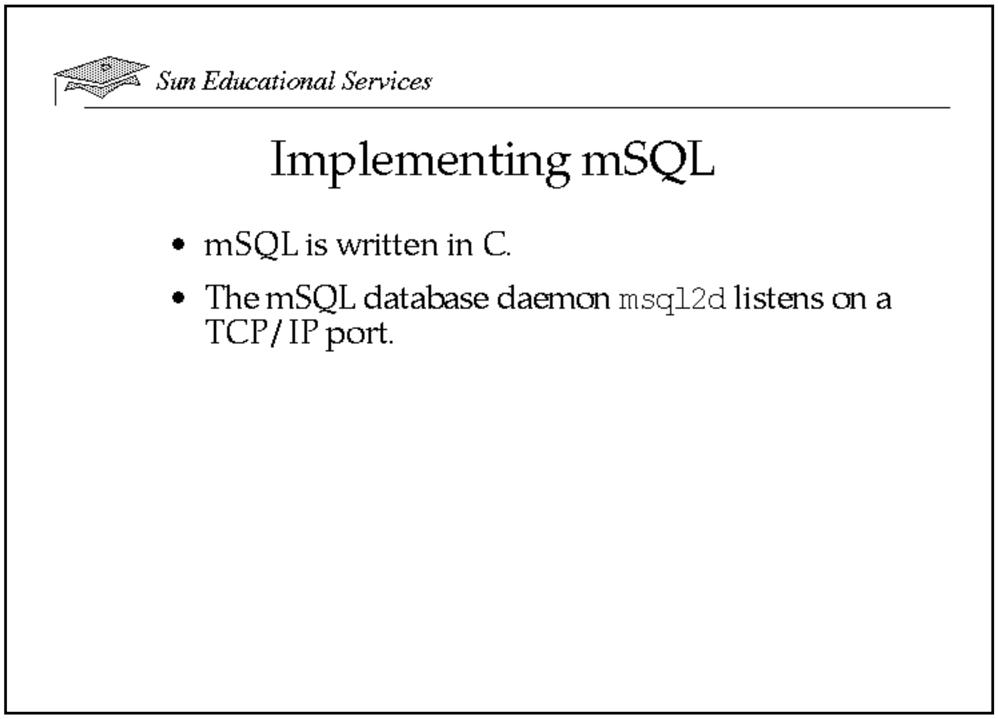
Introducing mSQL

- mSQL is an implementation of a SQL database that supports a large resource-intensive system.
- mSQL is a subset of the ANSI SQL specification.
- mSQL allows storing, manipulation, and retrieval of data in table structures.

Introducing mSQL

Mini SQL, or mSQL, as it is often called, is a light weight relational database management system. It has been designed to provide rapid access to data sets with as little system overhead as possible. The system itself is comprised of a database server and various tools that allow a user or a client application to communicate with the server.

mSQL enables a program or user to store, manipulate, and retrieve data in table structures. mSQL does not support all the relational operations defined in the ANSI SQL specification but it does provide the capability of "joins" between multiple tables.



Sun Educational Services

Implementing mSQL

- mSQL is written in C.
- The mSQL database daemon `msql2d` listens on a TCP/IP port.

Implementing mSQL

mSQL is written in native C and, for this class, has been compiled using the Solaris 7 Operating Environment. At the heart of the mSQL database is the `msql2d` daemon, which listens on a TCP/IP port. The following description is from the *Mini SQL 2.0 User Guide*:

> The philosophy of mSQL has been to provide a database management system capable of rapidly handling simple tasks. It has not been developed for use in critical financial environments (banking applications for example). The software is capable of performing the supported operations with exceptional speed whilst utilizing very few system resources. Some database systems require high-end hardware platforms and vast quantities of memory before they can provide rapid access to stored data. mSQL has been designed to provide exceptional data access performance on 'small hardware' platforms (such as PC class hardware). Because of these characteristics, mSQL is well suited to the vast majority of data management tasks.

mSQL Tools

The mSQL commercial package provides a rich toolset. From the Hughes Technologies web site, `http://www.Hughes.com.au`, you can download the mSQL manual, or view the online version of it. The following sections contain information from that manual.

`msql2d` – *mSQL Database Engine*

The database engine is called `msql2d` and it is written in C and compiled using the Solaris Operating Environment. The daemon is started on the server machine and expects to run as either root or a local user name. For the classroom use, the database is running as root; the instructor will let you know on which host.

`relshow` – *mSQL Schema Viewer*

The schema viewer is called `relshow` and it allows you to query the database for table names or for field names and field descriptions. You can start the schema viewer either locally or remotely.

```
relshow [-h hostname] database
relshow [-h hostname] database tablename
```

`msql` – *mSQL Terminal Monitor*

The terminal monitor is called `mSQL` and it provides a program to enable you to interact with the database while it is "live." The terminal emulator accepts all of the mSQL commands and you can run it in script mode where a preconfigured script of SQL statements is run against the database.

You can start the mSQL terminal monitor either locally or remotely.

```
msql [-h hostname] database
```

If the host name is not specified, `msql` reads the environment variable `MSQL_HOST;` if this variable is not set, `msql` uses the local host name.

`msql` accepts the following four runtime commands that have a back slash (\) prefix to distinguish them from SQL statements.

- `\g –` Go, execute the preceding SQL statement
- `\p` – Print, display the contents of the query buffer
- `\e –` Edit the last command in the default editor
- `\q` – Quit mSQL

mSQL Commands

The SQL standard specifies a set of commands and a specific syntax for the retrieval and modification of data, as well as commands for the administration of tables. Each SQL statement is issued to the database system and parsed. SQL statements begin with a command keyword. mSQL supports the following keywords:

- `SELECT` – Retrieves 0 or more records from the named table
- `INSERT` – Adds a new record to the named table
- `DELETE` – Removes one or more record(s) from the table
- `UPDATE` – Modifies one or more field(s) of particular record(s)
- `CREATE` – Builds a new table with the specified field names and types
- `DROP` – Completely removes a table from the database

SQL commands are meant to be read and spoken aloud; for example, "get me all of the fields in the table named employee data, where the employee ID is 10223." If the employee data table contained fields with the name, employee identifier (ID), date of hire, social security number, and current salary, you would expect to receive a single employee record with these values.

In mSQL, you could write this command using the following statement syntax:[3]

```
SELECT * FROM employee_data WHERE employee_id =
'10223'
```

3. mSQL statements are not case sensitive, but the examples shown highlight the keywords with capital letters.

The SELECT Statement

The SELECT statement is the primary command used for data retrieval from a SQL database. It supports the following:

- Joins
- DISTINCT row selection
- ORDER BY clauses
- Regular expression matching
- Column-to-column comparisons in WHERE clauses

The formal syntax for SELECT is the following:

```
SELECT [table.]column [ , [table.]column ]...
FROM table [ , table]...
[ WHERE [table.]column OPERATOR VALUE
[ AND | OR [table.]column OPERATOR VALUE]... ]
[ ORDER BY [table.]column [DESC] [, [table.]column
[DESC] ]
```

Where:

- OPERATOR can be <, >, =, <=, >=, <>, or LIKE
- VALUE can be a literal value or a column name

The regular expression syntax supported by LIKE clauses is the same as that in standard SQL.

- An underscore (_) matches any single character.
- A percent sign (%) matches 0 or more characters of any value.
- A back slash (\) escapes special characters (for example, \% matches % and \\ matches \).
- All other characters match themselves.

Examples

A basic inquiry would appear as follows:

```
SELECT first_name, last_name FROM emp_details
WHERE dept = 'finance'
```

To sort the returned data in ascending order by `last_name` and descending order by `first_name`, the query would look like the following:

```
SELECT first_name, last_name FROM emp_details
WHERE dept = 'finance'
ORDER BY last_name, first_name DESC
```

To remove any duplicate rows, you could use the following `DISTINCT` operator:

```
SELECT DISTINCT first_name, last_name FROM emp_details
WHERE dept = 'finance'
ORDER BY last_name, first_name DESC
```

To search for anyone in the finance department whose last name consists of a letter followed by "ughes", such as Hughes, the query could look like the following:

```
SELECT first_name, last_name FROM emp_details
WHERE dept = 'finance' AND last_name LIKE '_ughes'
```

Joins

The power of a relational query language is apparent when you start joining tables during a SELECT operation. For example, if you have two tables defined—one containing staff details and another listing the projects being worked on by each staff member—and each staff member has been assigned an employee number that is unique to that person, you can generate a sorted list of who was working on what project with the following query:

```
SELECT emp_details.first_name,
emp_details.last_name,
     project_details.project
FROM emp_details, project_details
WHERE emp_details.emp_id = project_details.emp_id
ORDER BY emp_details.last_name,
emp_details.first_name
```

mSQL places no restriction on the number of tables joined during a query; so if there are 15 tables, all containing information related to an employee ID in some manner, data from each of those tables can be extracted (albeit slowly) by a single query.

Note – You must qualify all column names with a table name. mSQL does not support the concept of uniquely named columns spanning multiple tables. You must qualify every column name as soon as you access more than one table in a single SELECT.

The INSERT *Statement*

The INSERT statement is used to add new SQL records to a table. You specify the names of the fields into which the data is to be inserted. You cannot specify the values without the field name and expect the server to insert the data into the correct fields by default. The syntax for the INSERT state is the following:

```
INSERT INTO table_name ( column [ , column ]... )
VALUES (value [, value]... )
```

For example:

```
INSERT INTO emp_details ( first_name, last_name, dept,
salary)
VALUES ('David', 'Hughes', 'I.T.S.','12345')
```

The number of values supplied must match the number of columns. However, the column names are optional if every column value is matched with an INSERT value.

The DELETE *Statement*

The DELETE statement is used to remove records from a SQL table. The syntax for the mSQL DELETE statement is the following:

```
DELETE FROM table_name
WHERE column OPERATOR value
[ AND | OR column OPERATOR value ]...
```

OPERATOR can be <, >, =, <=, >=, <>, or the keyword LIKE

For example:

```
DELETE FROM emp_details WHERE emp_id = ’12345’
```

The UPDATE *Statement*

The UPDATE statement is the SQL mechanism for changing the contents of a SQL record. To change a particular record, you must identify what record from the table you want to change. The mSQL

UPDATE statement cannot use a column name as a value. Only literal values can by used as an UPDATE value. The syntax for the UPDATE statement is the following:

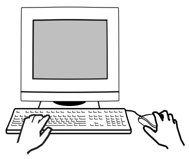
```
UPDATE table_name SET column=value [ , column=value
]...
WHERE column OPERATOR value
[ AND | OR column OPERATOR value ]...
```

OPERATOR can be <, >, =, <=, >=, <>, or LIKE.

For example:

```
UPDATE emp_details SET salary=30000 WHERE emp_id =
'1234'
```

Exercise: mSQL Database Queries



Exercise objective – Create and execute queries on an mSQL database.

Preparation

You should already have installed the mSQL tools to perform this lab exercise. To use the JDBC in the next module, become familiar with the operations of the mSQL database. You will be looking at the `StockMarket` database using SQL statements from the mSQL command-line interface.

Tasks

Complete the following steps:

1. Run `msql` against the `StockMarket` database. Your instructor will provide the name of the server. Display all of the information in the three tables. For example:

```
% msql -h server StockMarket
Welcome to the miniSQL monitor. Type \h for help.
mSQL > select * from Stock
     > \g
Query OK.
10 rows matched.
 +-----------------+--------------+
 | symbol          | price        |
 +-----------------+--------------+
 | SUNW            | 68.75        |
 | CyAs            | 22.625       |
 | DUKE            | 6.25         |
 | ABStk           | 18.5         |
 | JSVCo           | 9.125        |
 | TMAs            | 82.375       |
 | BWInc           | 11.375       |
 | GMEnt           | 44.625       |
 | PMLtd           | 203.375      |
 | JDK             | 33.5         |
 +-----------------+--------------+
mSQL >
```

2. Create and execute queries for each of the following:

Note – Do not delete or create any tables.

- Display all stocks with a price greater than 50.
- Add your own name, social security number, and address to the database.
- Display all customers alphabetically.
- Add two stocks to your portfolio, and verify your purchases by viewing the `Shares` table.
- Sell all shares of one of your stocks, and sell half of your shares of another stock.

Exercise Summary

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

- Experiences

- Interpretations

- Conclusions

- Applications

Check Your Progress

Before continuing on to the next module, check that you are able to accomplish or answer the following:

- ❑ Describe relational database design
- ❑ Explain Codd's first three rules of relational database design
- ❑ Construct mSQL queries
- ❑ Create a connection to an mSQL database
- ❑ Extract information from an mSQL database

Think Beyond

SQL, has evolved over time due to its standardization of the syntax of “how to talk to” a database.

What parallels exist between what SQL means to database programming and what the evolution and standardization of the Java programming language means to programming in general?

Introduction to JDBC 3

Objectives

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

- Describe JDBC
- Explain how using the abstraction layer provided by JDBC can make a database front-end portable across platforms
- Describe the five major tasks involved with the JDBC programmer’s interface
- State the requirements of a JDBC driver and its relationship to the JDBC driver manager

This module covers the basics of JDBC.

Relevance

Discussion – Attaching to a database with a provided tool (as was done with mSQL in the last module) is fine. However, you need to attach to databases from within code. Without a layer of database connectivity, you would have to learn a lot about the database's API and invoke API calls from each and every manufacturer's API libraries using the Java Native Interface (JNI) to do so.

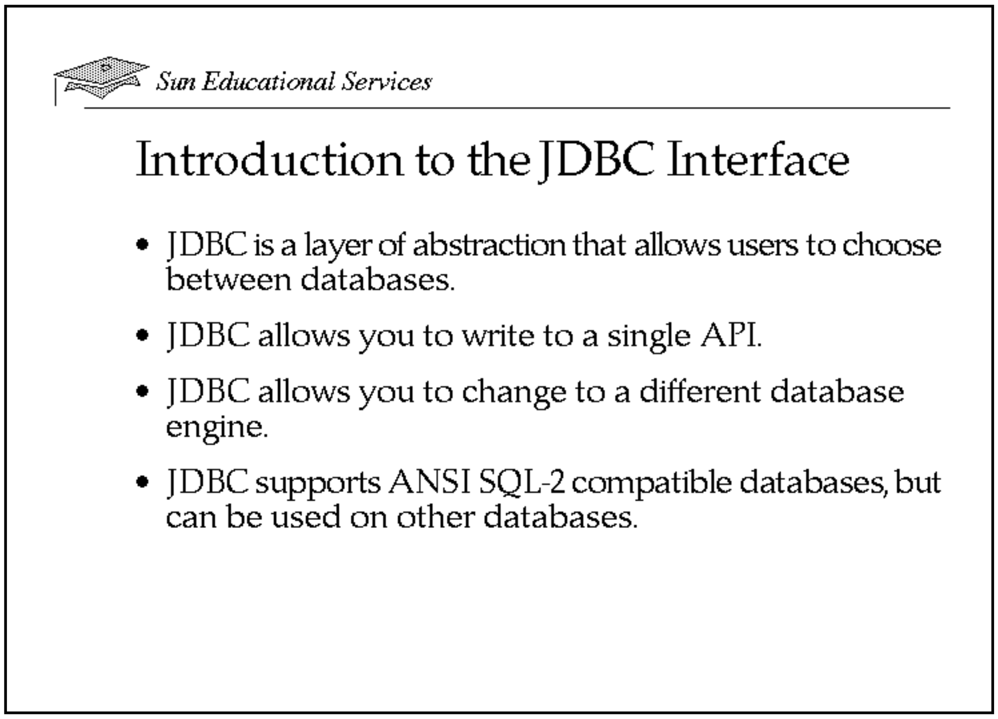
In the Java programming language, there is a well-defined specification for multiple levels of adherence to a specification called the JDBC layer. This layer simplifies what you need to connect to and identifies the information you need from a database.

What difficulties would arise from calling a manufacturer's API libraries directly when you switch from one major vendor's database to another?

Additional Resources

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

- JDBC specification. Available:
  `http://splash.javasoft.com/jdbc`
- Emerson, Darnovsky, and Bowman. 1989. *The Practical SQL Handbook.* Addison-Wesley.


Sun Educational Services

Introduction to the JDBC Interface

- JDBC is a layer of abstraction that allows users to choose between databases.
- JDBC allows you to write to a single API.
- JDBC allows you to change to a different database engine.
- JDBC supports ANSI SQL-2 compatible databases, but can be used on other databases.

Introducing the JDBC Interface

When you use a database, you have a number of choices about the database engine on which to build. Should you build on an SQL database, an Oracle database, or a Sybase database?

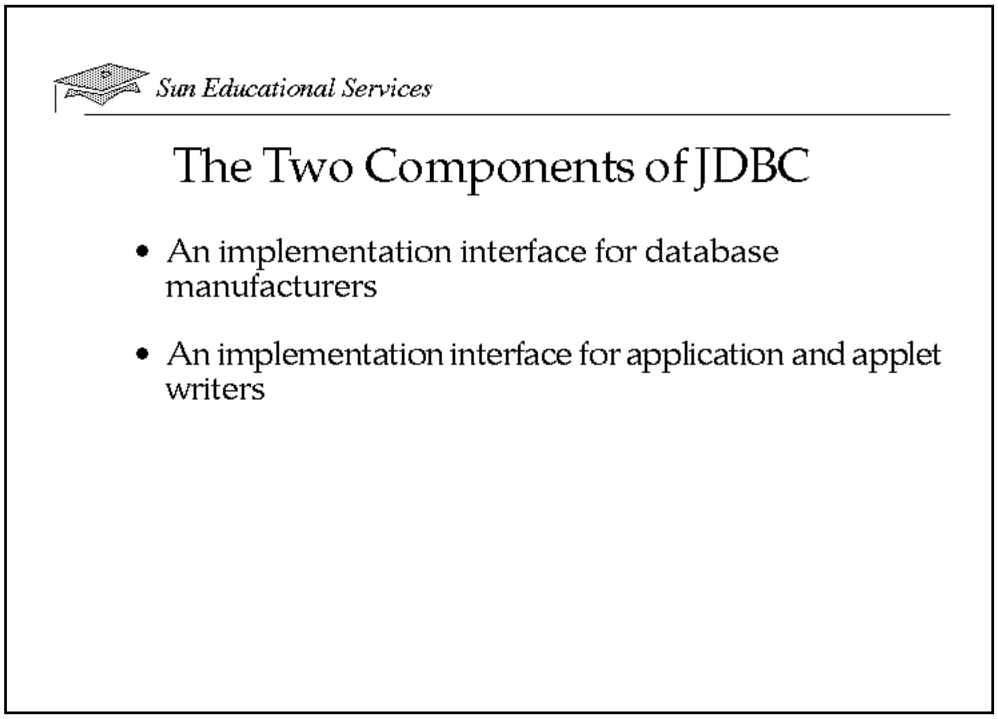
By introducing a layer of abstraction, JDBC leaves the choice of using SQL and a certain vendor up to the integrator. JDBC takes this freedom of choice a step further and enables you to write a single API that takes care of the necessary communication between your front end and the database's back end.

If you subsequently decide to change the back end to use another database engine, you can easily substitute this alternative database engine, provided a JDBC-compliant driver has been created for that database engine.

ANSI SQL-2 Conformance

Database systems support a wide range of SQL syntax and semantics. While they can vary on more advanced functionality, such as outer joins, they share common support for ANSI SQL-2. Because you can write Java applications to use only statements compliant with this ANSI standard, these applications are portable across the various databases.

However, this should not be construed to say that JDBC only supports ANSI SQL-2. JDBC enables any query string to be passed through, so an application can use as much SQL functionality as desired, at the risk of receiving an error on other databases.


Sun Educational Services

The Two Components of JDBC

- An implementation interface for database manufacturers
- An implementation interface for application and applet writers

The Two Components of JDBC

There are two major components of JDBC: an implementation interface for database manufacturers, and an interface for application and applet writers. The following sections describe JDBC from the perspective of a vendor writing a JDBC driver, and then cover in greater detail the steps involved in writing a Java application using a vendor's JDBC driver interface.

JDBC Driver Interface

The JDBC Driver interface provides vendor-specific implementations of the abstract classes provided by the JDBC API. Each vendor's driver must provide implementations of the following:

- `java.sql.Connection`
- `java.sql.Statement`
- `java.sql.PreparedStatement`
- `java.sql.CallableStatement`
- `java.sql.ResultSet`
- `java.sql.Driver`

Each database driver must provide a class that implements the `java.sql.Driver` interface used by the generic `java.sql.DriverManager` class when it needs to locate a driver for a particular database using a uniform resource locator (URL) string. JDBC is patterned after ODBC (open database connectivity); this makes providing a JDBC implementation on top of ODBC easy and efficient.

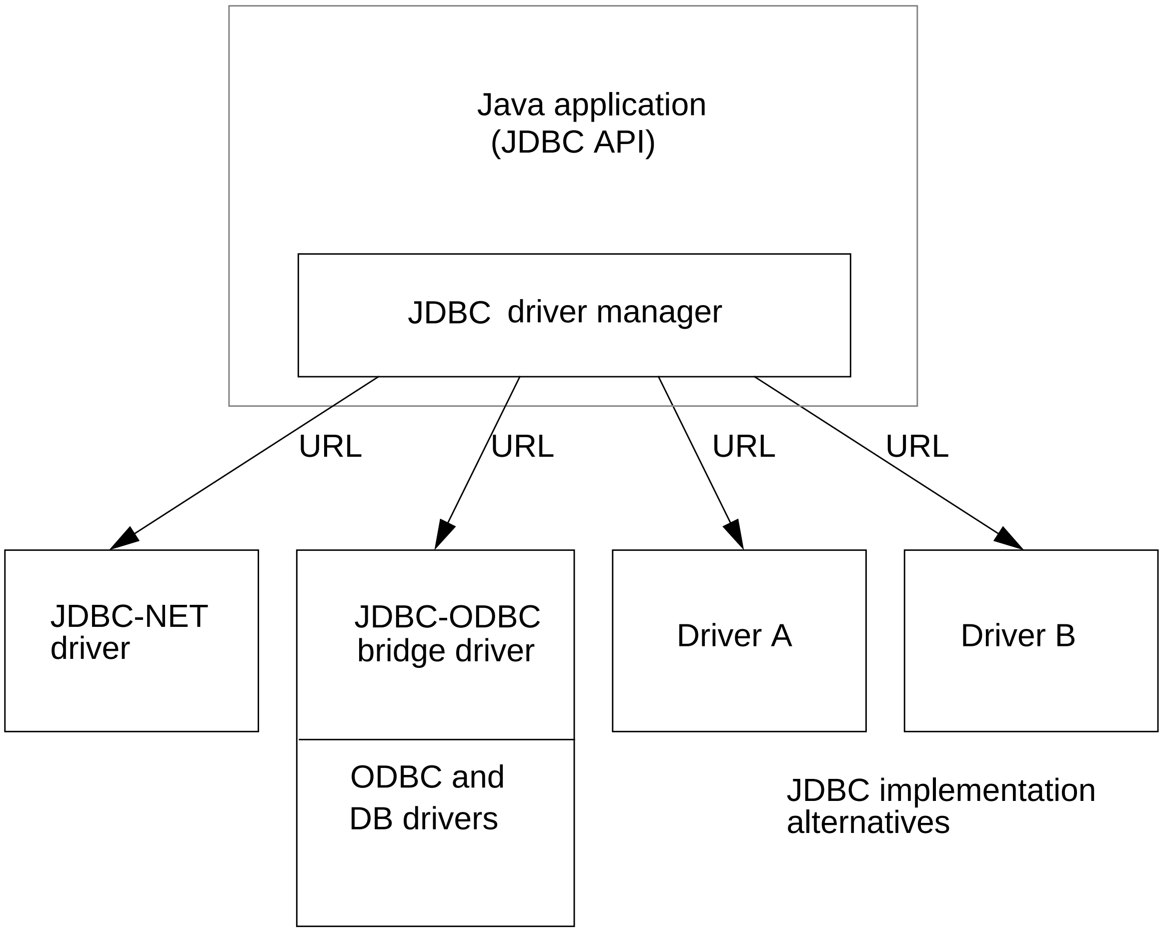
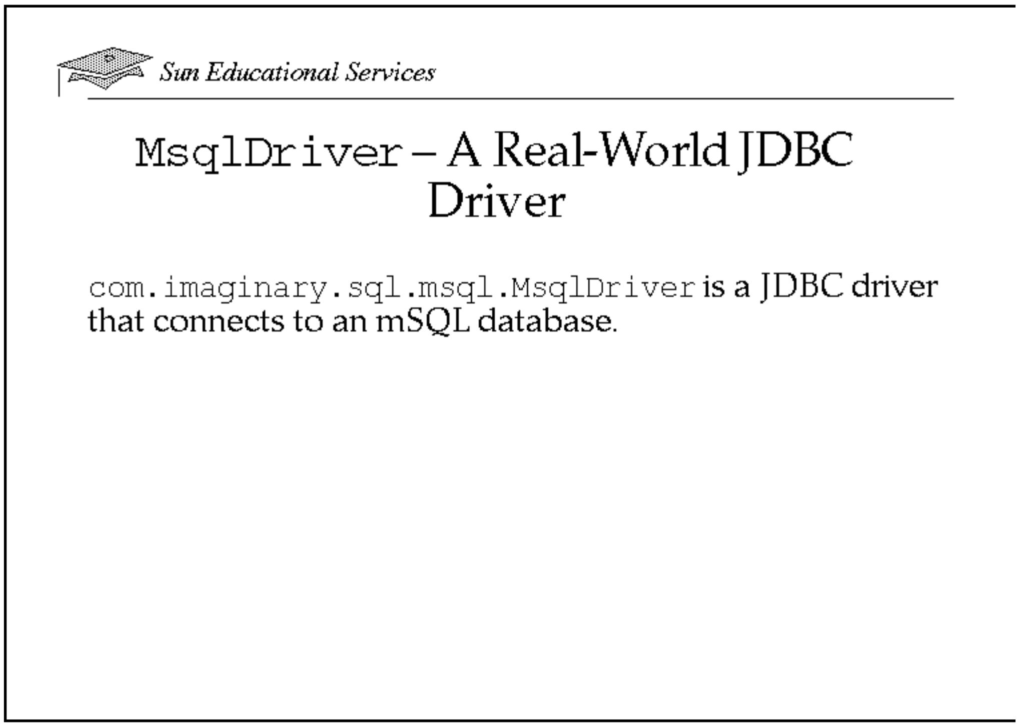


Figure 3-1 JDBC Drivers

Figure 3-1 illustrates how a single Java application (or applet) can access multiple database systems through one or more drivers.



Sun Educational Services

`MsqlDriver` – A Real-World JDBC Driver

`com.imaginary.sql.msql.MsqlDriver` is a JDBC driver that connects to an mSQL database.

`MsqlDriver` – *A Real-World JDBC Driver*

JDBC allows you to write database applications in Java without having to concern yourself with the underlying details of a particular database. All it requires is a driver that passes the appropriate JDBC API calls to the database engine. The JDBC driver insulates you from the specifics of the database. As a programmer, you can write generic JDBC API calls that work with any JDBC-compliant database, as long as the appropriate JDBC driver is present that interprets your JDBC API calls.

Mini SQL (also called mSQL) is a type of SQL database created by Hughes Technologies. The Center for Imaginary Environments (CIE, http://www.imaginary.com/), created a JDBC driver for mSQL databases. Their JDBC driver, MsqlDriver, allows Java applications and applets to access mSQL databases. It uses the mSQL-JDBC API, a database access API for mSQL databases that conforms to the Sun Microsystems JDBC access API.

The MsqlDriver file comes packaged in a .jar file called msql-jdbcxxx.jar. To make this driver file available to your Java runtime environment, copy it to a location that is included in your class path.

In this class, you use `com.imaginary.sql.msql.MsqlDriver`[1]; a JDBC driver written to connect to a mSQL database.

1. mSQL-JDBC API is provided courtesy of George Reese (borg@imaginary.com). For more information and latest developments, consult George's home page at `http://www.imaginary.com/~borg`.

Connecting Through the JDBC Interface

JDBC Programming Tasks

This section covers some of the common tasks you perform with JDBC.

- Create an instance of a JDBC driver or load JDBC drivers through `jdbc.drivers`
- Register a driver
- Specify a database
- Open a database connection
- Submit a query
- Receive results

Again, the following sections assume that you communicate with the `MsqlDriver` class using the JDBC API.

The `java.sql` Package

The eight interfaces associated with the JDBC are the following:

- `java.sql.Driver`
- `java.sql.Connection`
- `java.sql.Statement`
- `java.sql.PreparedStatement`
- `java.sql.CallableStatement`
- `java.sql.ResultSet`
- `java.sql.ResultSetMetaData`
- `java.sql.DatabaseMetaData`

The JDBC Flow

Each of these interfaces enables an application programmer to open connections to particular databases, execute SQL statements, and process the results.

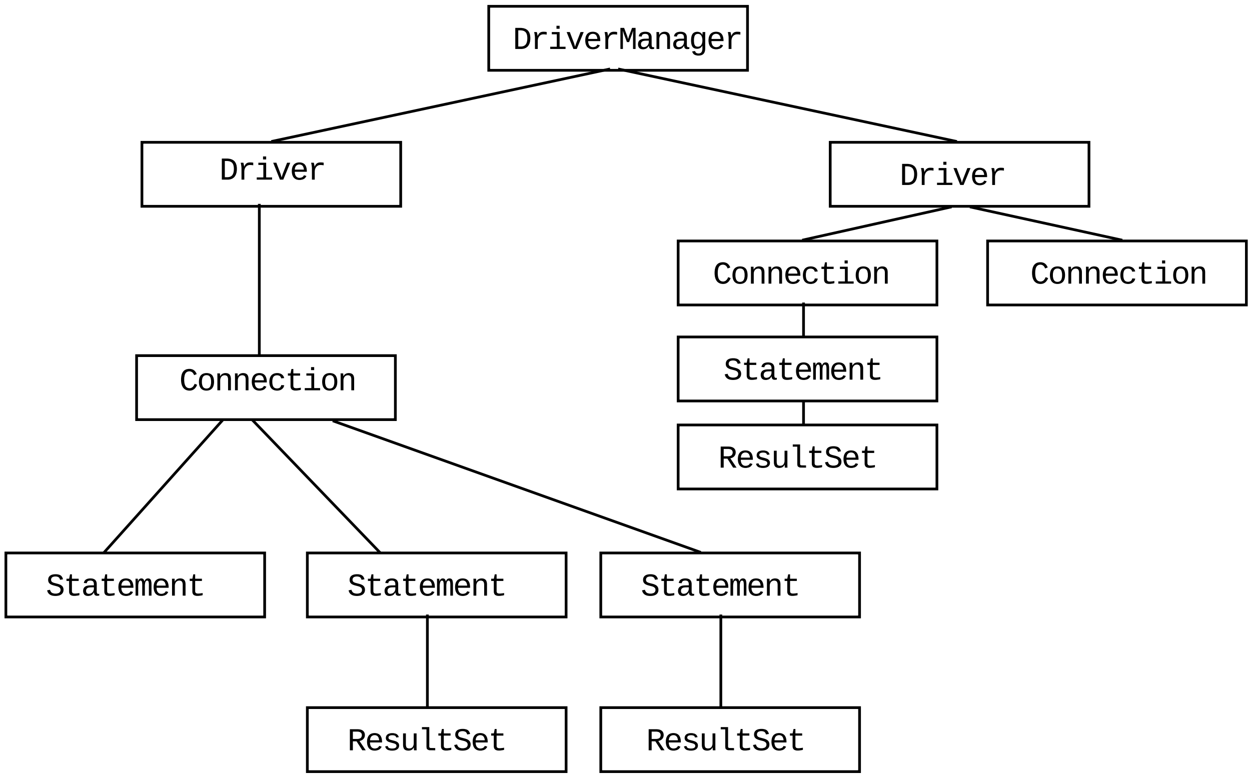


Figure 3-2 JDBC Flowchart

As illustrated by Figure 3-2:

- A URL string is passed to the `getConnection` method of the `DriverManager`, which in turn locates a `Driver`.
- With a `Driver`, you can obtain a `Connection`.
- With the `Connection`, you can create a `Statement`.
- When a `Statement` is executed with an `executeQuery` method, a `ResultSet` can be returned.

JDBC Example

The following is a simple example that uses the mSQL database in the lab. This example uses the elements of a JDBC application: creating a `Driver` instance, getting a `Connection` object, creating a `Statement` object, executing a query, and processing the returning `ResultSet` object.

Note – The source code for the following example can be found and compiled in `/SL285_XXX_LF/labfiles/complete2tier`.

```
import java.sql.*;
import com.imaginary.sql.msql.*;

public class JDBCExample {

   public static void main (String args[]) {

      if (args.length < 1) {
         System.err.println ("Usage:");
         System.err.println (" java JDBCExample <db server hostname>");
         System.exit (1);
      }
      String serverName = args[0];
      try {
         // Create the instance of the Msql Driver
         new MsqlDriver ();

         // Create the "url"
         String url = "jdbc:msql://" + serverName +
            ":1112/StockMarket";

         // Use the DriverManager to get a Connection
         Connection mSQLcon =
            DriverManager.getConnection (url);

         // Use the Connection to create a Statement object
         Statement stmt = mSQLcon.createStatement ();

         // Execute a query using the Statement and return a ResultSet
         ResultSet rs = stmt.executeQuery
            ("SELECT * FROM Customer ORDER BY ssn");
```

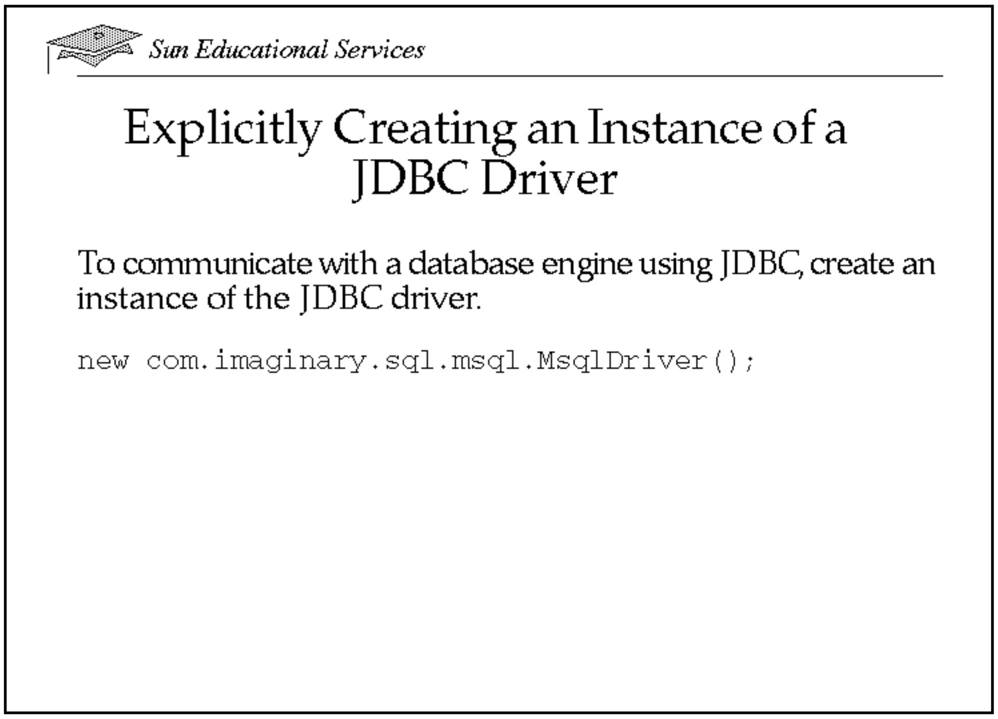
```
        // Print the results, row by row
        while (rs.next()) {
           System.out.println ("");
           System.out.println ("Customer: " + rs.getString(2));
           System.out.println ("Id:       " + rs.getString(1));
        }

     } catch (SQLException e) {
        e.printStackTrace();
     }
   }
}
```

If you run the code

`java JDBCExample <`*`server name`*`>`

the contents of the `Customer` table in the `StockMarket` database are similar to the following:

```
Customer: Tom McGinn
Id:       999-11-2222

Customer: Jennifer Sullivan Volpe
Id:       999-22-3333

Customer: Georgianna DG Meagher
Id:       999-33-4444

Customer: Priscilla Malcolm
Id:       999-44-5555
```



Sun Educational Services

Explicitly Creating an Instance of a JDBC Driver

To communicate with a database engine using JDBC, create an instance of the JDBC driver.

```
new com.imaginary.sql.msql.MsqlDriver();
```

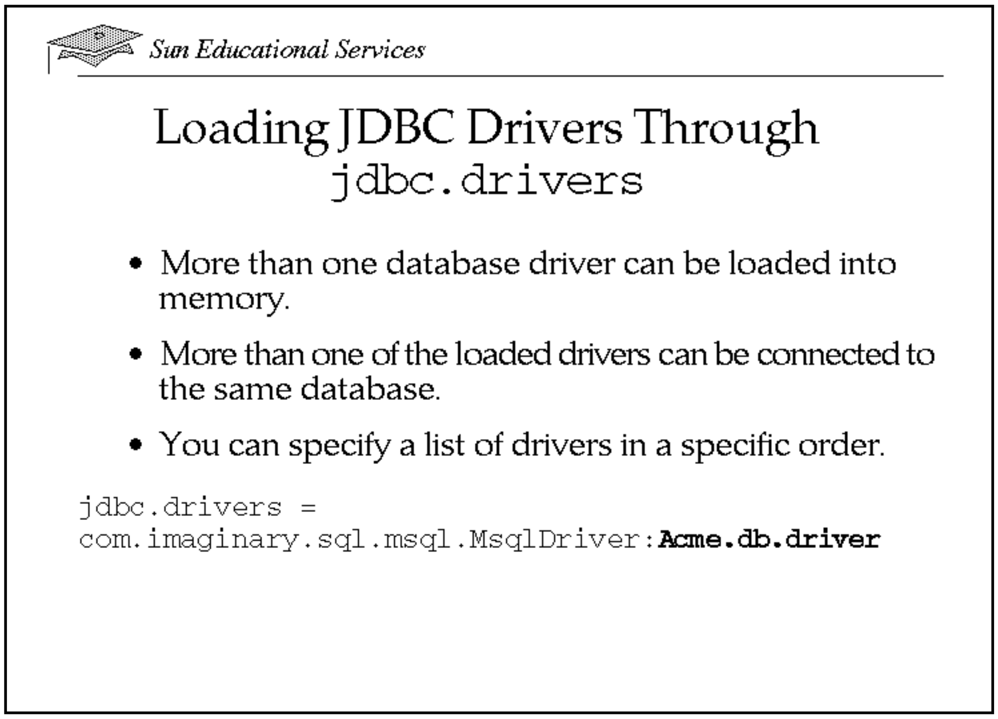
Connecting Through the JDBC Interface

Explicitly Creating an Instance of a JDBC Driver

To communicate with a particular database engine using JDBC, you must first create an instance of the JDBC driver. This driver remains behind the scenes, handling any requests for that type of database.

```
// Create an instance of Msql’s JDBC Driver
new com.imaginary.sql.msql.MsqlDriver();
```

You do not have to associate this driver with a variable, the driver exists after it is instantiated and successfully loaded into memory.


Sun Educational Services

Loading JDBC Drivers Through `jdbc.drivers`

- More than one database driver can be loaded into memory.
- More than one of the loaded drivers can be connected to the same database.
- You can specify a list of drivers in a specific order.

```
jdbc.drivers =
com.imaginary.sql.msql.MsqlDriver:Acme.db.driver
```

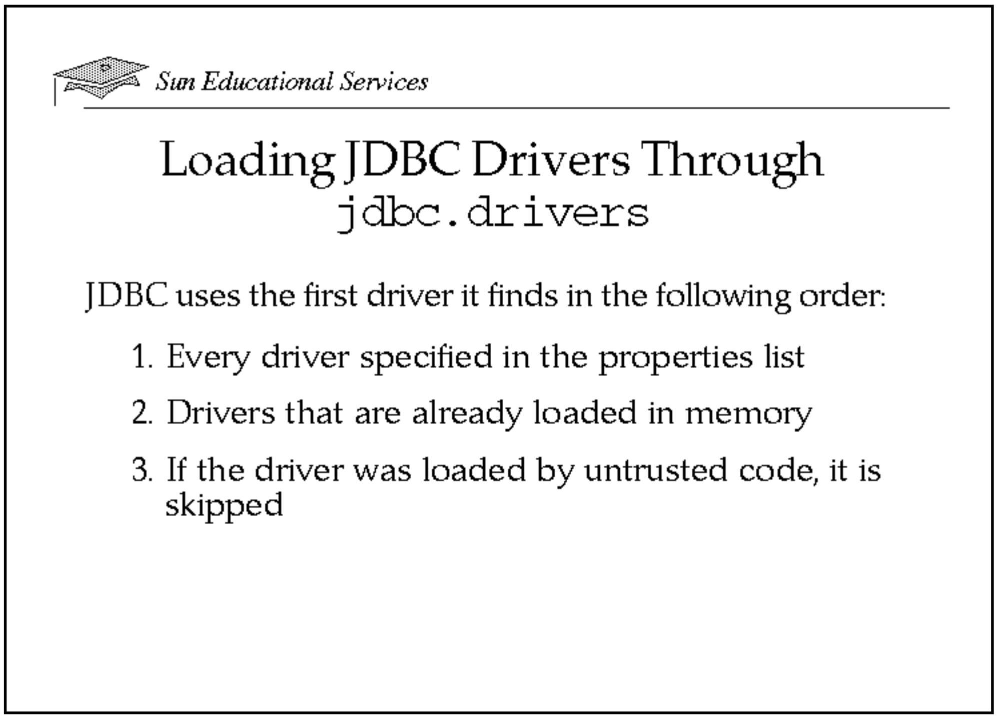
Loading JDBC Drivers Through `jdbc.drivers`

Sometimes more than one database driver is loaded into memory or more than one driver is loaded into memory (either ODBC or a JDBC generic network protocol) that is capable of connecting to the same database. If this is the case, JDBC allows you to specify a list of drivers in a specific order. The order of selection is specified by the `jdbc.drivers` properties tag. The `jdbc.drivers` property should be defined as a colon-separated list of driver class names.

```
jdbc.drivers=com.imaginary.sql.msql.MsqlDriver:Acme.db.driver
```

Properties are set through the `-D` option to the `java` interpreter (or the `-J` option to the `appletviewer` application). For example:

```
java -Djdbc.drivers=com.imaginary.sql.msql.MsqlDriver:Acme.db.driver
myApp
```



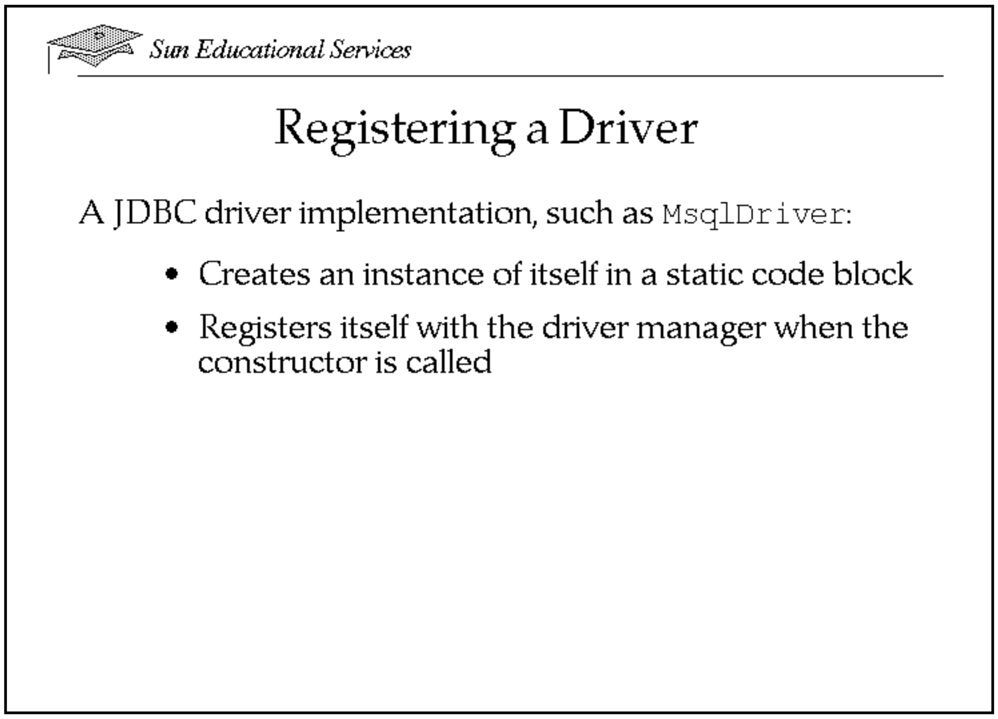
Sun Educational Services

Loading JDBC Drivers Through `jdbc.drivers`

JDBC uses the first driver it finds in the following order:

1. Every driver specified in the properties list
2. Drivers that are already loaded in memory
3. If the driver was loaded by untrusted code, it is skipped

When attempting to connect to a database, JDBC uses the first driver it finds that can successfully connect to the given URL. It first tries each driver specified in the properties list, in order from left to right. It then tries any drivers that are already loaded in memory, in the order that the drivers were loaded. If the driver was loaded by untrusted code, it is skipped, unless it has been loaded from the same source as the code that is trying to open the connection.



Sun Educational Services

Registering a Driver

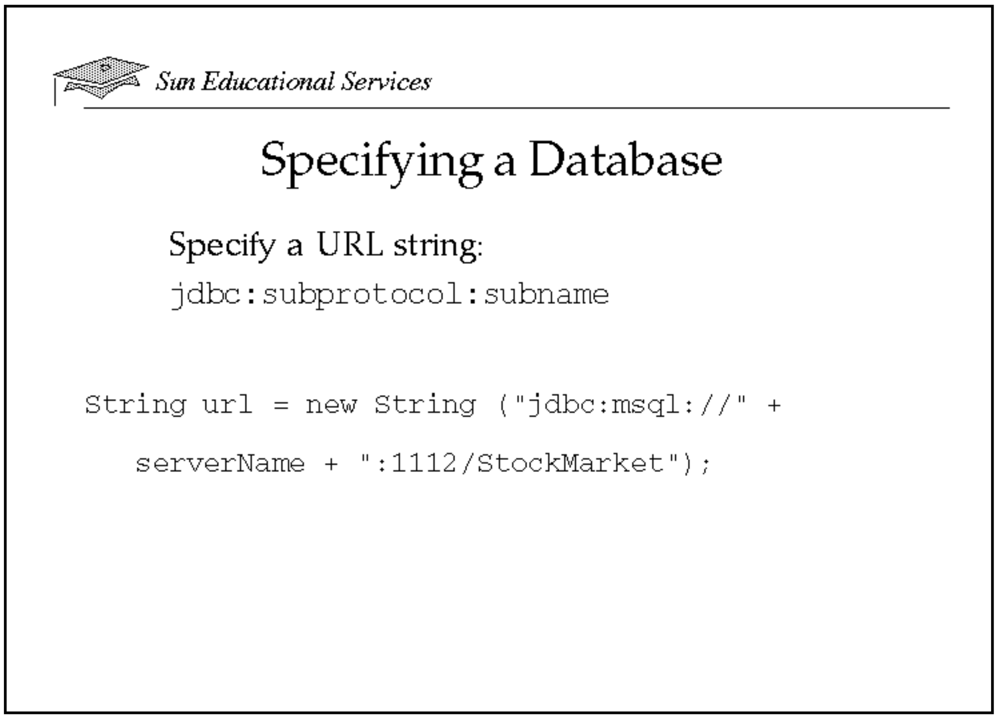
A JDBC driver implementation, such as `MsqlDriver`:

- Creates an instance of itself in a static code block
- Registers itself with the driver manager when the constructor is called

Registering a Driver

When a driver is loaded, it is the responsibility of the driver implementation to register itself with the driver manager. For example, the mSQL driver `com.imaginary.sql.msql.MsqlDriver`:

- Creates an instance of itself in a static code block
- When the constructor is called either explicitly or implicitly, registers itself with the driver manager



Sun Educational Services

Specifying a Database

Specify a URL string:

jdbc:subprotocol:subname

```
String url = new String ("jdbc:msql://" +
   serverName + ":1112/StockMarket");
```

Specifying a Database

Now that you have created the instance of the JDBC driver, you need some way to specify the database to which you want to connect. To do this in JDBC, specify a URL string that indicates the database type. The proposed URL syntax for a JDBC database is:

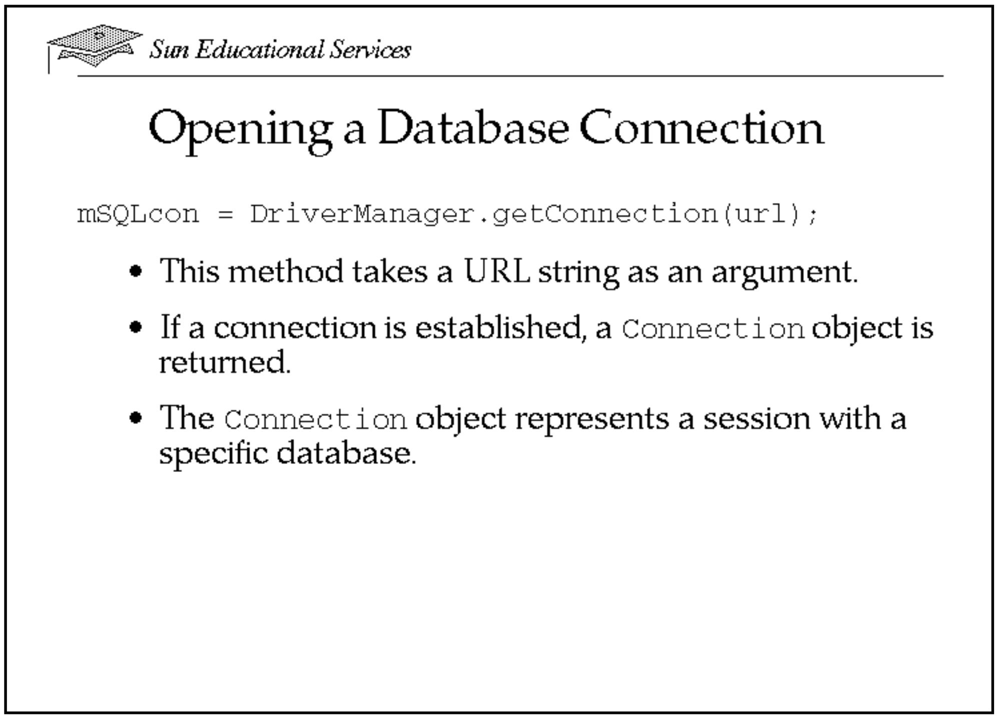
```
jdbc:subprotocol:subname
```

This is not a `java.net.URL`, but a `java.lang.String` in URL format, where `subprotocol` names a particular kind of database connectivity mechanism supported by one or more drivers. The contents and syntax of the `subname` depend on the `subprotocol`.

```
// Construct the URL for JDBC access
String url = new String ("jdbc:msql://" +
   serverName + ":1112/StockMarket");
```

This is the URL for JDBC access to the mSQL `StockMarket` database you have been using in the classroom. It could have been any other type of protocol that is accessed through a JDBC-ODBC bridge

```
    jdbc:odbc:Object.StockMarket
```



Sun Educational Services

Opening a Database Connection

```
mSQLcon = DriverManager.getConnection(url);
```

- This method takes a URL string as an argument.
- If a connection is established, a `Connection` object is returned.
- The `Connection` object represents a session with a specific database.

Opening a Database Connection

Now that you have created a URL specifying `msql` as the database engine, you are ready to make a database connection. To do this, you obtain a `java.sql.Connection` object by calling the JDBC driver's `java.sql.DriverManager.getConnection` method.

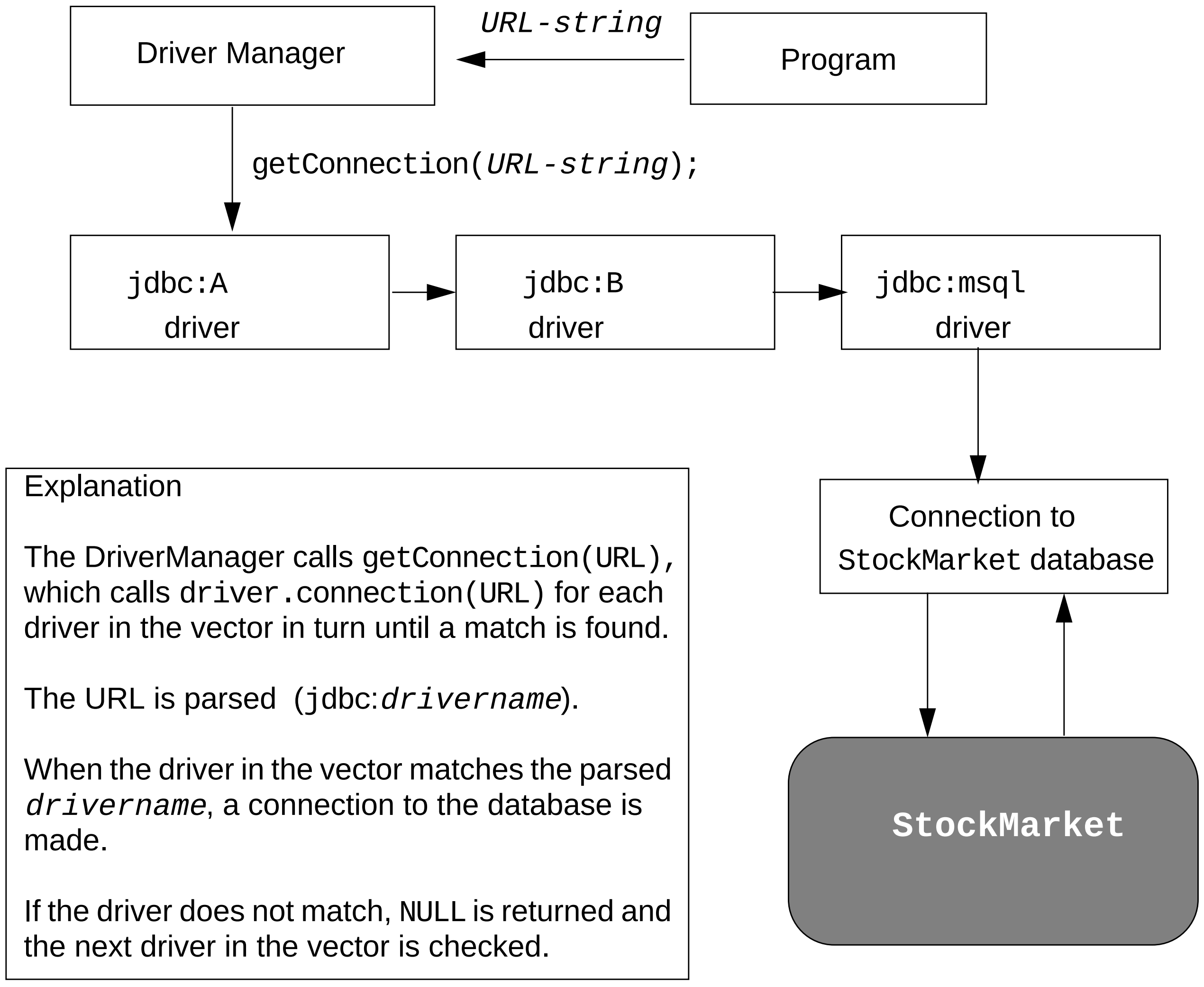
```
// Establish a database connection through the msql
// DriverManager
mSQLcon = DriverManager.getConnection(url);
```

The `DriverManager.getConnection` method takes a URL string as an argument. The JDBC driver management layer attempts to locate a driver that can connect to the database represented by the URL. If a driver succeeds in establishing a connection, it returns an appropriate `java.sql.Connection` object.

`mSQLcon` is defined as type `Connection` earlier in the code. The `Connection` represents a session with a specific database and provides methods that enable you to obtain `java.sql.Statement` and `java.sql.PreparedStatement` objects.

Using Database Resolution

Figure 3-3 illustrates how a `DriverManager` resolves a URL string passed by the `getConnection` method. A program can load more than one driver, so each registered driver gets stored in a vector. This vector is traversed in the order in which the drivers were loaded. When the driver returns a null, the driver manager continues to call the next registered driver in turn until either the list is exhausted or a `Connection` object is returned.



Explanation

The DriverManager calls `getConnection(URL)`, which calls `driver.connection(URL)` for each driver in the vector in turn until a match is found.

The URL is parsed (`jdbc:`*`drivername`*).

When the driver in the vector matches the parsed *`drivername`*, a connection to the database is made.

If the driver does not match, `NULL` is returned and the next driver in the vector is checked.

Figure 3-3 Example of Database Resolution

JDBC Statements

Submitting a Query

To submit a standard query, get a `Statement` object from the `Connection.createStatement` method.

```
   // Create a Statement object
try {
    stmt = mSQLcon.createStatement();
} catch (SQLException e) {
    System.out.println (e.getMessage());
}
```

Use the `Statement.executeUpdate` method to submit an `INSERT`, `UPDATE`, or `DELETE` statement to the database. JDBC passes the SQL statement to the underlying database connection unaltered, it does not attempt to interpret queries.

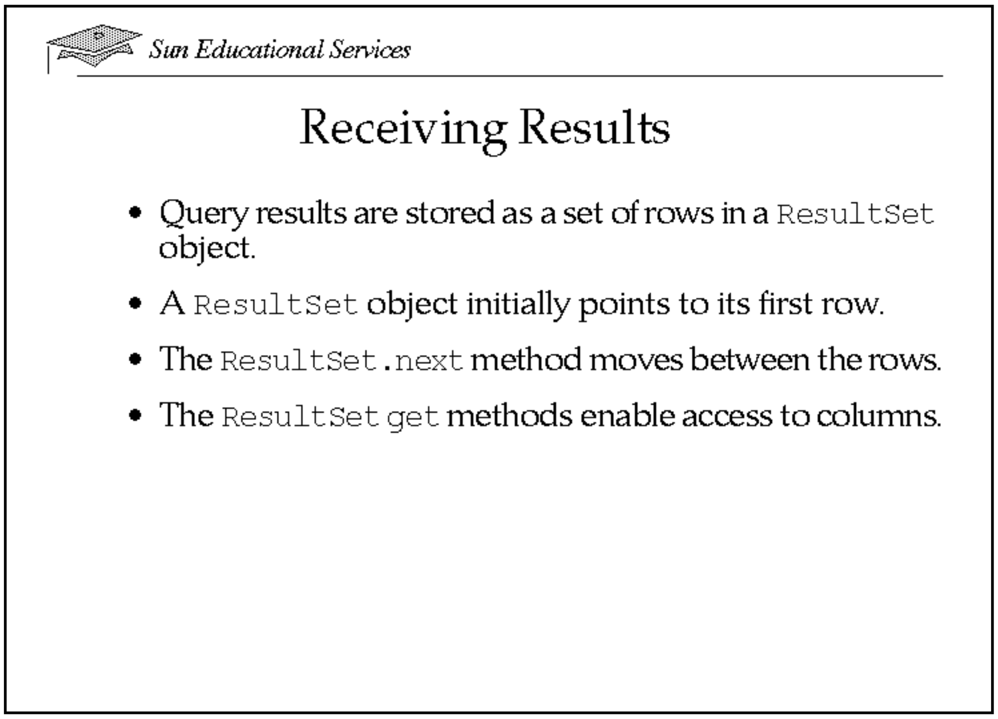
```
   // Pass a query via the Statement object
int count = stmt.executeUpdate("DELETE from
    Customer WHERE ssn='999-55-6666'");
```

The `Statement.executeUpdate` method returns an `int`, representing the number of rows affected by the `INSERT`, `UPDATE`, or `DELETE` statements.

Use the `Statement.executeQuery` method to submit a `SELECT` statement to the database.

```
   // Pass a query via the Statement object
ResultSet rs = stmt.executeQuery("SELECT * from
    Customer order by ssn");
```

The `Statement.executeQuery` method returns a `ResultSet` object for processing.



Sun Educational Services

Receiving Results

- Query results are stored as a set of rows in a ResultSet object.
- A ResultSet object initially points to its first row.
- The ResultSet.next method moves between the rows.
- The ResultSet get methods enable access to columns.

Receiving Results

The result of executing a query statement is a set of rows that are accessible using a `java.sql.ResultSet` object. The rows are received in order. A `ResultSet` object keeps a cursor pointing to the current row of data and is initially positioned before its first row. You can use the `ResultSet.next` method to move between the rows of the `ResultSet` object. The first call to `next` makes the first row the current row, the second call makes the second row the current row, and so on.

The `ResultSet` object provides a set of `get` methods that enable access to the various columns of the current row.

```
while (rs.next()) {
 System.out.println ("Customer: " + rs.getString(2));
 System.out.println ("Id:       " + rs.getString(1));
 System.out.println ("");
}
```

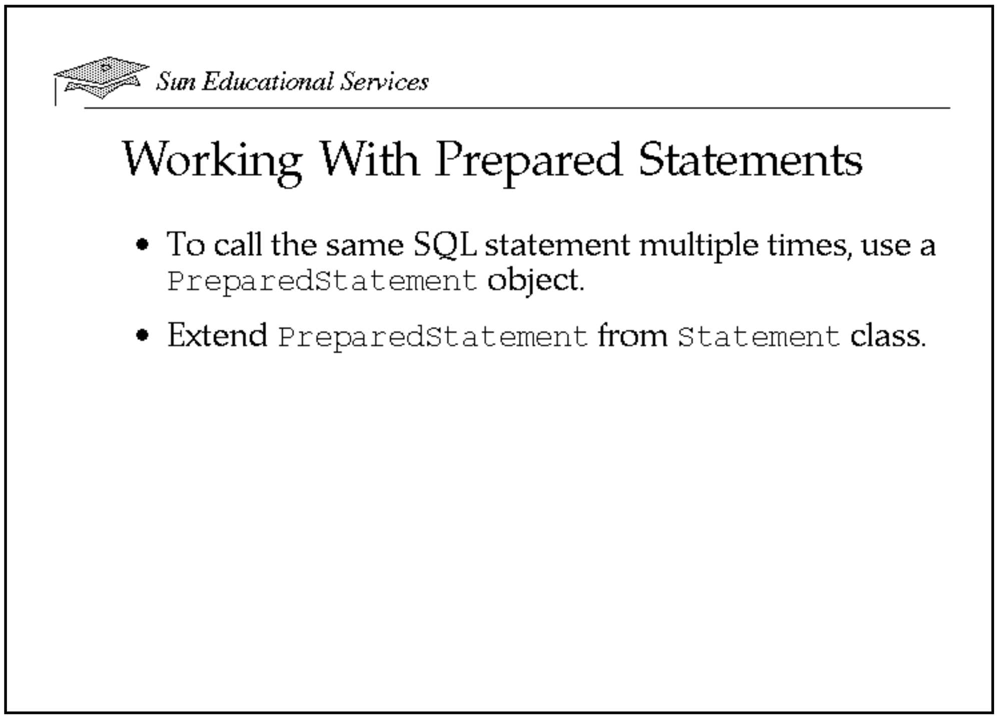
The various `getXXX` methods can take either a column name or an index as their argument. It is a good idea to use an index when referencing a column. Column indexes start at 1. When using a name to reference a column, more than one column can have the same name, thus causing a conflict. Within a given row, fields can be accessed in random order.

To retrieve data from the `ResultSet` object, you must be familiar with the columns returned and their data types. Tables 3-1 and 3-2 show the mapping between Java and SQL datatypes.

Using the `get`*`XXX`* *Methods*

Table 3-1 `get`*`XXX`* Methods and the Java Type Returned

Method	**Java Type Returned**
`getASCIIStream`	`java.io.InputStream`
`getBigDecimal`	`java.math.BigDecimal`
`getBinaryStream`	`java.io.InputStream`
`getBoolean`	`boolean`
`getByte`	`byte`
`getBytes`	`byte[]`
`getDate`	`java.sql.Date`
`getDouble`	`double`
`getFloat`	`float`
`getInt`	`int`
`getLong`	`long`
`getObject`	`Object`
`getShort`	`short`
`getString`	`java.lang.String`
`getTime`	`java.sql.Time`
`getTimestamp`	`java.sql.Timestamp`
`getUnicodeStream`	`java.io.InputStream` of Unicode characters



Sun Educational Services

Working With Prepared Statements

- To call the same SQL statement multiple times, use a `PreparedStatement` object.
- Extend `PreparedStatement` from `Statement` class.

Working With Prepared Statements

If the same SQL statements are going to be executed multiple times, it is advantageous to use a `PreparedStatement` object. A prepared statement is a precompiled SQL statement that is more efficient than calling the same SQL statement over and over. The `PreparedStatement` class extends the `Statement` class to add the capability of setting parameters inside of a statement.

When declaring the `PreparedStatement` object, use the question mark (`?`) character as a placeholder for the incoming parameter. When passing the parameter to the statement, indicate which placeholder you are referencing by its sequential position in the statement.

An Example of Using a Prepared Statement

An example of using a PreparedStatement object is shown in the following code:

```
Connection conn = DriverManager.getConnection(url);
.
.
java.sql.PreparedStatement stmt =
   conn.prepareStatement
      ("UPDATE table3 set m = ? WHERE x = ?");

// We pass two parameters. One varies each time around
// the for loop, the other remains constant.
stmt.setString(1, "Hi");
for (int i = 0; i < 10; i++) {
   stmt.setInt(2, i);
   int j = stmt.executeUpdate();
   System.out.println(j +" rows affected when i=" +i);
}
```

The `PreparedStatement` object has an `executeQuery` method as well, which returns a `ResultSet` object.

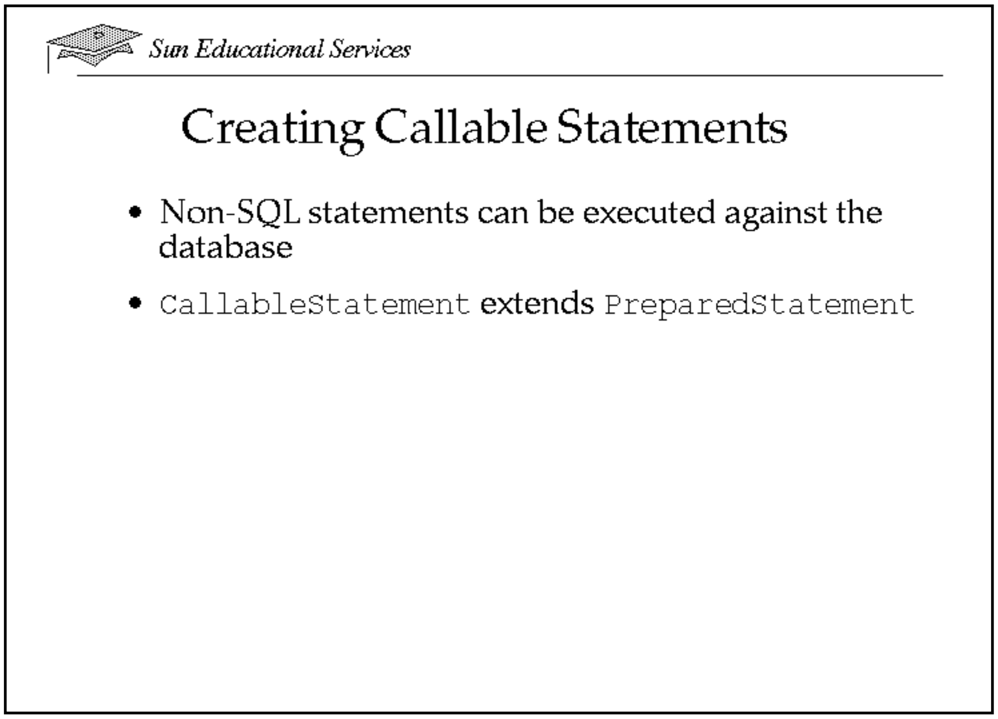
The `setXXX` methods for setting SQL `IN` parameter values must specify types that are compatible with the defined SQL type of the input parameter. For example, if the `IN` parameter has SQL type `Integer`, then you should use the `setInt` parameter.

Once a parameter value has been defined for a given statement, you can use it for multiple executions of that statement until it is cleared by a call to the `PreparedStatement.clearParameters` method.

Using the `setXXX` *Methods*

Table 3-2 `setXXX` Methods and SQL Types

Method	**SQL Type(s)**
`setASCIIStream`	`LONGVARCHAR` produced by an ASCII stream
`setBigDecimal`	`NUMERIC`
`setBinaryStream`	`LONGVARBINARY`
`setBoolean`	`BIT`
`setByte`	`TINYINT`
`setBytes`	`VARBINARY` or `LONGVARBINARY` (depending on the size relative to the limits on `VARBINARY`)
`setDate`	`DATE`
`setDouble`	`DOUBLE`
`setFloat`	`FLOAT`
`setInt`	`INTEGER`
`setLong`	`BIGINT`
`setNull`	`NULL`
`setObject`	The given object that is converted to the target SQL type before being sent
`setShort`	`SMALLINT`
`setString`	`VARCHAR` or `LONGVARCHAR` (depending on the size relative to the driver's limits on `VARCHAR`)
`setTime`	`TIME`
`setTimestamp`	`TIMESTAMP`
`setUnicodeStream`	`UNICODE`



Sun Educational Services

Creating Callable Statements

- Non-SQL statements can be executed against the database
- `CallableStatement` extends `PreparedStatement`

Creating Callable Statements

A *callable statement* allows non-SQL statements (such as stored procedures) to be executed against the database. The `CallableStatement` class extends the `PreparedStatement` class, which provides the methods for setting `IN` parameters. Because the `PreparedStatement` class extends the `Statement` class, a method for retrieving multiple results with a stored procedure is supported with the `Statement.getMoreResults` method.

An Example of Using a Callable Statement

For example, you could use a `CallableStatement` if you wanted to store a precompiled SQL statement to query a database containing available seat information about an airline flight

```
String planeID = "727";
CallableStatement querySeats = msqlConn.prepareCall("{call
return_seats[?, ?, ?, ?]}");
try {
    querySeats.setString(1, planeID);
    querySeats.registerOutParameter(2, java.sql.Types.INTEGER);
    querySeats.registerOutParameter(3, java.sql.Types.INTEGER);
    querySeats.registerOutParameter(4, java.sql.Types.INTEGER);
    querySeats.execute();
    int FCSeats = querySeats.getInt(2);
    int BCSeats = querySeats.getInt(3);
    int CCSeats = querySeats.getInt(4);
} catch (SQLException SQLEx){
        System.out.println("Query failed");
        SQLEx.printStackTrace();
}
```

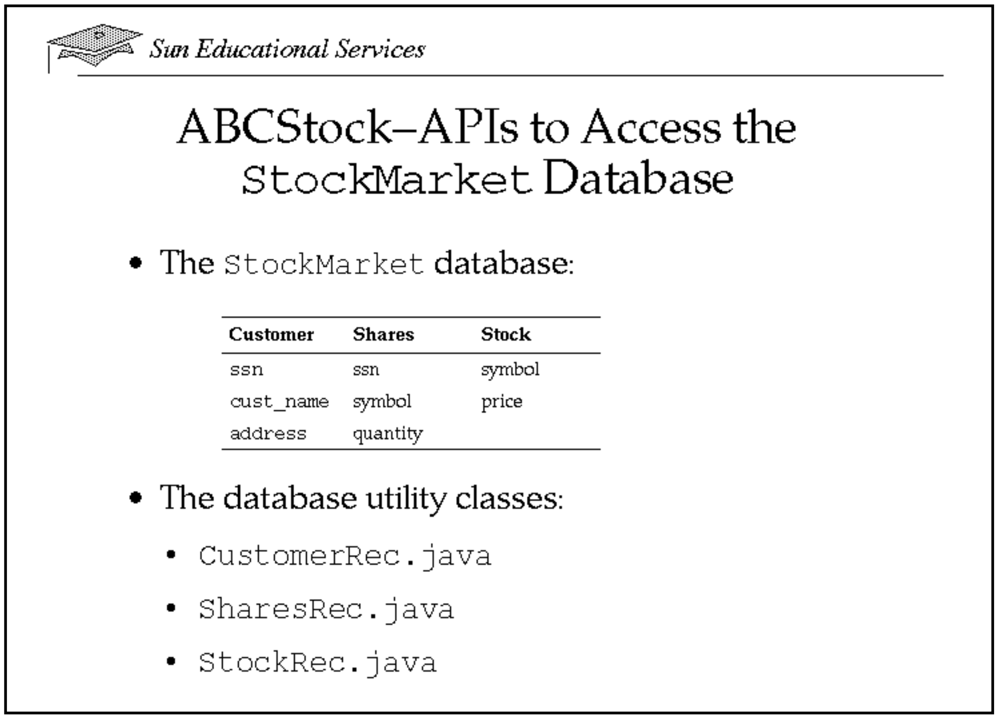
Before executing a stored procedure call, you must explicitly call `registerOutParameter` to register the `java.sql.Type` of any SQL `OUT` parameters.

Mapping SQL Data Types Into Java Data Types

Table 3-3 lists the standard Java types for mapping various common SQL types.

Table 3-3 Mapping SQL Types to Java Data Types

SQL Type	Java Type
`CHAR`	`String`
`VARCHAR`	`String`
`LONGVARCHAR`	`String`
`NUMERIC`	`java.math.BigDecimal`
`DECIMAL`	`java.math.BigDecimal`
`BIT`	`boolean`
`TINYINT`	`byte`
`SMALLINT`	`short`
`INTEGER`	`int`
`BIGINT`	`long`
`REAL`	`float`
`FLOAT`	`double`
`DOUBLE`	`double`
`BINARY`	`byte[]`
`VARBINARY`	`byte[]`
`LONGVARBINARY`	`byte[]`
`DATE`	`java.sql.Date`
`TIME`	`java.sql.Time`
`TIMESTAMP`	`java.sql.Timestamp`


Sun Educational Services

ABCStock–APIs to Access the StockMarket Database

- The StockMarket database:

Customer	Shares	Stock
ssn	ssn	symbol
cust_name	symbol	price
address	quantity	

- The database utility classes:
  - CustomerRec.java
  - SharesRec.java
  - StockRec.java

ABCStock – APIs to Access the `StockMarket` *Database*

The `StockMarket` *Database*

In the lab exercise, you use the `StockMarket` database. As described in Module 1, the `StockMarket` database contains three tables: `Customer`, `Shares`, and `Stock`, each with the following columns:

Customer	**Shares**	**Stock**
`ssn`	ssn	symbol
`cust_name`	symbol	price
`address`	quantity	

The Database Utility Classes

To access this mSQL database, you must create a class containing methods specific to the tables and create classes to represent the information retrieved. When accessing the `StockMarket` database, classes to hold information from the specific records could be `CustomerRec`, `SharesRec`, and `StockRec`. The class to handle all the database access could be called `Database`.

ABCStock – Support Classes for the StockMarket Database

CustomerRec.java

An instance of the following could represent a single customer record in the database.

```
package broker.database;

import java.io.Serializable;
import java.util.*;

// CustomerRec class
// This class represents a single customer record in the
// database, including the number of shares the customer
// owns.
//
public class CustomerRec implements Serializable {

   private String ssn;
   private String name;
   private String addr;
   private Vector portfolio;

   // Constructors
   public CustomerRec (String ssn, String name, String addr,
      Vector portfolio) {
      this.ssn = ssn;
      this.name = name;
      this.addr = addr;
      this.portfolio = portfolio;
   }

   public CustomerRec (String ssn, String name, String addr) {
      this (ssn, name, addr, null);
   }

   public CustomerRec (String ssn) {
      this (ssn, null, null, null);
   }

   public CustomerRec () {
      this (null, null, null, null);
   }
```

```
      // Accessor methods

   public String getSSN () {
      return ssn;
   }

   public String getName () {
      return name;
   }

   public String getAddr () {
      return addr;
   }

   // Get and return portfolio for this customer.
   // This method will return the Vector object
   // that contains the portfolio
   public Vector getPortfolio () {
      return portfolio;
   }

   // Mutator methods - you cannot change ssn

   public void setName (String newName) {
      name = newName;
   }

   public void setAddr (String newAddr) {
      addr = newAddr;
   }

   public void setPortfolio (Vector newPortfolio) {
      portfolio = newPortfolio;
   }
}
```

SharesRec.java

An instance of the following could represent a single shares record in the database.

```
package broker.database;
import java.io.Serializable;

// The SharesRec class
// Stores a single instance of a Shares record
// These are dynamic objects that belong to customers
public class SharesRec implements Serializable {

   private String ssn;
   private String symbol;
   private int quantity;

   // **SharesRec constructor**
   public SharesRec (String ssn, String symbol, int quantity) {
      this.ssn = ssn;
      this.symbol = symbol;
      this.quantity = quantity;
   }

   public SharesRec (String ssn) {
      this (ssn, "", 0);
   }

   public SharesRec () {
      this ("", "", 0);
   }

   // **Accessor methods**
   public String getSSN () {
      return ssn;
   }

   public String getSymbol () {
      return symbol;
   }

   public int getQuantity () {
      return quantity;
   }
   // **Mutator methods** - note no setSSN method

```

```
    public void setSymbol (String newSymbol) {
       symbol = newSymbol;
    }

    public void setQuantity (int newQuantity) {
       quantity = newQuantity;
    }
}
```

StockRec.java

An instance of the following could represent a single stock record in the database.

```
package broker.database;
import java.io.Serializable;

// The StockRec class
// Keeps a single record instance of a stock object
//
// These are created from a dynamic query against the DB
public class StockRec implements Serializable {

  private String symbol;
  private float price;

  // StockRec constructors
  // Create a instance of a stock from the queried data
  public StockRec (String symbol, float price) {
    this.symbol = symbol;
    this.price = price;
  }

  // create a instance of a stock with no data
  public StockRec () {
    symbol = "";
    price = 0.0f;
  }

  // Accessor Methods
  public float getPrice () {
    return price;
  }

  public String getSymbol () {
    return symbol;
  }

  public void setPrice (float newPrice) {
    price = newPrice;
  }

    public void printPrice () {
    System.out.print(price);
  }

  public void printStock () {
     System.out.print(symbol);
  }
}
```

ABCStock – Database API

The instance variables and methods that you should use to connect to the database are listed on the following pages.

The instance variables are:

- `Connection mSQLcon` – Creates a `java.sql.Statement`
- `Statement stmt` – Enables queries to be submitted to the database and, optionally, `ResultSets` to be returned.
- `ResultSet` result – Stores rows returned by queries
- `static String database` = "`StockMarket`" – Stores the name of the database

Methods

You use the following methods to access the database.

`public Database (String serverName) throws SQLException`

Return Type:	(None) – Constructor
Exceptions:	`SQLException` if there is a problem connecting to the server or accessing `StockMarket` database
Arguments:	String *serverName*
Behavior:	Establishes a connection with the database server

`public void close () throws SQLException`

Return Type:	`void`
Exceptions:	`SQLException` if there is a problem connecting to the server
Arguments:	None
Behavior:	Closes the connection with the database server

```
public void addCustomer (String name, String ssn, String
address) throws DuplicateIDException
```

Return Type:	`void`
Exceptions:	`DuplicateIDException` if *ssn* already exists in the `Customer` table
Arguments:	String *name*, String *ssn*, String *address*
Behavior:	Adds a customer to the `Customer` table

```
public void deleteCustomer (String ssn) throws
RecordNotFoundException
```

Return Type:	`void`
Exceptions:	`RecordNotFoundException` if *ssn* is not found in the `Customer` table
Arguments:	String *ssn*
Behavior:	Deletes a customer from the `Customer` table

```
public void updateCustomer (String name, String ssn,
String address) throws RecordNotFoundException
```

Return Type:	`void`
Exceptions:	`RecordNotFoundException` if *ssn* is not found in the `Customer` table
Arguments:	String *name*, String *ssn*, String *address*
Behavior:	Updates customer information in the `Customer` table; *ssn* cannot be changed

`public CustomerRec getCustomer (String ssn) throws RecordNotFoundException`

Return Type:	`CustomerRec`
Exceptions:	`RecordNotFoundException` if *ssn* is not found in the `Customer` table
Arguments:	String *ssn*
Behavior:	Returns a `CustomerRec` populated with data from the `Customer` and `Shares` tables

`public CustomerRec [] getAllCustomers ()`

Return Type:	`CustomerRec [ ]`
Exceptions:	None
Arguments:	None
Behavior:	Returns an array of `CustomerRecs` populated with data from the `Customer` and `Shares` tables

`public Vector getPortfolio (String ssn) throws RecordNotFoundException`

Return Type:	`Vector`
Exceptions:	`RecordNotFoundException` if *ssn* is not found in the `Customer` table
Arguments:	String *ssn*
Behavior:	Returns a Vector of `SharesRec` objects, representing the portfolio for the customer

`public StockRec [] getAllStocks ()`

Return Type:	`StockRec []`
Exceptions:	None
Arguments:	None
Behavior:	Returns an array of `StockRecs` populated with data from the `Stock` table

‘`public void sellShares (String ssn, String symbol, int quantity) throws RecordNotFoundException, InvalidTransactionException`

Return Type:	`void`
Exceptions:	`RecordNotFoundException` if *ssn* is not in the `Customer` table, `InvalidTransactionException` if *ssn* does not own any shares of given stock or does not own enough shares of given stock
Arguments:	String *ssn*, String *symbol*, int *quantity*
Behavior:	Updates Shares table with new *quantity* of shares

`public void sellShares (String ssn, String symbol) throws RecordNotFoundException, InvalidTransactionException`

Return Type:	`void`
Exceptions:	`RecordNotFoundException` if *ssn* is not in the `Customer` table, `InvalidTransactionException` if *ssn* does not own any shares of given stock
Arguments:	String *ssn*, String *symbol*
Behavior:	Deletes an entry from the `Shares` table (sells all shares of a stock for a customer)

```
public void buyShares (String ssn, String symbol, int
quantity) throws RecordNotFoundException
```

Return Type:	`void`
Exceptions:	`RecordNotFoundException` if *ssn* is not found in the `Customer` table
Arguments:	String *ssn*, String *symbol*, int *quantity*
Behavior:	Either adds a new entry to the `Shares` table or updates an existing entry

```
public void updateStockPrice (String symbol, float
price)
```

Return Type:	`void`
Exceptions:	None
Arguments:	String *symbol*, float *price*
Behavior:	Updates the price for a stock in `Stock` table: this assumes valid stock symbol

```
public float getStockPrice (String symbol)
```

Return Type:	`float`
Exceptions:	None
Arguments:	String *symbol*
Behavior:	Returns a price from `Stock` table, -1.0 on failure

Exercise: Implementing A Database Class Wrapper

Exercise objective – Create and implement a database using the `java.sql` interface methods.

Preparation

In this exercise you will construct a class to wrap the JDBC classes and methods to access the database. While technically two-tiered, you will still experience the "implementation detail hiding" aspect of code wrapping that allows others to use a class without having to know about the JDBC layer and database access details.

Tasks

Complete the following steps:

1. Implement all methods in the `Database` class by making calls to the database through the `java.sql` interface. In the constructor, create an instance of the driver implementation, and get an instance of a connection to the `StockMarket` database.

Note – The model outlined in this module is one of many options. Your team can use another object model, just make sure it meets the functional specifications. Add supporting methods to your `Database` class, if so desired.

---

2. Write a harness program to test the `Database` class. Do not create or remove any existing tables— just add and delete information from these tables.

3. Verify that your program is actually updating the `StockMarket` database by running `msql` in another window.

4. Prepare a brief report highlighting the features of your class design.

Notes

Exercise Summary

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

- Experiences
- Interpretations
- Conclusions
- Applications

Check Your Progress

Before continuing on to the next module, check that you are able to accomplish or answer the following:

- ❑ Describe JDBC
- ❑ Explain how using the abstraction layer provided by JDBC can make your database front-end portable across platforms
- ❑ Describe the five major tasks involved with the JDBC programmer's interface
- ❑ State the requirements of a JDBC driver and its relationship to the JDBC driver manager

Think Beyond

Using JDBC requires a working knowledge of the SQL language syntax.

How does wrapping JDBC functionality with Java classes and packages help reduce what programmers need to know about SQL?

Building GUIs 4

Objectives

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

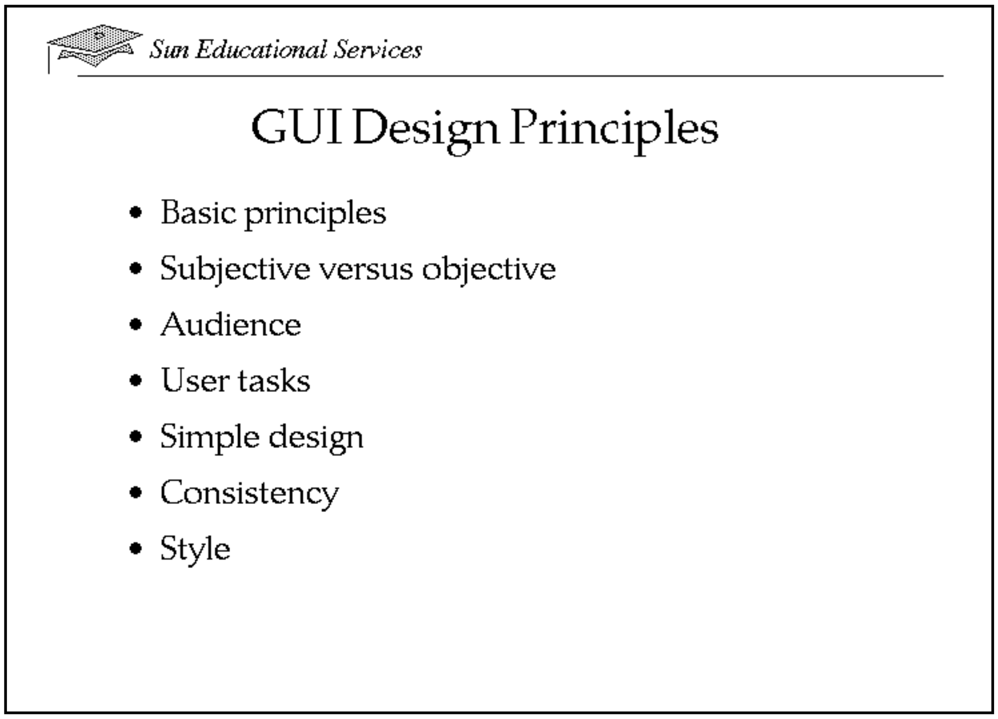
- Apply the principles of good GUI design
- Differentiate, at a high level, between the *new* Java Foundation Classes (*Swing components*) and the AWT model
- Explain how to create the class structure needed for an object-oriented GUI
- Design and implement a GUI for the BrokerTool project using your choice of containers, components, and layout managers

This module reinforces the foundations of GUI design and creation using layout managers and components from the Abstract Window Toolkit (AWT). This module also compares the Swing package found in the Java foundation classes with the AWT.

Relevance

Discussion – GUIs are part of everyday life. They make interacting with software easy, intuitive, and fun. Most modern software is expected to have a GUI interface if it is to be adopted and accepted by a user base.

- What would happen if you released a complex piece of software with only a command-line interface?
- How do you think the software and user communities would view your software product?


Sun Educational Services

GUI Design Principles

- Basic principles
- Subjective versus objective
- Audience
- User tasks
- Simple design
- Consistency
- Style

GUI Design Principles

How can you design GUIs that guide the end-user, and how can you tell when a GUI has the potential to hamper the user's productivity? This section addresses these questions and describes how to put this information to work for your user.

Basic Principles

Why is it important to have an effective GUI? How can you design an application that improves the efficiency and effectiveness of the user? Much of a user's opinion of an application is a direct result of the experience with its GUI, so it is imperative that you put time and thought into designing the GUI. A tremendously powerful and productivity-enhancing application can be significantly diminished in the mind of the user if the GUI appears to be lacking, despite the application's underlying technical excellence.

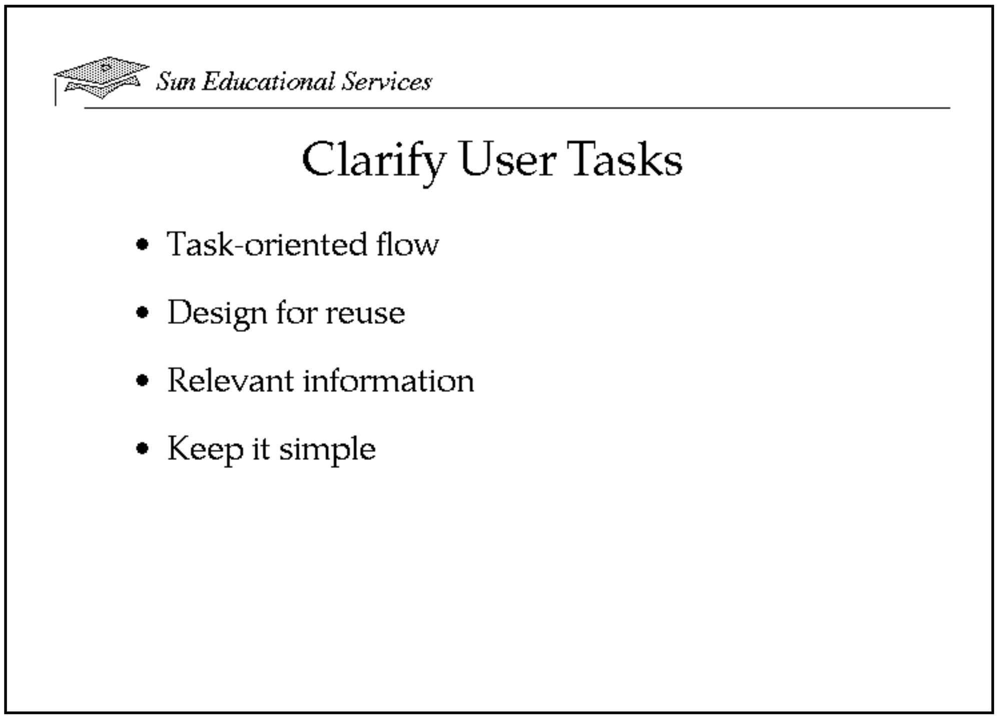
Subjective Versus Objective

Remember that the decisions you make about your GUI might be based on subjective impressions, objective impressions, or more likely, a combination of the two. Often it is best to gather subjective information through usability studies with actual end-users. You can meet objective system goals based on several criteria: knowing the audience, having task-oriented organization and visual structure, striving for an elegant and simple design that provides intuitive functionality, being consistent, using scale and contrast with care, and keeping a style that is familiar to your users.

Assess Your Audience

The first step is to start with a thorough assessment of the users of your system. Their experience levels, typical working environments, and expectations of your application greatly influence how they react to your front-end interface. A beginner user does not want complex cross-screen tasks, while an expert might want to customize the GUI to accomplish specific goals more efficiently.

Design the GUI so that it can grow gracefully. As the needs of the audience increase, your GUI must expand. An adaptable GUI that can have features added without a complete redesign serves users better over an extended period of time.



Sun Educational Services

Clarify User Tasks

- Task-oriented flow
- Design for reuse
- Relevant information
- Keep it simple

Clarify User Tasks

Once you know your audience, you must plan the tasks that your GUI will assist the user in completing. With separate tasks laid out, you can group components related to each task. If needed, provide individual screens to step the user through the tasks. Logical flow within an area keeps the user on the right track. Keep in mind that simplicity is almost always better than complexity so limit the number of screens, menus, and dialogs to those necessary for each job.

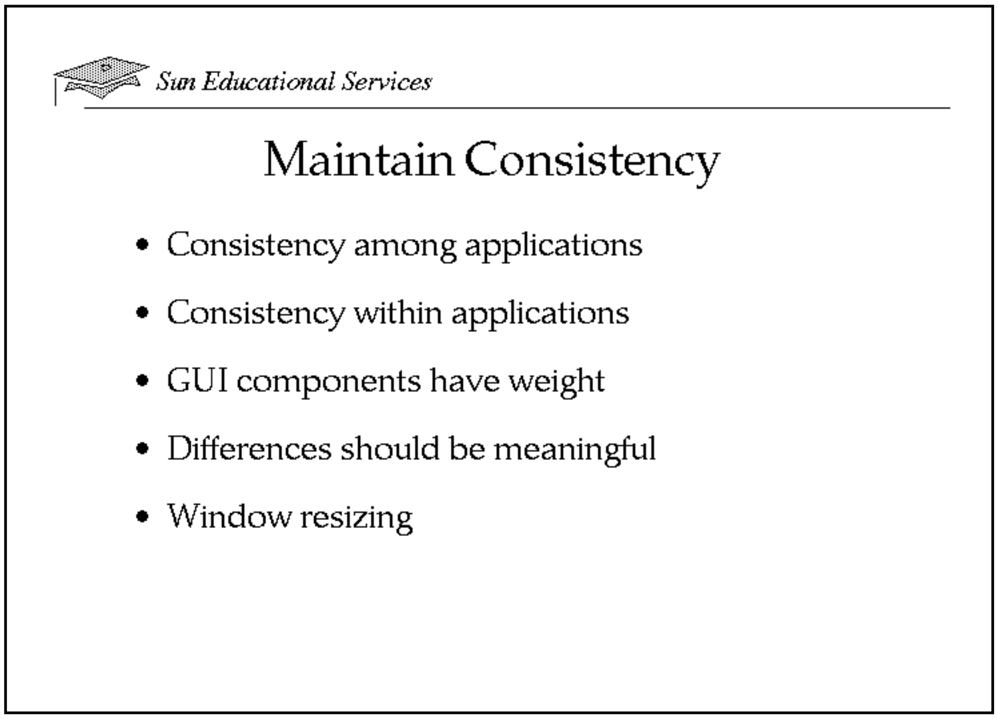
Similarly, avoid placing nonmeaningful information on the screen. The two extremes of user categories, beginner versus expert, usually have different information needs. Design your GUI so that you can reuse as many setups as possible. This allows for a smooth transition as the user's experience level increases. In this way, information that is inappropriate for some users can be accessed only when necessary.

One indicator of a less than optimally designed GUI is when the end-user frequently asks, “Where am I?” This is often the result of chopping up a single task into multiple, disconnected steps. With a task-oriented design, keep the relevant information within the user's view as much as possible. If it is necessary to use multiple screens or menus to accomplish a task, ensure that enough information is available at each stage so the user does not have to rely on memory to track the process. This helps continuity and flow.

Keep It Simple

There is a fine line between providing enough information and providing too much information. The determination of how much is enough is often subjective. Usability studies with a prototype of your GUI can help you determine this.

You can make more objective decisions about the amount of visual stimulus caused by the GUI. Presenting the user with too many visual stimulants such as colorful patterns or excessive graphics can turn using the application into a mentally exhausting experience. Strive to simplify the visual complexity while not eliminating relevant information.



Sun Educational Services

Maintain Consistency

- Consistency among applications
- Consistency within applications
- GUI components have weight
- Differences should be meaningful
- Window resizing

Maintain Consistency

An extremely important aspect of good GUI design is maintaining consistency. Without a consistent interface, users are frequently left with questions and confusion about the user interface. It is particularly important to be consistent across screens and dialogs used for similar tasks.

Consistency Among Applications

The design philosophy behind Java technology goes a long way towards maintaining consistency on a platform-by-platform basis. Java technology is designed so that you have complete control over how each of your GUI components appears. You do not have to specify every component's look and feel, but you do have the choice. Should your button look like a Motif button or should it have a customized design? These options are provided by the AWT and Java Foundation Classes (JFC) Swing packages.

Java technology’s layout managers help maintain the consistency of your GUI’s visual layout, regardless of the platform on which your application is run. This is a compelling reason why exact placement of components within containers is not recommended. A further discussion of AWT, Swing and Layout Managers follows the GUI principles section of this module.

Consistency Within Applications

While maintaining visual consistency among applications across platforms can be fully automated if you choose, you do need to work to maintain consistency within your application. If you label most of your buttons in the affirmative, do not suddenly throw one in labeled in the negative. For example, if most of your buttons say “Save,” do not suddenly switch to “Don’t save.” The size and color of components also affect consistency.

GUI Components Have Weight

Avoid buttons that vary in size because GUI components, such as buttons have “weight,” and weight denotes importance. If the user is presented with three buttons, two small and one considerably larger, the user’s attention is drawn to the larger button. Font sizes and styles can have the same effect.

But size is not the only aspect that gives a GUI component weight. Color has the same effect. If a button stands out boldly due to its unusual color, it draws the user’s attention.

Differences Should Be Meaningful

The human brain appears to be wired in such a way that it has a natural disposition toward assigning meaning to colors. If your screen has six buttons, each a different color, your user will become confused trying to sort out the meaning behind the colors (especially if there is no meaning).

You can use visual differences to convey information to the end-user. For example, graying out a disabled menu option or visually distinguishing an editable versus a noneditable text field informs the end-user of what is allowed at that given point in time.

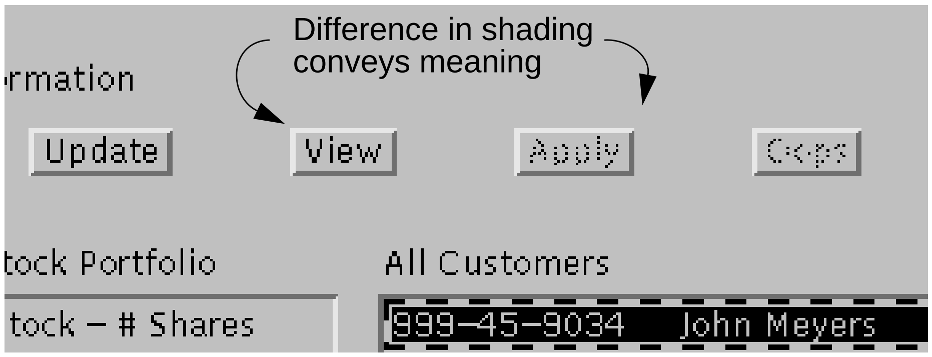


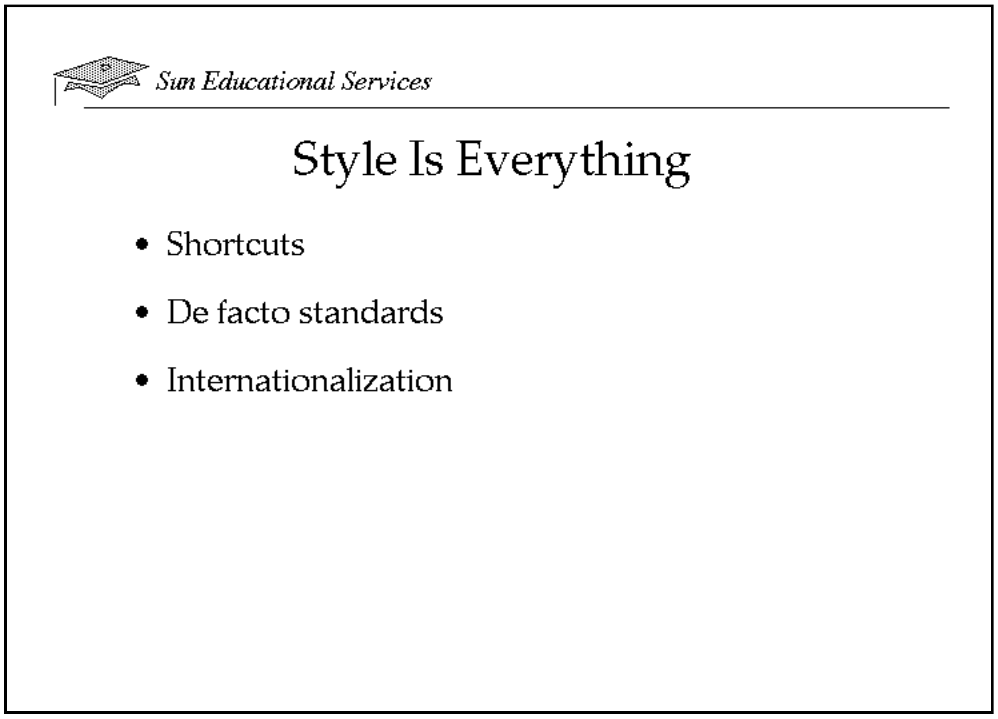
Figure 4-1 GUI Component Visual Differences

Remember judicious use of weight, size, and color can have noticeable effects. However, international audiences can interpret the elements differently.

Window Resizing

When the user resizes the windows, how does this affect the layout of the GUI? If the window is reduced in size, the application might not be usable, even if the user increases the window's size. If the application does not adjust appropriately to an increase in size, the user is likely to conclude that the application is misbehaving.

If increasing the size of the window causes a trivial button instead of the text area to increase in size, the user might ask, "Why did it do that?" Ensure that changes, particularly enlargements, to your window do not increase confusion due to weight issues, and thus decrease your GUI's functionality.



Sun Educational Services

Style Is Everything

- Shortcuts
- De facto standards
- Internationalization

Style Is Everything

You should strive to maintain the style in which your user is accustomed to working. This involves not only everything that has been mentioned previously, but also topics, such as menu handling, use of standard language, look-and-feel for the user's platform, and challenges presented by a multinational user group.

Provide Shortcuts

You should provide key stroke equivalents, pop-up menus, and tool bars to increase the efficiency of experienced users. However, make sure you do not violate the "Keep It Simple" principle. That is, do not provide shortcuts or alternatives that cause confusion due to the inability to track the task's process or the experience level of the user.

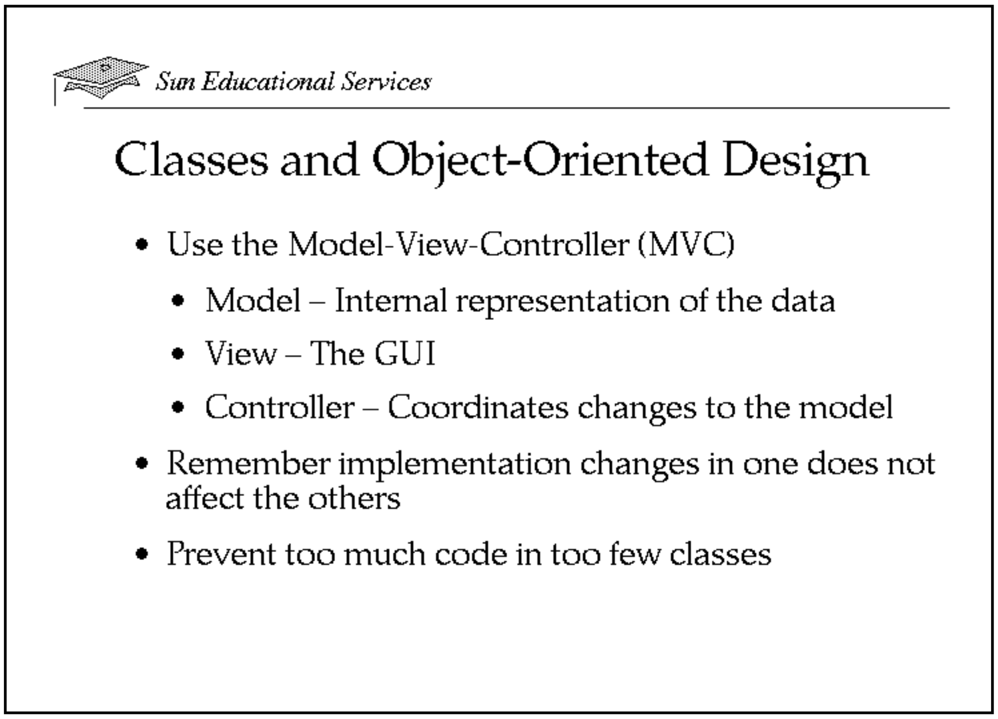
De Facto Standards

Consider the types of environments in which your application is deployed. For example, if the environment contains Macintosh machines, then you must consider the language that Macintosh users understand. Not only does the language you choose for your components affect usability, issues, such as mouse buttons, menu positioning, and components' look-and-feel also come into play. A single-platform environment makes these decisions easier. If the environment is mixed, then it is a matter of agreeing on a set of standards that is acceptable to all users.

Internationalization

Internationalization is receiving more attention now because of the advent of the World Wide Web and Java technology. If you plan to distribute your GUI to other countries, be sensitive to cultural differences. Experiment with prototype user interface designs before committing major programming resources.

A good example of this is how people read dates and numbers. Is 7/10/1998 July the tenth or October the seventh? Is 5.000 five thousand or five with three decimal places? These issues can have an impact on your users.



Sun Educational Services

Classes and Object-Oriented Design

- Use the Model-View-Controller (MVC)
  - Model – Internal representation of the data
  - View – The GUI
  - Controller – Coordinates changes to the model
- Remember implementation changes in one does not affect the others
- Prevent too much code in too few classes

Classes and Object-Oriented Design

Despite good intentions, it is deceptively easy to code a GUI that breaks all of your goals for proper object-oriented design. You do your user-analysis, set out all of your tasks, and plan for a smooth, consistent layout. Then a terrible thing happens—it all lands in one big, overgrown class!

Since it is just as important to use good object-oriented design (OOD) principles when creating GUI classes as it is when designing business logic or database access classes, this section describes how to do this. One valuable technique in OOD is the use of design patterns. Design patterns are language-independent strategies for solving common object-oriented design problems. These design patterns are found throughout the Java programming language; for example, the "Singleton" pattern is used in the `System` class. An important design pattern for GUI programming is the Model-View-Controller (MVC) pattern made popular by the SmallTalk™ language.

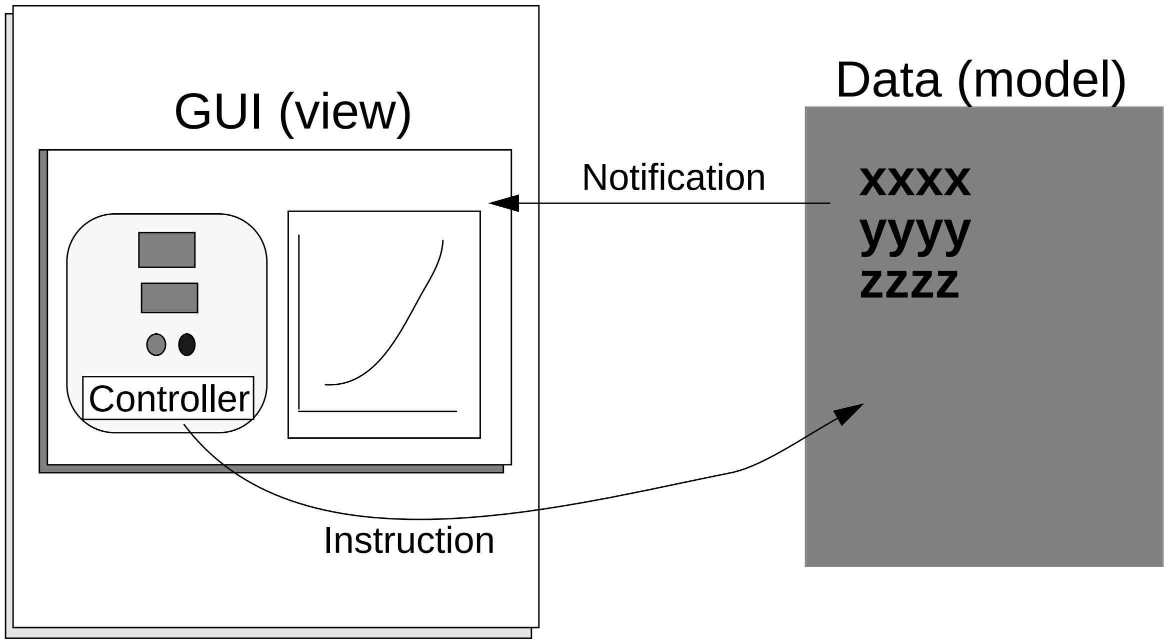


Figure 4-2 Model-View-Controller Design Pattern

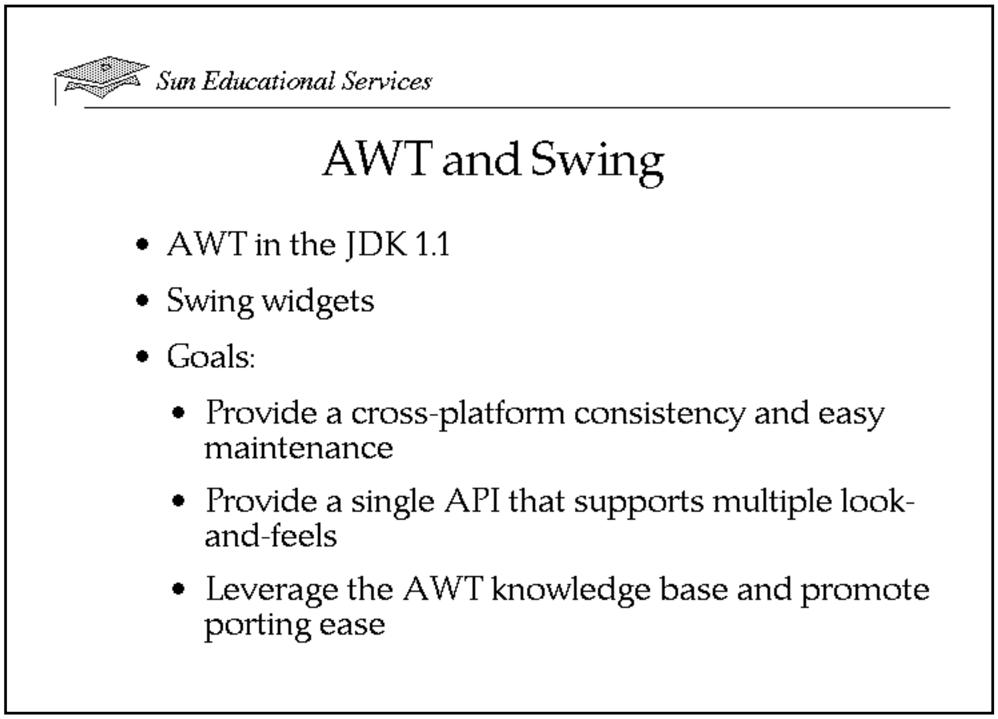
Figure 4-2 illustrates the MVC design pattern. The *view* of the data is separated from the *model* of the data. In this case, the view of the data is the user's representation of the data (shown on the GUI). The model of the data is the internal representation that contains the state of the data. If any change in the state occurs, the model notifies the view. The *controller* is a way for the user to interact with the model and view. It is a GUI aspect that sends instructions to the *model*. These instructions commonly cause the model's state to change, which requires notification to the *view*.

This design approach allows GUI classes to change without affecting the underlying data, and it allows the underlying data representation to change without modifying the GUI. That is, MVC decouples the GUI view from the data model.

You can extend this approach to the GUI components themselves, in which the look-and-feel of the components is separated from the behavior of the components. You can see this in the design of the JFC Swing classes in which a single component's interface—the view—and its controller code are in one class, while the data model for the component is in another class. This allows for maximum flexibility in modifying the behavior or appearance of Swing components without necessarily changing the class definition for the other part of the component.

Using the principles of MVC when you implement a GUI-based Java application, you can put the data in a flat file, an SQL RDBMS, or an object-oriented database management system (OODBMS) without concern for the particulars of the storage mechanism. If you design your model-view interface to provide a consistent transition back and forth between the data and the GUI, then changes in implementation of any one will not affect the implementation of the others. That is, changes in the data storage affect the implementation of the model alone, not the view or the controller. This reduced dependency makes system upgrades substantially easier.

Following the MVC design pattern also reduces the temptation to put too much code in too few classes. The following is a review of some of the classes you use in designing and building your GUI.



Sun Educational Services

AWT and Swing

- AWT in the JDK 1.1
- Swing widgets
- Goals:
  - Provide a cross-platform consistency and easy maintenance
  - Provide a single API that supports multiple look-and-feels
  - Leverage the AWT knowledge base and promote porting ease

AWT and Swing

The Java Foundation Classes (JFC) is a group of packages used to enhance GUI design and is built on top of some fundamental aspects of AWT, specifically `Window` and `Container`. These packages include Swing, 2D graphics, and Accessibility.

The JDK 1.1 revamped AWT event processing and some of the GUI components. The `Graphics`, `Component`, and `LayoutManager` classes are part of the core AWT package. Other AWT packages provide functionality for cut and paste, event handling, and image manipulation.

Swing is an enhanced component set. Some of the components provided by Swing overlap and extend the functionality found in AWT components. In general, the Swing equivalent of an AWT component has the same name with the addition of the letter J, so `Button` becomes `JButton`, and `Label` becomes `JLabel`. Other Swing components implement completely new and more flexible GUI elements (for example, the JTree control for hierarchical display of information). Swing components are *lightweight*, meaning that they do not rely on the underlying operating system for their implementation.

Although the Swing components work well when used alongside AWT components, complications can arise. Because of this, a user interface should use Swing components exclusively and not mix Swing and AWT in the same program.

Swing is implemented entirely in the Java programming language to promote cross-platform consistency and easier maintenance. It provides a single API capable of supporting multiple look-and-feels so that developers and end-users are not locked into a single platform's look-and-feel. It is compatible with AWT APIs when there is overlapping functionality, the AWT knowledge base is leveraged, and porting ease is required.

A complete description of Swing is outside the scope of this course, but information about the basics of Swing is included in Appendix D, "Swing Foundations." For in-depth information, refer to the Sun Educational Services course SL-320: *User Interface, JFC, and Swing.*

The `java.awt` *and* `javax.swing` *Packages*

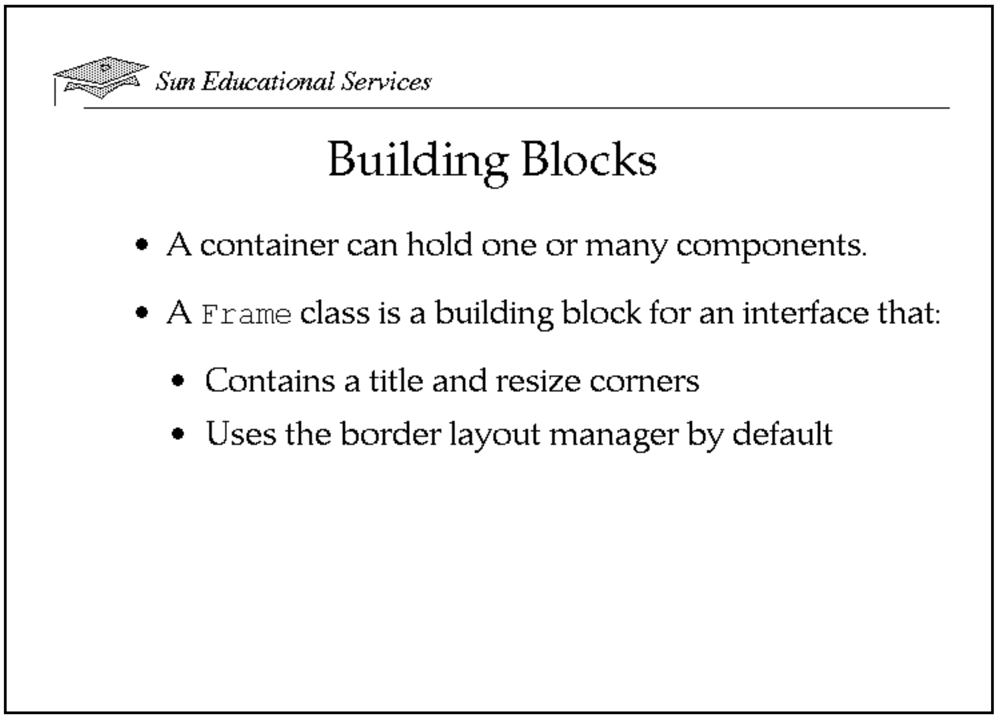
The `java.awt` and `javax.swing` packages contain classes to generate GUI components.

The three functional categories of classes in the GUI packages are:

- Containers
- Components
- Layout managers

Other packages that are subordinate in the package hierarchy are:

- `java.awt.event`
- `java.awt.image`
- `javax.swing.table`
- `javax.swing.plaf`



Sun Educational Services

Building Blocks

- A container can hold one or many components.
- A `Frame` class is a building block for an interface that:
  - Contains a title and resize corners
  - Uses the border layout manager by default

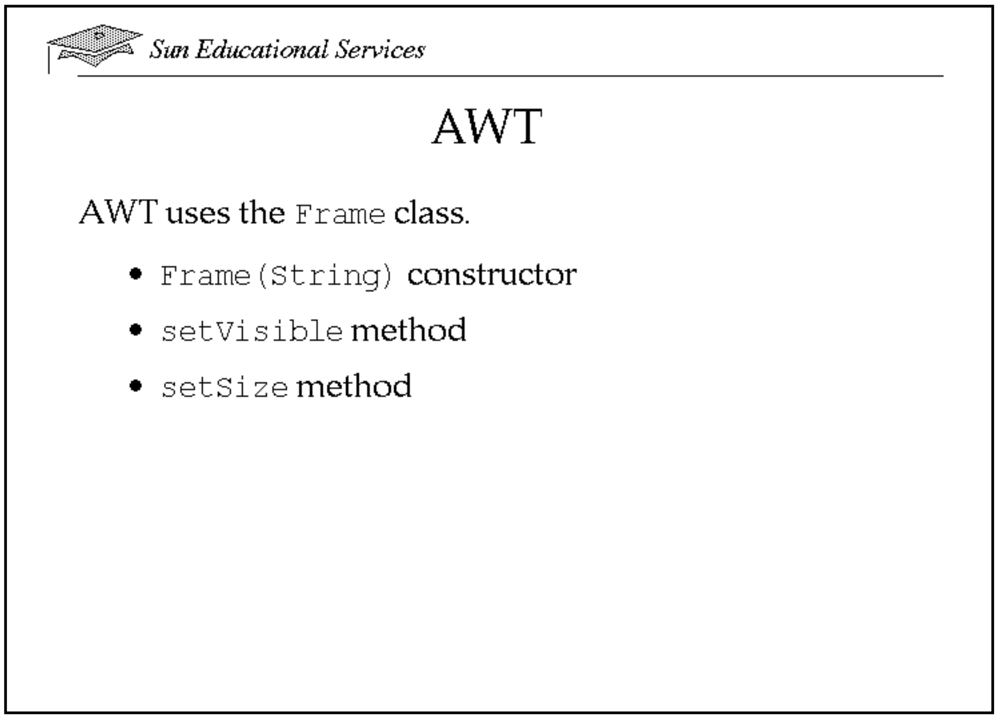
Building Blocks

Containers

A container can hold one or many components and, if desired, can hold other containers.

Note – The fact that a container can hold not only components, but also other containers is important and fundamental to building more complex layouts.

A `Frame` is a building block for an interface and represents a window on the screen. The `Frame` has a title and resize corners. If you do not explicitly use the `setLayout` method, a `Frame` uses a border layout manager by default.



Sun Educational Services

AWT

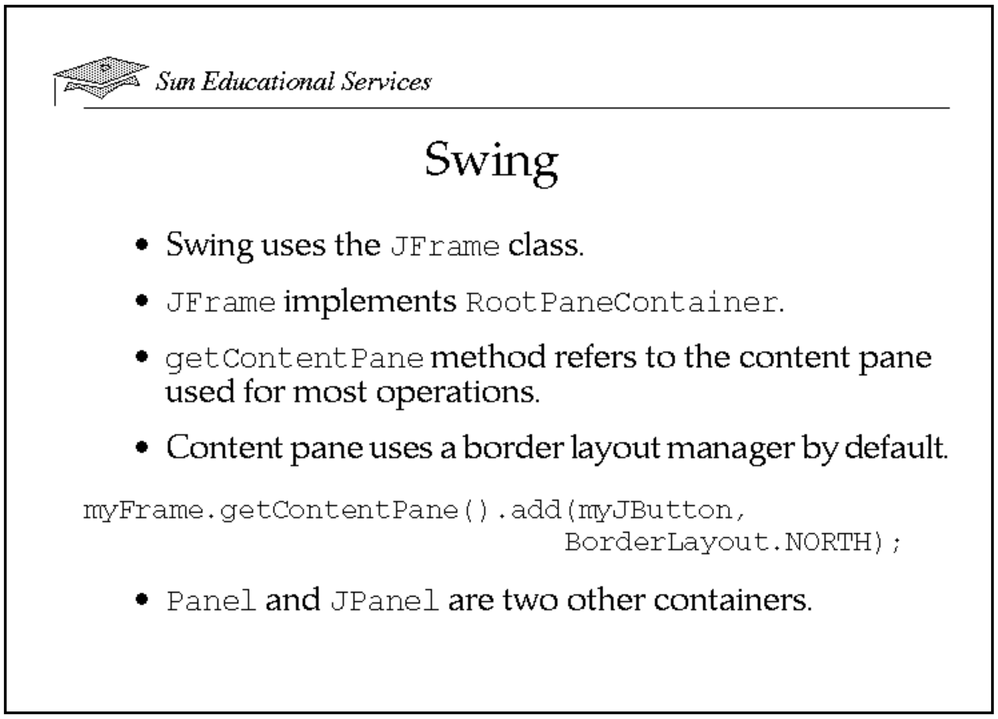
AWT uses the `Frame` class.

- `Frame(String)` constructor
- `setVisible` method
- `setSize` method

AWT

In AWT, the class you use is `Frame`. The constructor `Frame(String)` creates a new, invisible frame object with the title specified by the string. You can resize a `Frame` using the `setSize` method inherited from the `Component` class.

Note – You must call the `setVisible` and `setSize` methods to make the `Frame` visible and of usable size.



Sun Educational Services

Swing

- Swing uses the `JFrame` class.
- `JFrame` implements `RootPaneContainer`.
- `getContentPane` method refers to the content pane used for most operations.
- Content pane uses a border layout manager by default.

```
myFrame.getContentPane().add(myJButton,
                             BorderLayout.NORTH);
```

- `Panel` and `JPanel` are two other containers.

Swing

As previously mentioned, you should not use AWT and Swing components in the same layout, so you need to know about `JFrame`, a top-level Swing container. Like all top-level Swing containers, `JFrame` implements a special interface called `RootPaneContainer`. A `RootPaneContainer` is actually a container for a number of other panes; the root, glass, layered, and content panes. Most of the time, you need to be concerned only with the content pane.

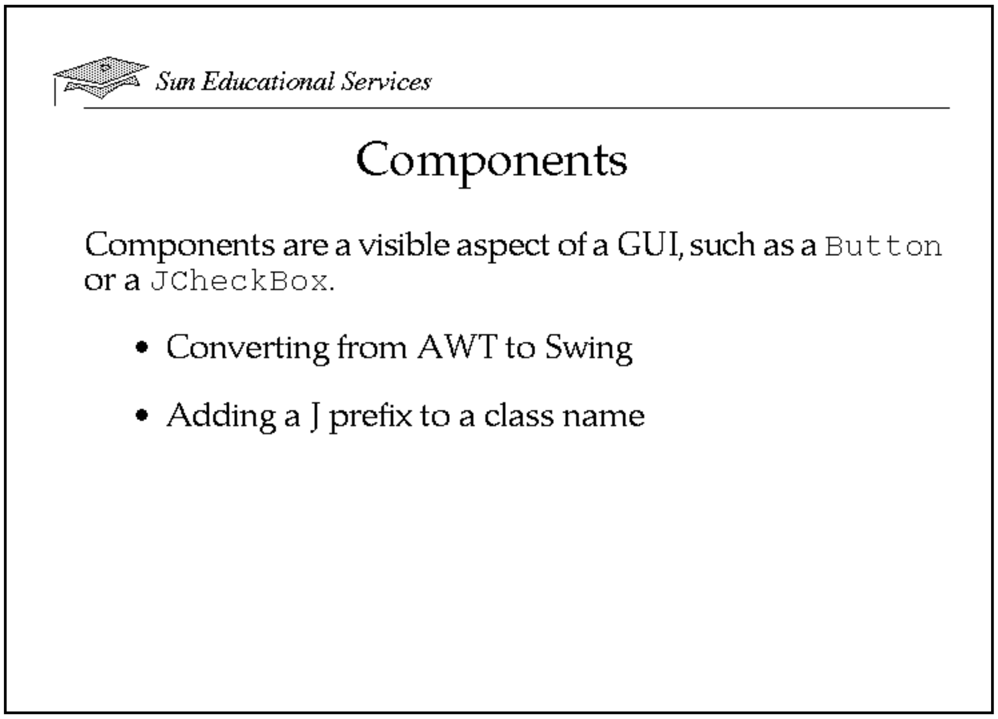
To perform most operations, such as setting a layout manager or adding new components to the container as a whole, you must refer to the content pane. This is obtained from a `RootPaneContainer` using the method `getContentPane`.

Content panes have a border layout manager by default, so, to add a `JButton` to a `JFrame` referred to by the variable `myFrame`, you would use code similar to the following:

```
myFrame.getContentPane().add(myJButton, BorderLayout.NORTH);
```

Panels

Two other containers to investigate are `Panel` and `JPanel`. Unlike `JFrame`, `JPanel` does not have a special content pane to which you add components.


Sun Educational Services

Components

Components are a visible aspect of a GUI, such as a `Button` or a `JCheckBox`.

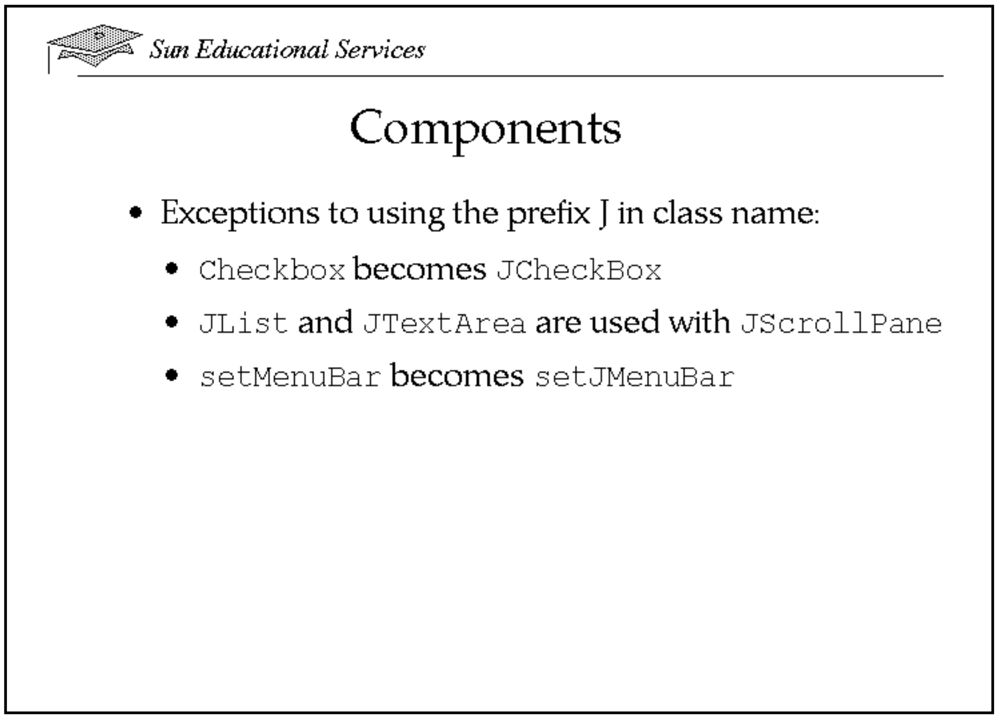
- Converting from AWT to Swing
- Adding a J prefix to a class name

Components

Components are the visible aspect of a GUI, such as a `Button` or a `JCheckBox`. AWT components rely on the underlying operating system for their appearance and behavior. Swing components are more flexible in that they can either have a system- or user-defined look-and-feel.

Converting From AWT to Swing

Because you are familiar with AWT components, using Swing is fairly straightforward. Often it requires little more than the addition of "J" in front of the AWT component class name. For example, rather than adding a `Button` to a `Panel`, you would add a `JButton` to a `JPanel`.

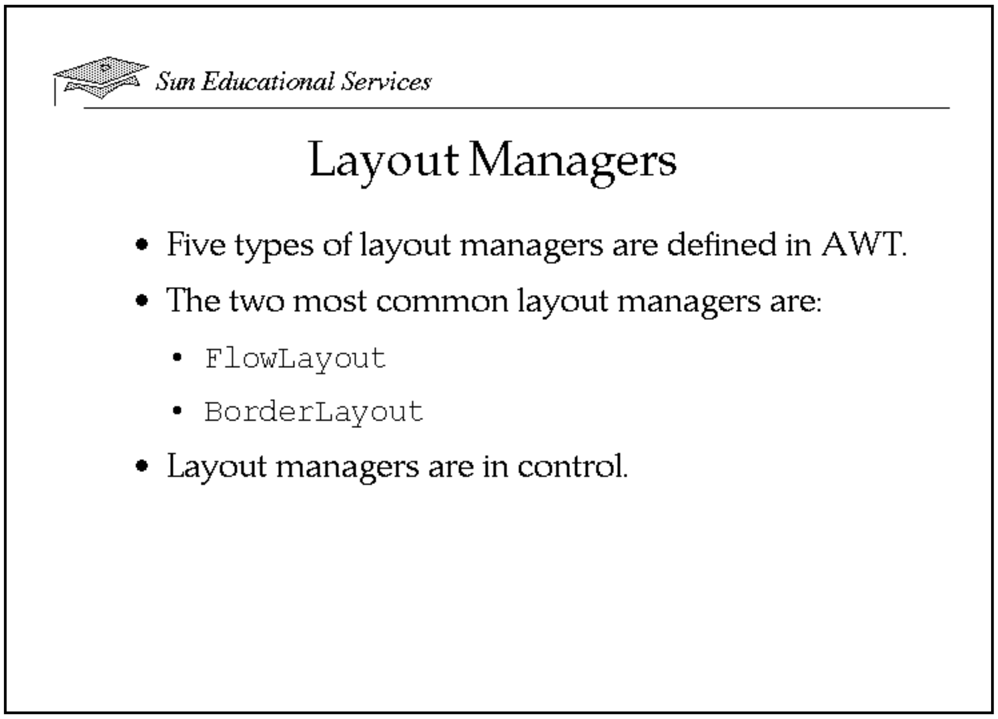

Sun Educational Services

Components

- Exceptions to using the prefix J in class name:
  - `Checkbox` becomes `JCheckBox`
  - `JList` and `JTextArea` are used with `JScrollPane`
  - `setMenuBar` becomes `setJMenuBar`

However, sometimes adding a “J” to the front of a class name is not enough to make a complete translation from AWT to Swing. For example:

- The `Checkbox` class in AWT is replaced by the `JCheckBox` class. There are two aspects to note here:
  - ▼ The spelling of `JCheckBox` has a capital B, unlike `Checkbox`
  - ▼ The Swing set has a separate class, `JRadioButton`, which should be used with `ButtonGroup` objects to implement radio button behavior.
- In Swing, components do not have automatic scrolling. Instead, components such as `JList` and `JTextArea` are added to the `JScrollPane` container if they need scrollbars.
- As the class names change by the addition of a “J”, so the method `setMenuBar` of a `java.awt.Frame` is paralleled with a new method `setJMenuBar` in a `JFrame`.



Sun Educational Services

Layout Managers

- Five types of layout managers are defined in AWT.
- The two most common layout managers are:
  - `FlowLayout`
  - `BorderLayout`
- Layout managers are in control.

Layout Managers

The position of a component in a container is determined by the container's *layout manager*. There are several types of layout managers defined in both AWT and Swing. The two most common AWT layout managers are `FlowLayout`, the default layout manager of panels and applets, and `BorderLayout`, the default layout manager of windows, dialogs, and frames. Other layout managers from AWT are presented in Appendix A, "Building GUIs With AWT." For more information about layout managers, see the online API documentation.

Because the layout manager is generally responsible for the size and position of components on a container, do not attempt to set the size or position of components yourself. If you use `setLocation`, `setSize,` or `setBounds`), certain layout managers can and will override your decision.

Wrap-up

With all of the topics discussed in this module, you now have the tools to design and build a GUI that is effective, functional, object-oriented, and visually pleasing. The following pages describe the requirements specifications that you must include.

Exercise: Creating the Stock Market GUI

Exercise objective – Given the functional GUI specifications, your team must develop a reasonable GUI design, first on paper, then in code, using the standard delegation event model and classes.

Preparation

Get together in a group, sketch out your GUI design based on the inputs and outputs required to interface with the `StockMarket` database.

The GUI specifications are listed on the following page.

Tasks

GUI Specifications

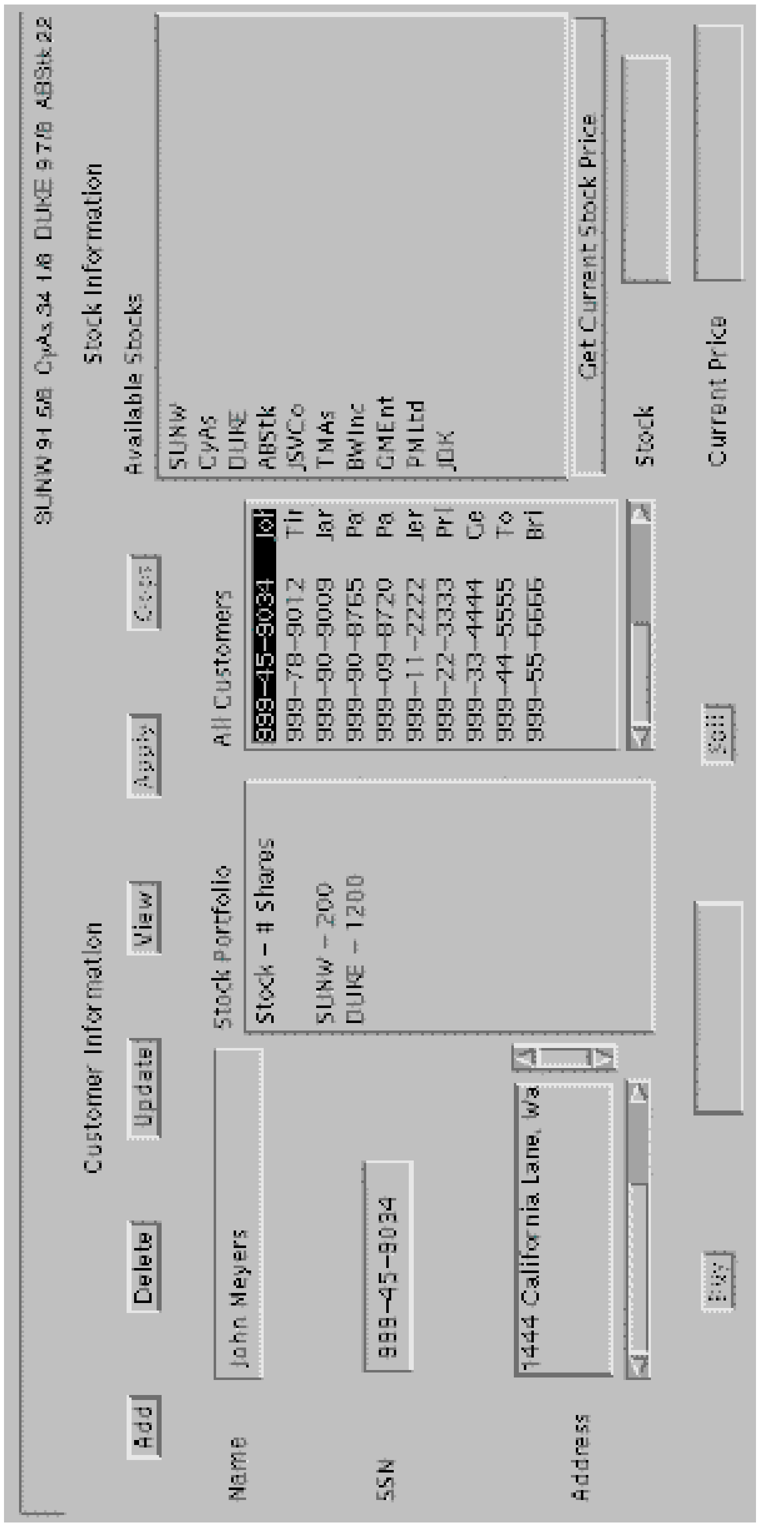
The BrokerTool program was designed with a functional specification in mind. To continue to provide the end-user with the necessary functionality, you must ensure that any modifications made to the BrokerTool program continue to meet the following *minimal* functional requirements.

The GUI must enable the end-user to:

- Add a customer record with a name, social security number, and address
- Delete a customer record
- Update a customer record
- View a customer record

- View the current price of any stock in the database
- View the customer's portfolio
- Buy an arbitrary quantity of a single stock for a particular customer at the current price
- Sell any quantity of a single stock that the selected customer owns
- Sell *all* shares of a single stock that the selected customer owns
- Determine current stock prices by way of a live-data ticker tape

Your group should decide how your GUI looks and behaves. However, remember the principles of GUI design discussed earlier, and work to ensure that your GUI remains interchangeable with the GUIs created by the other groups.



Name: John Meyers

SSN: 999–45–9034

Address: 1444 California Lane, Wa

Stock Portfolio:

Stock – # Shares

SUNW – 200
DUKE – 1200

All Customers:

```
999–45–9034  Jo
999–78–9012  Ti
999–90–9009  Jar
999–90–8765  Pa
999–09–8720  Pa
999–11–2222  Jer
999–22–3333  Pri
999–33–4444  Ge
999–44–5555  To
999–55–6666  Bri
```

Available Stocks:

```
SUNW
CyAs
DUKE
ABStk
JSVCo
TMAs
BWInc
GMEnt
PMLtd
JDK
```

Figure 4-3 Sample GUI

GUI Design and Coding

Complete the following steps:

1. As a team, design a GUI based on the requirements on the previous page.

2. Compare your design to the sample screen illustrated in Figure 4-3. What problems have you solved?

3. Code the GUI based on your design.

4. Integrate the database classes you created in Module 2 with the new GUI.

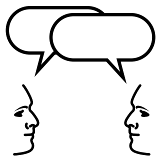
5. Test your GUI and (optionally) present it to the rest of class.

Going Further (Optional)

6. Modify your GUI design so that it can operate as either an applet or an application. What, if any, concerns are there with this approach?

7. Add a password or login screen to your design.

Exercise Summary



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

- Experiences
- Interpretations
- Conclusions
- Applications

Check Your Progress

Before continuing on to the next module, check that you are able to accomplish or answer the following:

- ❑ Apply the principles of good GUI design
- ❑ Differentiate, at a high level, between the *new* Java Foundation Classes (Swing components) and the AWT model
- ❑ Explain how to create the class structure needed for an object-oriented GUI
- ❑ Design and implement a GUI for the BrokerTool project using your choice of containers, components, and layout managers

Think Beyond

Could you write the entire GUI interface as an applet to better enable the BrokerTool program to operate within a Web browser?

How difficult would it be?

What complications would arise?

Networking Connections 5

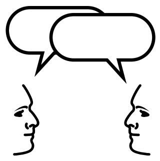
Objectives

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

- Create a class that reads data from a socket connection
- Integrate a class into a GUI application

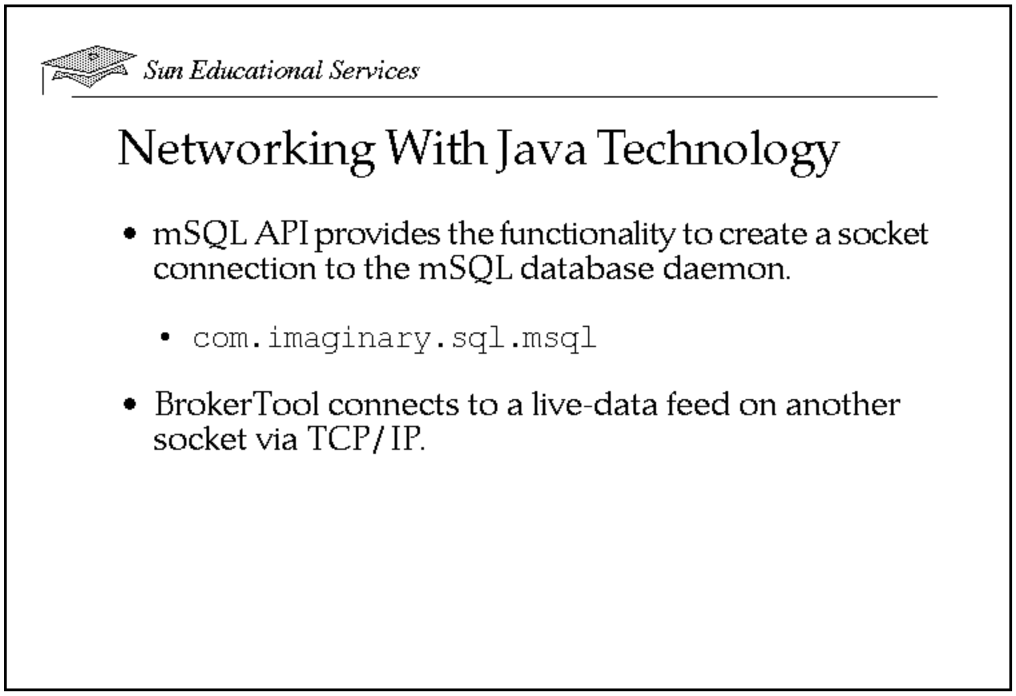
Network programming with sockets is a built-in language feature that makes Java technology powerful for creating network applications. This module explains wrapping functionality as it applies to socket connections for transporting data in usable formats (for example, stock market fractional representations).

Relevance



Discussion – Data packaging with layers of objects is fundamental to computing with networked applications.

What do you hope to accomplish by massaging data at higher and higher levels in the class hierarchy? Why is this important?



Sun Educational Services

Networking With Java Technology

- mSQL API provides the functionality to create a socket connection to the mSQL database daemon.
  - `com.imaginary.sql.msql`
- BrokerTool connects to a live-data feed on another socket via TCP/IP.

Networking With Java Technology

The BrokerTool program uses a socket connection to the mSQL database daemon to transmit queries and receive data. This functionality is provided in the mSQL API (`com.imaginary.sql.msql`). In this module, you integrate another component into the BrokerTool program, one that displays information from another socket that presents live-feed data.

This component receives data by way of another TCP/IP socket when a synchronous request is made. The data provided on the port is a simulated stream of current stock symbols and prices that are dynamically changed every 45 seconds.

You create a class that makes a client socket connection and reads data from the server. The class to display your data is already written. Integrate both classes into your GUI application.

The Live-Feed Application

In addition to a database connection, the host server also includes an application that continuously provides the most recent stock prices. This application simulates a live-feed socket. Data requested from the live-feed represents the most recent stock prices.

In this module, you write class methods that access this data. Use the ticker tape class provided to display the data in the BrokerTool GUI. The ticker tape provides a continuously scrolling data area, as illustrated in Figure 5-1.

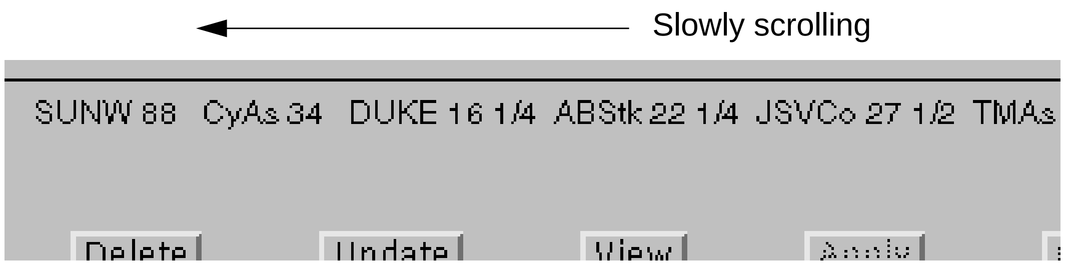


Figure 5-1 Live-Feed TickerTape

The Live-Feed Application Specifications

The live-feed application is located on the same host as the database. This application updates the current price in the `Stock` table every
45 seconds. You can request a current quote from the live-feed application at any time by issuing a string to the live-feed application over a TCP/IP socket connection.

The following describes additional live-feed specifications:

- The TCP/IP port for the live-feed application is port number 5432.
- You can request a quote by issuing any file-system safe, universal character set transformation format (UTF) string to the live-feed application at any time.
- The live-feed application responds by sending a response in the following format:

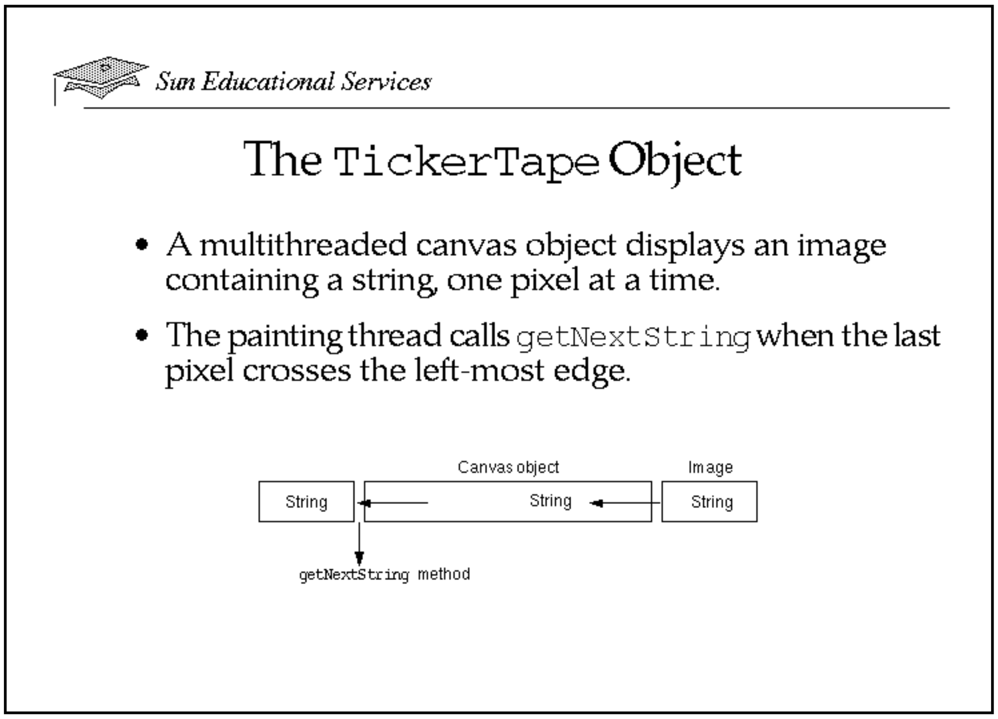
  `int String float String float` ...

  where:

  `int` – The number of symbol/price pairs (header)

  `String` – A UTF string containing the Stock symbol, such as SUNW

  `float` – A floating-point value that is the current Stock price
- The live-feed application updates the `Stock` table in the database at each 45-second interval.
- The live-feed application creates a new socket and thread for each client connection.



Sun Educational Services

The TickerTape Object

- A multithreaded canvas object displays an image containing a string, one pixel at a time.
- The painting thread calls getNextString when the last pixel crosses the left-most edge.

Canvas object
Image
String
String
String
getNextString method

The TickerTape *Object*

The `TickerTape` object is a multithreaded canvas object that is used to display an image containing a string, one pixel at a time. An `Image` object is created, and a string is drawn onto its graphic context. The `Image` object is "stepped" across the `Canvas`, one pixel at a time, from right to left. When the last pixel crosses the left-most edge, the thread that controls the painting of the Canvas calls the `getNextString` method.

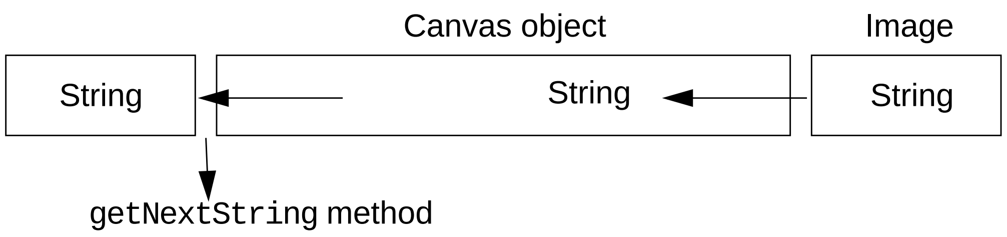


Figure 5-2 `TickerTape` Overview

You must implement the class that establishes the socket connection with the live feed and supplies the `getNextString` method.

Figure 5-3 illustrates the flow of the `TickerTape` object.

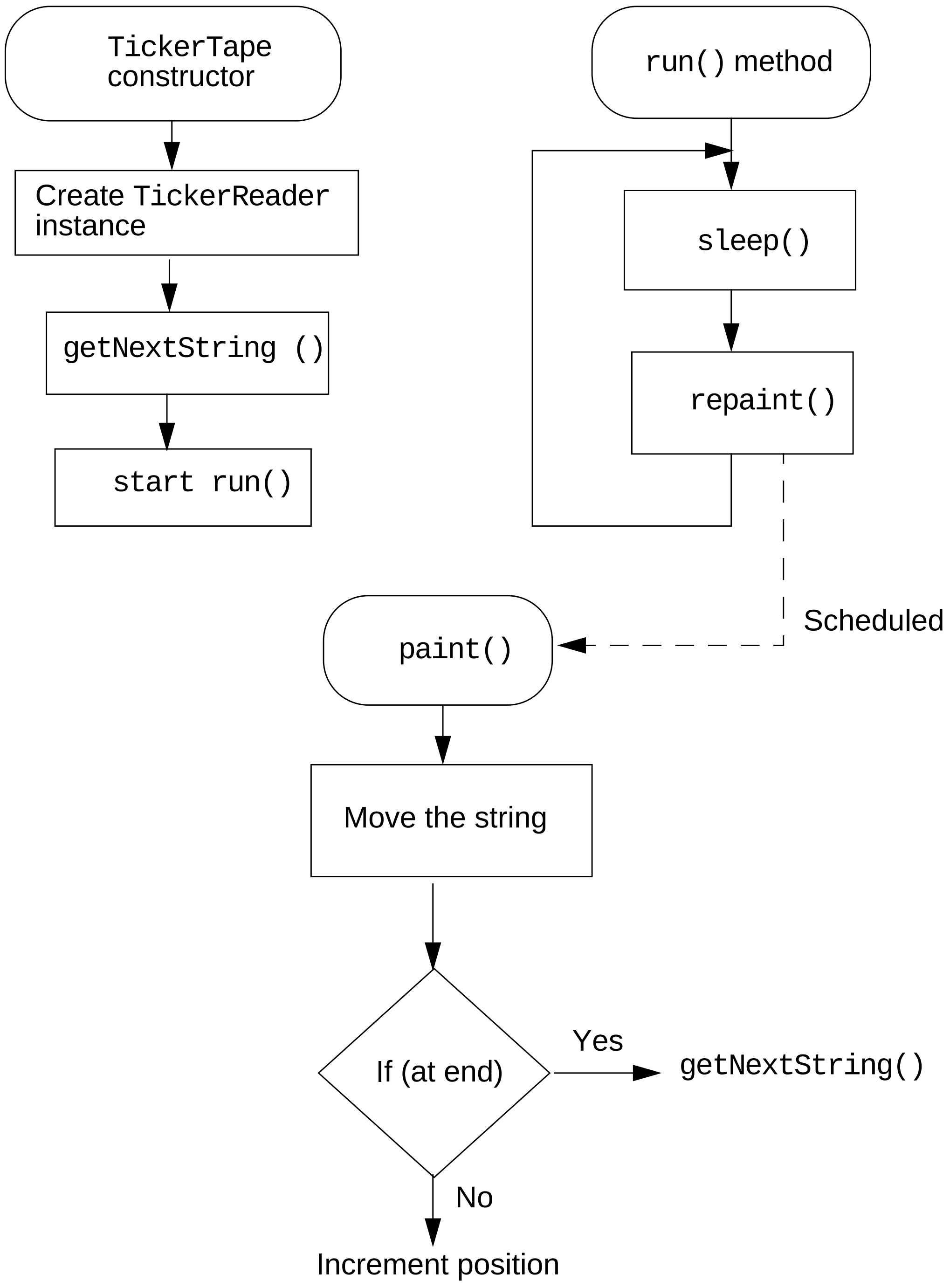


Figure 5-3 `TickerTape` Flow Chart

Adding the `TickerTape` *Object*

The BrokerTool GUI needs a placeholder for the ticker-tape component, so initially you create an instance of that component for BrokerTool. The API for a `TickerTape` class is provided in the following sections. This class enables you to scroll a string slowly across a `Canvas` object.

The `TickerTape` object must obtain text strings from a class that you will write in the lab. A sample of the API for a recommended class is provided, as well as a utility class that converts floating-point numbers to a ticker-tape format.

Class Hierarchy

The `TickerTape` class inherits from the `Canvas` class. The `Runnable` interface is implemented so that the `TickerTape` instances can be executed by a thread.

```
class TickerTape extends Canvas implements Runnable
```

Instance Variables

The `TickerTape` class provides several instance variables, but the most important ones are highlighted as follows:

- `TickerReader tickerHost` – An object that represents the socket connection manager between the live-feed application and `TickerTape`.

- `private static final int TICKERHOSTPORT` – The port that the `TickerReader` connects to on the live-feed server is number 5432.

The `TickerTape` *Class*

You use the following methods to access the `TickerTape` class.

Methods

`public TickerTape (String feedhost, int width)`

Return type:	`(None) – Constructor`
Exceptions:	
Arguments:	`String` *feedhost*`, int` *width*
What it does:	Constructs a `TickerTape` object that communicates on the live-feed port to *feedhost* and has an initial size of *width* in pixels (height is fixed by the height of the font).

`public String getNextString ()`

Return type:	`String`
Exceptions:	
Arguments:	
What it does:	Gets the next string (formatted for display) from the live-feed by calling the `TickerReader method, readData`.

`public void setupTape ()`

Return type:	`void`
Exceptions:	
Arguments:	
What it does:	Adjusts the size of the ticker tape after a resize (for example, the applet resize).

`public void paint (Graphics g)`

Return type:	`void`
Exceptions:	
Arguments:	`graphics g`
What it does:	Overloads the `Canvas` paint method. In this method, the String is drawn one pixel at a time, starting with the string at the far right edge of the `Canvas`. This method illustrates the use of graphics buffering for smooth animation of the ticker tape.

`public void run ()`

Return type:	`void`
Exceptions:	
Arguments:	
What it does:	Controls the thread body. The timing of the animated scrolling is controlled by sleeping for a period of time, then calling the `repaint` method.

`public void close ()`

Return type:	`void`
Exceptions:	
Arguments:	
What it does:	Calls the TickerReader method `closePort` to close the socket.

The `MakeFraction` *Class*

This class resembles the `java.lang.Math` class. Its constructor is private, and each of its methods is declared as static. This enables you to use the class through the class name without having to create an instance of the class. `MakeFraction` cannot be subclassed.

Class Hierarchy

```
final class MakeFraction
```

Methods

```
public static String convertToFraction (float num)
```

Return type:	`String`
Exceptions:	
Arguments:	`float` *num*
What it does:	Converts a floating point representation of a number into its ticker-tape format. For example, 80.25 is converted to a String containing "80 1/4"

```
public static String convertToFraction (String num)
```

Return type:	`String`
Exceptions:	
Arguments:	`String` *num*
What it does:	Converts a string representation of a floating-point number into a fraction by converting *num* into a float, and then calling the previous `convertToFraction` method.

The `TickerReader` *Class*

This is the class that you will implement in the lab. In the following API, some of the methods are provided.

The following are the instance variables in the `TickerReader` class.

- `Socket tickerSocket`
- `DataInputStream recvStream`
- `DataOutputStream sendStream`
- `String symbol [ ]`
- `float price [ ]`

Methods

The following are the methods of the `TickerReader` class.

`public TickerReader (String feedhost, int port)`

Return type:	`(None) – Constructor`
Exceptions:	
Arguments:	`String` *feedhost*`, int` *port*
What it does:	Saves local copies of *`feedhost`* and *port*. Call `makeConnection`.

`public void makeConnection(String hostname, int port)`

Return type:	`void`
Exceptions:	
Arguments:	`String` *hostname*`, int` *port*
What it does:	Opens a connection to the socket at *`hostname`* and *port*, and creates both an input and output stream object.

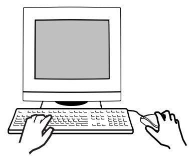
`public String readData ()`

Return type:	`String`
Exceptions:	
Arguments:	
What it does:	Requests symbol/price pairs from the live-feed application, formats each symbol/price pair, and concatenates them into a single string which is passed back to the ticker tape. Returns `null` if connection to the live- feed fails.

`public void closePort ()`

Return type:	`void`
Exceptions:	
Arguments:	
What it does:	Closes the connection with the live-feed application.

Exercise: Creating the `TickerReader`



Exercise objective – Given the specifications for the `TickerReader` class and its methods, write the code to implement them.

Preparation

Review the code for the `TickerTape` class in your `labfiles/mod4` directory.

Tasks

Complete the following steps:

1. Discuss with your team how you should make the connection.
2. Review the existing `MakeFraction` class.
3. Write your own `TickerReader` class, and implement the following methods:
   - ▼ `TickerReader` constructor
   - ▼ `makeConnection` (or equivalent)
   - ▼ `readData` method
   - ▼ `closePort` method
4. Build a test application to test the `TickerReader` class directly from the `mod5` directory. Make the test application output three lines of data read from the socket.
5. Test the `TickerReader` class with `TickerTape` and `MakeFraction` in a test applet.
6. (Optional) Integrate these classes into your GUI.
7. (Optional) Write the `MakeFraction` class from the provided API.

Going Further

Consider what happens when the connection with the live-feed socket goes down. How can you write a method or class that will reconnect the socket when it becomes available?

8. Create a method or class that periodically polls the connection and reconnect when the live-feed application is once again available.

9. Ask the instructor to take the live feed off line when you are ready to test your code, or write a `close` method that simulates the connection terminating.

10. Modify `TickerTape` so that a mouse click in the `Canvas` causes `TickerTape` to obtain the latest stock report data from the live-feed.

11. Modify your GUI to update the stock prices currently displayed elsewhere on the screen when a String is read from the `TickerReader`.

Exercise Summary

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

- Experiences
- Interpretations
- Conclusions
- Applications

Check Your Progress

Before continuing on to the next module, check that you are able to accomplish or answer the following:

- ❑ Create a class that reads data from a socket connection
- ❑ Integrate a class into a GUI application

Think Beyond

How would you create a daemon called `StockTicker` that provides the stock information on a port? How would you make the data random?

Multiple-Tier Database Design 6

Objectives

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

- Describe one-, two-, and three-tier database architectures
- Explain the issues related to implementing a three-tier design
- Create a multiple-tier Java applet or application

In this module, you will explore the issues associated with one-, two-, and multiple-tier database design.

Relevance

Discussion – *N*-tier systems allow for better flexibility and maintenance of a software system. The most common one is *three-tier*. How do the components of a three-tier system interact with each other?

What is the deciding factor for these divisions?

Additional Resources

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

- Balick, Fritzinger, and Siegel. 1996. *Effective 3-tiered Engineering.* SunSoft, Sun Microsystems Inc.
- Appendix F, "Object Serialization"

The Tiered Database Model

The BrokerTool program is based on a two-tier database design. An upcoming section discusses why modifying the BrokerTool program to a three-tier database design might be desirable. The following is a brief discussion of the tier model of databases.

One-Tier Databases

Originally, databases were written as a single unit with both the database engine and the user interface tightly coupled.

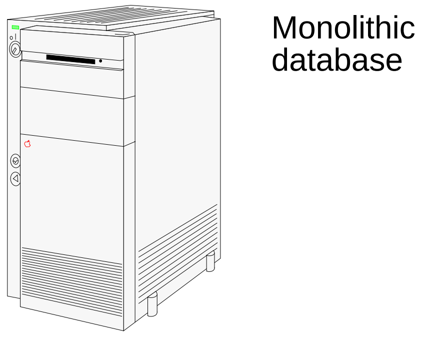


Figure 6-1 The One-Tier Database Model

There are two disadvantages to the design illustrated in Figure 6-1. The one-tier database is:

- Not readily extensible
- Not easily accessible through a network

Two-Tier Databases

The two-tier model addresses the one-tier design limitations by separating the database front end from the database engine.

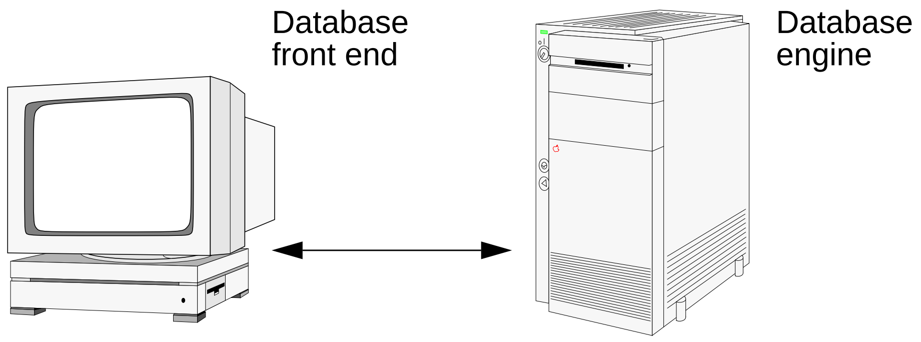


Figure 6-2 The Two-Tier Database Model

This enables the data to reside locally or remotely. If other users need a different interface to the same data, you need to develop only a database front end.

There are some disadvantages to this design; issues arise if the decision is made to add any of the following to the database:

- Mirroring
- Caching
- Proxy services
- Secure transactions

While you can add functionality to the database engine, this tends to result in database engines that are feature rich. That is, if the database you are implementing requires only mirroring, you might not want the extra baggage associated with the other functionalities.

Three-Tier Databases

The three-tier database design resolves all of the issues that a two-tier design addresses without the limitations associated with a two-tier database.

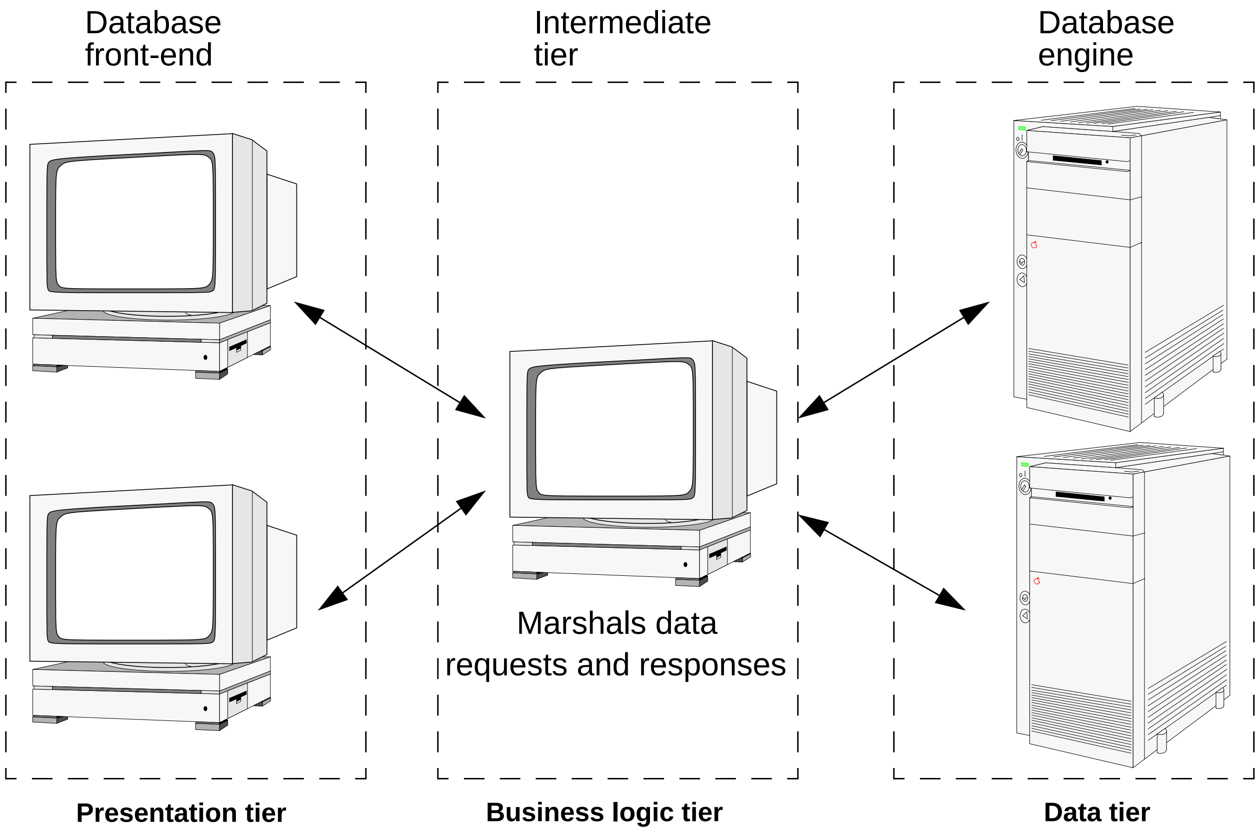
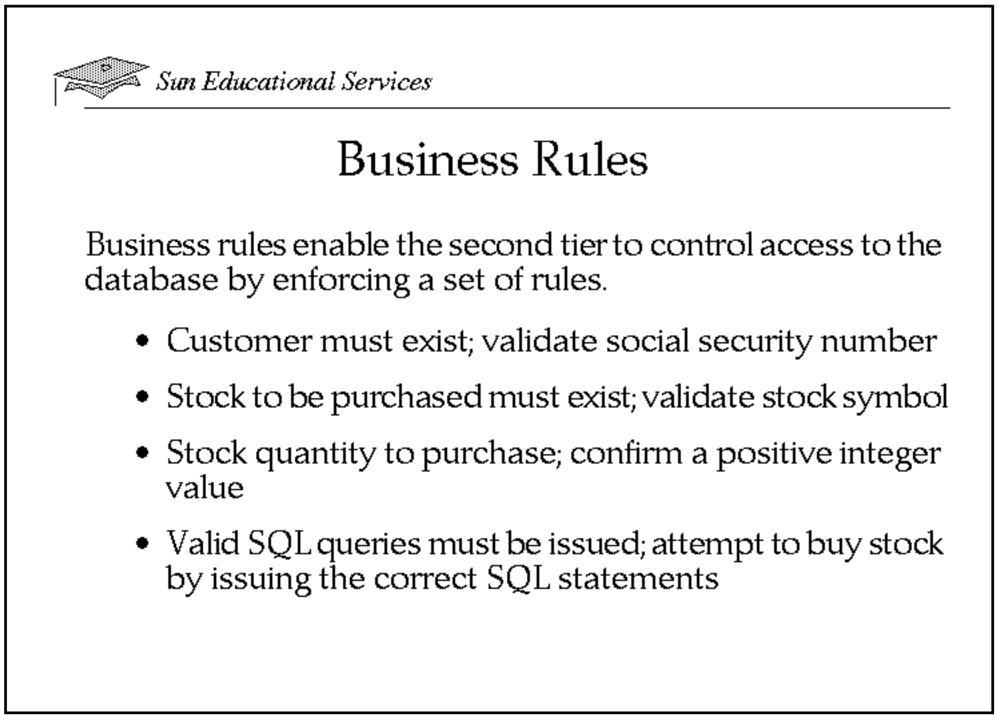


Figure 6-3 The Three-Tier Database Model

Functionality, such as mirroring and secure transactions can be introduced through the intermediary tier. Additionally, the intermediary tier can be tailored to your needs. The three tiers are:

- Presentation – This tier receives the data and data processing requests.
- Business logic – This tier implements business rules.
- Data – This tier stores and allows access to the data.


Sun Educational Services

Business Rules

Business rules enable the second tier to control access to the database by enforcing a set of rules.

- Customer must exist; validate social security number
- Stock to be purchased must exist; validate stock symbol
- Stock quantity to purchase; confirm a positive integer value
- Valid SQL queries must be issued; attempt to buy stock by issuing the correct SQL statements

Business Rules

Until now, you have probably created code to do error checking on the client side of the BrokerTool program. This is because the server (mSQL) does not provide any error checking. The database cannot recognize negative stock values or invalid social security number tables.

Business rules enable the second tier to control access to the database by enforcing a set of rules. For example, consider what kind of error checking would be required to purchase stock:

- Customer must exist; validate social security number
- Stock to be purchased must exist; validate stock symbol
- Stock quantity to purchase; confirm a positive integer value
- Valid SQL queries must be issued; buy stock by issuing the correct SQL statements

Moving the enforcement of the rules to the middle tier, between the client and database, enables the rules to be created once and then kept in a single, easily maintained location.

The BrokerTool Program

The BrokerTool program is built upon a two-tier database design, as illustrated in Figure 6-4.

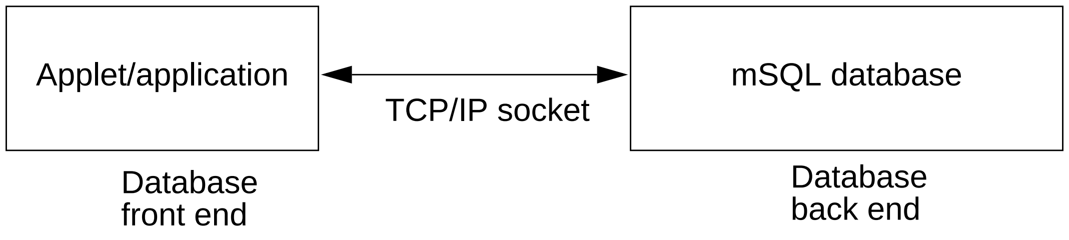


Figure 6-4 The Two-Tier Database Design

Caching of database queries can be supported by introducing the caching functionality into the database front end (for example, Java technology) or the database engine (for example, mSQL).

If you decided to switch to a three-tier design and introduce caching into the intermediate tier, your front-end application can remain unchanged. The Java applet or application communicates with an intermediate layer of code. You can write code in the Java programming language and use native methods, or you can write in C or C++. This new tier enables you to develop your own functionality: secure socket classes, cache data, or implement business rules.

Figure 6-5 illustrates the three-tier database design.

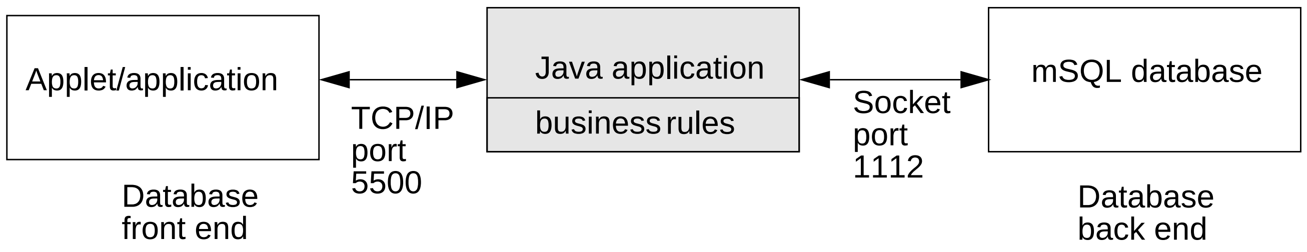


Figure 6-5 The Three-Tier Database Design

The Three-Tier Database Design

In this course, to insulate or abstract the client program from the database, another tier is going to be created. This new tier receives commands that are specific to the BrokerTool program. For example, you know that the current program supports the following functions:

- Buy shares
- Sell shares
- Get a list of stock symbols and prices
- Get the current price for a particular stock
- Get a list of all the customer names and social security numbers (SSNs)
- View a customer, including the customer's portfolio
- Add a new customer
- Update an existing customer
- Delete an existing customer

The middle tier receives these commands and executes the appropriate business rules. A status that includes an error code or argument list is returned.

This middle-tier application introduces the concept of a *protocol handler* in Java technology. The design precipitates a new protocol that the middle tier receives and interprets. In this case, it converts the new command protocol to SQL using the appropriate business rules.

Figure 6-6 illustrates the modified two-tier database design.

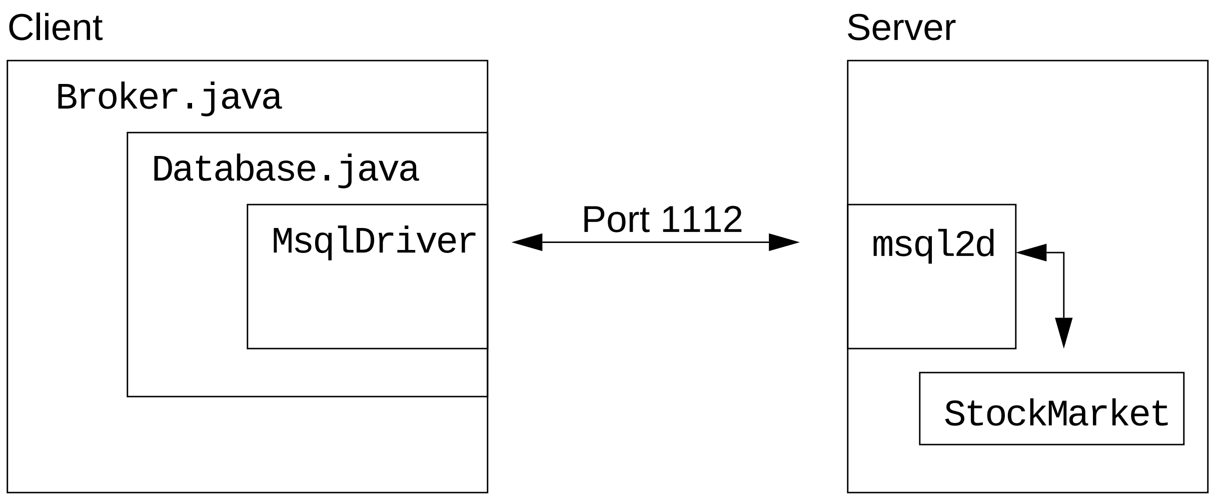


Figure 6-6 The Modified Two-Tier Database Design

Figure 6-7 illustrates the modified three-tier database design.

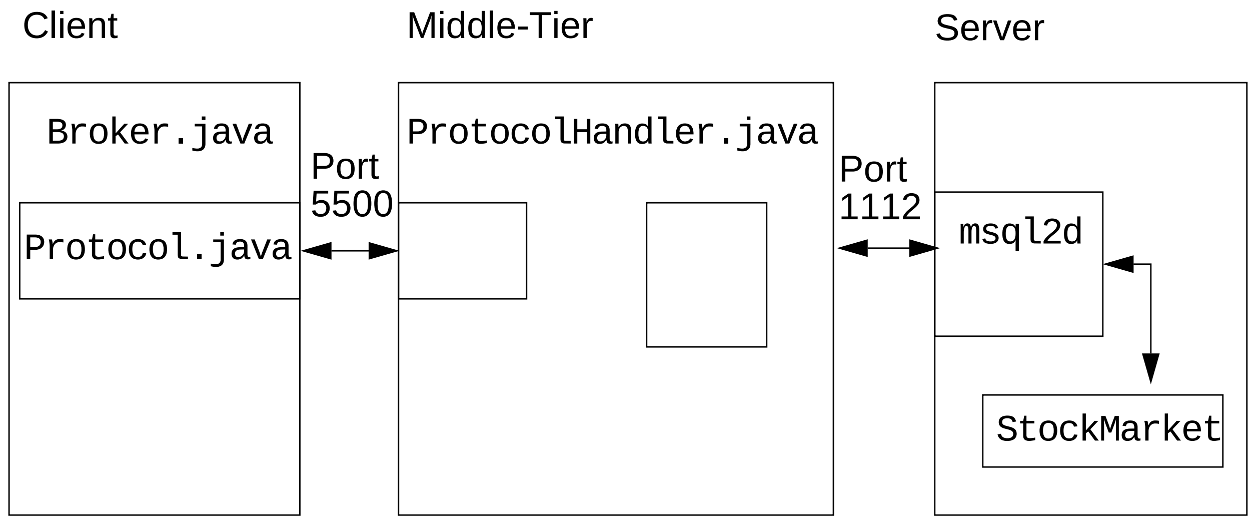
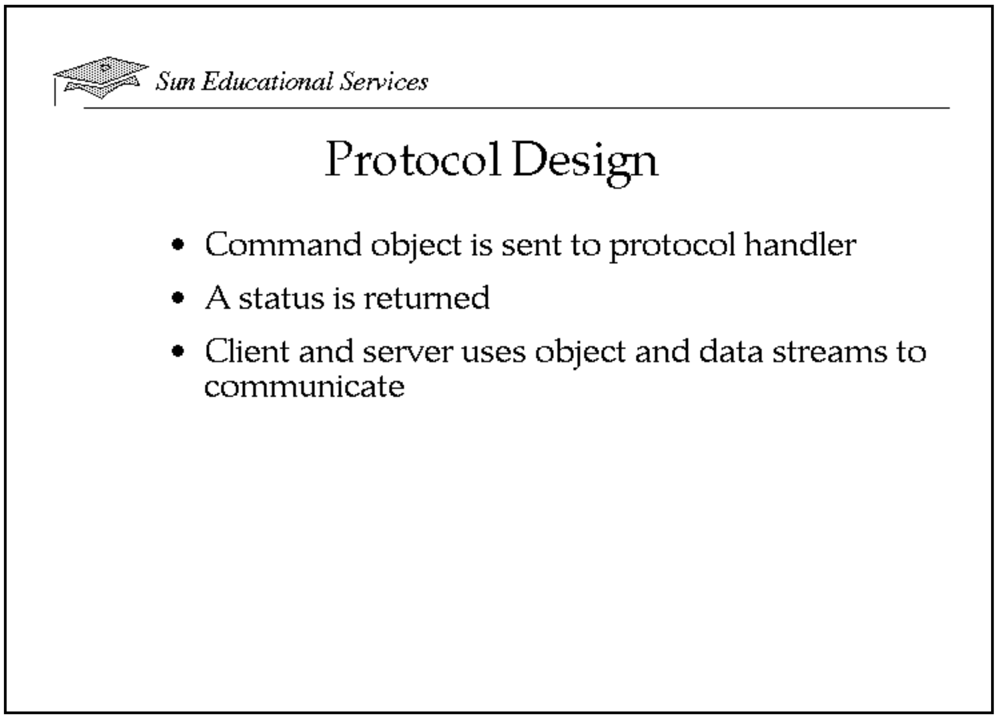


Figure 6-7 The Modified Three-Tier Database Design



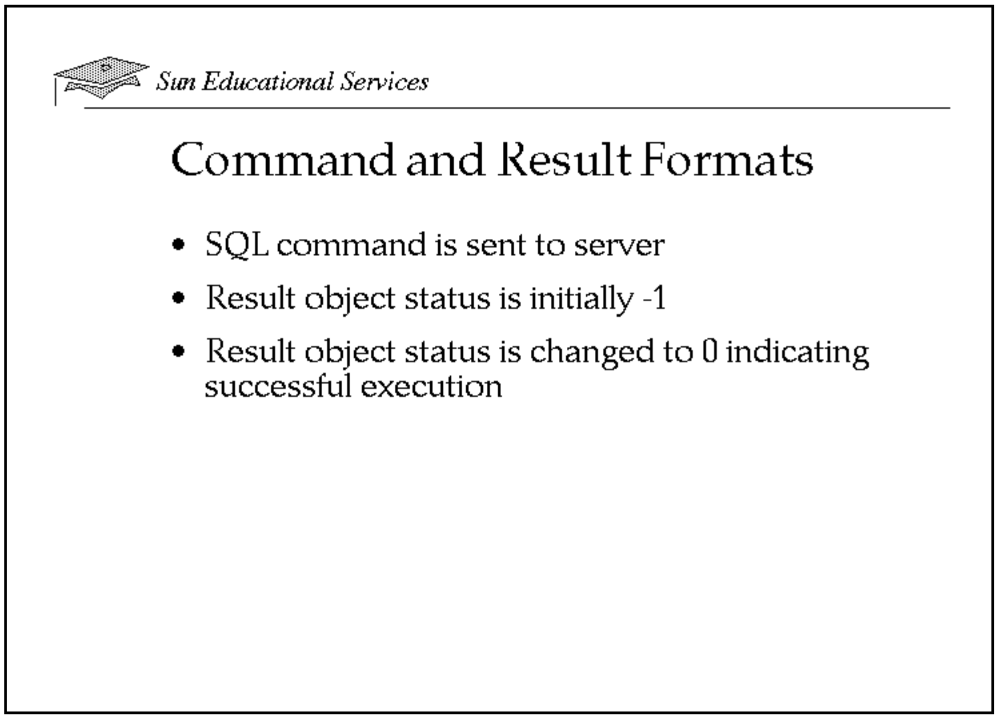
Sun Educational Services

Protocol Design

- Command object is sent to protocol handler
- A status is returned
- Client and server uses object and data streams to communicate

Protocol Design

For the purposes of this course, the protocol of the middle tier is simple and straightforward. A command object is sent by way of an `ObjectOutputStream` associated with a TCP/IP socket to the middle tier, and a status of some kind is expected on the other end. Both the client and server use object and data streams to handle communication of the commands and status.



Sun Educational Services

Command and Result Formats

- SQL command is sent to server
- Result object status is initially -1
- Result object status is changed to 0 indicating successful execution

Command and Result Formats

Command objects are sent to the protocol handler in the middle tier. The resulting SQL command is executed by the middle tier, and the `Command` object is returned to the client with the appropriate modified `Result`. Each `Command` instance indicates the type of request being made by having an instance of a `String` array pass arguments, and a `Result` to extract return values.

The `Result` object is created in the `Command` constructor, and has an initial status of -1, indicating an error occurred. If all goes well in the middle tier (no exceptions are thrown as a result of the query), the status field of the `Result` instance is changed to 0, indicating that the command was executed successfully.

Result.java

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

public class Result extends Vector
   implements Serializable {

   // Default to error status
   private int status = -1;

   public Result () {
       super(1,1);
   }

   public int getStatus () {
       return status;
   }

   public void setStatus(int value) {
       status = value;
   }

   public int getNumRows () {
       return this.size();
   }
}
```

Command.java

```
import java.io.*;

public class Command implements Serializable {

   private int commandValue;
   private Result results;
   private String [] args = {""};

   public static final int BUYSHARES = 10;
   public static final int SELLSHARES = 20;
   public static final int READSTOCKLIST = 30;
   public static final int READASTOCK = 40;
   public static final int READCUSTLIST = 50;
   public static final int VIEWACUSTOMER = 60;
   public static final int ADDACUSTOMER = 70;
   public static final int UPDATECUSTOMER  = 80;
   public static final int DELETECUSTOMER  = 90;

   // Constructor - takes an int that represents
   // the type of action that was requested from
   // the GUI
   public Command (int comm) {
       commandValue = comm;
       results = new Result();
   }

   // Determine which type of action was requested
   // from the GUI
   public int getCommandValue() {
       return commandValue;
   }

   // Assign any args to pass to the SQL statement
   public void setArgs(String [] params) {
       args = params;
   }

```

```
    // Get any args to pass to the SQL statement
    public String [] getArgs() {
        return args;
    }

    // Get result returned from DB transaction
    public Result getResult() {
        return results;
    }
}
```

Command Implementation

Each command requires certain arguments. For example, the Buy Shares command would consist of the following:

`Command (int)`	`args[0]= ssn`	`args[1]= symbol`	`args[2]= quantity`
10	999-55-3434	SUNW	100

Status

Every command returns a status associated with its `Result` instance. For example, the Buy Shares command could return the following status values:

- 0 – OK
- -1 – Customer not found
- -2 – Invalid stock symbol
- -3 – Invalid quantity

Note – If the middle tier receives a command that it does not recognize, it returns a command setting the result status to -4.

The Buy Shares Command

`Command (int)`	`commandValue`
`10`	
Social security number	`args[0]`
Stock symbol	`args[1]`
Quantity shares	`args[2]`

This command returns one of the following statuses:

- 0 – OK
- -1 – Customer not found
- -2 – Invalid stock symbol
- -3 – Invalid quantity

The Sell Shares Command

`Command (int)`	`commandValue`
`20`	
Social security number	`args[0]`
Stock symbol	`args[1]`
Quantity shares	`args[2]`

This command returns one of the following statuses:

- 0 – OK
- -1 – Customer not found
- -2 – No shares owned
- -3 – Invalid quantity
- -5 – Invalid stock symbol

The Read Available Stock List Command

`Command (int)`	`commandValue`
`30`	

This command returns one of the following statuses:

- 0 – OK (results Vector contains one element, an array of `StockRec`)
- -1 – Error

The Read a Specific Stock Price Command

`Command (int)`	`commandValue`
`40`	
Stock symbol	`args[0]`

This command returns one of the following statuses:

- 0 – OK (results Vector contains one element, the price, stored in a `Float` object)
- -1 – Error

The Read Available Customer List Command

`Command (int)`	`commandValue`
`50`	

This command returns one of the following statuses:

- 0 – OK (results Vector contains one element, an array of `CustomerRec`)
- -1 – Error

The View a Customer Record Command

`Command (int)`	`commandValue`

`60`
`Social security number`
	
	`args [0]`

This command returns one of the following statuses:

- 0 – OK (results Vector contains one element, a `CustomerRec`)
- -1 – Customer not found

The Add a New Customer Record Command

Command (int)	commandValue
70	
Social security number	args[0]
Name	args[1]
Address	args[2]

This command returns one of the following statuses:

- 0 – OK
- -1 – This social security number already exists in the database

The Update a Customer Record Command

Command (int)	commandValue
80	
Social security number	args[0]
Name	args[1]
Address	args[2]

This command returns one of the following statuses:

- 0 – OK
- -1 – Customer not found

The Delete a Customer Record Command

`Command (int)`	`commandValue`
`90`	
Social security number	`args[0]`

This command returns one of the following statuses:

- 0 – OK
- -1 – Customer not found

Exercise: Implementing the Protocol

Exercise objective – Given the specifications for the `Database` class, implement the methods in this class.

Preparation

Back up your `Database.java` file by copying it to another file, for example, `Database.java.orig`. Then, using the same method signatures (and retaining any error checking not explicitly done by the protocol handler), strip out all references to `java.sql`. Create `Command` instances to pass your database queries and updates to the middle tier through a TCP socket connection.

Tasks

Complete the following steps:

1. Create a class that implements the protocol as it is described in this module and replaces the existing `Database` class. If you name your new class anything other than `Database.java`, remember to make the appropriate changes in your GUI code to reference this class.

2. Use the same method signatures that are in your original `Database` class (retain any error checking not explicitly done by the protocol handler). Strip out all references to `java.sql`. Create *Command* instances to pass your database queries and updates to the middle tier through a TCP socket connection.

3. Modify your code to implement the new class. Remember that each of the classes, `CustomerRec.java`, `SharesRec.java,` and `StockRec.java`, now need to implement the `java.io.Serializable` interface to be passed across the object stream.

4. Test your new BrokerTool program.

5. Prepare a short report.

Going Further (Optional)

6. Create a transaction system by modifying both the `Protocol` and `ProtocolHandler` class.

7. Keep the current command protocol, but do not “commit” a transaction until a command + 1 is sent.

   For example, to commit an Add Customer transaction, send the 70 command string followed by a 71 command.

8. Allow “rollbacks” by sending a negative command. To rollback the Add Customer, send -71.

Notes

Exercise Summary

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

- Experiences
- Interpretations
- Conclusions
- Applications

Check Your Progress

Before continuing on to the next module, check that you are able to accomplish or answer the following:

- ❑ Describe one-, two-, and three-tier database architectures
- ❑ Explain the issues related to implementing a three-tier design
- ❑ Create a multiple-tier Java applet or application

Think Beyond

In implementing additional *n*-layers of abstraction, you add to the amount of design and coding (and the project costs). At what point does the real return on investment come for the additional engineering? How can you estimate the break-even point?

Porting Considerations and Wrap-Up 7

Objectives

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

- Enumerate issues involved in porting from a Solaris Operating Environment to a Windows platform and from a Windows to a Solaris Operating Environment platform.
- Demonstrate the success of your modular design by building hybrid applications, mixing server and client modules
- Debug problems when interchanging modules
- Describe how design decisions made in program design now affect the extensibility of the application

This module covers issues relating to porting between the Solaris Operating Environment and Windows platforms.

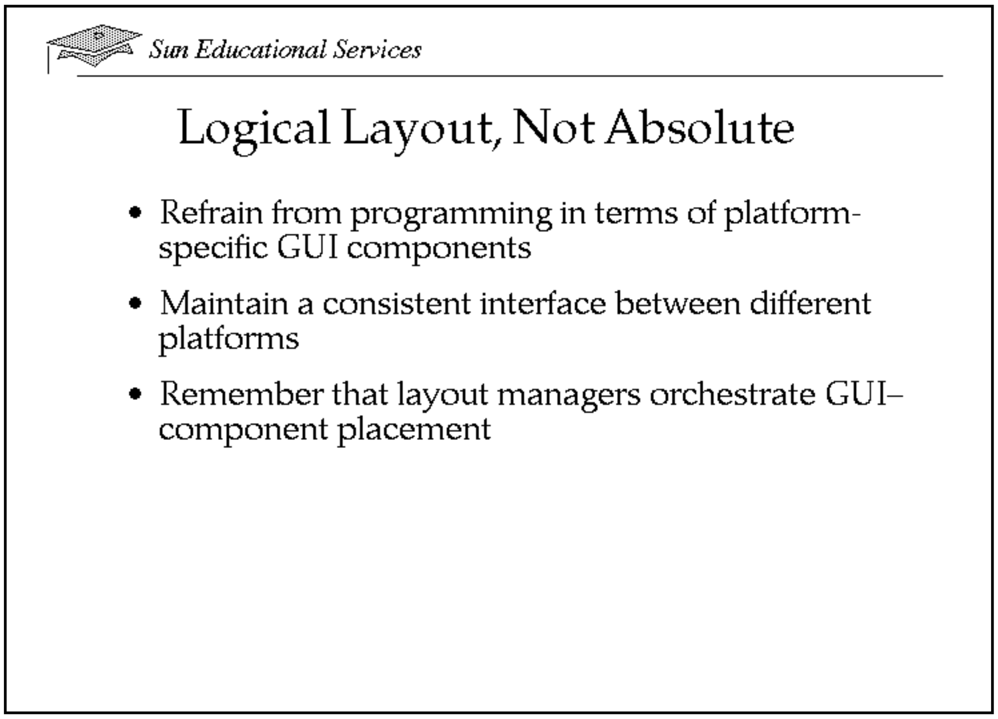
Relevance

Discussion – While the 100% Pure Java™ technology paradigm means you can port bytecodes successfully across platforms, what other issues can you expect to face?

Solaris Operating Environment-to-Windows Porting Issues

While Java technology tries to maintain the Write Once, Run Anywhere™ paradigm, some of the issues to remember when porting from a Solaris Operating Environment to a Windows platform and from a Windows platform to a Solaris Operating Environment are the following:

- Logical layout, not absolute
- File access
- Font availability
- Mouse buttons
- Threads
- Platform-specific implementation bugs


Sun Educational Services

Logical Layout, Not Absolute

- Refrain from programming in terms of platform-specific GUI components
- Maintain a consistent interface between different platforms
- Remember that layout managers orchestrate GUI–component placement

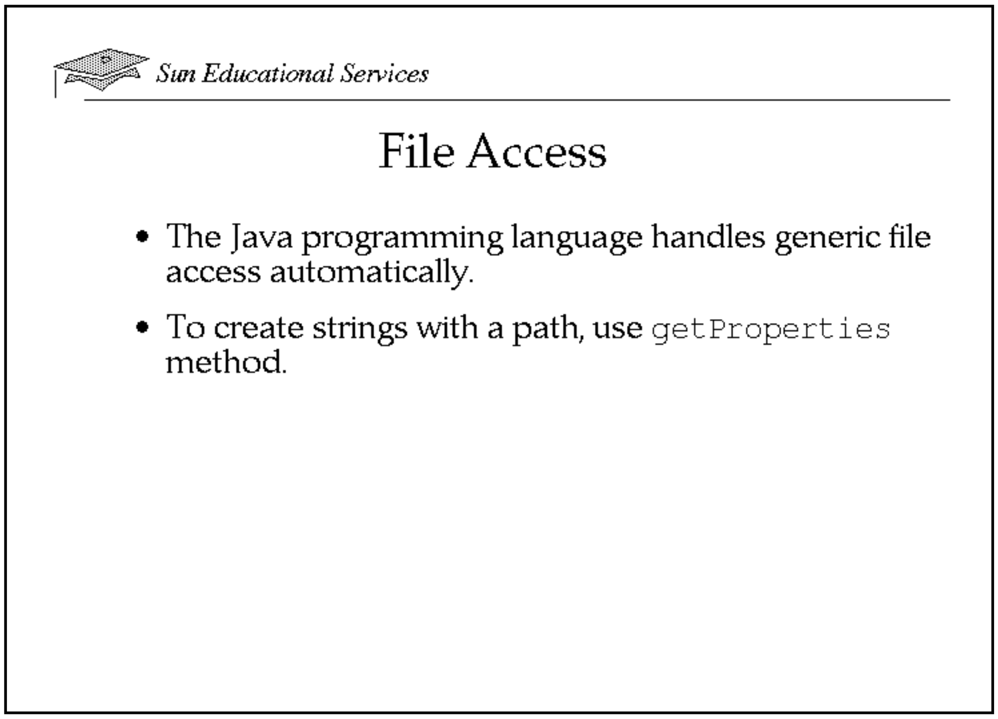
Logical Layout, Not Absolute

When writing for both the Solaris Operating Environment and Windows platforms, refrain from thinking in terms of either Motif or Windows GUI components. You are not programming in Motif or Windows. You are programming in the Java programming language.

The size, shape, and appearance of buttons, text fields, lists, and so on, will vary between the two platforms. Solaris Operating Environment has Motif buttons, whereas Windows has Windows buttons. This, of course, is by design. The intent is to maintain a consistent interface on the respective platforms.

While you might be accustomed to placing GUI components using absolute layout (that is, using absolute coordinates) in other windowing systems, GUI-component placement in Java technology is orchestrated by layout managers.

The variance in the shape and size of each platform's GUI components is part of the reason behind the logical layout design decision. The need for logical rather than absolute layout becomes apparent when you consider the wide variance in monitor sizes on which your Java code can run.



Sun Educational Services

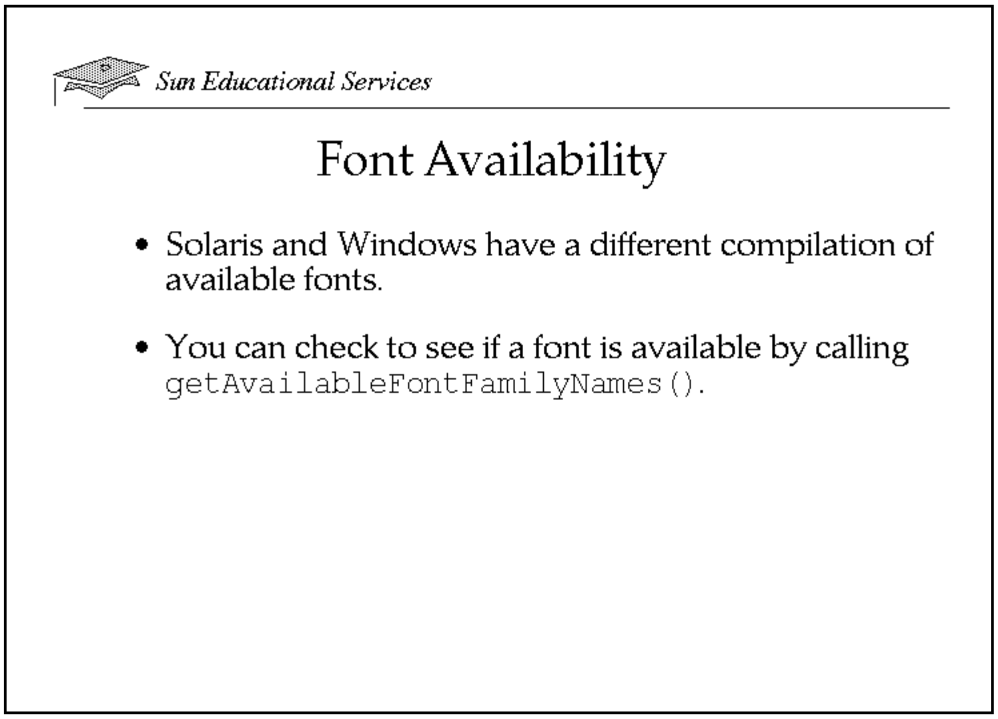
File Access

- The Java programming language handles generic file access automatically.
- To create strings with a path, use `getProperties` method.

File Access

Solaris Operating Environment file-naming conventions differ from those of Windows. Whereas the Solaris Operating Environment uses the forward slash (/) as the path separator, Windows uses the back slash (\).

Generic file access is handled automatically by the Java programming language. However, if you are doing something, such as creating strings that contain the path name, use the `getProperties` method to retrieve the path separator.



Sun Educational Services

Font Availability

- Solaris and Windows have a different compilation of available fonts.
- You can check to see if a font is available by calling `getAvailableFontFamilyNames()`.

Font Availability

Fonts available in the Solaris Operating Environment might not be available on a Windows platform and fonts for Windows might not be available in Solaris Operating Environment. Do not assume that fonts available on one platform are available on the other. Always check the availability of a font before attempting to use it. You can get a list of available fonts by calling `getAvailableFontFamilyNames()`. This method returns an array containing the names of all font families available in this `GraphicsEnvironment`.

Mouse Buttons

Due to the variety of mouse-types (one-, two-, and three-button devices), Java technology enables you to hide the mouse functionality by generating a mouse event for any mouse key. However, you can determine the number of the button that was pressed by examining the modifier portion of the event generated.

The `java.awt.event.InputEvent` class defines the input masks associated with each mouse button press. Figure 7-1 illustrates the event mask that is associated with a mouse click on a Solaris Operating Environment system using a type 5c keyboard.

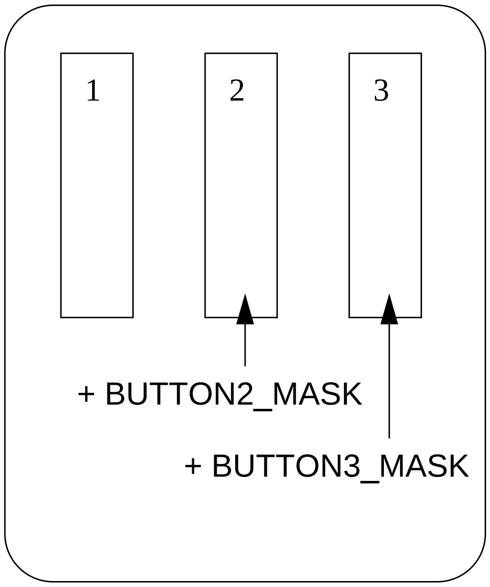
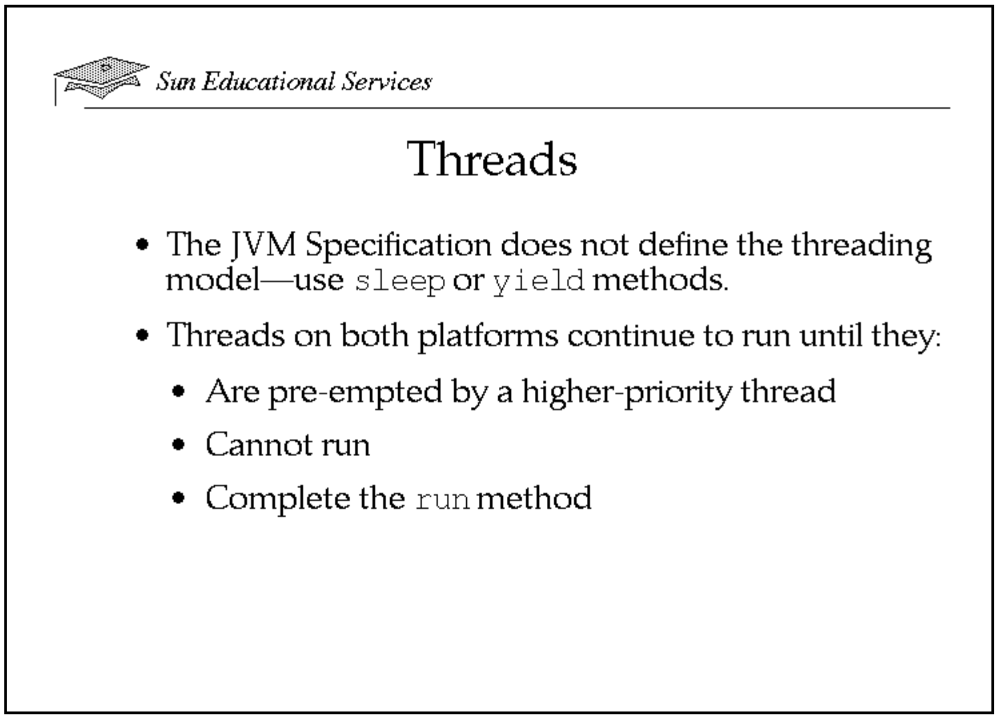


Figure 7-1 Mouse Buttons

You can use the following test code to determine what event masks you would receive on the platform you are using.

MouseTest.java

```
import java.awt.*;
import java.awt.event.*;

public class MouseTest extends Canvas {

   public static void main (String args[]) {

      MouseTest mt = new MouseTest ();
      Frame f = new Frame ("mouse test");
      f.add(mt, BorderLayout.CENTER);
      f.pack();
      f.setVisible (true);

   }

   public MouseTest () {
      setSize (100, 100);
      addMouseListener (new MouseHandler());
   }  // Inner class
   private class MouseHandler extends MouseAdapter {
      public void mousePressed (MouseEvent me) {

         System.out.println ("Mouse pressed");

         if ((me.getModifiers() & InputEvent.BUTTON1_MASK) != 0) {
            System.out.println ("Mouse button 1 pressed");
         }
         if ((me.getModifiers() & InputEvent.BUTTON2_MASK) != 0) {
            System.out.println ("Mouse button 2 pressed");
         }
         if ((me.getModifiers() & InputEvent.BUTTON3_MASK) != 0) {
            System.out.println ("Mouse button 3 pressed");
         }
         if ((me.getModifiers() & InputEvent.CTRL_MASK) != 0) {
            System.out.println ("Mouse button and
               Control key pressed");
         }
         if ((me.getModifiers() & InputEvent.META_MASK) != 0) {
            System.out.println ("Mouse button and
               Meta key pressed");
         }
         if ((me.getModifiers() & InputEvent.SHIFT_MASK) != 0) {
            System.out.println ("Mouse button and
               Shift key pressed");
         }
      }
   }
}
```



Sun Educational Services

Threads

- The JVM Specification does not define the threading model—use `sleep` or `yield` methods.
- Threads on both platforms continue to run until they:
  - Are pre-empted by a higher-priority thread
  - Cannot run
  - Complete the `run` method

Threads

The JVM Specification does not define the threading model, nor does the Java runtime environment (JRE) provide methods for determining the model under which you are running. The only genuinely portable solutions involve making no assumptions, and using the `sleep` or `yield` methods.

Under a Solaris Operating Environment, once a thread is running it continues to run until it is pre-empted by a higher-priority thread, it becomes "not runnable," or its `run` method is completed.

Under a Microsoft Windows platform, the JVM uses Windows threads. Like Solaris Operating Environment, threads running at a given priority are pre-empted by threads running at a higher priority. Similarly, under Windows, a thread runs until it is pre-empted by a higher priority thread, it becomes "not runnable", or its `run` method is completed. However, unlike the Java scheduler for Solaris Operating Environment, threads of equal priority are time-sliced.

Other Java-enabled browsers, such as Netscape Navigator, use their own threading packages. In instances such as this, you might not know how threads are implemented. Make no assumptions, use the `sleep` or `yield` methods.

Platform-Specific Implementation Bugs

Keep abreast of the latest information on platform-specific issues. The best resource for this information is found by searching JavaSoft's home page:

`http://www.javasoft.com`

Making the BrokerTool Program Fully Functional

The BrokerTool application has additional functionality not covered in this module. To understand this functionality, you should discuss the following questions.

- What issues must be kept in mind when porting Java programs between platforms?
- What modifications are required to allow the BrokerTool program to be run as an applet or as an application?
- How have the original design decisions affected the extensibility of your BrokerTool program?
- What modifications should be made in the following:
    - Original design?
    - GUI module?
    - Query processing module?
    - Network connection module?

Exercise: Integrating Modules

Exercise objective – Given modules written by other students in the class, create a hybrid application and locate potential integration problems.

Preparation

All of the modules in the previous exercises should be completed before beginning this exercise.

Tasks

Complete the following steps:

1. Exchange the GUI or database module(s) with those developed by your classmates, and create your own hybrid applications.

2. Mix both server and client modules.

If you run into unexpected integration problems, view this not as a failure but as an opportunity to learn from hands-on experience. Work in conjunction with the classmate who developed the other module, debug it, and be prepared to report back to the class what you learned.

Exercise Summary

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

- Experiences
- Interpretations
- Conclusions
- Applications

Check Your Progress

Before continuing, check that you are able to accomplish or answer the following:

- ❑ Enumerate issues involved in porting from a Solaris Operating Environment to a Windows platform, and from a Windows to Solaris Operating Environment platform.
- ❑ Demonstrate the success of your modular design by building hybrid applications, mixing server and client modules
- ❑ Debug problems when interchanging modules
- ❑ Describe how design decisions made in program design now affect the extensibility of the application

Think Beyond

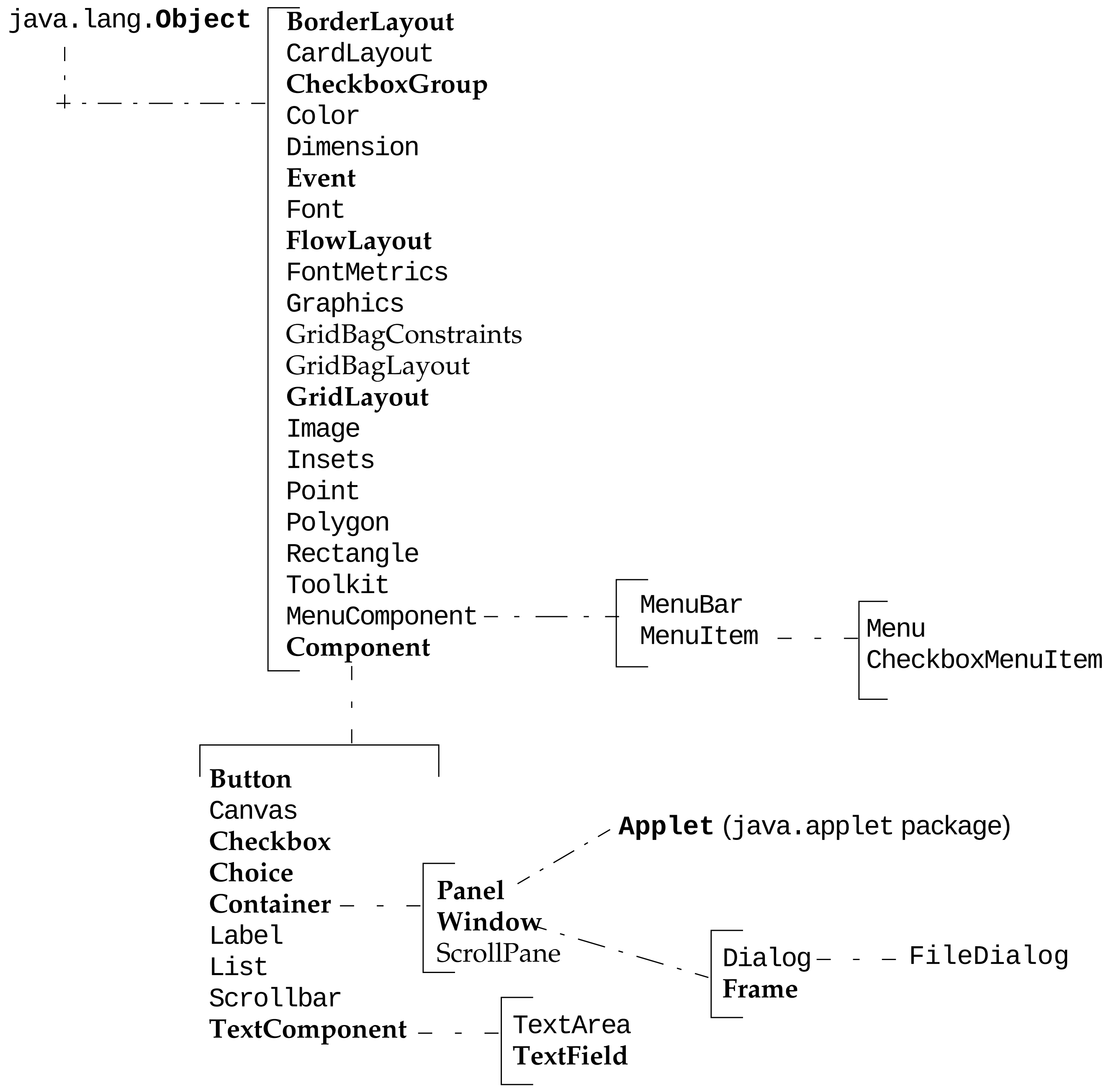
How can porting issues be permanently overcome in the computing world? What standards organization would oversee these issues?

Building GUIs With AWT *A*

This appendix describes how to build GUIs using AWT components such as containers, frames, panels, and layout managers.

The `java.awt` *Package*

The `java.awt` package contains classes to generate widgets and GUI components. A basic overview of this package is shown in Figure A-1. The classes shown in bold highlight the main points covered in this section.



Exceptions – `AWTException` **Errors** – `AWTError`

Figure A-1 The `java.awt` Package

Building Graphical User Interfaces

Containers and Components

Containers and components are fundamental to the AWT. A container can hold one or many components and, if desired, it can hold other containers.

Components are the visible aspect of a GUI, such as a button or a label. You place components into a display by "adding" them to a container.

Note – Because containers can hold not only components but also containers, they are important and fundamental to building layouts of realistic complexity.

Positioning Components

The position of a component in a container is determined by a *layout manager*. A container keeps a reference to a particular instance of `LayoutManager`. 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.

Component Sizing

Because the layout manager is responsible for the size and position of components on its container, do not attempt to set the size or position of components yourself. If you try to do so (using any of the `setLocation`, `setSize`, or `setBounds` methods), the layout manager overrides your decision.

If you must control the size or position of components in a way that cannot be done using the standard layout managers, disable the layout manager by issuing this method call to your container:

```
setLayout(null);
```

Then use the `setLocation`, `setSize`, or `setBounds` methods on components to locate them in the container.

Be aware that 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 class of `LayoutManager.`

Frames

A `Frame` is a subclass of `Window`. It is a `Window` with a title and resize corners.

Creating a Simple Frame

The constructor `Frame(String)` in the `Frame` class creates a new, invisible `Frame` object with the title specified by `String`. A `Frame` can be resized using the `setSize` method inherited from the `Component` class.

Note – The `setVisible` and `setSize` methods must be called to make the `Frame` visible.

The following program creates a simple frame with a specific title, size, and background color:

```
import java.awt.*;
public class MyFrame extends Frame {
   public static void main (String args[]) {
      MyFrame fr = new MyFrame("Hello Out There!");
      // Component method setSize()
      fr.setSize(500,500);
      fr.setBackground(Color.blue);
      fr.setVisible(true); // Component method
show()
   }
   public MyFrame (String str) {
      super (str);
   }
   ...
}

```

Running the Program

The following is an example of compiling and then running the program `MyFrame`.

```
javac MyFrame.java
java MyFrame
```

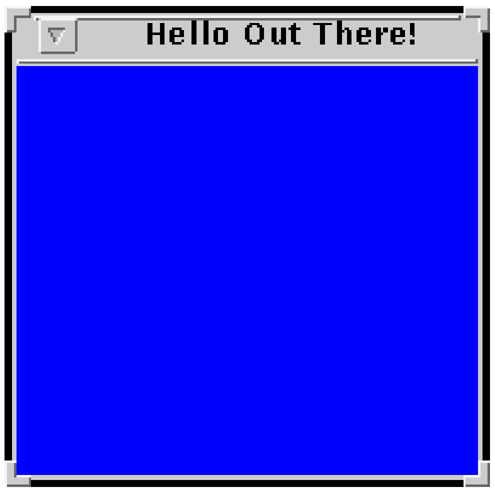


Figure A-2 Example of a Frame

Panels

Panels, like frames, provide the space for you to attach any GUI component, including other panels.

Creating Panels

You use the constructor `Panel` to create panels. Once you create a `Panel` object, you must add it to a `Window` or `Frame` object 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` object:

```
import java.awt.*;
public class FrameWithPanel extends Frame {

   // Constructor
   public FrameWithPanel (String str) {
      super (str);
   }

   public static void main (String args[]) {
      FrameWithPanel fr =
      new FrameWithPanel ("Frame with Panel");
      Panel pan = new Panel();

      fr.setSize(200,200);
      fr.setBackground(Color.blue);
      fr.setLayout(null); //override default layout mgr
      pan.setSize (100,100);
      pan.setBackground(Color.yellow);

      fr.add(pan);
      fr.setVisible(true);
   }
   ...
}
```

Running the Program

The following is an example of compiling and then running the program `FrameWithPanel`.

```
javac FrameWithPanel.java
java FrameWithPanel
```

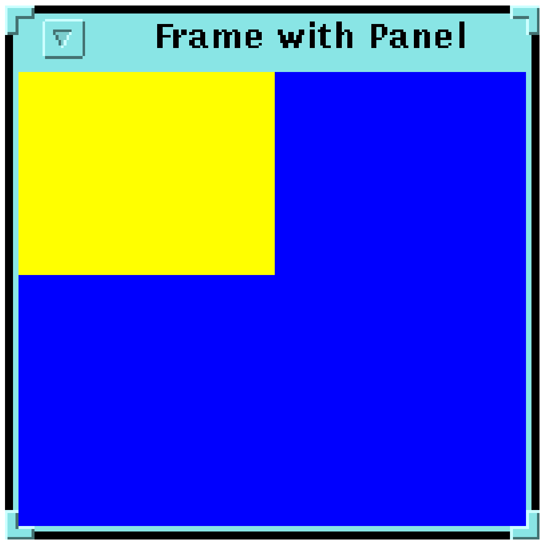


Figure A-3 Example of a Panel in a Frame

Container Layouts

The layout of components in a container can be governed by a layout manager. Each container, such as a panel or a frame, has a default layout manager associated with it, which can be changed by the Java software developer when an instance of that container is created.

Layout Managers

The layout managers included with the Java programming language are the following:

- `FlowLayout` – The default layout manager of `Panels` and `Applets`.
- `BorderLayout` – The default layout manager of `Windows`, `Dialogs`, and `Frames`.
- `GridLayout`
- `CardLayout`
- `GridBagLayout`

The `GridBagLayout` manager is not discussed in this appendix.

A Simple GUI Example

This simple example code demonstrates several important points, and is discussed in the following sections.

```
import java.awt.*;

public class ExGui {
   private Frame f;
   private Button b1;
   private Button b2;

   public static void main(String args[]) {
      ExGui that = new ExGui();
      that.go();
   }

   public void go() {
      f = new Frame(“GUI example”);
      f.setLayout(new FlowLayout());
      b1 = new Button(“Press Me”);
      b2 = new Button(“Don’t press Me”);
      f.add(b1);
      f.add(b2);
      f.pack();
      f.setVisible(true);
   }
}
```

The `main` *Method*

The `main` method in this example does two jobs: it creates an instance of the `ExGui` object and once the data space has been created, calls the instance method `go` in the context of that instance. Until an instance exists, there are no real data items called `f`, `b1`, and `b2` for use. It is in `go`, that the real action occurs.

`new Frame("GUI Example")`

This method creates an instance of the class `java.awt.Frame`. A `frame` in the Java programming language is a top-level window, with a title bar—defined by the constructor argument "GUI Example" in this case—and resize handles and other conventional decorations. The `frame` has a zero size and is currently not visible.

`f.setLayout(new FlowLayout())`

This method creates an instance of the flow layout manager, and installs it into the frame. There is a default layout manager for every frame, but it is not used in this example. The flow layout manager is the simplest manager in the AWT and positions components somewhat like words on a page, line by line. The flow layout manager centers each line by default.

`new Button("Press Me")`

This method creates an instance of the class `java.awt.Button`. A button is the standard push button taken from the local window toolkit. The button label is defined by the string argument to the constructor.

`f.add(b1)`

This method tells frame f, which is a container, that it is to contain the component b1, a button. The size and position of b1 are under the control of the frame's layout manager from this point onward.

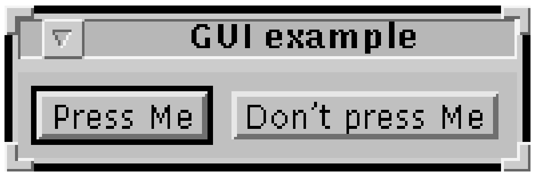
`f.pack()`

This method tells the frame to set a size that "neatly encloses" the components that it contains. The layout manager, which is responsible for the size and position of everything it contains, accomplishes this.

`f.setVisible(true)`

This method causes the frame and all its contents to become visible to the user.

The final result of this code, on an OpenLook system is the following:



Layout Managers

Flow Layout Manager

The flow layout manager used in the first example positions components on a line by line basis. Each time a line is filled, a new line is started.

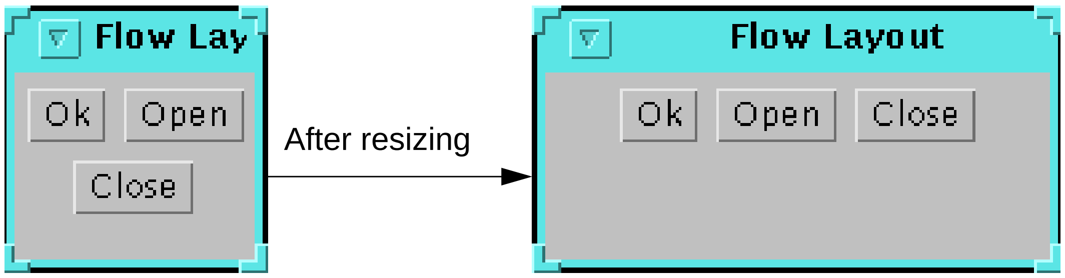
Unlike other layout managers, the flow layout manager does not constrain the size of the components it manages, but instead allows them to have their preferred size.

Note – All components have a method called `getPreferredSize`, which is used by layout managers to determine the size of the component.

Options on a flow layout allow the components to justify the components to the left or to the right. By default, the components are centered.

You can specify insets if you want to create a bigger border area between each component.

When you resize the area that is being managed by a flow layout, the layout might change.



The following examples show how to implement the FlowLayout, using the setLayout method inherited from Component:

```
setLayout(new FlowLayout(FlowLayout.RIGHT, 20, 40));
setLayout(new FlowLayout(FlowLayout.LEFT));
setLayout(new FlowLayout());
```

The following example adds several buttons to a flow layout on a frame:

```
import java.awt.*;

public class MyFlow {
   private Frame f;
   private Button button1, button2, button3;

   public static void main (String args[]) {
      MyFlow mflow = new MyFlow ();
      mflow.go();
   }

   public void go() {
      f = new Frame ("Flow Layout");
      f.setLayout(new FlowLayout());
      button1 = new Button("Ok");
      button2 = new Button("Open");
      button3 = new Button("Close");
      f.add(button1);
      f.add(button2);
      f.add(button3);
      f.setSize (100,100);
      f.setVisible(true);
    }
}
```

Border Layout Manager

The BorderLayout manager provides a more complex scheme for placing your components within a panel or window. The BorderLayout contains five distinct areas: North, South, East, West, and Center.

North occupies the top of a panel, 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.

The BorderLayout manager is the default layout manager for dialogs and frames. The code for the following image can be found on the next page.

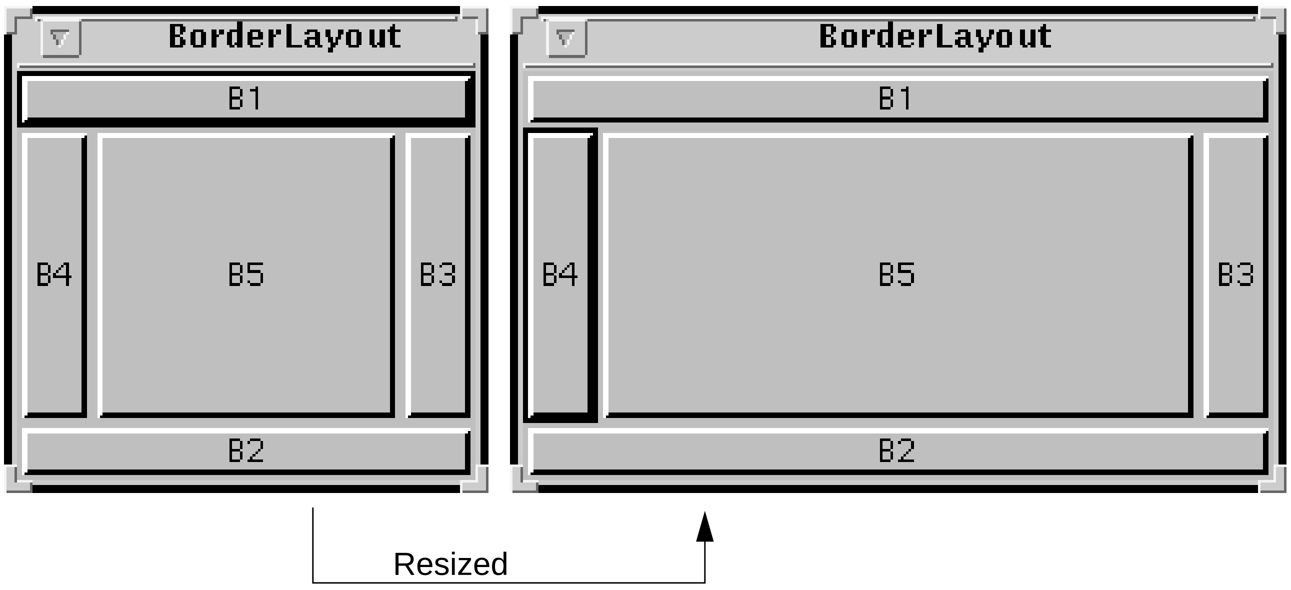


Figure A-4 Border Layout Manager

Note – The relative positions of the buttons do not change as the window is resized, but the sizes of the buttons do change.

The following code is a modification of the previous example and demonstrates the behavior of the border layout manager:

```
import java.awt.*;

public class ExGui2 {
   private Frame f;
   private Button bn, bs, bw, be, bc;

   public static void main(String args[]) {
      ExGui2 that = new ExGui2();
      that.go();
   }

   public void go() {
      f = new Frame("Border Layout");
      bn = new Button("B1");
      bs = new Button("B2");
      bw = new Button("B3");
      be = new Button("B4");
      bc = new Button("B5");

      f.add(bn, BorderLayout.NORTH);
      f.add(bs, BorderLayout.SOUTH);
      f.add(bw, BorderLayout.WEST);
      f.add(be, BorderLayout.EAST);
      f.add(bc, BorderLayout.CENTER);

      f.setSize (200, 200);
      f.setVisible(true);
   }
}
```

You must add components to named regions in the border layout manager, otherwise they are not visible.

You can use a border layout manager to produce layouts with elements that stretch in one direction or the other, or both, upon resizing.

If you leave a region of a border layout unused, it behaves as if its preferred size was zero by zero. So the center region still appears as background even if it contains no components, but the four peripheral regions effectively shrink to a line of zero thickness and disappear.

You must add only a single component to each of the five regions of the border layout manager. If you try to add more than one, only one is visible. You will see later how you can use intermediate containers to allow more than one component to be laid out in the space of a single border layout manager region.

Grid Layout Manager

The grid layout 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 grid layout with three rows and two columns, created by the statement `new GridLayout(3, 2)` creates six cells as follows:

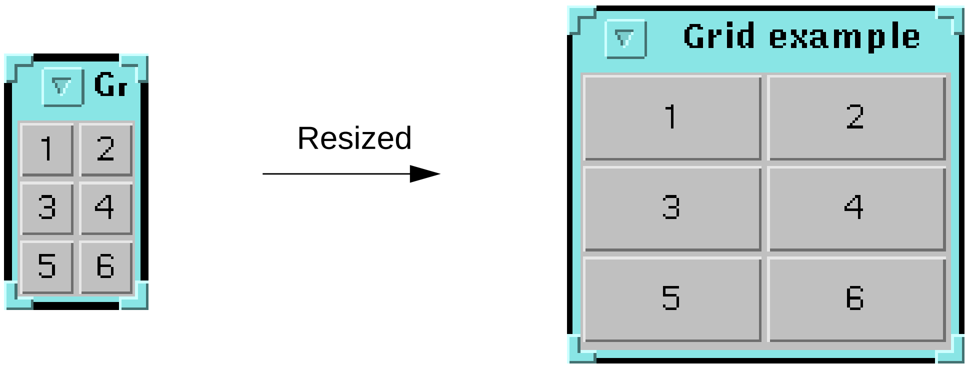


Figure A-5 Grid Layout Manager

As with the `BorderLayout` manager, the relative position of components does not change as the area is resized. Only the size of the components change.

Observe that the width of all cells is identical, and is determined as a simple division of the available width by the number of cells. Similarly the height of all cells is determined by the available height divided by the number of rows.

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 code displays the application shown on the previous page:

```
import java.awt.*;
public class GridEx {
   private Frame f;
   private Button b1, b2, b3, b4, b5, b6;

   public static void main(String args[]) {
      GridEx grid = new GridEx();
      grid.go();
   }

   public void go() {
      f = new Frame(“Grid example”);

      f.setLayout (new GridLayout (3, 2));
      b1 = new Button(“1”);
      b2 = new Button(“2”);
      b3 = new Button(“3”);
      b4 = new Button(“4”);
      b5 = new Button(“5”);
      b6 = new Button(“6”);

      f.add(b1);
      f.add(b2);
      f.add(b3);
      f.add(b4);
      f.add(b5);
      f.add(b6);

      f.pack();
      f.setVisible(true);
   }
}
```

Card Layout Manager

The CardLayout Manager enables you to treat the interface as a series of cards, one of which you can view at any one time. A CardLayout object is a layout manager for a container. It treats each component in the container as a card. Only one card is visible at a time, and the container acts as a stack of cards. The first component added to a CardLayout object is the visible component when the container is first displayed.

The ordering of cards is determined by the container's own internal ordering of its component objects. CardLayout defines a set of methods that allow an application to flip through these cards sequentially, or to show a specified card.

The following application `CardTest` demonstrates the use of the CardLayout Manager.

```
import java.awt.*;
import java.awt.event.*;

public class CardTest implements MouseListener {
  private Panel p1, p2, p3, p4, p5;
  private Label lb1, lb2, lb3, lb4, lb5;

  // Declare a CardLayout object to call its
methods.
  private CardLayout myCard;
  private Frame f;

  public void go() {
    f = new Frame ("Card Test");
    myCard = new CardLayout();
    f.setLayout(myCard);

    // Create the panels that I want
    // to use as cards.
    p1 = new Panel();
    p2 = new Panel();
    p3 = new Panel();
    p4 = new Panel();
    p5 = new Panel();

    // Create a label to attach to each panel, and
    // change the color of each panel, so they are
    // easily distinguishable

    lb1 = new Label("This is the first Panel");
    p1.setBackground(Color.yellow);
    p1.add(lb1);

    lb2 = new Label("This is the second Panel");
    p2.setBackground(Color.green);
    p2.add(lb2);

    lb3 = new Label("This is the third Panel");
    p3.setBackground(Color.magenta);
    p3.add(lb3);

    lb4 = new Label("This is the fourth Panel");
    p4.setBackground(Color.white);
    p4.add(lb4);

    lb5 = new Label("This is the fifth Panel");
    p5.setBackground(Color.cyan);
    p5.add(lb5);

    // Set up the event handling here.
    p1.addMouseListener(this);
    p2.addMouseListener(this);
    p3.addMouseListener(this);
    p4.addMouseListener(this);
    p5.addMouseListener(this);

    // Add each panel to my CardLayout
    f.add(p1, "First");
    f.add(p2, "Second");
    f.add(p3, "Third");
    f.add(p4, "Fourth");
    f.add(p5, "Fifth");

    // Display the first panel.
    myCard.show(f, "First");

    f.setSize(200,200);
    f.setVisible(true);
  }

  public void mousePressed(MouseEvent e) {
    myCard.next(f);
  }

  public void mouseReleased(MouseEvent e) { }
  public void mouseClicked(MouseEvent e) { }
  public void mouseEntered(MouseEvent e) { }
  public void mouseExited(MouseEvent e) { }

  public static void main (String args[]) {
    CardTest ct = new CardTest();
    ct.go();
  }
}
```

Other Layout Managers

In addition to the flow, border, grid, and card layout managers, the core AWT also provides the `GridBagLayout` Manager.

The `GridBagLayout` manager provides complex layout facilities, based on a grid, and allows single components to take their preferred size within a cell, rather than fill the whole cell. Also, a grid bag layout manager allows a single component to extend over more than one cell.

Containers

The AWT provides several containers. This section discusses the two essential ones.

Frames

You have already seen the frame used in the preceding examples. It presents a "top-level" window with a title, border, and resizeable corners according to the local platform conventions.

If you do not explicitly use the `setLayout` method, a frame uses a border layout manager by default.

Most applications use at least one frame as the starting point for their GUIs, but it is possible to use multiple frames in a single piece of code.

Panels

Panels are containers and almost nothing else. They do not have an appearance of their own, and they cannot be used as stand-alone windows. By default, a panel has a flow layout manager associated with it, but you can change this by using the `setLayout` method used earlier on frames.

Panels are created and added to other containers in the same way components, such as buttons are created and added. However, when a panel is added to a container, you can do the following two crucial tasks in the resulting panel:

- Give it a layout manager of its own, imposing a different layout approach on a region of the display
- Add components to the panel even if, for example, the panel itself constitutes the only component that can be properly added to a region of a border layout

Creating Panels and Complex Layouts

Panels are created using the constructor Panel. Once a Panel object is created, you must add it to another container. This is done as before using the add method of the container.

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.

```
import java.awt.*;
public class ExGui3 {
   private Frame f;
   private Panel p;
   private Button bw, bc;
   private Button bfile, bhelp;

   public static void main(String args[]) {
      ExGui3 gui = new ExGui3();
      gui.go();
   }
   public void go() {
      f = new Frame(“GUI example 3”);
      bw = new Button(“West”);
      bc = new Button(“Work space region”);
      f.add(bw, BorderLayout.WEST);
      f.add(bc, BorderLayout.CENTER);
      p = new Panel();
      f.add(p, BorderLayout.NORTH);
      bfile = new Button(“File”);
      bhelp = new Button(“Help”);
      p.add(bfile);
      p.add(bhelp);
      f.pack();
      f.setVisible(true);
   }
}
```

When the previous example is run, the resulting display is illustrated in Figure A-6.

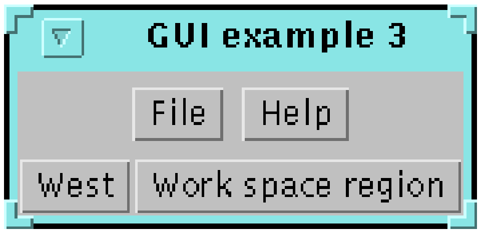


Figure A-6 Window

If the window is resized, the resulting display is illustrated in Figure A-7.

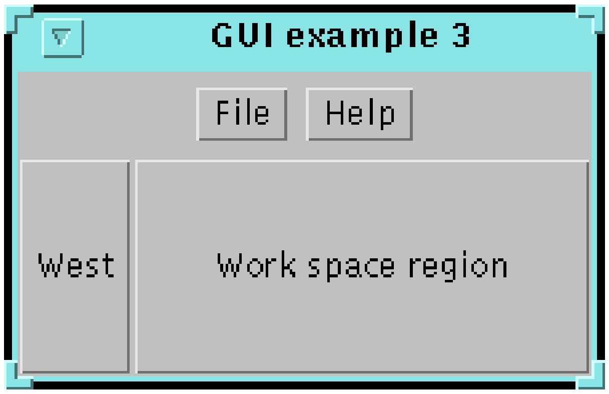


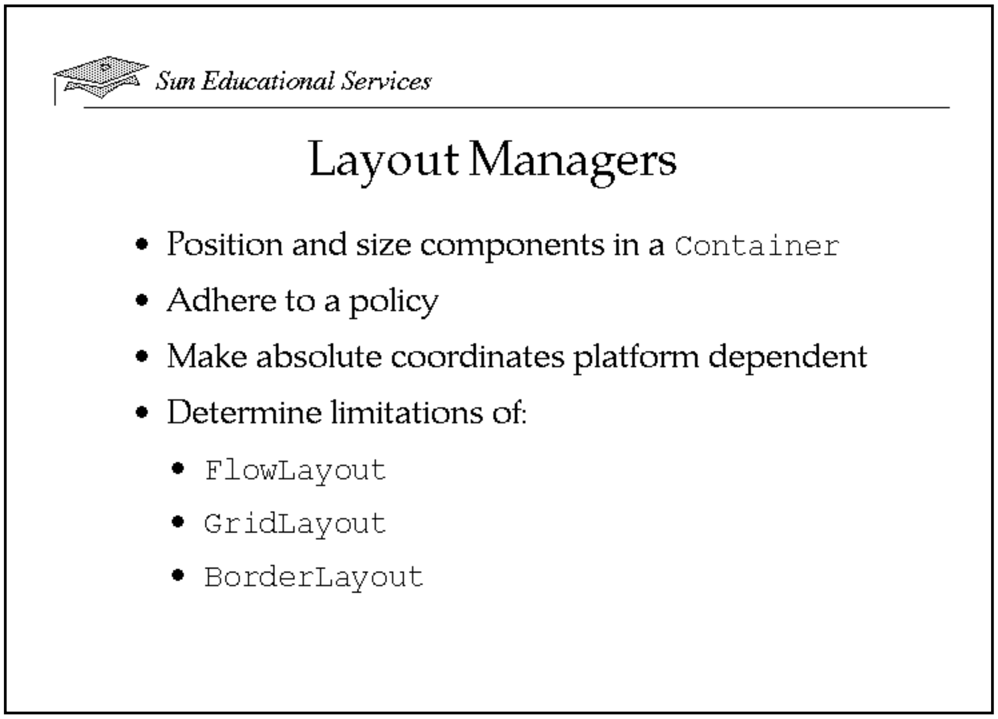
Figure A-7 Resized Window

Observe that the North region of the border layout is now effectively 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 border layout manager: 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 flow layout manager that is associated with the panel by default.

Using the `GridBagLayout` *Manager* *B*

This appendix describes the use of the `GridBagLayout` manager in the production of complex user interfaces.


Sun Educational Services

Layout Managers

- Position and size components in a `Container`
- Adhere to a policy
- Make absolute coordinates platform dependent
- Determine limitations of:
  - `FlowLayout`
  - `GridLayout`
  - `BorderLayout`

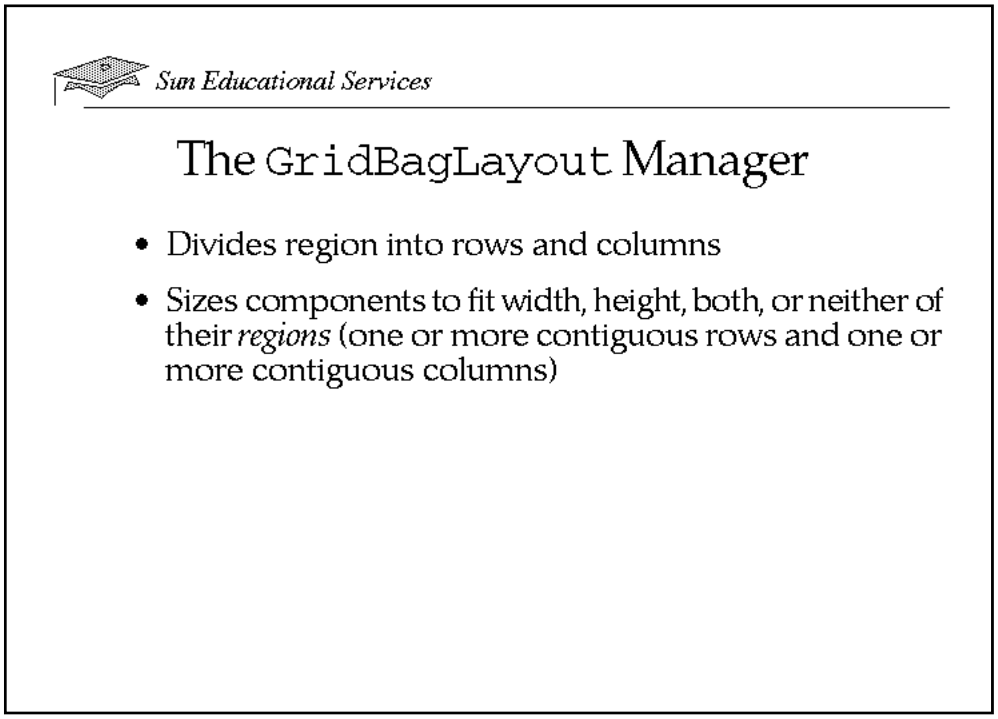
Layout Managers

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 will be 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` manager 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.

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` manager, which is more powerful.


Sun Educational Services

The GridBagLayout Manager

- Divides 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` *Manager*

The `GridBagLayout` manager lays out components using a grid. However, unlike the `GridLayout` manager, child components are not necessarily constrained to occupy exactly one entire grid cell, neither are all rows and columns equal in size. Rather, you can assign a component multiple cells, horizontally, vertically, or both, and the component can exist within that region.

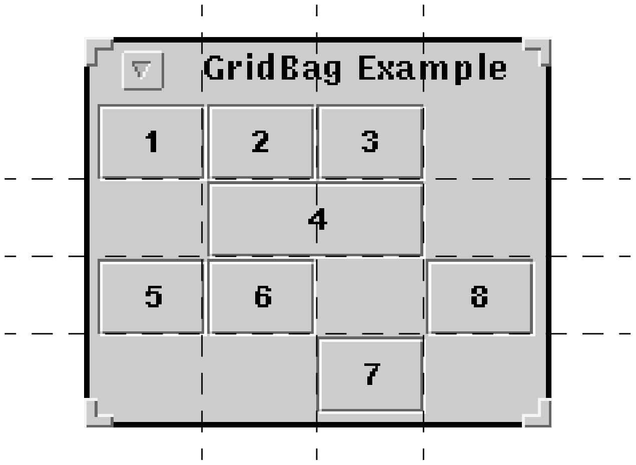


Figure B-1 Sample `GridBagLayout` With Four Rows and Four Columns

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 B-1, each grid cell is the basic size of a `JButton` with a single-digit label.

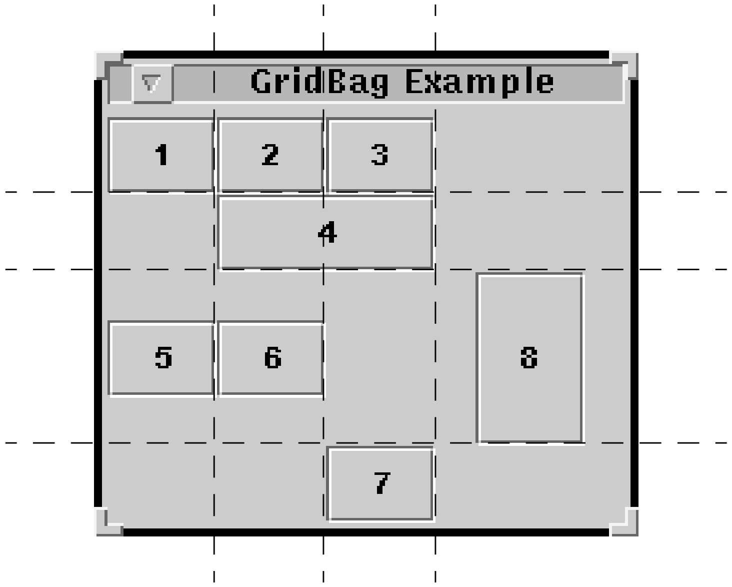


Figure B-2 Sample `GridBagLayout` Showing Cells Expanded by Weight

Where 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 B-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 alloted to one component is referred to as the component's *region*. In Figure B-2, the button labelled "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 B-2, the buttons labelled "5" and "6" do not fill the vertical space available to them; similarly, the button labelled "8" fills vertically, but not horizontally.

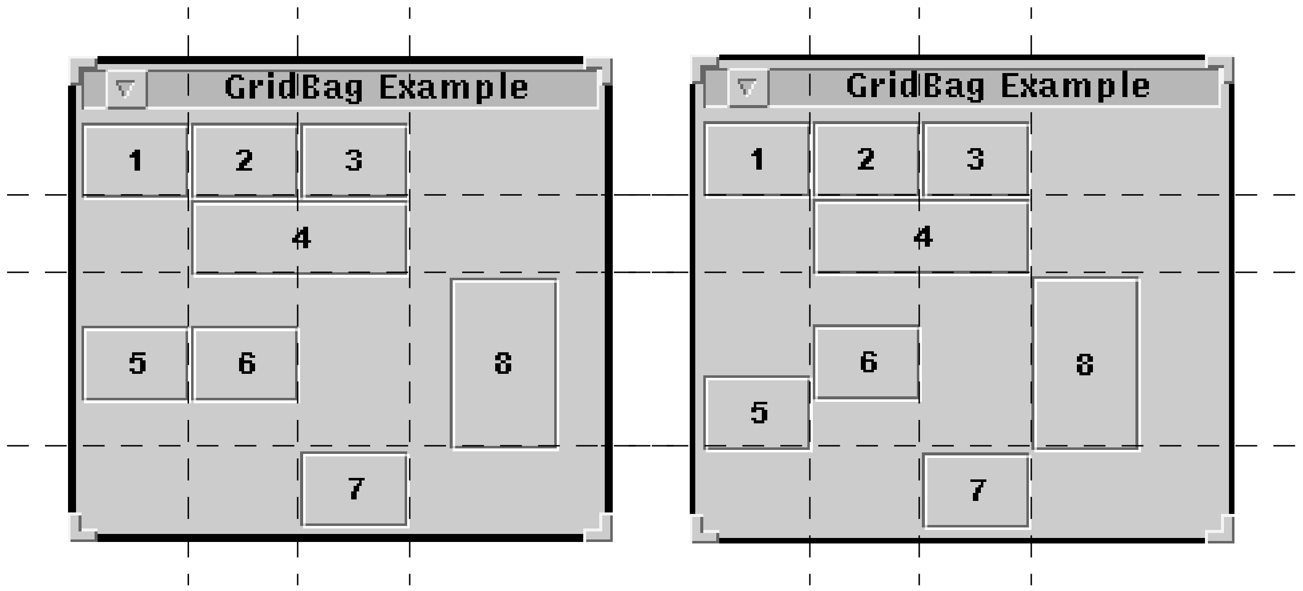
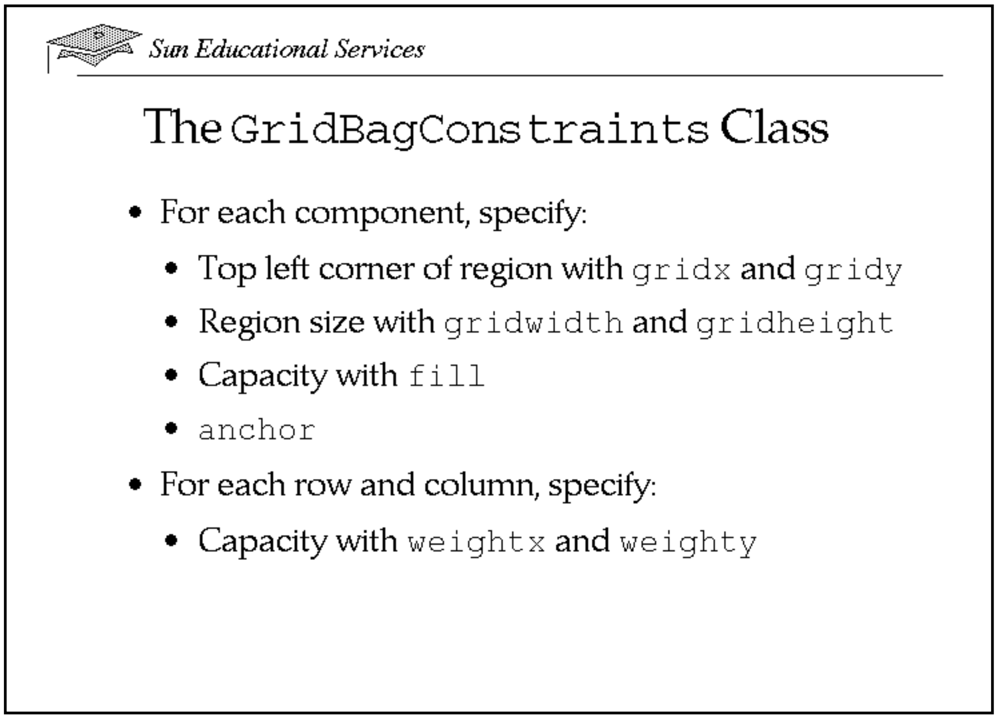


Figure B-3 Sample `GridBagLayout` Showing the Effect of Anchor

Where 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 is positioned at the top left of its allocated region.

In Figure B-3, the two examples have differing anchor settings. Specifically, the button labelled "5" has a `CENTER` anchor in the left-hand example, but a `SOUTH` anchor in the right-hand example. The button labelled "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 alloted 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.`



Sun Educational Services

The GridBagConstraints Class

- For each component, specify:
  - Top left corner of region with gridx and gridy
  - Region size with gridwidth and gridheight
  - Capacity with fill
  - anchor
- For each row and column, specify:
  - Capacity with weightx and weighty

The `GridBagConstraints` *Class*

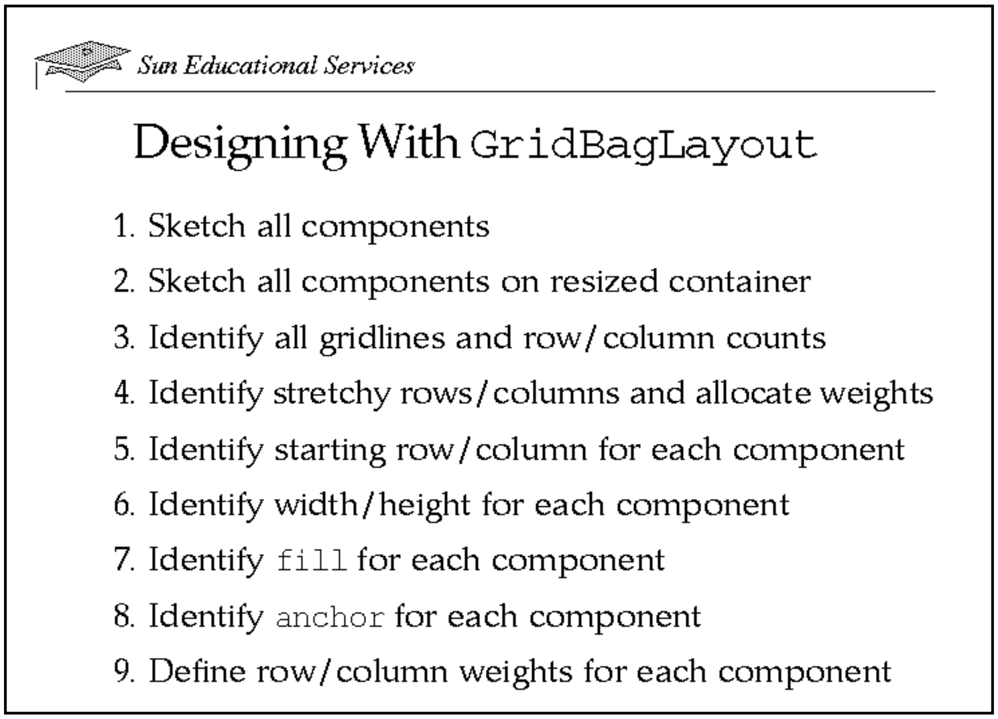
You have seen the principles by which the `GridBagLayout` manager makes positioning decisions, but not how those preferences are supplied to it. This is done using an object of the class `GridBagConstraints`. Each time you add a `Component` to a `Container` that has a `GridBagLayout`, you provide an 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 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`.
- `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 doesn't 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 confuses anyone reading the program.



Sun Educational Services

Designing With `GridBagLayout`

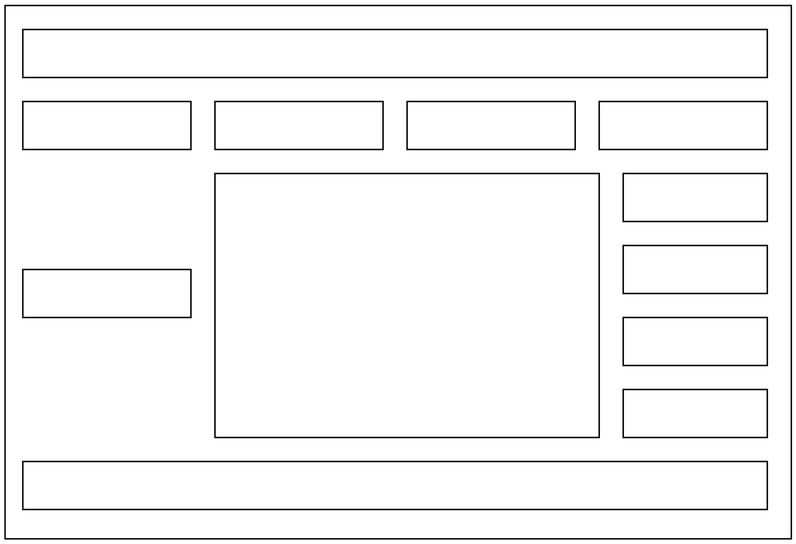
1. Sketch all components
2. Sketch all components on resized container
3. Identify all gridlines and row/column counts
4. Identify stretchy rows/columns and allocate weights
5. Identify starting row/column for each component
6. Identify width/height for each component
7. Identify `fill` for each component
8. Identify `anchor` for each component
9. Define row/column weights for each component

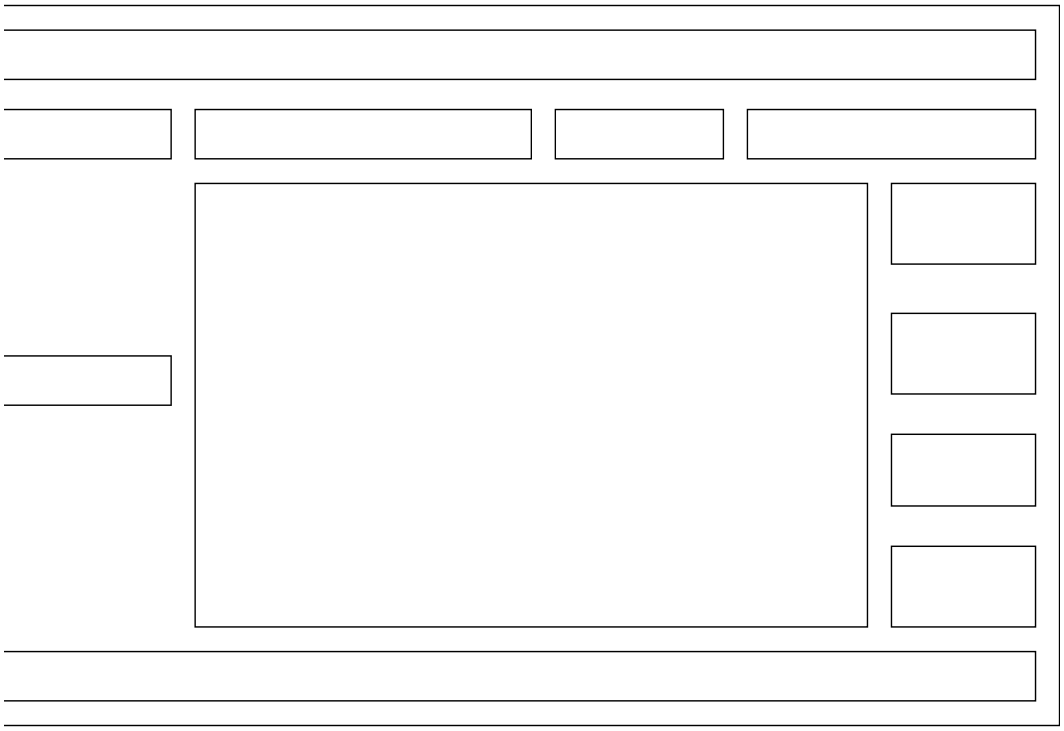
Designing With `GridBagLayout`

Design Steps

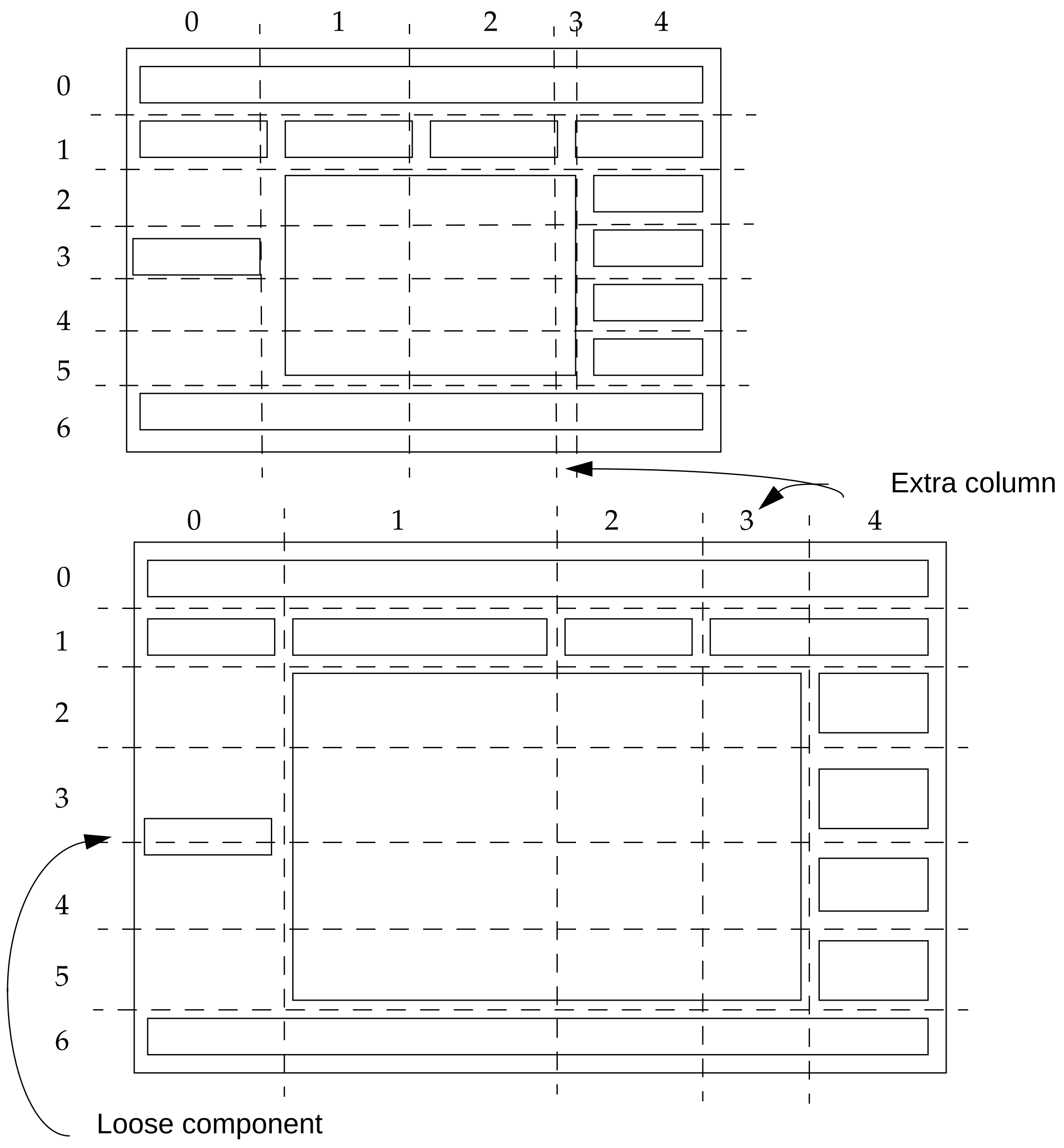
When designing with a `GridBagLayout`:

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.


Basic, unexpanded layout proposal


Basic, expanded layout proposal

3. 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.



4. Decide how the extra space is to be allocated. 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).

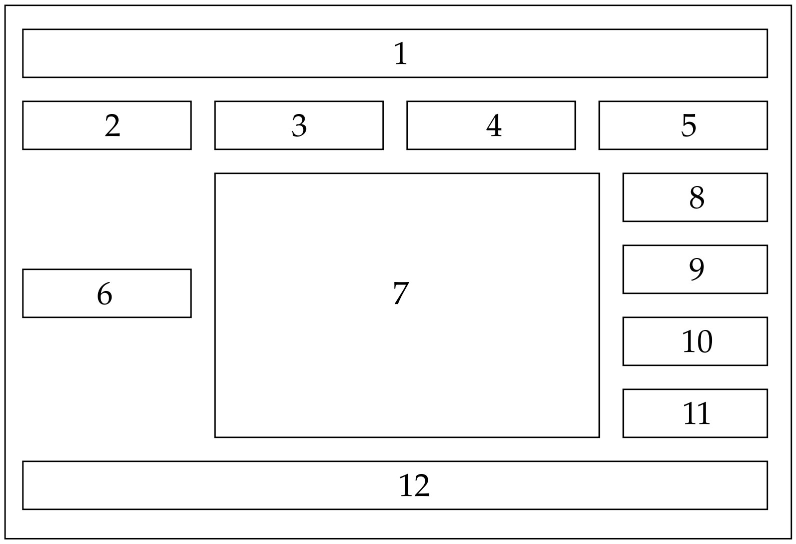
The expanded version brings out the existence of an extra column, which is not really noticeable until the display is expanded. You would be unlikely to recognize this column's existence in the unexpanded diagram.

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 the region that extends over rows 3 through to 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.

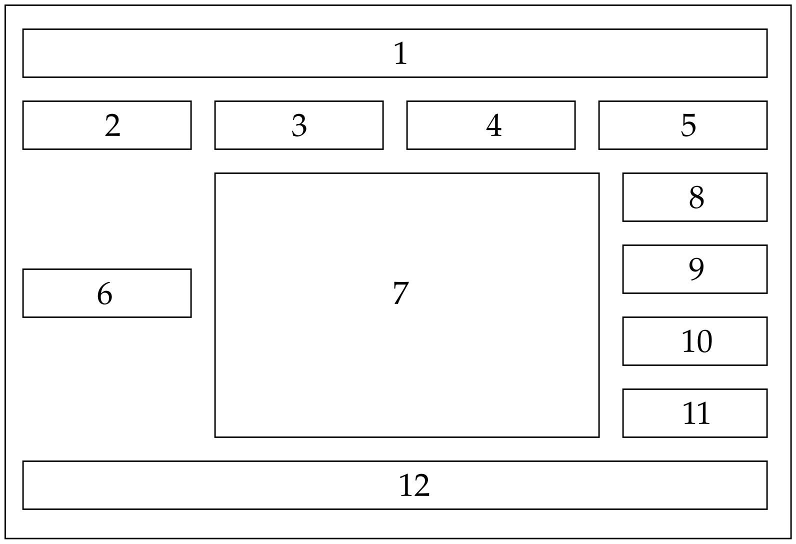
5. Now that you have designed the underlying grid, you can start to position each component over that grid. Start by identifying the top left row and column for each component region; this gives you the `gridy` and `gridx` values for each.



6. 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

7. 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.

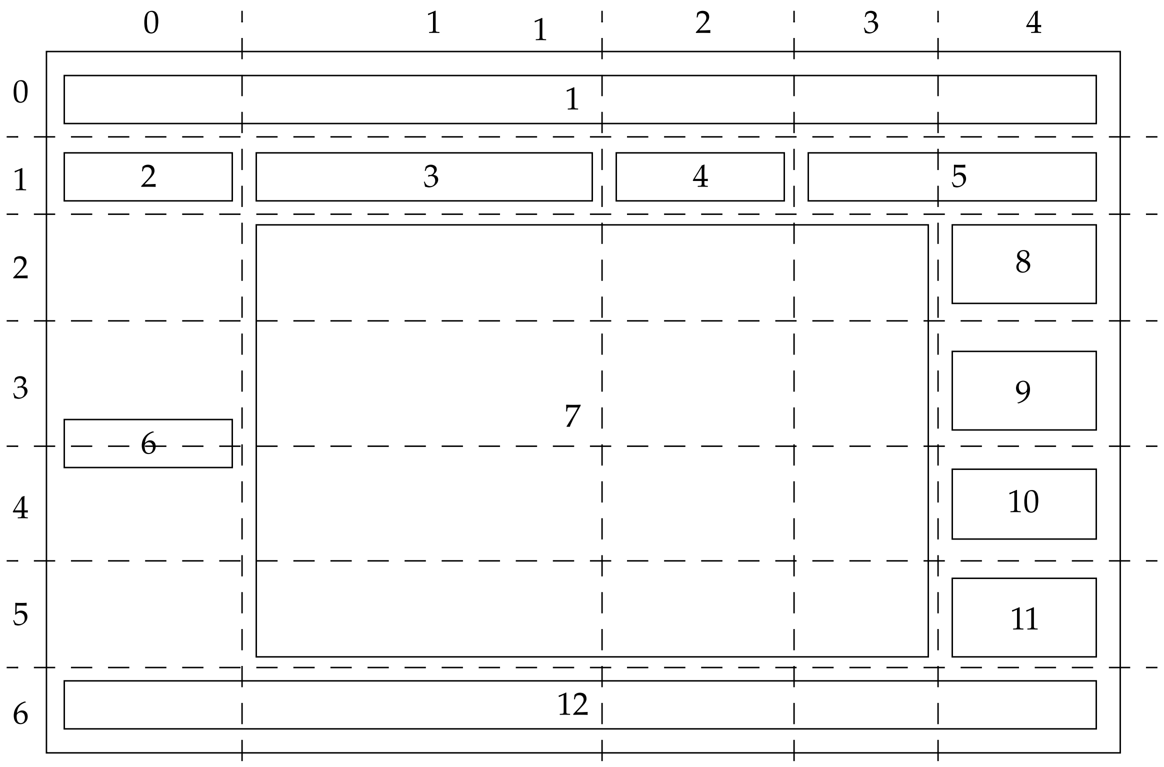
So, in this example, the only component that is not set to fill BOTH is component 6. This should have a fill value of HORIZONTAL, to ensure that it takes up the full width of its region.

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 where 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. Any of the anchor values EAST, WEST, or CENTER would result in the required behavior, but CENTER is the most reasonable because it most directly expresses the required result.

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`.



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 `new Component(){}` 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`.

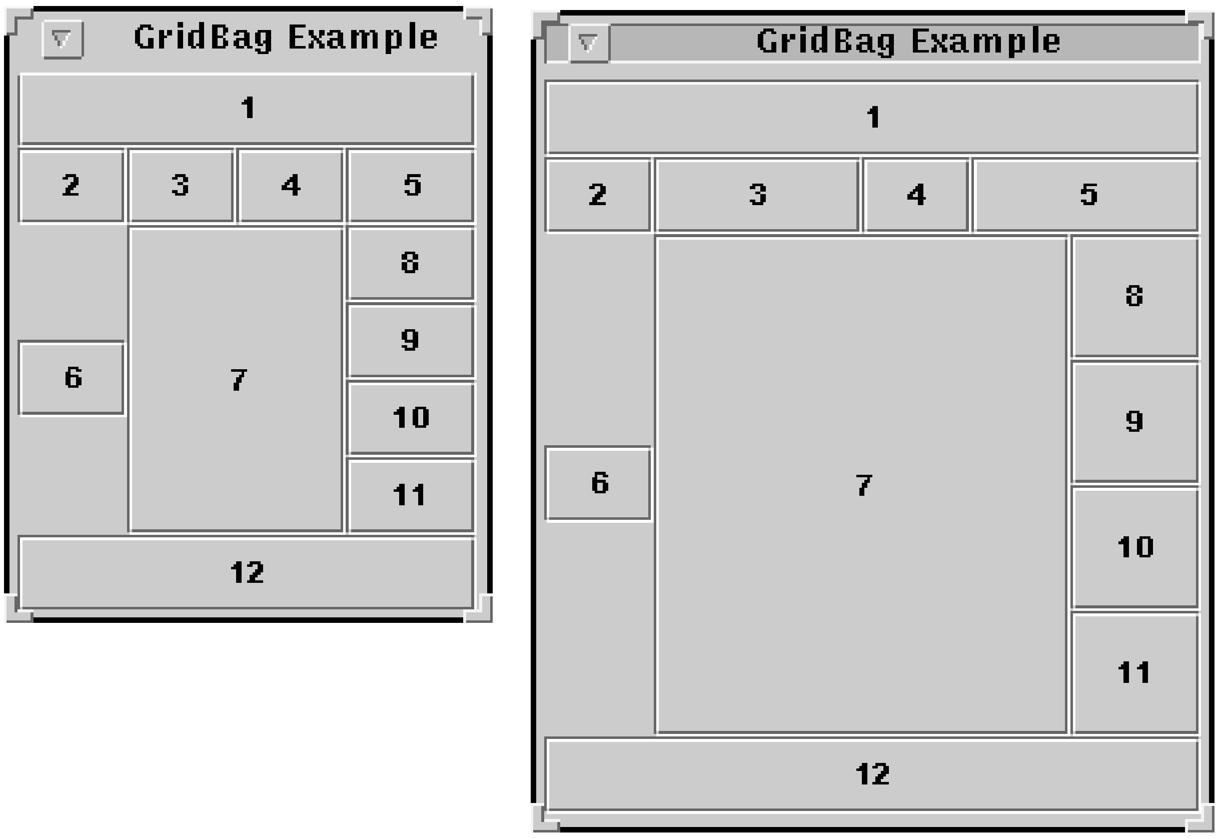


Figure 2-4 Sample Screen Shots

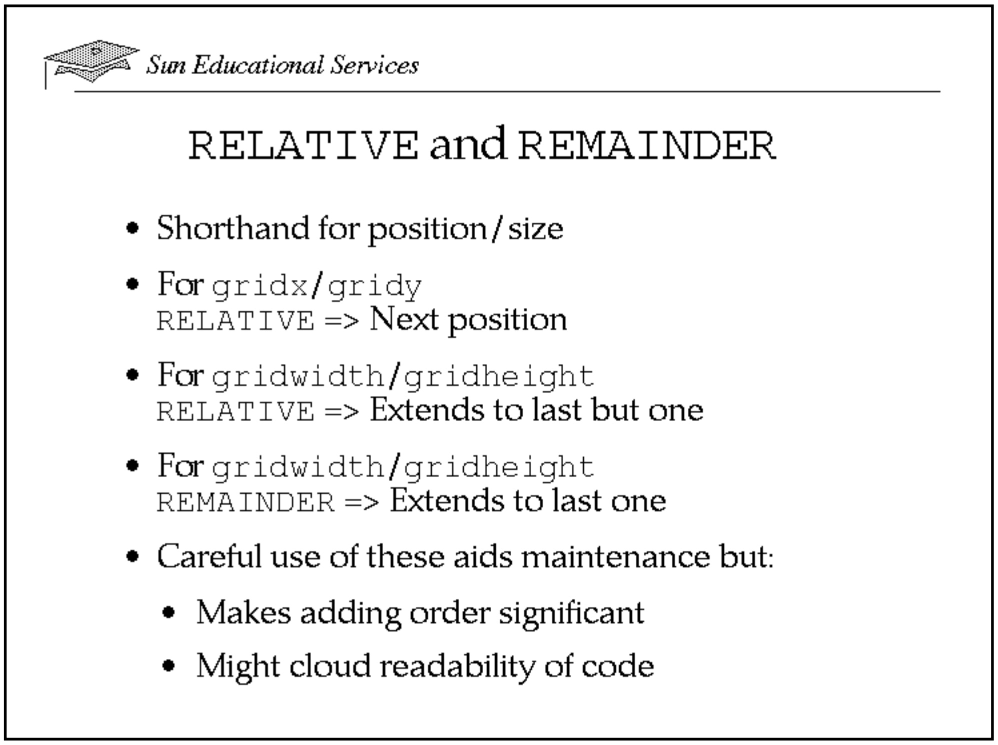
Once the `GridBagConstraints` values have been applied to components added to a `GridBagLayout`, the desired behavior is achieved. The screen shots shown here are derived from the implementation program listed on the next page.

The main part of the program for this example, with the values used in the `GridBagConstraints`, is shown as follows:

```
import java.awt.*;
import javax.swing.*;

public class ExampleGB {
  public static void main(String args[]) {
    JFrame f = new JFrame("GridBag Example");
    Container c = f.getContentPane();
    c.setLayout(new GridBagLayout());
    GridBagAdder.add(c, new Canvas(), 3, 2, 1, 1, 1, 0,
      GridBagConstraints.NONE, GridBagConstraints.CENTER);
    GridBagAdder.add(c, new JButton("1"), 0, 0, 5, 1, 0, 0,
      GridBagConstraints.HORIZONTAL, GridBagConstraints.CENTER);
    GridBagAdder.add(c, new JButton("2"), 0, 1, 1, 1, 0, 0,
      GridBagConstraints.BOTH, GridBagConstraints.CENTER);
    GridBagAdder.add(c, new JButton("3"), 1, 1, 1, 1, 1, 0,
      GridBagConstraints.HORIZONTAL, GridBagConstraints.CENTER);
    GridBagAdder.add(c, new JButton("4"), 2, 1, 1, 1, 0, 0,
      GridBagConstraints.BOTH, GridBagConstraints.CENTER);
    GridBagAdder.add(c, new JButton("5"), 3, 1, 2, 1, 0, 0,
      GridBagConstraints.HORIZONTAL, GridBagConstraints.CENTER);
    GridBagAdder.add(c, new JButton("6"), 0, 2, 1, 4, 0, 0,
      GridBagConstraints.HORIZONTAL, GridBagConstraints.CENTER);
    GridBagAdder.add(c, new JButton("7"), 1, 2, 3, 4, 0, 0,
      GridBagConstraints.BOTH, GridBagConstraints.CENTER);
    GridBagAdder.add(c, new JButton("8"), 4, 2, 1, 1, 0, 1,
      GridBagConstraints.BOTH, GridBagConstraints.CENTER);
    GridBagAdder.add(c, new JButton("9"), 4, 3, 1, 1, 0, 1,
      GridBagConstraints.BOTH, GridBagConstraints.CENTER);
    GridBagAdder.add(c, new JButton("10"), 4, 4, 1, 1, 0, 1,
      GridBagConstraints.BOTH, GridBagConstraints.CENTER);
    GridBagAdder.add(c, new JButton("11"), 4, 5, 1, 1, 0, 1,
      GridBagConstraints.BOTH, GridBagConstraints.CENTER);
    GridBagAdder.add(c, new JButton("12"), 0, 6, 5, 1, 0, 0,
      GridBagConstraints.HORIZONTAL, GridBagConstraints.CENTER);
    f.pack();
    f.setVisible(true);
  }
```

Supporting the code on the previous page is this inner class. It provides the `add` method that simplifies setting up the `GridBagConstraints` values.

```
  static class GridBagAdder {
    // OK to reuse this as we overwrite all elements every time
    // Note that this is not threadsafe however!
    static GridBagConstraints cons = new GridBagConstraints();
    public static void add(Container cont,Component comp,int x, int y,
      int width,int height,int weightx,int weighty,
      int fill,int anchor) {

      cons.gridx = x;
      cons.gridy = y;
      cons.gridwidth = width;
      cons.gridheight = height;
      cons.weightx = weightx;
      cons.weighty = weighty;
      cons.fill = fill;
      cons.anchor = anchor;
      cont.add(comp, cons);
    }
  }
}
```


Sun Educational Services

RELATIVE and REMAINDER

- Shorthand for position/size
- For gridx/gridy
  RELATIVE => Next position
- For gridwidth/gridheight
  RELATIVE => Extends to last but one
- For gridwidth/gridheight
  REMAINDER => Extends to last one
- Careful use of these aids maintenance but:
  - Makes adding order significant
  - Might cloud readability of code

RELATIVE and REMAINDER

Where a layout involves a large number of components in a fairly simple layout, it can be time consuming to set up the `gridx` and `gridy` values for each one. This situation is aggravated when you make changes; 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 last row or column.

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.

The AWT Event Model C

This appendix describes how to write code to handle events that occur at the user interface.

What Is an Event?

When the user performs an action at the user interface, this causes an *event* to be issued. Events are objects that describe what happened. A number of different types of event classes exist to describe different general categories of user action.

Event Sources

An event source (at the user interface level) is the result of some user action on an AWT component. For example, a mouse click on a button component generates (sources) an `ActionEvent`. The `ActionEvent` is an object (class) that contains the following information about the status of the event:

- `ActionCommand` – The command name associated with the action
- Modifiers – Any modifiers held during the action

Event Handlers

When an event occurs, the event is received by the component with which the user interacted; for example, the button, slider, text field, and so on. An event handler is a method that receives the Event object so that the program can process the user's interaction.

How Events Are Processed

Between JDK 1.0 and JDK 1.1, there were significant changes in the way that events are received and processed. This section compares the previous event model (hierarchical) and the current event model (delegation).

Hierarchical Versus Delegation Event Model

JDK 1.0 uses a hierarchical event model and the Java 2 Platform (as well as the JDK 1.1) uses a delegation event model. While this course covers the Java 2 Platform, it is important to recognize how these two event models compare.

Hierarchical Model (JDK 1.0)

The hierarchical event model is based on containment. Events are sent to the component first, and then propagate up the containment hierarchy. Events that are not handled at the component level *automatically* continue to propagate to the component's container.

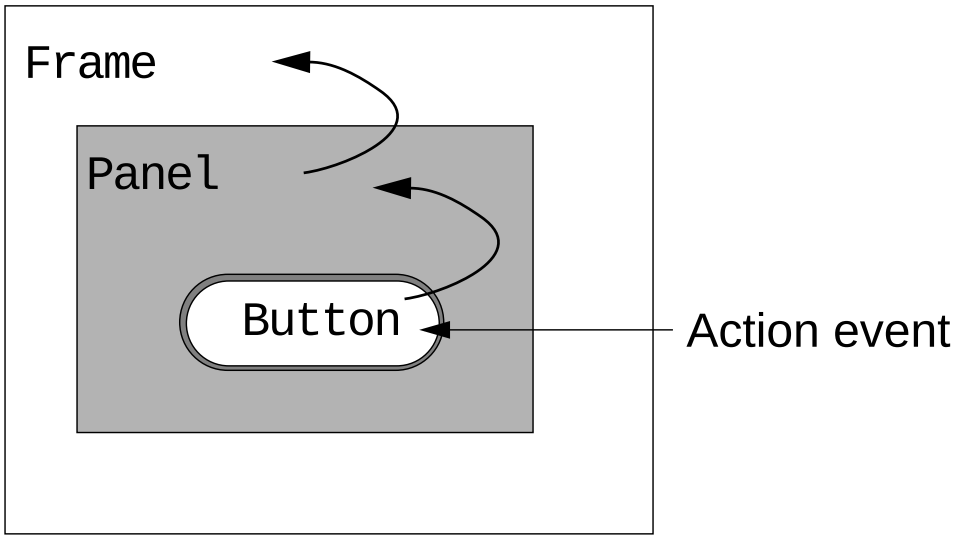


Figure C-1 Hierarchical Event Handling

For example, in Figure C-1, a mouse click on the `Button` object (contained by a `Panel` on a `Frame`) sends an action event to the `Button`. If it is not handled by the `Button`, the event is then sent to the `Panel`, and if not handled there, the event is sent to the `Frame`.

There is an obvious advantage to this model:

- It is simple and well-suited to an object-oriented programming environment; after all, Java technology components extend from the `java.awt.Component` class, which is where the `handleEvent` method resides.

However, there are some disadvantages:

- There is no simple way to filter events.
- To handle events, you must either subclass the component that receives the event or create a big `handleEvent` method at the base container.

Delegation Model

A new event model was introduced in JDK 1.1, called the delegation event model. In the delegation event model, events are sent to a component, but it is up to each component to register an event handler routine (called a listener) to receive the event. In this way, the event handler can be in a class separate from the component. The handling of the event is then delegated to the separate class.

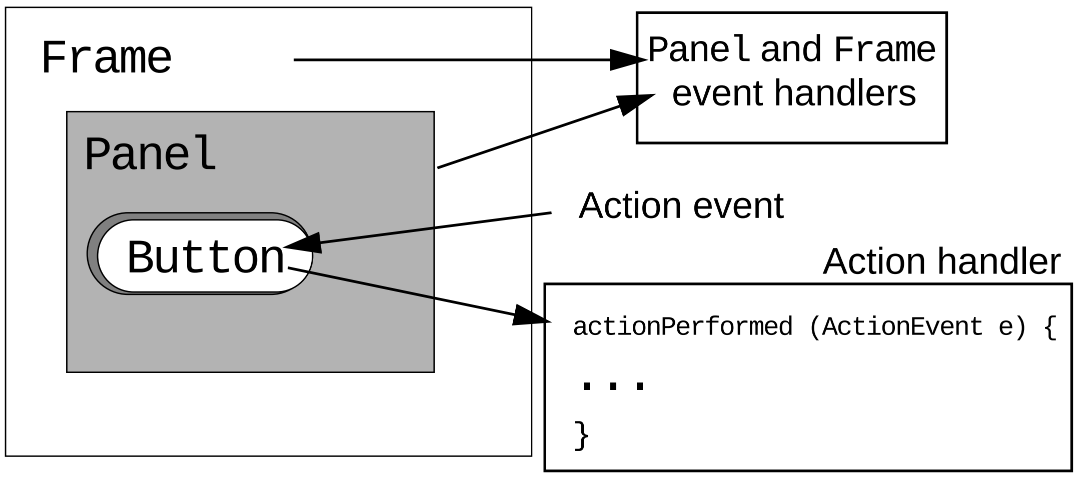

Frame
Panel
Button
Panel and Frame event handlers
Action event
Action handler
actionPerformed (ActionEvent e) {
...
}

Figure C-2 Delegation Event Handling

Events are objects that are reported only to registered listeners. Every event has a corresponding listener class (interface). The class that implements the listener interface defines each method that can receive an `Event` object.

For example, the following is a simple `Frame` with a single `Button` on it:

```
import java.awt.*;
import ButtonHandler;
public class TestButton {
   public static void main(String args[]) {
      Frame f = new Frame("Test");
      Button b = new Button("Press Me!");
      b.addActionListener(new ButtonHandler());
      f.add("Center", b);
      f.pack();
      f.setVisible(true);
   }
}
```

The `ButtonHandler` class is the handler class to which the event is delegated.

```
import java.awt.event.*;
public class ButtonHandler implements
      ActionListener {
   public void actionPerformed(ActionEvent e) {
      System.out.println("Action occurred");
   }
}
```

Where:

- 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, the object can register a listener for `ActionEvents` through the `addActionListener` method, specifying the class object that implements the `ActionListener` interface.
- When the `Button` object is clicked on using the mouse, an `ActionEvent` is sent to any `ActionListener` that is registered through the `actionPerformed (ActionEvent)` method.

There are several advantages to this approach:

- Events are not accidentally handled; in the hierarchical model it is possible for an event to propagate to a container and get handled at a level that is not expected.
- You can create filter (adapter) classes to classify event actions.
- There is better distribution of work among classes.
- It provides support for JavaBeans™.

There are also some issues and disadvantages with this model that are worth considering:

- It is more complex, at least initially, to understand.
- Moving code with the hierarchical event model to the delegation event model is not easy.
- Although the current release of the Java programming language supports the JDK 1.0 hierarchical event model in addition to the delegation model, you cannot mix the two event models.

GUI Behavior

Categories of Events

The general mechanism for receiving events from components has been described in the context of one single type of event. A number of events are defined in the `java.awt.event` package, and you can add third-party components to this list.

For each category of events, there is an interface that must be defined by any class that wants to receive the events. That interface demands that one or more methods be defined. Those methods are called when particular events arise. Table C-3 lists the event categories, and gives the interface name for each category and the method names demanded. The method names are mnemonic, indicating the conditions that cause the method to be called.

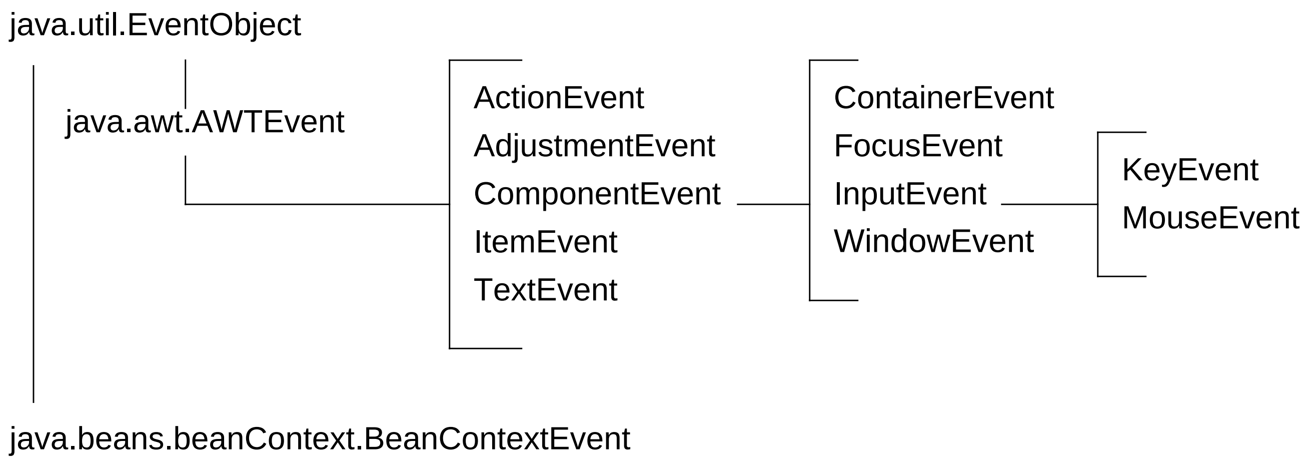


Figure C-3 Event Categories

Table C-1 Event Categories

Category	Interface Name	Methods
Action	`ActionListener`	`actionPerformed(ActionEvent)`
Item	`ItemListener`	`itemStateChanged(ItemEvent)`
Mouse motion	`MouseMotionListener`	`mouseDragged(MouseEvent)`
		`mouseMoved(MouseEvent)`
Mouse button	`MouseListener`	`mousePressed(MouseEvent)`
		`mouseReleased(MouseEvent)`
		`mouseEntered(MouseEvent)`
		`mouseExited(MouseEvent)`
		`mouseClicked(MouseEvent)`
Key	`KeyListener`	`keyPressed(keyEvent)`
		`keyReleased(keyEvent)`
		`keyTyped(keyEvent)`
Focus	`FocusListener`	`focusGained(FocusEvent)`
		`focusLost(FocusEvent)`
Adjustment	`AdjustmentListener`	`adjustmentValueChanged(AdjustmentEvent)`
Component	`ComponentListener`	`componentMoved(ComponentEvent)`
		`componentHidden(ComponentEvent)`
		`componentResized(ComponentEvent)`
		`componentShown(ComponentEvent)`
Window	`WindowListener`	`windowClosing(WindowEvent)`
		`windowOpened(WindowEvent)`
		`windowIconified(WindowEvent)`
		`windowDeiconified(WindowEvent)`
		`windowClosed(WindowEvent)`

Table C-1 Event Categories

Category	Interface Name	Methods
		windowActivated(WindowEvent)
		windowDeactivated(WindowEvent)
Container	ContainerListener	componentAdded(ContainerEvent)
		componentRemoved(ContainerEvent)
Text	TextListener	textValueChanged(TextEvent)

A More Complex Example

A more complex example might be tracking the movement of the mouse while the mouse button is pressed (*mouse dragging)*, and detecting mouse movement.

The events caused by moving the mouse with a button pressed can be picked up by a class that implements the `MouseMotionListener` interface. That 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 mouse events, including mouse clicking, implement the `MouseListener` interface. This interface includes several events, including `mouseEntered`, `mouseExited`, `mousePressed`, `mouseReleased`, and `mouseClicked`.

When the mouse drag or key typed events occur, report the information about the position of the mouse and the key that was pressed into label fields.

The following program tracks the movement of the mouse when the mouse button is pressed.

```
import java.awt.*;
import java.awt.event.*;

public class TwoListen implements
      MouseMotionListener, MouseListener {
   private Frame f;
   private TextField tf;

   public static void main(String args[]) {
      TwoListen two = new TwoListen();
      two.go();
   }

   public void go() {
      f = new Frame("Two listeners example");
      f.add (new Label ("Click and drag the mouse"),
      BorderLayout.NORTH);
      tf = new TextField (30);
      f.add (tf, BorderLayout.SOUTH);

      f.addMouseMotionListener(this);
      f.addMouseListener (this);
      f.setSize(300, 200);
      f.setVisible(true);
   }

   // These are MouseMotionListener events
   public void mouseDragged (MouseEvent e) {
      String s =
      "Mouse dragging:  X = " + e.getX() +
      " Y = " + e.getY();
      tf.setText (s);
   }

   public void mouseMoved (MouseEvent e) {
   }

   // These are MouseListener events
   public void mouseClicked (MouseEvent e) {
   }

   public void mouseEntered (MouseEvent e) {
      String s = "The mouse entered";
      tf.setText (s);
   }

   public void mouseExited (MouseEvent e) {
      String s = "The mouse has left the building";
      tf.setText (s);
   }

   public void mousePressed (MouseEvent e) {
   }

   public void mouseReleased (MouseEvent e) {
   }
}
```

Declaring Multiple Interfaces

The class is declared using the following:

```
implements MouseMotionListener, MouseListener
```

You can declare multiple interfaces by using comma separation.

Listening to Multiple Sources

The following methods:

```
f.addMouseListener(this);
f.addMouseMotionListener(this);
```

cause handler methods to be called in the `TwoListen` class. You can listen to as many event sources as you want, both different categories of event and different sources (for example, `Frames` and `Buttons`).

Obtaining Details About the Event

When the handler methods, such as `mouseDragged,` are called, they receive an argument that contains potentially important information about the original event. To determine the details of what information is available for each category of event, check the appropriate class documentation in the `java.awt.event` package.

Multiple Listeners

The AWT event listening framework 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, say, a context sensitive help system is being added to an existing program.

The listener mechanism allows you to call an `addListener` method as many times as is needed, specifying as many different listeners as your design requires. All registered listeners have their handler methods called when the event occurs.

Note – The order in which the handler methods are called is undefined. Generally, if the order of invocation matters, then the handlers are not unrelated, and this facility is not used to call them. Instead, register only the first listener, and have that one call the others directly.

Event Adapters

It is a lot of work to have to implement all of the methods in each of the `Listener` interfaces, particularly the `MouseListener` interface and `ComponentListener` interface.

The `MouseListener` interface, for example, defines the following methods:

- `mouseClicked (MouseEvent)`
- `mouseEntered (MouseEvent)`
- `mouseExited (MouseEvent)`
- `mousePressed (MouseEvent)`
- `mouseReleased (MouseEvent)`

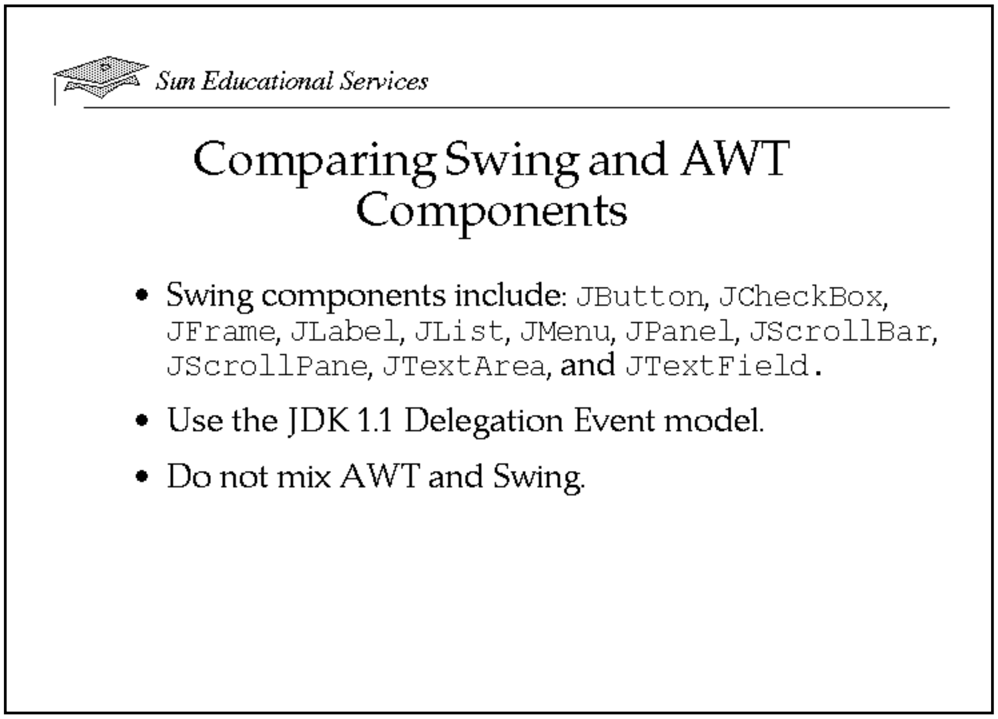
As a convenience, the Java programming language provides an adapter class for each `Listener` interface that implements the appropriate `Listener` interface, but leaves the implemented methods empty.

This way, the `Listener` routine that you define can extend the `Adapter` class and override only the methods that you need. For example:

```
import java.awt.*;
import java.awt.event.*;

public class MouseClickHandler extends MouseAdapter {

   // We just need the mouseClick handler, so we use
   // the Adapter to avoid having to write all the
   // event handler methods
   public void mouseClicked (MouseEvent e) {
      // Do stuff with the mouse click...
   }
}
```

Swing Foundations D

This appendix introduces some of the essential aspects of the Swing set and addresses issues that you must understand to succeed in translating a GUI from the AWT to Swing. This appendix also introduces some fundamental Swing components, including Icons and Buttons.


Sun Educational Services

Comparing Swing and AWT Components

- Swing components include: `JButton`, `JCheckBox`, `JFrame`, `JLabel`, `JList`, `JMenu`, `JPanel`, `JScrollBar`, `JScrollPane`, `JTextArea`, and `JTextField`.
- Use the JDK 1.1 Delegation Event model.
- Do not mix AWT and Swing.

Comparing Swing and AWT Components

Naming and Event Model

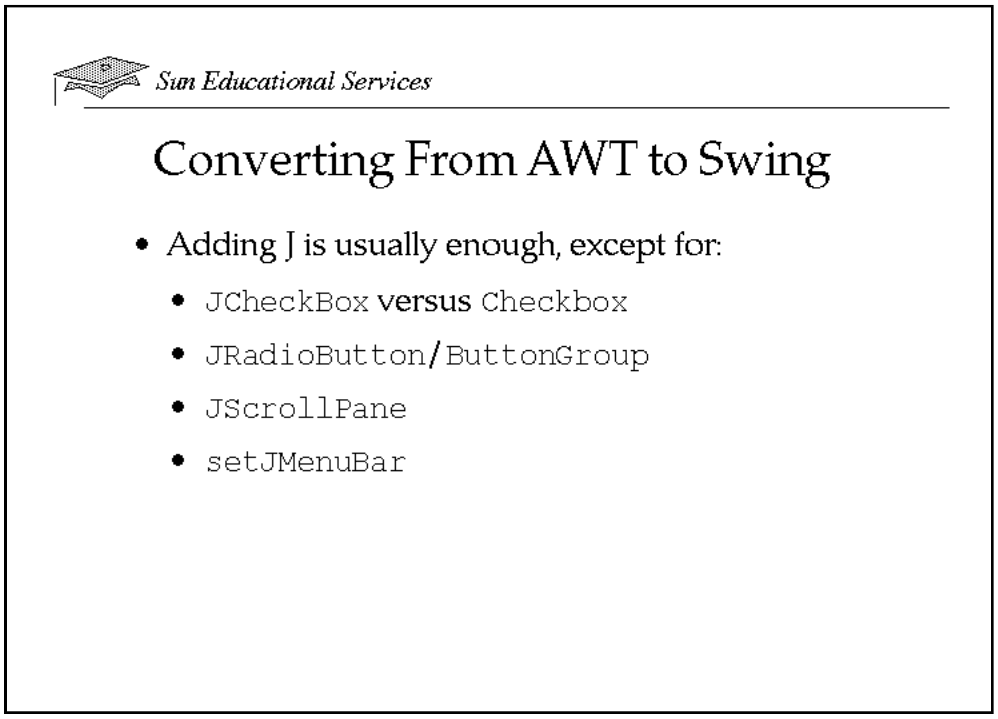
The AWT, while limited, provides a selection of important components, and the functionality is replicated in the Swing set. In general, the Swing equivalent of an AWT component has the same name with the addition of the letter "J," so `Button` becomes `JButton`, `Label` becomes `JLabel`, and so on.

For the most part, the behavior of these AWT-equivalent Swing components is an exact replica of the AWT version, although there might be some additional functionality. However, one important restriction, is that the Swing components require the use of the delegation event model.

Therefore, converting a user interface from AWT to Swing is fairly straightforward, requiring little more than the addition of "J" in front of the AWT component class names.

Selecting Swing or *AWT*

Although the Swing components do work fairly well when used alongside AWT components, complications can arise. Because of this, you should ensure that a user interface uses exclusively Swing components and does not mix Swing and AWT in the same program.


Sun Educational Services

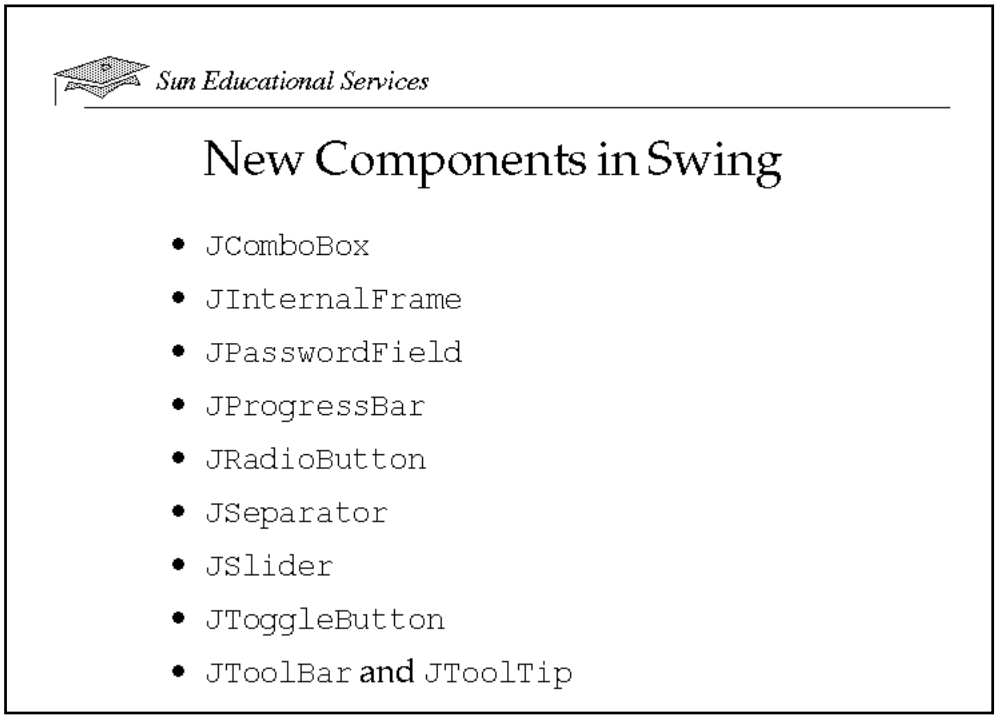
Converting From AWT to Swing

- Adding J is usually enough, except for:
  - `JCheckBox` versus `Checkbox`
  - `JRadioButton/ButtonGroup`
  - `JScrollPane`
  - `setJMenuBar`

Converting From AWT to Swing

Sometimes, simply adding a "J" to the front of classnames is not enough to make a complete translation from AWT to Swing.

- The `Checkbox` class in AWT is replaced by the `JCheckBox` class. There are two aspects to note here:
  - The spelling of `JCheckBox` has a capital B, unlike `Checkbox.`
  - The Swing set has a separate class, `JRadioButton`, that should be used with `ButtonGroup` objects to implement radio button behavior.
- In Swing, components do not have automatic scrolling. Instead, components like `JList` and `JTextArea` are added to the `JScrollPane` container if they need scrollbars.
- As the class names change by the addition of a "J," the method `setMenuBar` of a `java.awt.Frame` becomes `setJMenuBar` in a `JFrame`.



Sun Educational Services

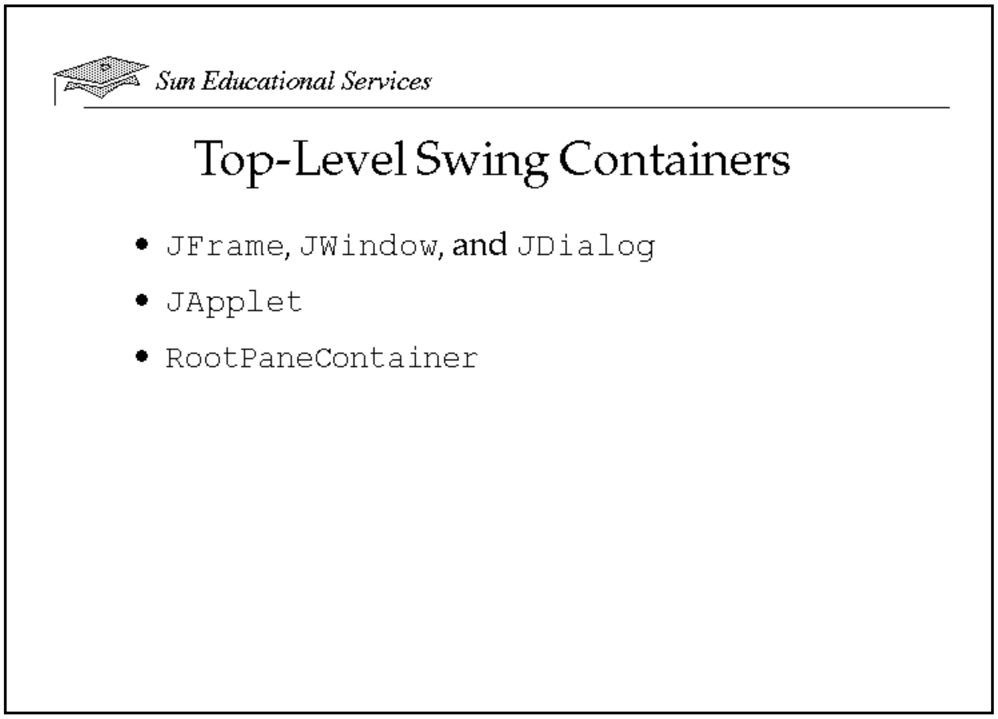
New Components in Swing

- `JComboBox`
- `JInternalFrame`
- `JPasswordField`
- `JProgressBar`
- `JRadioButton`
- `JSeparator`
- `JSlider`
- `JToggleButton`
- `JToolBar` and `JToolTip`

New Components in Swing

In addition to replacements for the existing AWT component set, Swing provides many new components. Some of these are relatively simple to use, and probably do not require much discussion. A few, however, do need to be covered in more detail. They are reserved for discussion later in this appendix.

You can investigate the full set of Swing components using the SwingSet demonstration that is provided with the Swing or Java 2 platform distribution.



Sun Educational Services

Top-Level Swing Containers

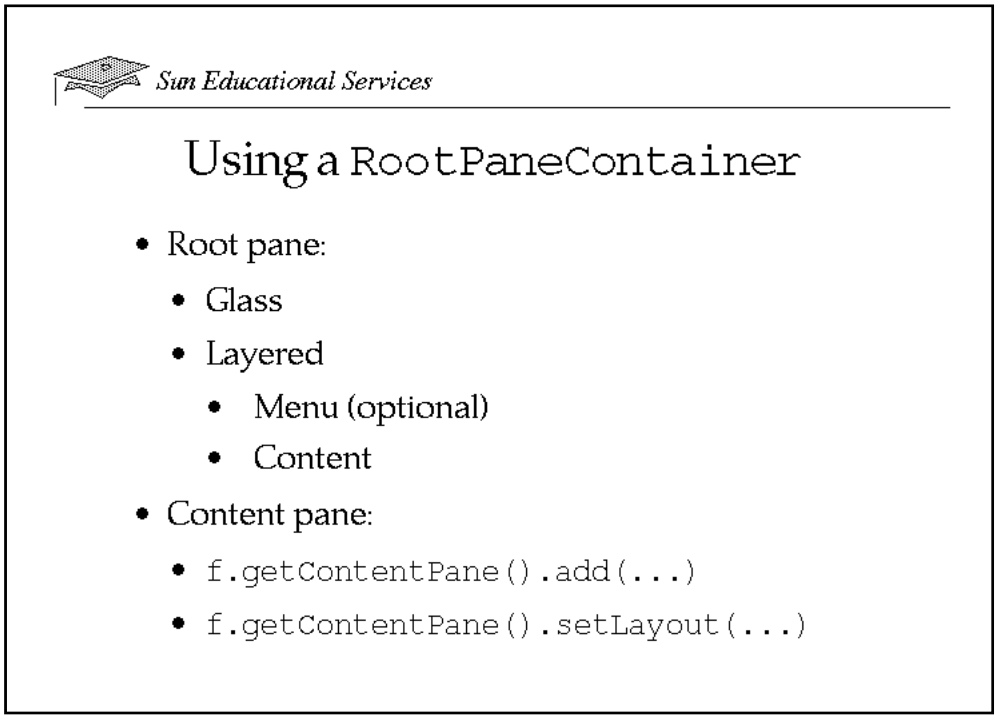
- JFrame, JWindow, and JDialog
- JApplet
- RootPaneContainer

Top-Level Swing Containers

As previously mentioned, you should not use AWT and Swing components in the same layout, so you should investigate the top-level Swing containers that form the starting point for your interfaces.

There are three top-level Swing containers: `JFrame`, `JWindow`, and `JDialog`. There is also a special class, `JApplet,` which is not strictly a top-level container but is worth mentioning here because it should be used as the "top level" of any applet that uses Swing components.

Each of these four containers (including `JApplet`) implements a special interface called `RootPaneContainer`.



Sun Educational Services

Using a RootPaneContainer

- Root pane:
  - Glass
  - Layered
    - Menu (optional)
    - Content
- Content pane:
  - `f.getContentPane().add(...)`
  - `f.getContentPane().setLayout(...)`

Using a `RootPaneContainer`

`RootPaneContainer` is a container for a number of other panes; these are the root, glass, layered, and content panes. Most of the time, you should be concerned with only the content pane.

To perform most operations, such as setting a layout manager or adding new components to the container as a whole, refer to the content pane. It is obtained from a `RootPaneContainer` using the method `getContentPane`.

Content panes have a `BorderLayout` by default, so, to add a `JButton` to a `JFrame` referred to by the variable `myFrame`, use code of this form:

```
myFrame.getContentPane().add(myJButton, BorderLayout.NORTH);
```

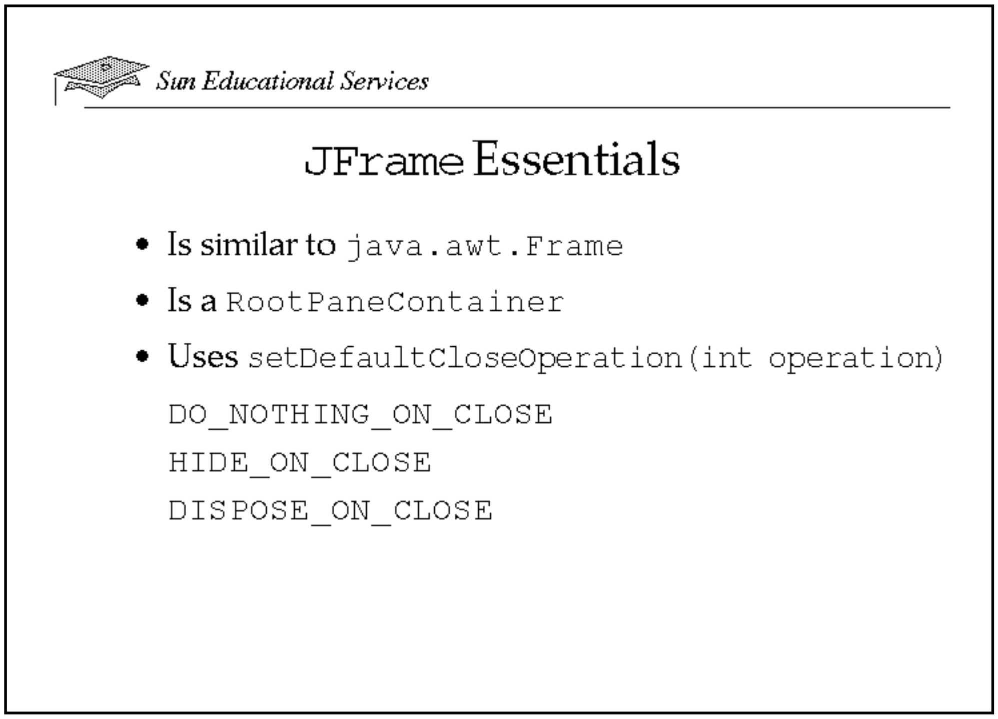
The Root, Glass, and Layered Panes

The `JRootPane` forms the basis of a stack of several components. It contains a `GlassPane` and a `LayeredPane`, and has a layout manager that controls all the other panes and the menu if it is present.

The `LayeredPane` can contain a menu bar, and the all-important `ContentPane`.

A `LayeredPane` allows a specific stacking order to be enforced on the components that it contains. This is used for pop-up menus, tool tips, and similar elements.

The `GlassPane` is a transparent pane that is used to collect events.



Sun Educational Services

JFrame Essentials

- Is similar to java.awt.Frame
- Is a RootPaneContainer
- Uses setDefaultCloseOperation(int operation)

  DO_NOTHING_ON_CLOSE
  HIDE_ON_CLOSE
  DISPOSE_ON_CLOSE

`JFrame` *Essentials*

Much of the time when you use a `RootPaneContainer`, it is actually a `JFrame`. This class is the Swing replacement for a `java.awt.Frame` class.

Because the `JFrame` is a `RootPaneContainer`, you must take care referring to the content pane while adding components or setting a layout manager.

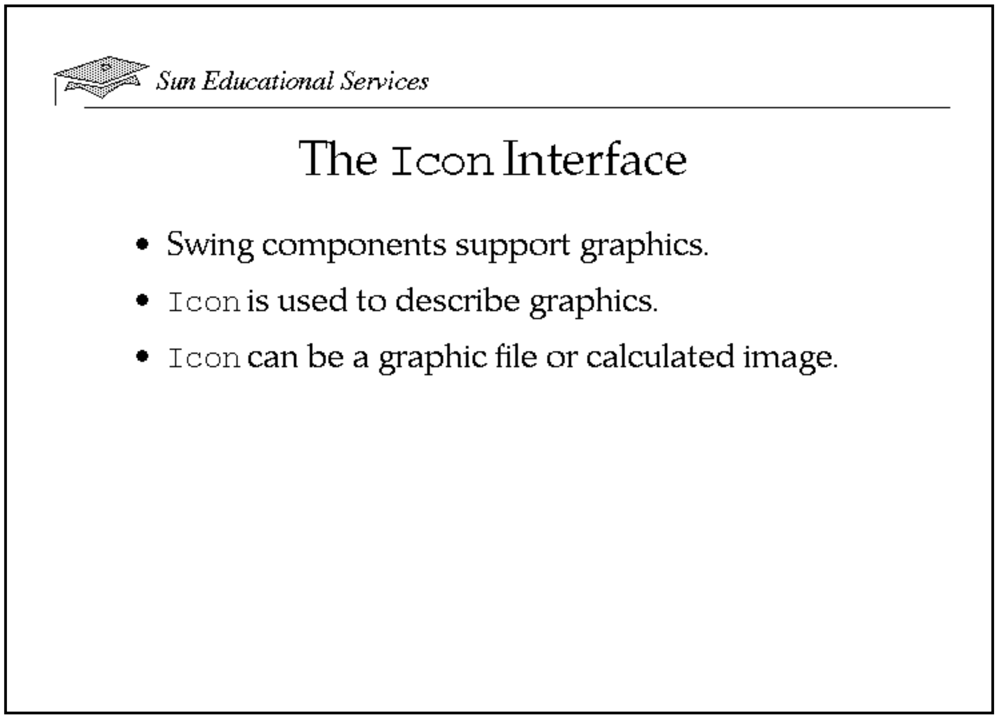
Reacting to the System Menu

In addition to being the most common top-level Swing container, the `JFrame` has a useful enhancement over the original `java.awt.Frame`. It has behavior attached to its system menu button. In AWT, if you wanted a window to close when the system menu was used, you had to code this behavior using a listener. With a `JFrame`, you have a choice of three reactions that can be set using the `setDefaultCloseOperation` method. The options are:

- `DO_NOTHING_ON_CLOSE`
- `HIDE_ON_CLOSE`
- `DISPOSE_ON_CLOSE`

These constants are declared in the `JFrame` class because `JFrame` implements the interface `javax.swing.WindowConstants`.

Note – If you close the last or only `JFrame`, and request `DISPOSE_ON_CLOSE` behavior, the AWT thread and the Java Virtual Machine (JVM) do not exit.



Sun Educational Services

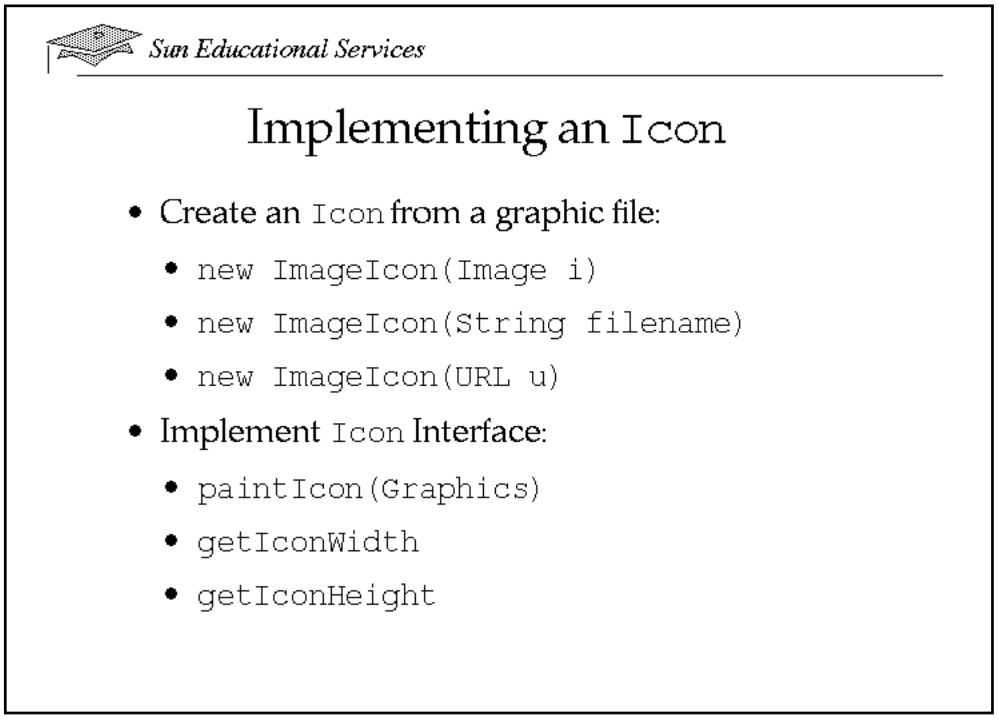
The Icon Interface

- Swing components support graphics.
- Icon is used to describe graphics.
- Icon can be a graphic file or calculated image.

The `Icon` *Interface*

Most Swing components support graphics. You can place images in buttons, labels, lists, and menus. At the heart of all of the graphics capabilities in Swing is the `Icon` interface.

The `Icon` interface provides you with a means of drawing a picture. An implementation of the `Icon` interface can display a graphic file, such as a graphics interchange format (GIF) or joint photographic group (JPEG) image. You can also use it to create a "calculated" image.



Sun Educational Services

Implementing an `Icon`

- Create an `Icon` from a graphic file:
  - `new ImageIcon(Image i)`
  - `new ImageIcon(String filename)`
  - `new ImageIcon(URL u)`
- Implement `Icon` Interface:
  - `paintIcon(Graphics)`
  - `getIconWidth`
  - `getIconHeight`

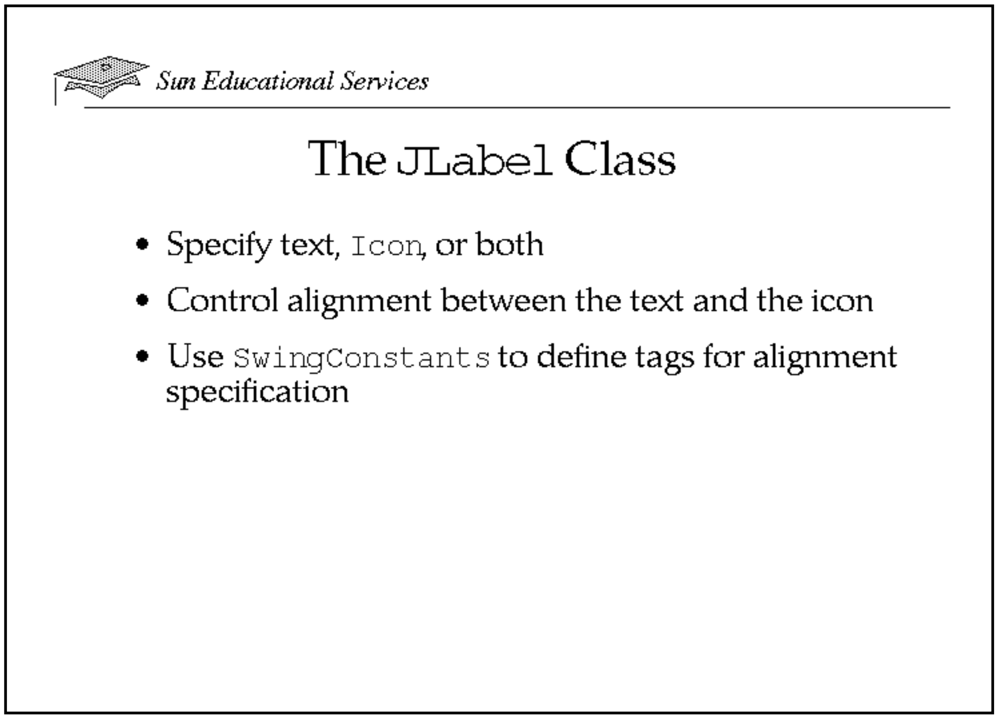
Implementing an `Icon`

The simplest way to create an `Icon` interface is to supply an image file to the constructor of the `ImageIcon` class. Three constructors provide different ways of specifying the location of the image file:

```
Icon icon = new ImageIcon(Image i);
Icon icon = new ImageIcon(String filename);
Icon icon = new ImageIcon(URL url);
```

These constructors create an `Icon` object that you can use with most of the other Swing components.

You can also create an `Icon` by implementing the `Icon` interface in a new class of your own. To do this, you must implement the methods `paintIcon`, `getIconWidth`, and `getIconHeight`.



Sun Educational Services

The JLabel Class

- Specify text, Icon, or both
- Control alignment between the text and the icon
- Use SwingConstants to define tags for alignment specification

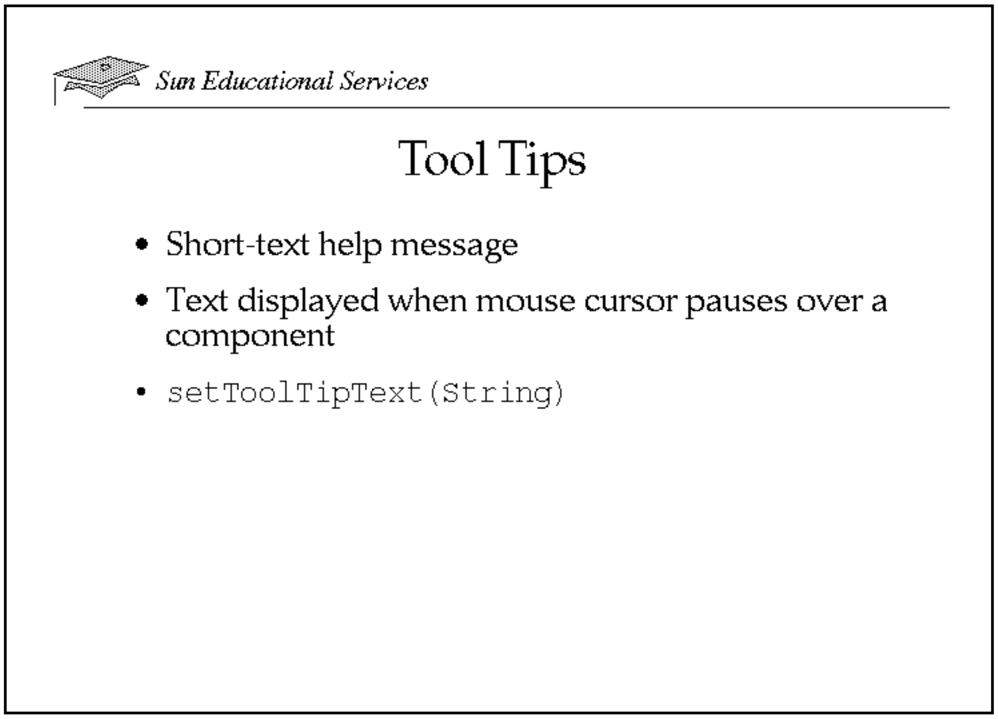
The JLabel *Class*

One of the simplest Swing components is the JLabel. This class contains all of the functionality associated with the AWT Label and supports icons.

JLabel has a variety of constructors that enable you to specify text, an Icon, or both.

You can construct a label using only an Icon. This is a convenient way to put an image into an application.

Additionally, you can specify how the text and the Icon are to align. The alignment feature uses values defined in the SwingConstants interface.



Sun Educational Services

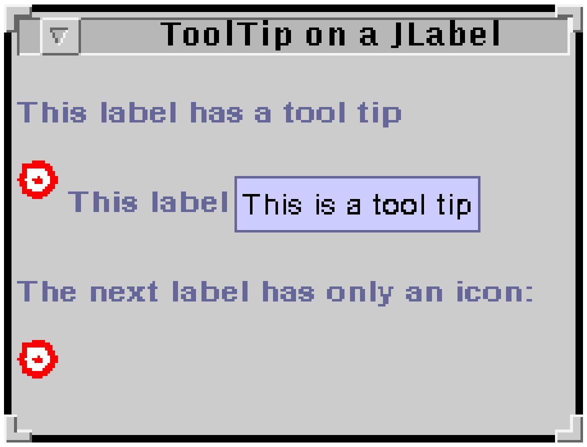
Tool Tips

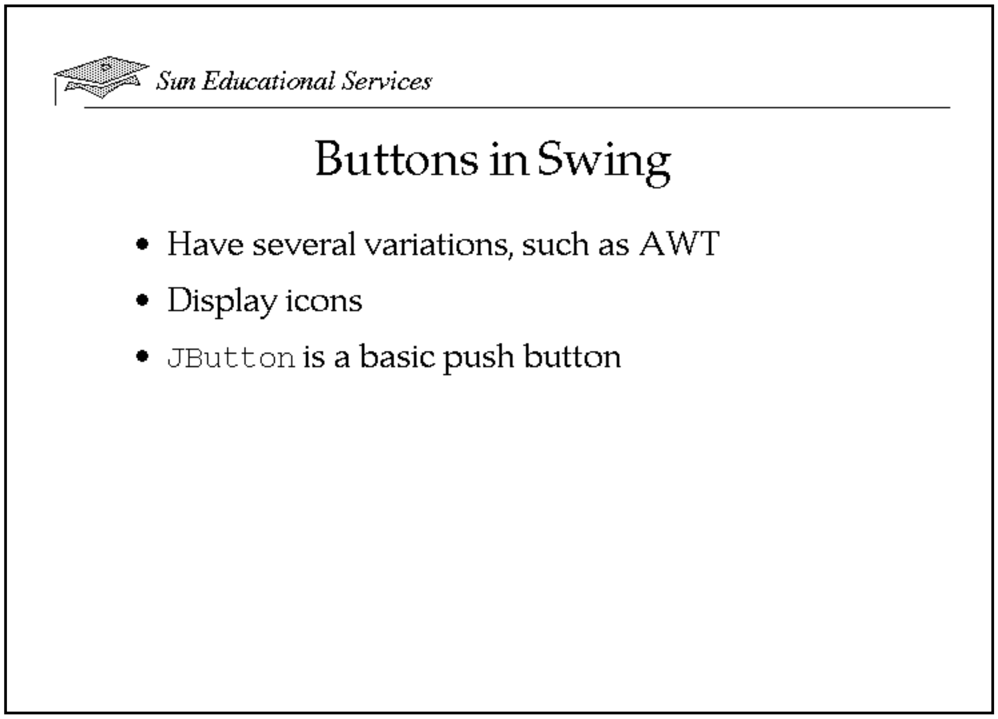
- Short-text help message
- Text displayed when mouse cursor pauses over a component
- `setToolTipText(String)`

Tool Tips

Swing components support tool tips. Tool tips are short help messages that are displayed when the mouse cursor pauses over a component.

```
JLabel tipLabel = new JLabel("This label has a
tooltip");
tipLabel.setToolTipText("This is a tooltip");
add(tipLabel);
```





Sun Educational Services

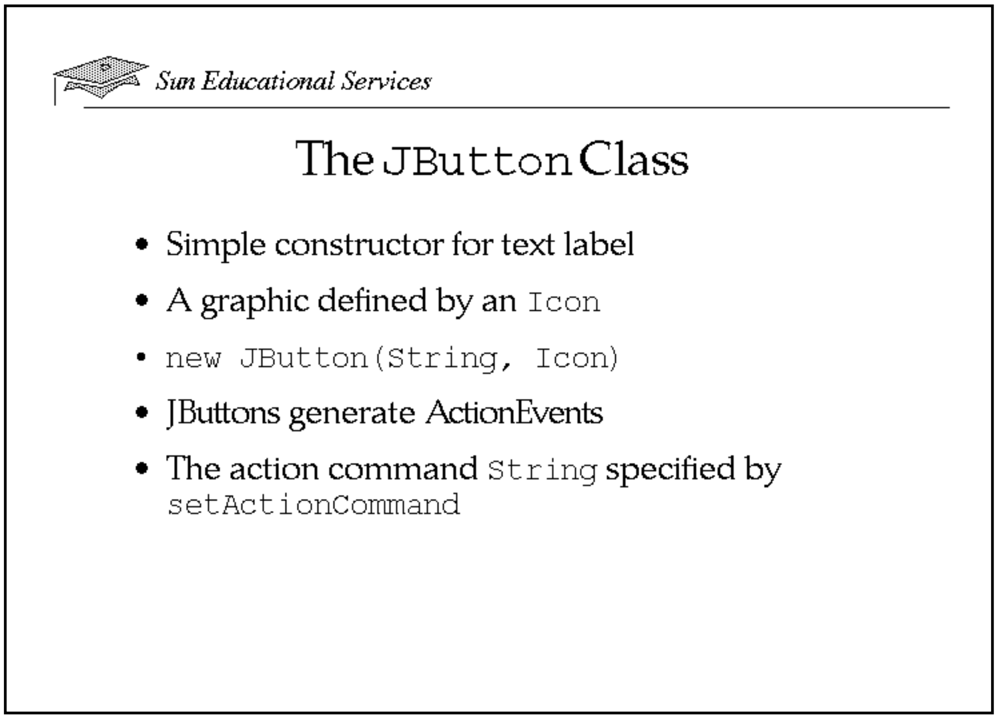
Buttons in Swing

- Have several variations, such as AWT
- Display icons
- `JButton` is a basic push button

Buttons in Swing

The Swing package provides versions of all the AWT buttons. Like other Swing components, Swing buttons can display graphics as well as text.

You use the `JButton` class to create a basic push button in Swing.



Sun Educational Services

The JButton Class

- Simple constructor for text label
- A graphic defined by an Icon
- new JButton(String, Icon)
- JButtons generate ActionEvents
- The action command String specified by setActionCommand

The JButton *Class*

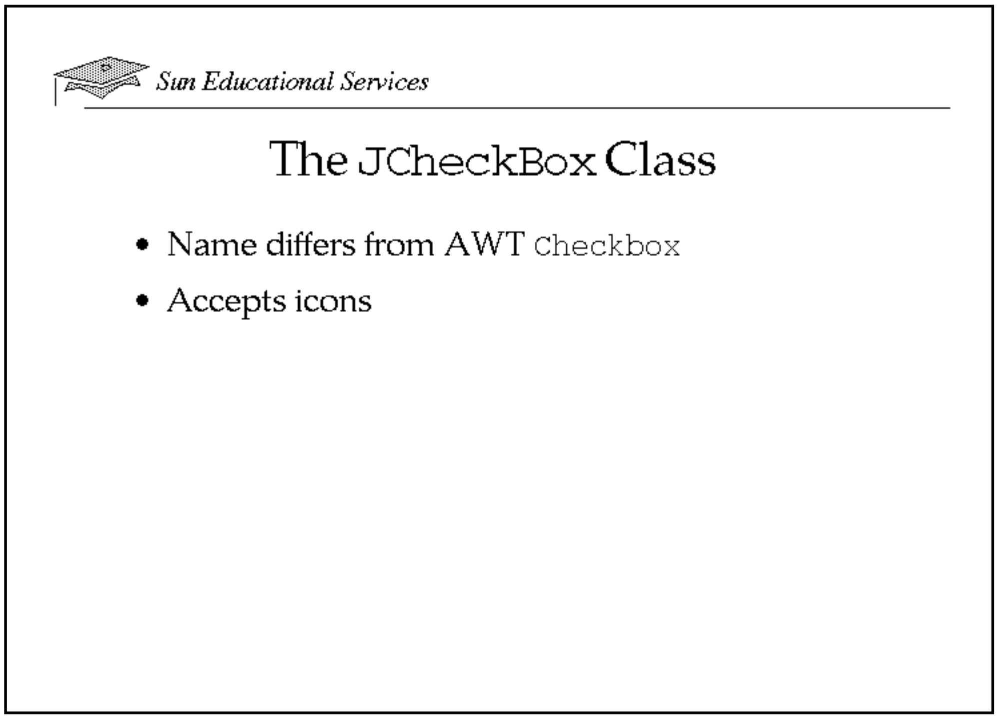
You can create JButton objects with a simple String argument to the constructor, in which case they display that text as their label, just as the AWT button did.

Creating graphical buttons is straightforward if you use an Icon that defines the graphic you want displayed.

```
Icon cutIcon = new ImageIcon("cut32x32.gif");
JButton cutButton = new JButton("Cut", cutIcon);
```

Clicking on a JButton generates an ActionEvent. You might want to set the action command property of your button so that the ActionEvent carries a particular command string.

If you create a JButton with text, the label text is used by default as the action command string. However, if you create a graphics-only button or if the default action command string (which is the text label of the button) is not what you need, define the action command explicitly using the setActionCommand method.



Sun Educational Services

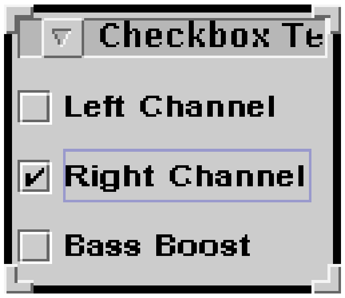
The JCheckBox Class

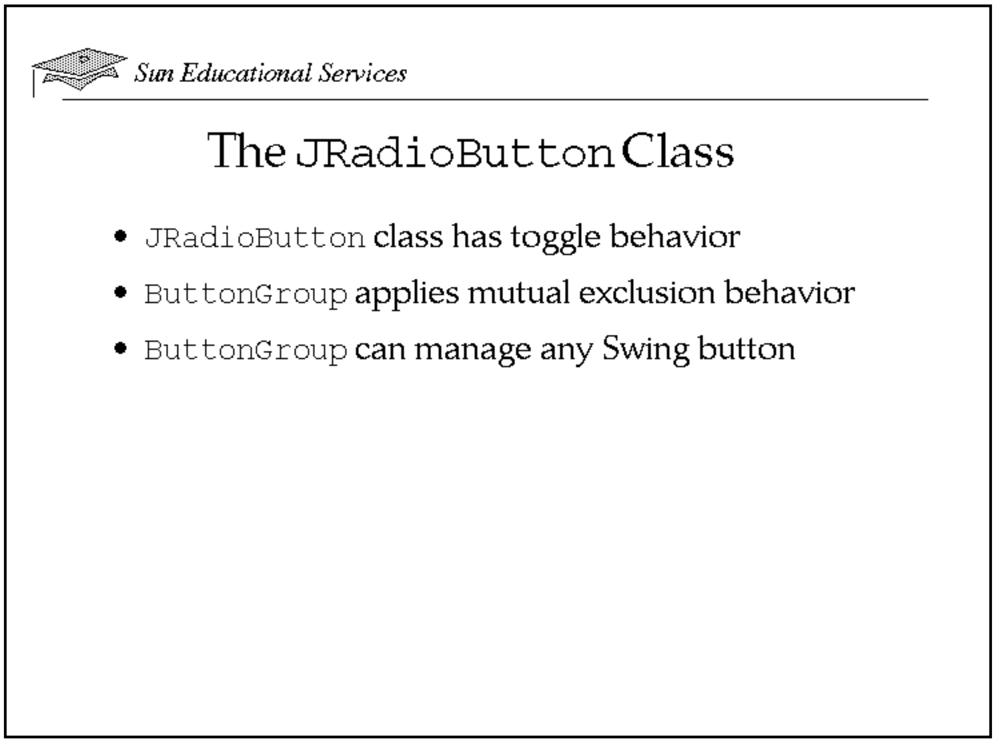
- Name differs from AWT Checkbox
- Accepts icons

The JCheckBox *Class*

Checkboxes are directly supported by Swing, using the JCheckBox class. The class name has a capital "B" in it, which is different from the corresponding java.awt.Checkbox class.

Like other Swing components, you can use icons on a JCheckBox.





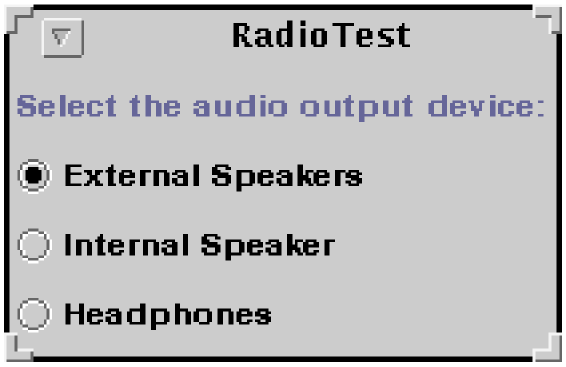
Sun Educational Services

The JRadioButton Class

- JRadioButton class has toggle behavior
- ButtonGroup applies mutual exclusion behavior
- ButtonGroup can manage any Swing button

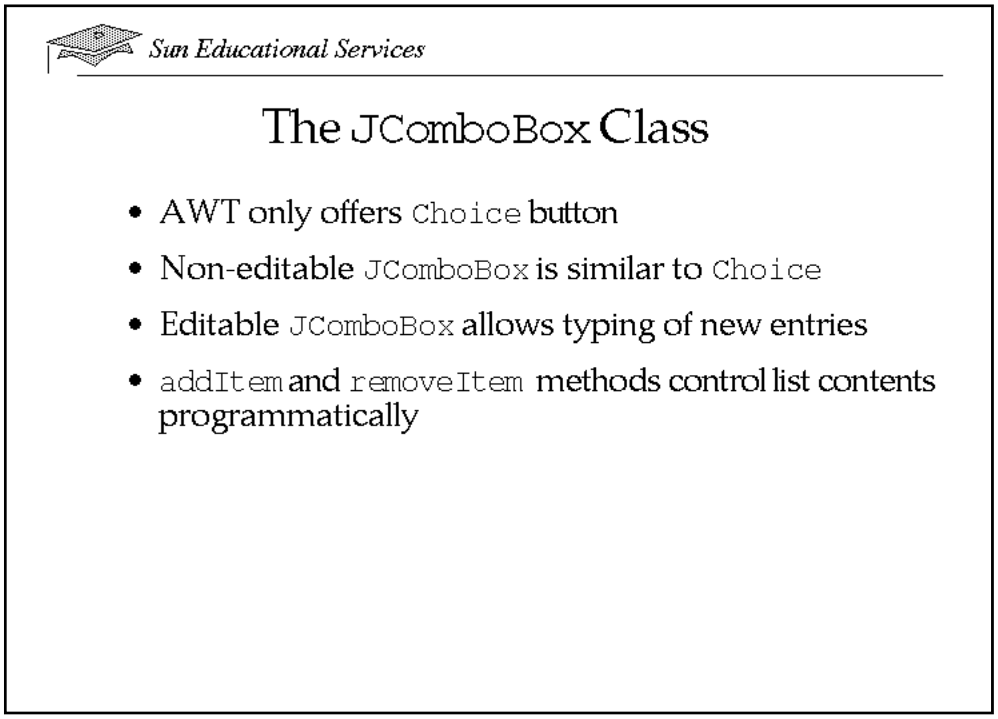
The JRadioButton *Class*

The JRadioButton class provides the radio buttons for Swing. Individually, a JRadioButton toggles on and off each time it is selected, just like a JCheckBox. To obtain the mutual exclusion behavior of a radio button, add the buttons to a ButtonGroup. A ButtonGroup is a manager that ensures that only one button is selected at one time. In the AWT package, this behavior was handled by the CheckboxGroup class, but Swing uses a more generic ButtonGroup class to manage all types of buttons.


RadioTest

Select the audio output device:

(● External Speakers) Internal Speaker Headphones


Sun Educational Services

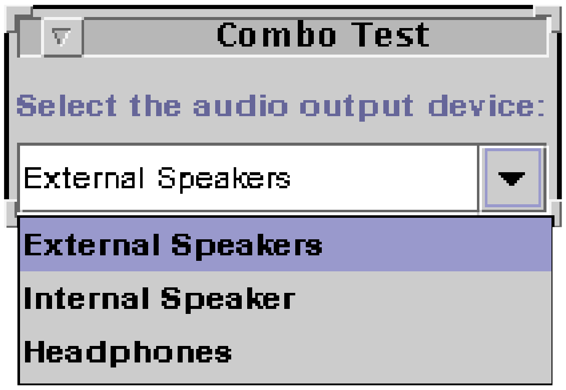
The JComboBox Class

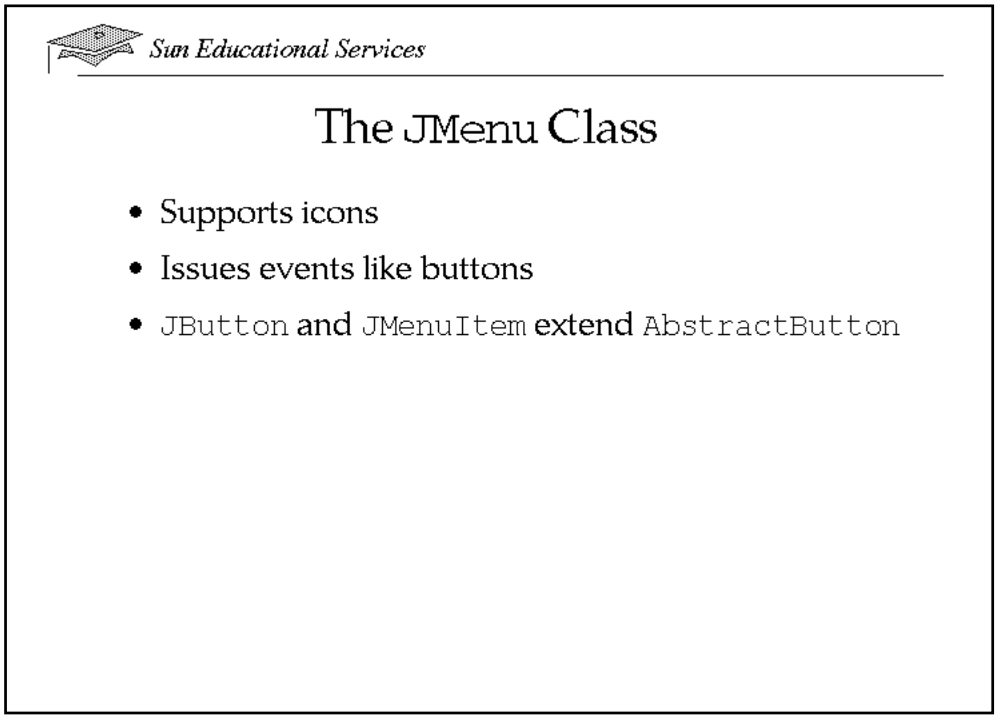
- AWT only offers Choice button
- Non-editable JComboBox is similar to Choice
- Editable JComboBox allows typing of new entries
- addItem and removeItem methods control list contents programmatically

The JComboBox *Class*

The Swing package also provides a combo box. The closest equivalent in the AWT package is the Choice button. However, the JComboBox can be made “editable.” If it is not editable, it behaves precisely like the Choice button. If it is editable, however, you have the option of selecting one of the standard choices or typing in a new value.

The JComboBox class has a convenient constructor that takes an array of objects to use as the initial choices. You can add and remove choices using the addItem and removeItem methods, respectively.





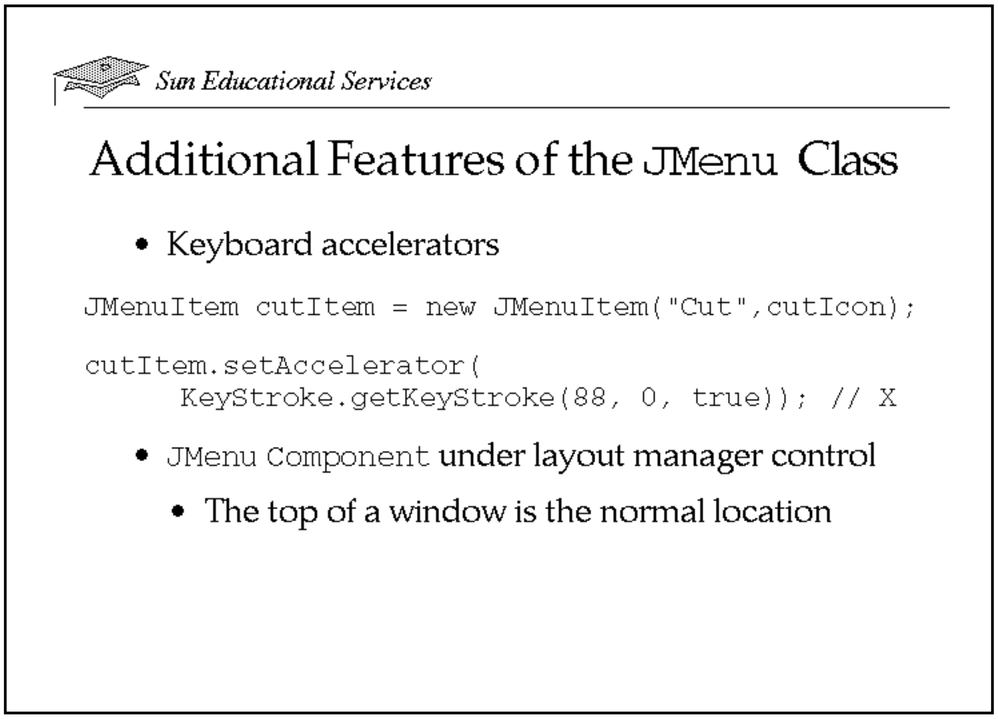
Sun Educational Services

The JMenu Class

- Supports icons
- Issues events like buttons
- `JButton` and `JMenuItem` extend `AbstractButton`

The `JMenu` *Class*

Swing provides a set of menu components. As with other Swing components, you can add icons to your menus. However, the menu items themselves are now subclasses of `AbstractButton`. The `AbstractButton` class is the parent class of both buttons and menu items, so this means that you can use the same event handler for buttons and menu items.



Sun Educational Services

Additional Features of the JMenu Class

- Keyboard accelerators

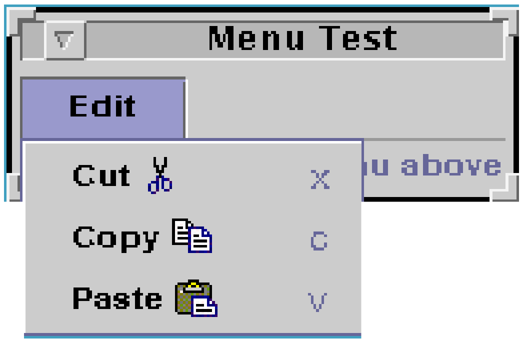
```
JMenuItem cutItem = new JMenuItem("Cut",cutIcon);

cutItem.setAccelerator(
      KeyStroke.getKeyStroke(88, 0, true)); // X
```

- JMenu Component under layout manager control
  - The top of a window is the normal location

Additional Features of the JMenu *Class*

Keyboard Accelerators

The JMenu class supports keyboard accelerators.



Menu Positioning

You can place menu bars anywhere in your application. This is a significant change from the menu behavior in AWT. Because the `JMenuBar` class is an extension of `JComponent`, you can position a `JMenuBar` anywhere you would add items, such as buttons or labels; they are no longer restricted to the physical frames of your application. Be aware that positioning menu bars at locations other than the top of a window might confuse users of your application.

The AWT Component Library E

This appendix describes the key AWT components used to build user interfaces for programs.

Facilities of the AWT

The AWT provides a wide variety of standard facilities. This appendix introduces the components that are available to you, and outlines any particular pitfalls that you might need to know when using these components.

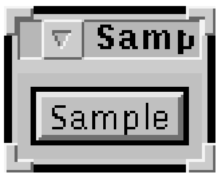
You should be aware of the full set of UI components so that you can choose the appropriate ones when building your own interfaces.

The AWT components provide mechanisms for controlling their appearance, specifically in terms of color and the font used for text display. They also support printing. This facility was added after JDK1.0.

Button

You have already learned about the `Button` component. It provides a basic "push to activate" user interface component. It can be constructed with a textual label that acts as a guide to the user as to its use.

```
Button b = new Button("Sample");
add(b);
```

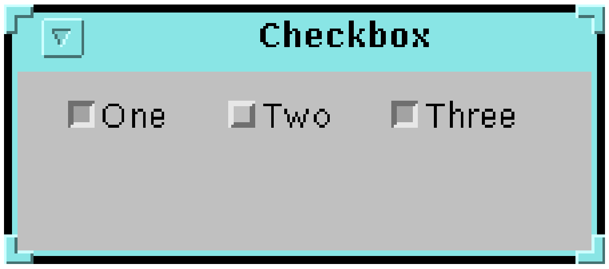


Use the `ActionListener` interface to react to button presses. The `getActionCommand` method of the `ActionEvent` that is issued when the button is clicked on is a label string by default. You can modify this using the button's `setActionCommand` method.

Checkbox

The `checkbox` component provides a simple “on/off” input device with a textual label beside it.

```
Checkbox one = new Checkbox("One", false);
Checkbox two = new Checkbox("Two", false);
Checkbox three = new Checkbox("Three", true);
add(one);
add(two);
add(three);
```



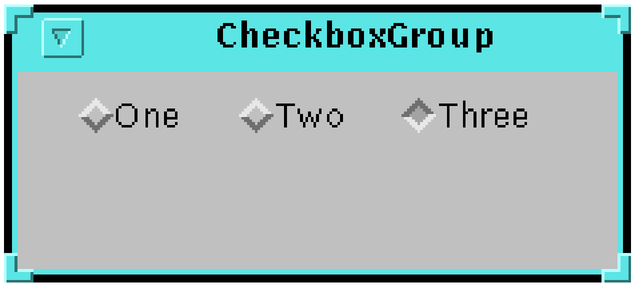
Selection, or deselection, of a checkbox is transmitted to the `ItemListener` interface. The `ItemEvent` that is passed indicates whether the operation selected or deselected a checkbox using the `getStateChange` method. This method returns the constant `ItemEvent.DESELECTED` or `ItemEvent.SELECTED` as appropriate. The method `getItem` returns a `String` object that represents the label string of the affected checkbox.

```
class Handler implements ItemListener {
   public void itemStateChanged(ItemEvent ev) {
      String state = “deselected”;
      if (ev.getStateChange() == ItemEvent.SELECTED){
      state = “selected”;
      }
      System.out.println(ev.getItem() + “ “ + state);
   }
}
```

CheckboxGroup – Radio Buttons

You can create checkboxes using a special constructor that takes an additional argument, a `CheckboxGroup`. If you do this, the appearance of the checkboxes is changed and all of the checkboxes that are related to the same checkbox group exhibit radio button behavior.

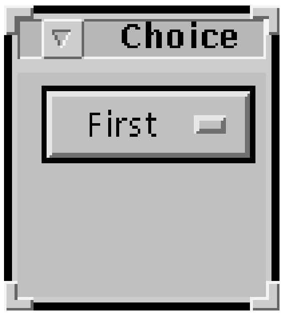
```
CheckboxGroup cbg = new CheckboxGroup();
Checkbox one = new Checkbox("One", cbg, false);
Checkbox two = new Checkbox("Two", cbg, false);
Checkbox three = new Checkbox("Three", cbg, true);
add(one);
add(two);
add(three);
```



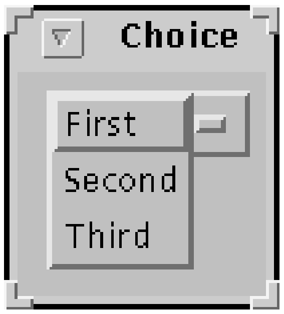
Choice

The `choice` component provides a "select one from this list" type input.

```
Choice c = new Choice();
c.addItem("First");
c.addItem("Second");
c.addItem("Third");
add(c);
```



When the choice is clicked on, it displays the list of items that have been added to it. The items added are `String` objects.

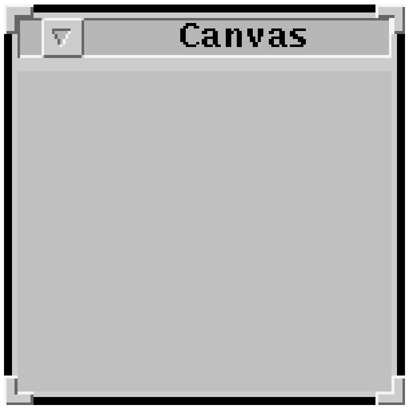


The `ItemListener` interface is used to observe changes in the choice. The details are the same as for the checkbox.

Canvas

A `Canvas` component provides a blank (background colored) space. It has a preferred size of zero by zero, so you must ensure that the layout manager gives it a nonzero size.

The space can be used to receive keyboard or mouse input, such as drawing or writing text. Generally the `Canvas` is used either as is to provide general drawing space, or as the basis of development to provide a working area for a custom component.



The `Canvas` can listen to all the events that are applicable to a general component. Notably you might want to add `KeyListener`, `MouseMotionListener`, or `MouseListener` objects to it to allow it to respond in some way to user input. The methods in these interfaces receive `KeyEvent` and `MouseEvent` objects, respectively.

Note – To receive key events in a canvas, it is necessary to call the `requestFocus` method of the canvas. If this is not done, you cannot direct keystrokes to the `Canvas`. Instead, the keys will go to another component, or perhaps be lost entirely. The sample code overleaf demonstrates this.

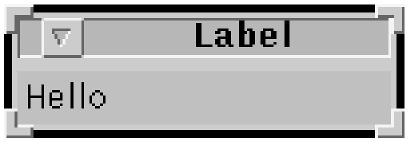
The following is an example of a `Canvas` component:

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class MyCanvas
   implements KeyListener, MouseListener {
   Canvas c;
   String s = "";

   public static void main(String args[]) {
      Frame f = new Frame("Canvas");
      MyCanvas mc = new MyCanvas();
      mc.c = new Canvas();
      f.add("Center", mc.c);
      f.setSize(150, 150);
      mc.c.addMouseListener(mc);
      mc.c.addKeyListener(mc);
      f.setVisible(true);
   }

   public void keyTyped(KeyEvent ev) {
      System.out.println("keyTyped");
      s += ev.getKeyChar();
      // Not a good drawing technique!!!
      c.getGraphics().drawString(s, 0, 20);
   }

   public void mouseClicked(MouseEvent ev) {
      System.out.println("mouseClicked");
      // force the focus onto the canvas
      c.requestFocus();
   }
   public void keyPressed(KeyEvent ev) {
      System.out.println("keyPressed");
   }

   public void keyReleased(KeyEvent ev) {
      System.out.println("keyReleased");
   }
   public void mousePressed(MouseEvent ev) {
      System.out.println("mousePressed");
   }

   public void mouseReleased(MouseEvent ev) {
      System.out.println("mouseReleased");
   }

   public void mouseEntered(MouseEvent ev) {
      System.out.println("mouseEntered");
   }

   public void mouseExited(MouseEvent ev) {
      System.out.println("mouseExited");
   }
}
```

Label

A `Label` object displays a single line of text. The program can change the text, but the user cannot. No special borders or other decoration are used to delineate a label.

```
Label l = new Label(“Hello”);
add(l);
```

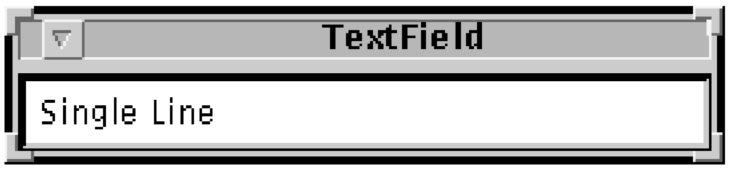


`Label`s are not usually expected to handle events, but can do so in the same manner as a `Canvas`. That is, keystrokes can be picked up reliably only by calling `requestFocus()`.

TextField

The `TextField` component is a single-line text device.

```
TextField f = new TextField("Single line", 30);
add(f);
```



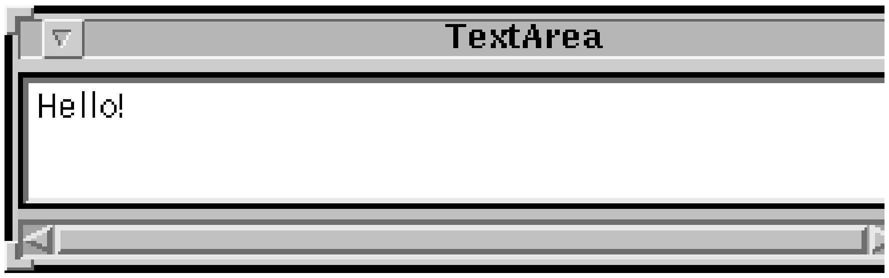
Because only one line is possible, an `ActionListener` can be informed using `actionPerformed` when the Enter or Return key is pressed. You can add other component listeners if desired.

This can be set to read only. It does not display scrollbars in either direction, but can scroll overly long text left to right if needed.

TextArea

The `TextArea` component is a multiple-line, multiple-column text input device. You can set it to non-editable using the method `setEditable(boolean)` in which case it becomes suitable for text browsing. It displays horizontal and vertical scrollbars.

```
TextArea t = new TextArea(“Initial text”, 4, 30);
add(t);
```



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, add a button that indicates Apply or Commit next to a text area to allow the user to indicate this.

TextComponent

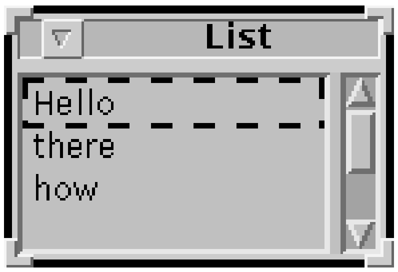
Both text area and text field are documented in two parts. If you look up a class called `TextComponent`, you will find many essential methods documented there. This is because text area and text field are subclasses of text component.

You have seen that the constructors for both `TextArea` and `TextField` classes allow you to specify a 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 can vary radically if a proportionally spaced font is used.

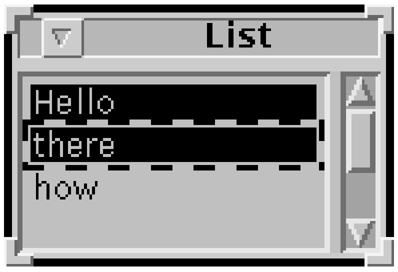
List

A `List` component allows you to present textual options, which are displayed in a region that allows several items to be viewed at one time to the user. The `List` is scrollable and supports both single- and multiple-selection modes.

```
List l = new List(4, true);
```



The numeric argument to the constructor defines the preferred height of the list in terms of a number of visible rows. As always, remember that this can be overridden by a layout manager. The Boolean argument indicates if the list should allow the user to make multiple selections.

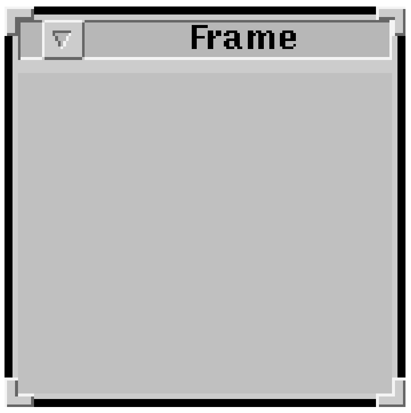


An ActionEvent, picked up using the `ActionListener` interface, is generated by the list in both single- and multiple-selection modes. Items are selected from the list with a single click. A double-click is needed to trigger the action of the list.

Frame

This is the general purpose top-level window. It has window manager decorations, such as a title bar and resize handles.

```
Frame f = new Frame("Frame");
```



You can set a `Frame`'s size using the `setSize` method. However, you should use the method `pack`, which causes the layout manager to calculate a size that neatly encloses all the components in the `Frame`. The size of the `Frame` is then set accordingly.

You can monitor the `Frame`'s events using all the listeners applicable to general components. You can use `WindowListener` to recognize, using the `windowClosing` method, that the Quit button on the Window Manager menu has been selected.

Do not try to listen to keyboard events from a `Frame` directly. Although the technique described for `Canvas` and `Label` components, that is, calling `requestFocus`, sometimes works, it is not reliable. If you need to follow keyboard events, add a `Canvas`, `Panel`, or something similar, to the `Frame` and add the listener to that component.

Panel

Panel is a general purpose container. You cannot use a panel in isolation, unlike Frames, Windows, and Dialogs.

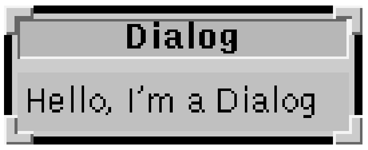
```
Panel p = new Panel();
```

Note – Because a Panel does not show up visibly, there is no screen shot of the Panel.

Panels can handle events, with the caveat that keyboard focus must be requested explicitly, as with the Canvas example earlier.

Dialog

A Dialog is similar to a Frame in that 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 takes over all forms of input until it is closed.



```
Dialog d = new Dialog(f, "Dialog", false);
d.add("Center",
new Label("Hello, I'm a Dialog"));
d.pack();
```

Dialogs are not normally made visible to the user when they are first created. Rather they are usually displayed in response to some other user interface action, such as clicking on a button.

```
public void actionPerformed(ActionEvent ev) {
d.setVisible(true);
}
```

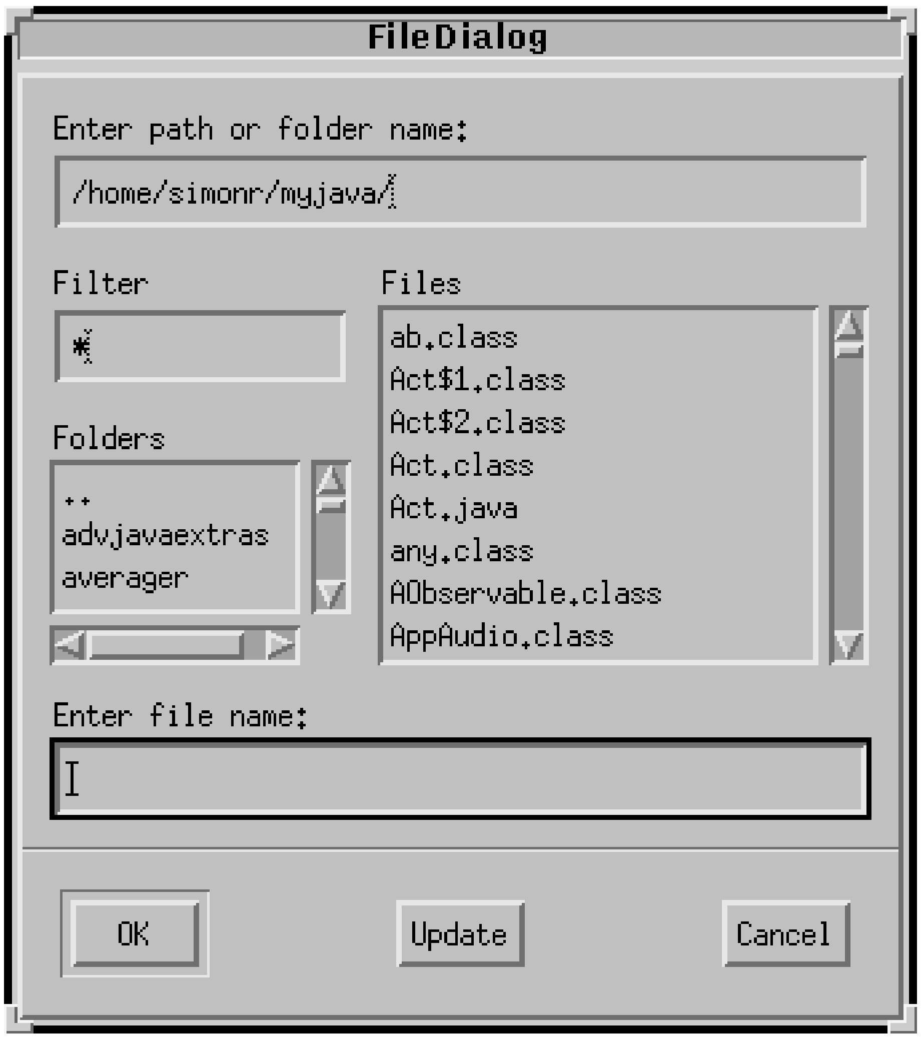
Note – Always treat a Dialog as a reusable device; that is, do not destroy the individual object when it is dismissed from the display, but keep it for later reuse. The garbage collector can make it too easy to waste memory, but remember that creating and initializing objects takes time and should not be done without some reason.

To hide a Dialog, you must call setVisible(false) on it. This is typically done by adding a WindowListener to it, and awaiting a call to the windowClosing method in that listener. This parallels the closing of a Frame.

FileDialog

This is an implementation of a file selection device. It has its own free- standing window, with decorations, and allows the user to browse the file system and select a particular file for further operations.

```
FileDialog d = new FileDialog(f, "FileDialog");
d.setVisible(true);
String fname = d.getFile();
```



/home/simonr/myjava/*

*

..
advjavaextras
averager

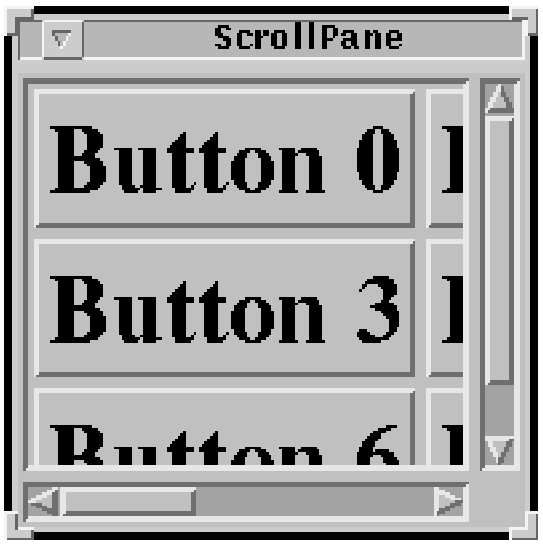
ab.class
Act$1.class
Act$2.class
Act.class
Act.java
any.class
AObservable.class
AppAudio.class

In general, it is not necessary to handle events from the `FileDialog`. The `setVisible(true)` call blocks until the user selects OK, at which point the name of the selected file can be retrieved. The file name is returned as a `String`.

ScrollPane

This provides a general container that cannot be used free standing. It provides a viewport onto a larger region and scrollbars to manipulate that viewport.

```
Frame f = new Frame("ScrollPane");
Panel p = new Panel();
ScrollPane sp = new ScrollPane();
p.setLayout(new GridLayout(3, 4));
sp.add(p);
f.add(sp);
f.setSize(200, 200);
f.setVisible(true);
```



The `ScrollPane` creates and manages the scroll bars as necessary. It holds a single component, and you cannot control the layout manager it uses. Instead, add a `Panel` to the `Scrollpane`, configure the layout manager of that `Panel`, and place your components into that `Panel`.

You will not generally handle events on a `ScrollPane`, rather you would do so from the components that it contains.

Menus

Menus are different from other components in one crucial way, you cannot add menus to ordinary containers and have them laid out by the layout manager. You can only add menus to *menu containers*. Generally, you can only start a menu "tree" by putting a menu bar into a `Frame`, using the `setMenuBar` method. From that point, you can add menus to the menu bar and add menus or menu items to the menus.

The exception to this is that a pop-up menu can be added to any component because no layout is required.

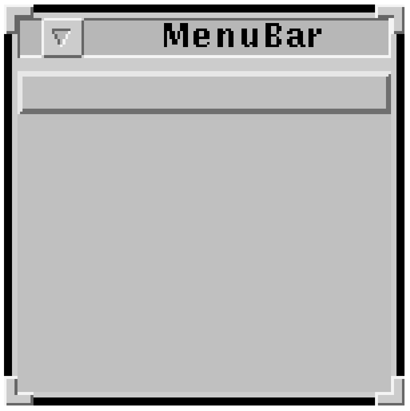
The Help Menu

One particular feature of the menu bar is that you can nominate one menu to be the Help menu. This is done with the method `setHelpMenu(Menu)`. The menu to be treated as the Help menu must be added to the menu bar, and treated in the appropriate fashion on the local platform. For X/Motif type systems, this involves flushing the menu entry to the right-hand end of the menu bar.

MenuBar

The MenuBar class provides the horizontal menu. It can be added only to a Frame object, and forms the root of all menu trees.

```
Frame f = new Frame("MenuBar");
MenuBar mb = new MenuBar();
f.setMenuBar(mb);
```

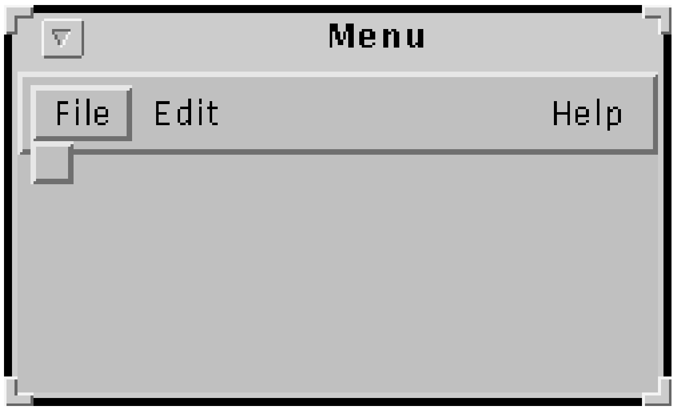


The MenuBar does not support listeners. This is because it is anticipated that all events that occur in the region of a MenuBar are processed automatically as part of the normal menu behavior.

Menu

The Menu class provides the basic pull-down menu. It can be added to either a MenuBar or to another Menu.

```
MenuBar mb = new MenuBar();
Menu m1 = new Menu("File");
Menu m2 = new Menu("Edit");
Menu m3 = new Menu("Help");
mb.add(m1);
mb.add(m2);
mb.setHelpMenu(m3);
```



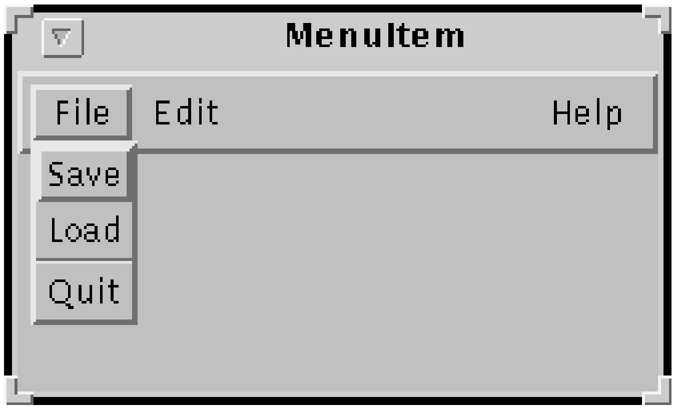
Note – The menus shown here are empty, which accounts for the appearance of the File menu.

You *can* add an ActionListener to a Menu object, but this would be unusual. Normally, menus are used to display and control menu items, which are discussed next.

MenuItem

MenuItems are the textual leaf nodes of a menu tree. They are generally added to a menu to complete the picture.

```
Menu m1 = new Menu("File");
MenuItem mi1 = new MenuItem("Save");
MenuItem mi2 = new MenuItem("Load");
MenuItem mi3 = new MenuItem("Quit");
m1.add(mi1);
m1.add(mi2);
m1.addSeparator();
m1.add(mi3);
```

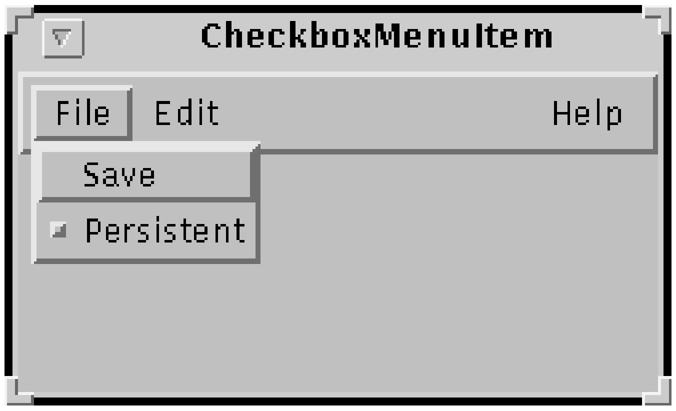


You usually add an ActionListener to a MenuItem object to provide behavior for your menus.

CheckboxMenuItem

This is a checkable menu item, so you can have selections—on or off choices—listed in menus.

```
Menu m1 = new Menu("File");
MenuItem mi1 = new MenuItem("Save");
CheckboxMenuItem mi2 =
new CheckboxMenuItem("Persistent");
m1.add(mi1);
m1.add(mi2);
```

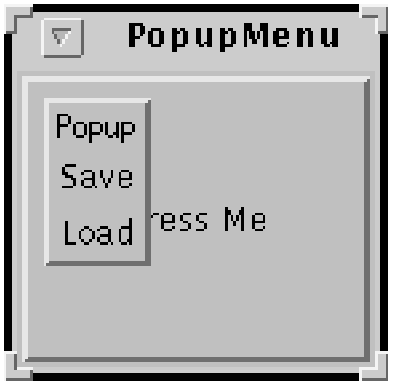


You should monitor the `CheckboxMenuItem` using the `ItemListener` interface. Therefore, the `itemStateChanged` method is called when the checkbox state is modified.

PopupMenu

This provides a standalone menu that can be displayed on any component. You can add menu items or menus to a PopupMenu.

```
Frame f = new Frame("PopupMenu");
Button b = new Button("Press Me");
PopupMenu p = new PopupMenu("Popup");
MenuItem s = new MenuItem("Save");
MenuItem l = new MenuItem("Load");
b.addActionListener(this);
f.add("Center", b);
p.add(s);
p.add(l);
f.add(p);
```



Note – The PopupMenu must be added to a parent component. This is not the same as adding ordinary components to containers. In this example, the PopupMenu has been added to the enclosing Frame.

To cause the PopupMenu to be displayed, you must call the show method. Showing requires a component reference to act as the origin for the x and y coordinates. Usually you would use the trigger component for this. In this case, that trigger is the button b.

```
public void actionPerformed(ActionEvent ev) {
p.show(b, 10, 10);
}
```

Note – The origin component must be beneath, or contained in, the parent component in the containment hierarchy.

Controlling Visual Aspects

You can control the appearance of AWT components in terms of colors used for the foreground and the background, and the font used for text.

Colors

Two methods that set the colors of a component are:

- `setForeground()`
- `setBackground()`

Both of these methods take an argument that is an instance of the `java.awt.Color` class. You can use constant colors that are 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.

In addition, you can create a specific color, such as:

```
int r = 255, g = 255, b = 0;
Color c = new Color(r, g, b);
```

Such a constructor creates a color based on the specified intensities of red, green, and blue, as a value in the range of 0 to 255 for each.

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);
```

Valid font names include:

- Dialog
- DialogInput
- Monospaced
- Serif
- SansSerif
- Symbol

Font style constants, which are actually `int` values, are:

- Font.BOLD
- Font.ITALIC
- Font.PLAIN
- Font.BOLD + Font.ITALIC

Point sizes should be specified using an `int` value.

Printing

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 is actually destined for the printer.

The printing system allows the use of local printer control conventions, so that the user will see a printer selection dialog box when a print operation is started. The user can then choose options, such as paper size, print quality, and printer to use.

```
Frame f = new Frame("Print test");
Toolkit t = f.getToolkit();
PrintJob job = t.getPrintJob(f, "MyPrintJob", null);
Graphics g = job.getGraphics();
```

If the user cancels a print request, `getPrintJob` returns null.

These lines create a `Graphics` that is connected to the printer of the user's choice. Obtain a new `Graphics` for each page to be printed. You can use any of the `Graphics` class drawing methods to write to the printer. Alternatively, as shown here, you can ask a component to draw itself onto the `Graphics`.

```
    f.printAll(g);
```

The `print` method asks a component to draw itself in this way, but it only relates to the component that it has been called for. In the case of a container, such as here, you can use the `printAll` method to cause the container and all the components that it contains to be drawn onto the printer.

```
g.dispose();
job.end();
```

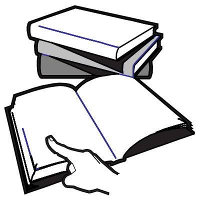
Once you have created a page of output, use the `dispose` method to submit that page to the printer.

When you have completed the job, call the `end` method on the print job object. This indicates that the print job is completed and allows the printer spooling system to run the job.

Object Serialization F

The Object Serialization API enables developers to write Java software code that creates persistent storage for objects.

Additional Resources



Parts of this appendix have been borrowed in whole or in part from: "Java Object Serialization Specification, Revision 1.2, December 2, 1996." Available at:

`http://www.javasoft.com/`

Introduction

Almost every application requires some way of storing data. Most applications use a database for the storage or persistence of data. However, databases are not typically used to store objects, particularly Java software objects. What is required is a way to keep the state of an object in such a way that the object can be stored easily and retrieved into its previous (stored) state.

These are the goals of the Java Object Serialization API—to provide a simple and transparent means of persisting objects.

Object serialization is fundamentally a single stream of data that represents the state of an object. The API provides methods to produce and consume the stream of data. Objects that need to act as containers to be serialized implement an interface that allows the contents of the object to be saved or restored in the single stream. These are the `java.io.ObjectOutput` and `java.io.ObjectInput` interfaces.

Object Serialization: The Old Way

To create a persistent storage mechanism without using the Object Serialization API:

- Create a flexible mechanism for naming the objects you wanted to store—some kind of code that identifies the kind of object to be stored.
- Create a standard method for reading the value of every field in the object and putting the fields onto the storage stream in a canonical fashion.
- Create a method to access the name and storage method for all of the objects referenced by the object that you are persisting.
- Create a mechanism for determining what field types are not candidates for persistence.

Object Serialization: The New Way

The `java.io.Serializable` interface was added after JDK1.0, as well as changes to the Java Virtual Machine to support the ability to save an object to a stream.

Saving an object to some type of permanent storage is called *persistence*. An object is said to be *persistent* when you can store that object on disk or tape, or send it to another machine to be stored in its 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 `Java.io.Serializable` cannot be serialized.

Serialization Architecture

`java.io` Package

The Serialization API is built upon two interfaces, `java.io.ObjectOutput` and `java.io.ObjectInput`. These interfaces are abstract stream-based interfaces designed to put or get objects onto an I/O stream. See Figure F-1.

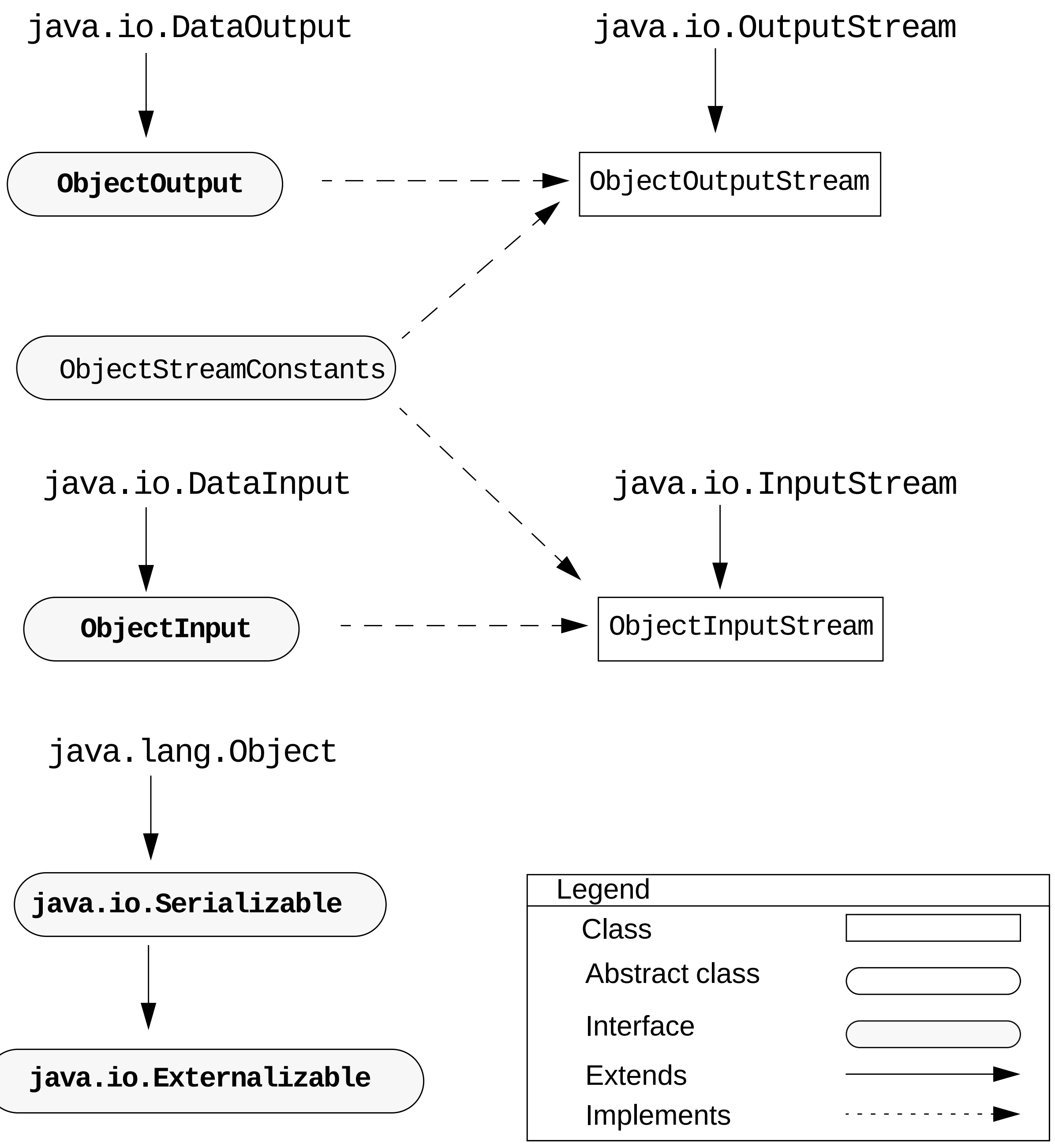


Figure F-1 Interfaces and Classes From the Serialization API

`ObjectOutput` *Interface*

The `ObjectOutput` interface extends `DataOutput` to write primitives. The important method in the interface is the `writeObject` method, which is used to write an object. Exceptions can occur while accessing the object or its fields, or when attempting to write to the storage stream.

```
package java.io;

public interface ObjectOutput extends DataOutput {

   public void writeObject(Object obj)
      throws IOException;

   public void write(byte b[]) throws IOException;

   public void write(byte b[], int off, int len)
      throws IOException;

   public void flush() throws IOException;

   public void close() throws IOException;
}
```

`ObjectInput` *Interface*

The `ObjectInput` interface is used for reading from the storage stream and returning an object. Exceptions are thrown when attempting to read the storage stream, or if the class name of the serialized object cannot be found.

```
package java.io;
public interface ObjectInput extends DataInput {

   public Object readObject()
      throws ClassNotFoundException, IOException;

   public int read() throws IOException;

   public int read(byte b[]) throws IOException;

   public int read(byte b[], int off, int len)
      throws IOException;
   public long skip(long n) throws IOException;

   public int available() throws IOException;

   public void close() throws IOException;
}
```

`Serializable` *Interface*

The `Serializable` interface is used to identify classes that can be serialized:

```
package java.io;

public interface Serializable {};
```

Any class can be serialized as long as the class meets the following criteria:

- The class (or a class in the class hierarchy of this class) must implement `java.io.Serializable`.
- Fields that should not be serialized must be marked with the `transient` keyword. These include classes, such as `java.io.FileInputStream`, `java.io.FileOutputStream`, and `java.lang.Threads`. If these fields are not marked `transient`, an attempt to call the `writeObject` method will throw a `NotSerializableException`.

What Gets Serialized

All of the fields (data) of a `Serializable` object are written to the storage stream. This includes primitive types, arrays, and references to other objects. Only the data (and class name) of the referenced objects is stored.

Static fields are not serialized.

The field accessor (`private`, `protected`, and `public`) has no effect on the field being serialized; you as the developer should mark your `private` fields as `private transient`.

Object Graphs

When an object is serialized, only the data of the object is preserved – class methods and constructors are not part of the serialized stream. When a data variable is an object, the data members of that object are serialized also. The tree, or structure of an object's data, including these subobjects, constitutes the object *graph*.

Some object classes are not serializable because the nature of the data they represent is constantly changing; for example, streams, such as `java.io.FileInputStream` and `java.io.FileOutputStream`; and `java.lang.Thread`. If a serializable object contains a reference to a non-serializable element, the entire serialization operation fails.

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 modifiers (`public, protected, private`) have no affect on the data field being serialized, and data is written to the stream in byte format, with Strings represented as UTF characters. Using 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. Consider the following code fragment that sends an instance of a `java.util.Date` object to a file:

```
1  Date d = new Date();
```

```
FileOutputStream f = new
FileOutputStream("date.ser");
ObjectOutputStream s = new ObjectOutputStream (f);
   s.writeObject (d);
   s.close ();
```

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.

```
Date d = null;
FileInputStream f = new FileInputStream
("date.ser");
ObjectInputStream s = new ObjectInputStream (f);
   d = (Date)s.readObject ();
System.out.println ("Date serialized at: "+ d);
```

Serialization Versus Externalization

Classes that implement the `Serializable` interface automatically have the capability to save and restore the state of the object. Classes can also implement the `Externalizable` interface, in which case the responsibility of the storage and retrieval of the object's state become the responsibility of the object itself.

```
package java.io;

public interface Externalizable extends Serializable {

   public void writeExternal (ObjectOutput out)
      throws IOException;

   public void readExternal (ObjectInput in)
      throws IOException, ClassNotFoundException;
}
```

Externalizable objects must:

- Implement the `java.io.Externalizable` interface

- Implement a `writeExternal` method to save the state of the object; the method must explicitly coordinate with the supertype to save its state

- Implement a `readExternal` method to read the data on the stream and restore the state of the object; the method must explicitly coordinate with the supertype to save its state

- Be responsible for the externally defined format; the `writeExternal` and `readExternal` methods are solely responsible for this format

Externalizable classes mean that the class is serializable, but you must provide the methods for reading and writing objects; none are provided by default.

Please
Recycle

Adobe PostScript