Distributed Programming With Java Technology

SL-301

Student Guide With Instructor Notes

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

Revision C.1, March 2000

Contents

About This Course

Course Goal

Distributed Programming With Java Technology provides you with the knowledge and skills necessary to program distributed computing applications using the distributed technologies from JavaSoft™:

- Java™ Database Connectivity (JDBC™) application programming interface (API)
- Remote Method Invocation (RMI) API
- JavaIDL (Java interface definition language) API
- Java technology servlets (Java servlets)

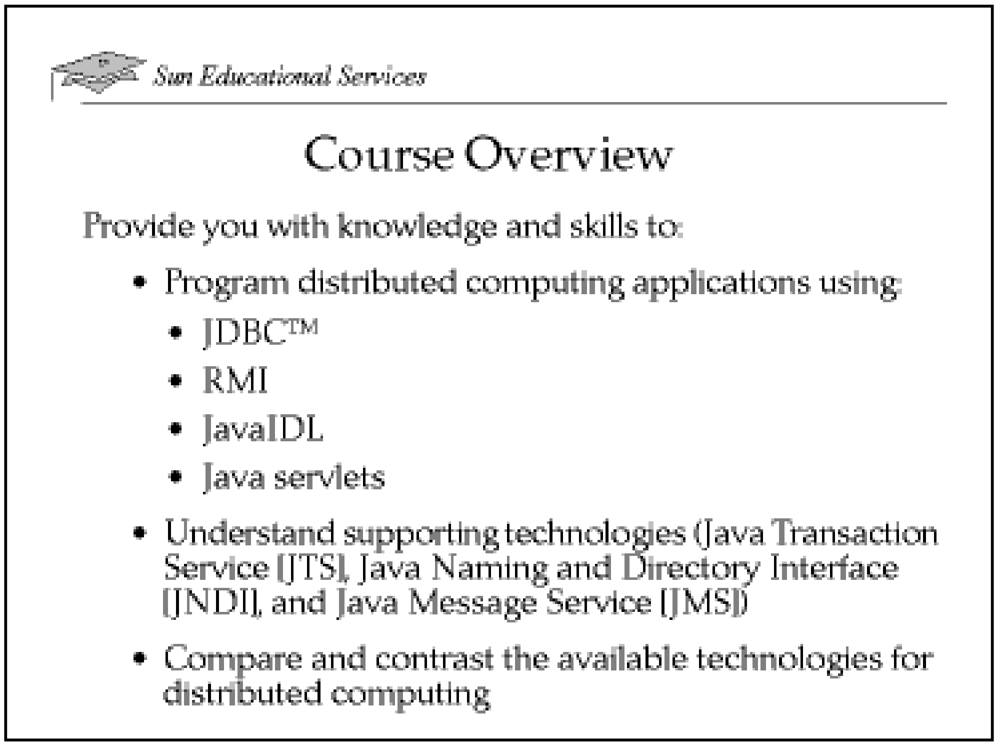
You will also learn about the supporting technologies, such as Java Transaction Services (JTS), Java Naming and Directory Interface (JNDI), and Java Message Service (JMS). At the end of the course, the knowledge gained should enable you to make informed decisions about which technology is best used under which circumstances.

✓ ***Use this module to get the students excited about this course.***

✓ ***With regard to the overheads: To avoid confusion among the students, it is very important to tell them that the page numbers on the overheads have no relation to the page numbers in their course materials. They should use the title of each overhead as a reference.***

✓ ***The strategy provided by the "About This Course" is to introduce students to the course before they introduce themselves to you and one another. By familiarizing them with the content of the course first, their introductions will have more meaning in relation to the course prerequisites and objectives.***

✓ ***Use this introduction to the course to determine how well students are equipped with the prerequisite knowledge and skills. The pacing chart on xx enables you to determine what adjustments you need to make in order to accommodate the learning needs of students.***



Sun Educational Services

Course Overview

Provide you with knowledge and skills to:

- Program distributed computing applications using:
  - JDBC™
  - RMI
  - JavaIDL
  - Java servlets
- Understand supporting technologies (Java Transaction Service [JTS], Java Naming and Directory Interface [JNDI], and Java Message Service [JMS])
- Compare and contrast the available technologies for distributed computing

Course Overview

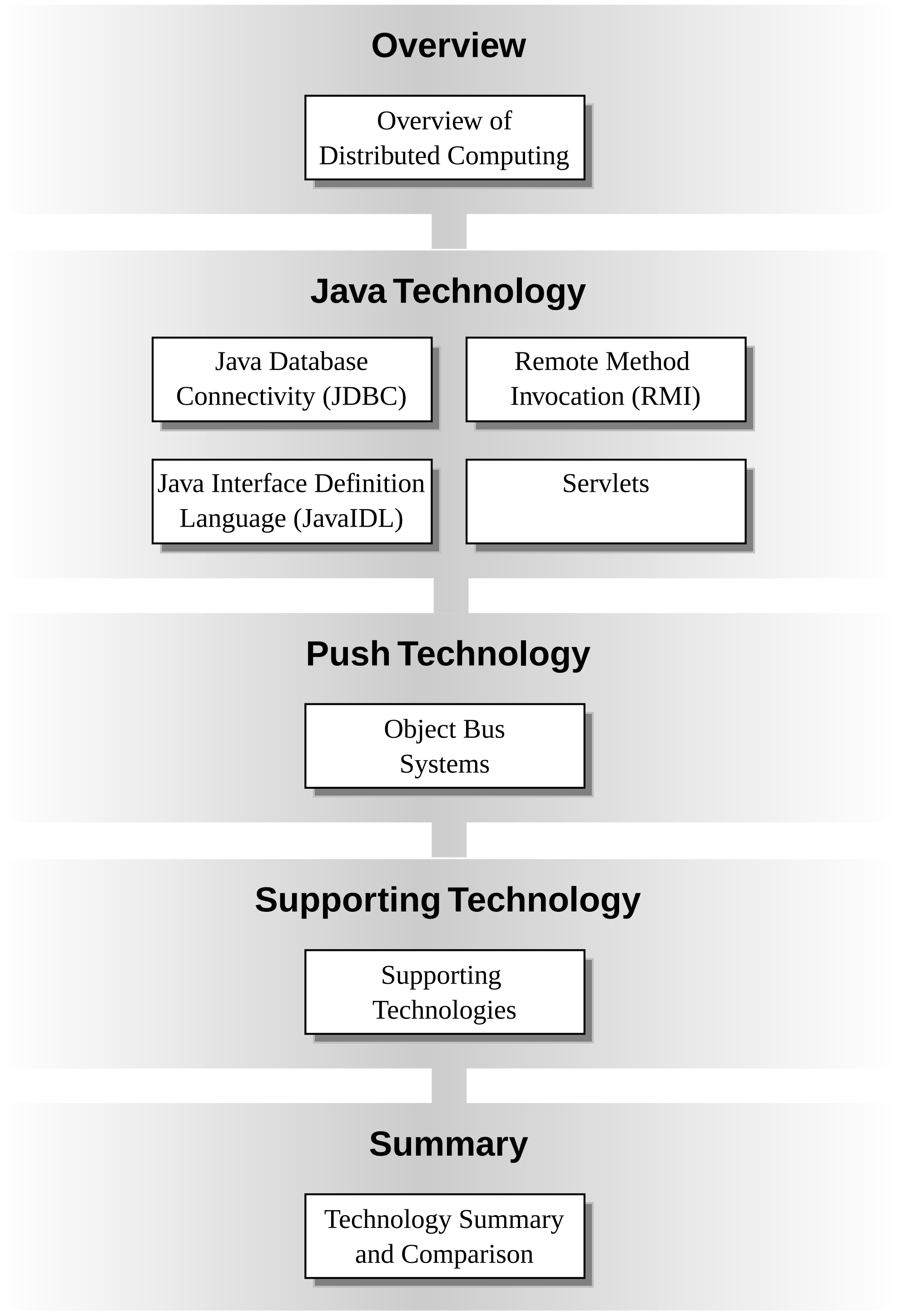
This course is concerned with providing you with the skills and concepts necessary to solve distributed computing problems using the four distributed computing technologies from JavaSoft:

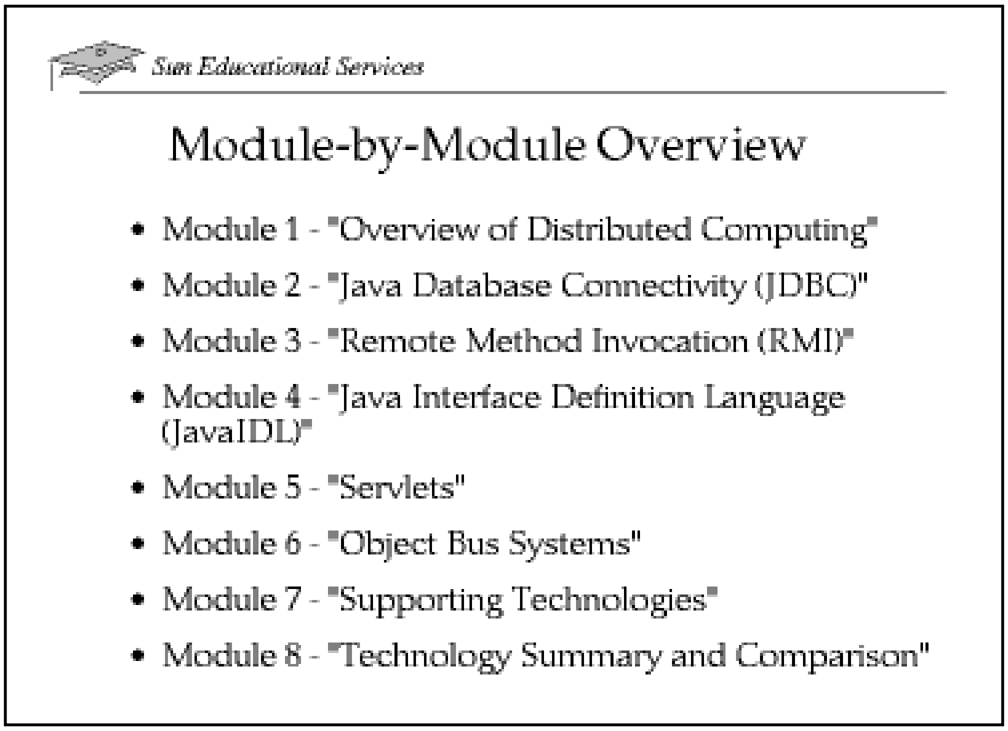
- JDBC
- RMI
- JavaIDL
- Java servlets

Issues related to distributed programming, such as a naming or transaction service, persistent objects, or security issues with remotely loaded code, are also addressed in this course.

Course Map

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


Overview
Overview of Distributed Computing
Java Technology
Java Database Connectivity (JDBC)
Remote Method Invocation (RMI)
Java Interface Definition Language (JavaIDL)
Servlets
Push Technology
Object Bus Systems
Supporting Technology
Supporting Technologies
Summary
Technology Summary and Comparison


Sun Educational Services

Module-by-Module Overview

- Module 1 - "Overview of Distributed Computing"
- Module 2 - "Java Database Connectivity (JDBC)"
- Module 3 - "Remote Method Invocation (RMI)"
- Module 4 - "Java Interface Definition Language (JavaIDL)"
- Module 5 - "Servlets"
- Module 6 - "Object Bus Systems"
- Module 7 - "Supporting Technologies"
- Module 8 - "Technology Summary and Comparison"

Module-by-Module Overview

This course contains the following modules:

- Module 1 – "Overview of Distributed Computing"

  This module describes the characteristics of distributed computing environments and of distributed object computing. Various techniques used to solve distributed computing problems are presented, with an overview of the JavaSoft technologies: JDBC, RMI, JavaIDL and Java servlets.

- Module 2 – "Java Database Connectivity (JDBC)"

  This module describes the main features of JDBC, the Java technology classes (Java classes) provided by JDBC, and how to use the JDBC to interface with a database system.

- Module 3 – “Remote Method Invocation (RMI)”

  This module describes the main features of RMI, the classes provided by RMI, and how to use RMI to solve distributed computing problems. The new features of RMI that are delivered with the Java 2 SDK, standard edition, Version 1.2 technology are also presented.

- Module 4 – “Java Interface Definition Language (JavaIDL)”

  This module describes the Common Object Request Broker Architecture (CORBA), the main features of JavaIDL, how to use JavaIDL to solve distributed computing problems, and information about the IDL-to-Java programming language mapping.

- Module 5 – “Servlets”

  This module describes generic Java servlets and hypertext transfer protocol (HTTP) servlets.

- Module 6 – “Object Bus Systems”

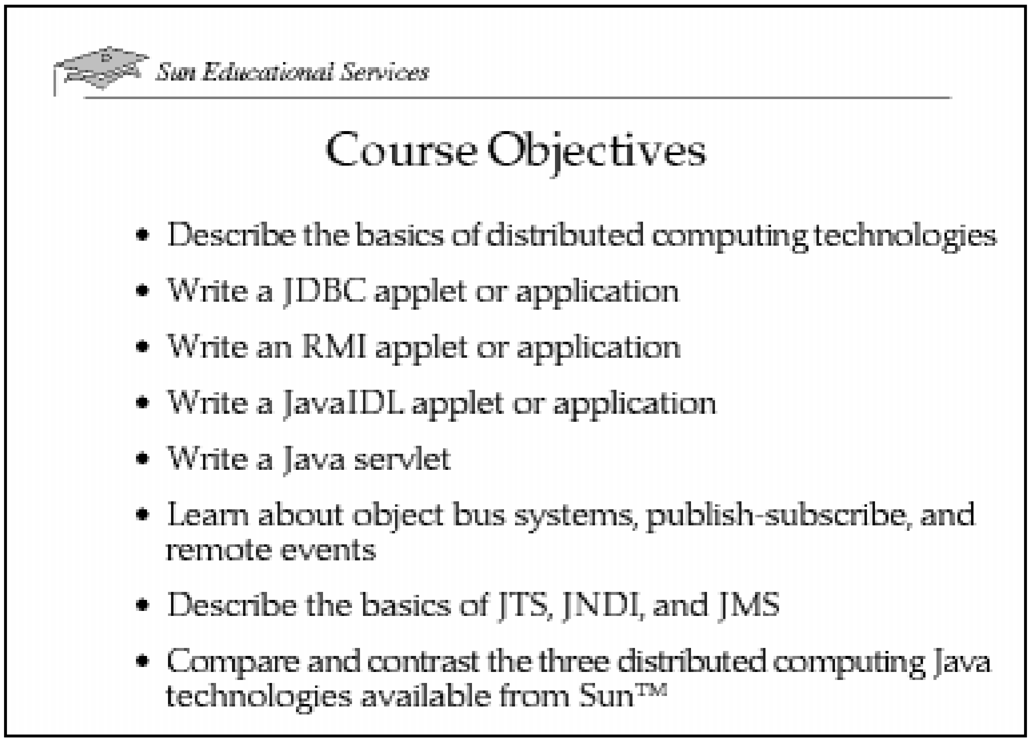
  This module describes object bus systems. Using object bus systems, you can build systems using a multicast, many-to-many object communication paradigm.

- Module 7 – “Supporting Technologies”

  This module describes three supporting technologies for doing distributed computing.

- Module 8 – “Technology Summary and Comparison”

  This module summarizes the different technologies explained in this course, and compares them to each other.


Sun Educational Services

Course Objectives

- Describe the basics of distributed computing technologies
- Write a JDBC applet or application
- Write an RMI applet or application
- Write a JavaIDL applet or application
- Write a Java servlet
- Learn about object bus systems, publish-subscribe, and remote events
- Describe the basics of JTS, JNDI, and JMS
- Compare and contrast the three distributed computing Java technologies available from Sun™

Course Objectives

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

- Describe the basics of distributed computing technologies
- Write a JDBC applet or application
- Write an RMI applet or application
- Write a JavaIDL applet or application
- Write a Java technology-based servlet
- Explain how object bus systems, publish-subscribe systems, and remote events work
- Explain the basics of JTS, JNDI, and JMS
- Compare and contrast the three distributed computing Java technologies available from Sun Microsystems™

Skills Gained by Module

The skills for *Distributed Programming With Java Technology* are shown in the first column of the matrix below. The black boxes indicate the main coverage for a topic; the gray boxes indicate that the topic is briefly discussed.

Skills Learned	Module							
	1	2	3	4	5	6	7	8
Describe the basics of distributed computing technologies	■	▨	▨	▨	▨		▨	▨
Write a JDBC applet or application		■		▨				
Write an RMI applet or application			■					
Write a JavaIDL applet or application				■				
Write a Java servlet					■			
Explain how object bus systems, publish-subscribe, and remote events work						■		
Explain the basics of JTS, JNDI, and JMS							■	
Compare and contrast the three distributed computing Java technologies available from Sun Microsystems								■

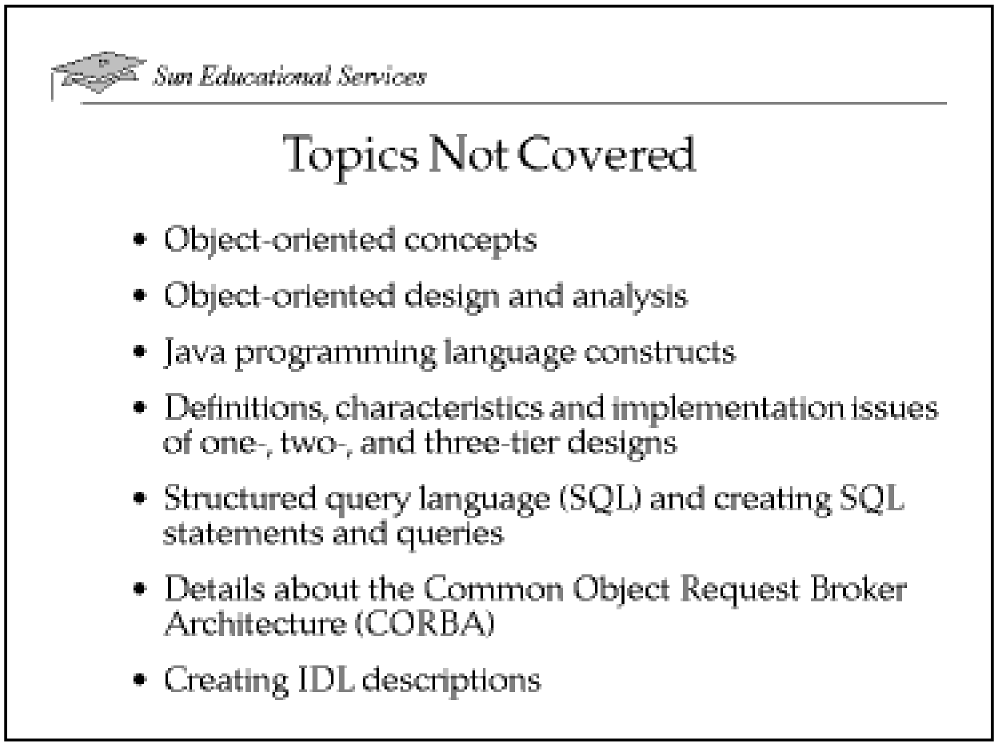
✓ ***Refer students back to this matrix as you progress through the course to show them the progress they are making in learning the skills advertised for this course.***

Guidelines for Module Pacing

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

Module	**Day 1**	**Day 2**	**Day 3**	**Day 4**	**Day 5**
“About This Course”	A.M.				
Module 1 – “Overview of Distributed Computing”	A.M.				
Module 2 – “Java Database Connectivity (JDBC)”	P.M.	A.M.			
Module 3 – “Remote Method Invocation (RMI)”		P.M.	A.M.		
Module 4 – “Java Interface Definition Language (JavaIDL)”			P.M.	A.M.	
Module 5 – “Servlets”				P.M.	
Module 6 – “Object Bus Systems”					A.M.
Module 7 – “Supporting Technologies”					A.M.
Module 8 – “Technology Summary and Comparison”					P.M.

✓ ***This table should only be used as a guideline for timing throughout the week. The time you spend on each module and lab may differ from class to class, as knowledge and Java programming language experience of the students varies.***

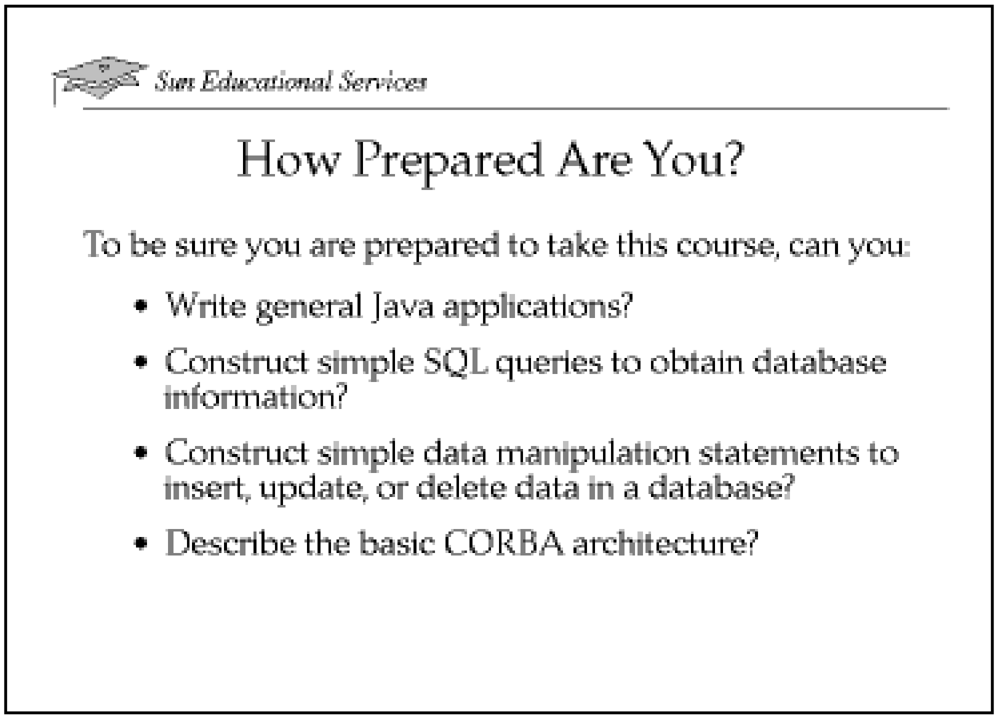

Sun Educational Services

Topics Not Covered

- Object-oriented concepts
- Object-oriented design and analysis
- Java programming language constructs
- Definitions, characteristics and implementation issues of one-, two-, and three-tier designs
- Structured query language (SQL) and creating SQL statements and queries
- Details about the Common Object Request Broker Architecture (CORBA)
- Creating IDL descriptions

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


Sun Educational Services

How Prepared Are You?

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

- Write general Java applications?
- Construct simple SQL queries to obtain database information?
- Construct simple data manipulation statements to insert, update, or delete data in a database?
- Describe the basic CORBA architecture?

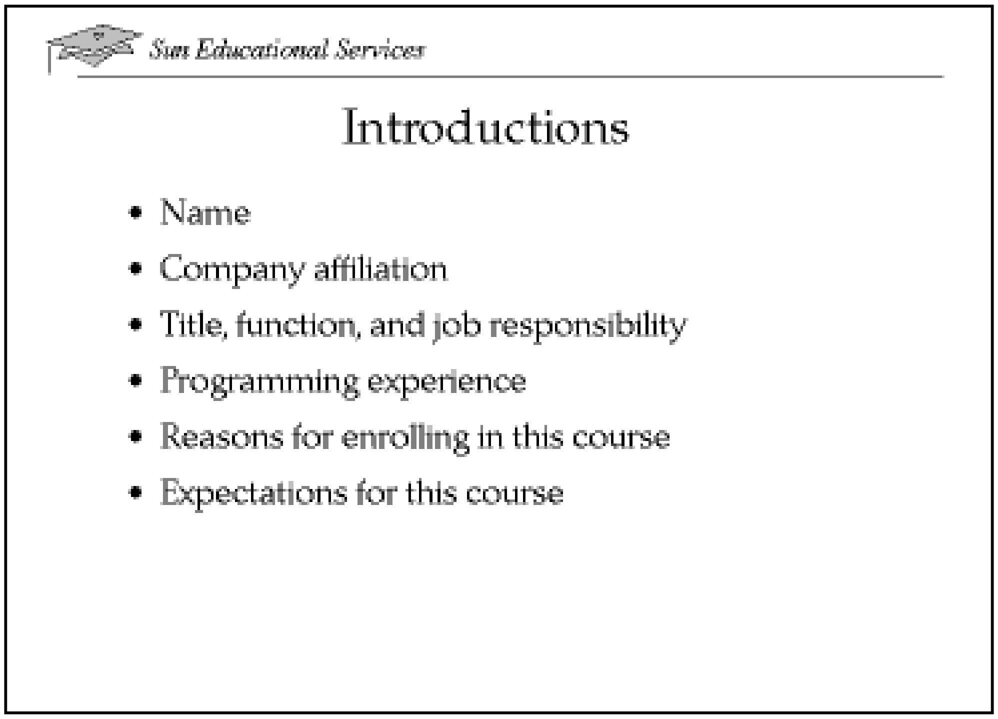
How Prepared Are You?

To be sure you are prepared to take this course, you should be able to do the following:

- Write general Java technology-based applications
- Construct simple structured query language (SQL) queries to obtain database information
- Construct simple data manipulation statements to insert, update, or delete data in a database
- Describe the basic CORBA architecture

✓ ***If any students indicate they cannot do the above, meet with them at the first break to decide how to proceed with the class. Do they want to take the class at a later date? Is there some way to get extra help?***

✓ ***It might be appropriate here to recommend resources from the Sun Educational Services catalog that provide training for topics not covered in this course.***

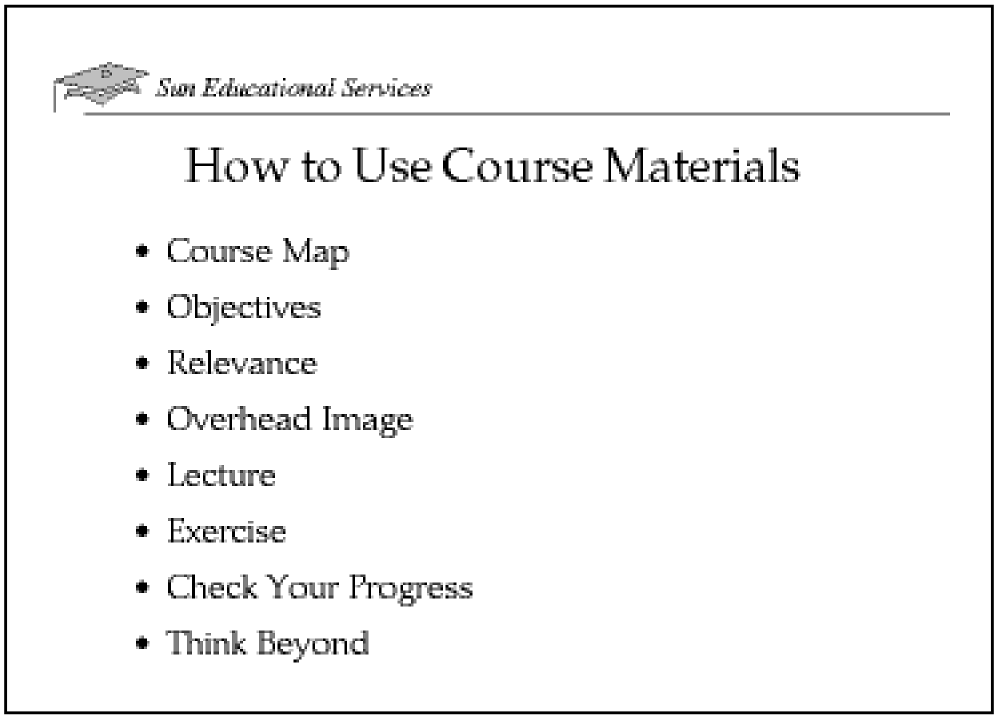

Sun Educational Services

Introductions

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

Introductions

Now that you have been introduced to the course, introduce yourself to each other and 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 provoke you to think about how the module content relates to your interest in learning Java technology programming (Java programming).
- **Overhead image** – Reduced overhead images for the course are included in the course materials to help you easily follow where the instructor is at any point in time. Overheads do not appear on every page.

- **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.

Icons

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

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

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

Caution – A potential hazard to data or machinery.

Warning – Anything that poses personal danger or irreversible damage to data or the operating system.

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
```

Notes to the Instructor

Philosophy

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

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

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

To allow the instructor flexibility and give time for meaningful discussions during the "relevance" periods, the lectures, and the small-group discussions, a timing table is included in "General Timing Recommendations."

Course Tools

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

- Relevance

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

- Course map

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

- Lecture overheads

  Overheads for the course are provided in two formats:

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

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

- Small-group discussion

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

- General timing recommendations

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

Module	Lecture (Minutes)	Lab (Minutes)	Total Time (Minutes)
Preface	60		60
Module 1	120		120
Module 2	180	180	360
Module 3	180	180	360
Module 4	180	180	360
Module 5	120	120	240
Module 6	60	60	120
Module 7	60	60	120
Module 8	120		120
Debriefing	120		120

- Module self-check

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

Instructor Setup Notes

Purpose of This Guide

This guide provides general information about setting up the classroom. Refer to the `README_setup_Instructions` file in the `SL301_IN` directory for specific information about how to set up this course.

Projection System and Workstation

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

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

  To install the HTML overheads on the machine connected to your overhead projection system, copy the `HTML` and `images` subdirectories provided in the SL301_`OH` directory to any directory on the overhead workstation machine.

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

  ```
  /SL301_OH/HTML/OH.Title.doc.html
  ```

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

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

Course Files

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

Course Components

This course consists of the following components:

- Instructor guide

  The `SL301_IG` directory contains the FrameMaker files for the instructor's guide (student's guide with instructor notes). The ART directory is required for printing this guide.

- Student guide

  The `SL301_SG` directory contains the FrameMaker files for the student's guide. The ART directory is required for printing this guide.

- Art

  The `SL301_ART` directory contains the supporting images and artwork for the student's and instructor's guides. This directory is *required* for the printing of the student's and instructor's guides and should be located in the same directory as `SL301_IG` and `SL301_SG`.

- Instructor notes

  The `SL301_IN` directory contains the text file `README_setup_Instructions`.

- Overheads

  The SL301_OH directory contains the instructor overheads. There are both HTML and FrameMaker versions of the overheads.

- Lab Files

  The SL301_SOL_LF and SL301_WIN_LF directories contain the lab files for this course.

Overview of Distributed Computing 1

Objectives

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

- List the five supporting technologies for distributed computing
- Compare and contrast the different architectures for distributed computing
- List the distributed computing technologies available for the Java programming language

Growing popularity and use of intranets and the Internet have increased the demand for distributed computing applications. This module provides an overview of distributed computing technologies, including the four distributed programming technologies available from the JavaSoft product.

Relevance

✓ *Present the following questions to stimulate the students and get them thinking about the issues and topics presented in this module. They are not expected to know the answers to these questions. The answers to these questions should be of interest to the students, and inspire them to learn the content presented in this module.*

Discussion – Suppose you have an intranet with a wide range of hardware and operating environments. You plan to implement a distributed computing strategy for your company. Consider the following questions:

- What is your understanding of distributed computing?

✓ *This is a very interesting question – pay close attention to the responses, it will help to define the course for the rest of the week. Some people believe that "distributed" is synonymous with "parallel" and may define distributed as a single task split over multiple CPUs. The definition used in this course is different: distributed means having program resources on multiple address spaces.*

- What advantages does distribution offer?

✓ *Obvious advantages are better control over the use/development of system resources, and the ability to share the load of a system over multiple address spaces, but be careful—keep in mind the previous discussion.*

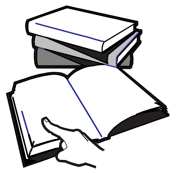
- What existing technologies use distributed computing?

✓ *NFS, client-server database systems, SunNet Manager, and so forth.*

- What distributed computing technologies do you know?

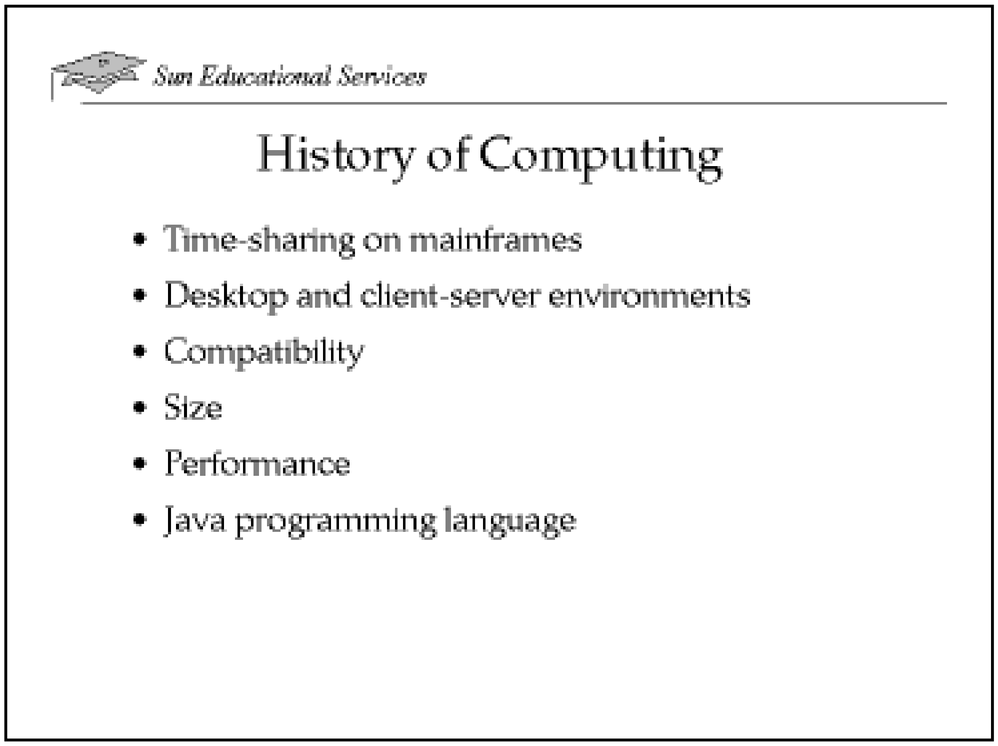
✓ *Daemons, RPC, Distributed Object, Object Bus, JDBC, Servlets, RMI, and so forth.*

Additional Resources



Additional resources – The following resources can provide additional detail on the topics presented in this module:

- Orfali, Robert, Bruce Harkey, and Bob Edwards. 1996. *The Essential Distributed Objects Survival Guide.* Wiley Press.
- Farley, Jim. 1997. *Java Distributed Computing*. O'Reilly & Associates.
- *A Note on Distributed Computing.* [Online]. Sun Microsystems, Inc. Available: `http://www.sunlabs.com/technical-reports/1994/abstract-29.html`



Sun Educational Services

History of Computing

- Time-sharing on mainframes
- Desktop and client-server environments
- Compatibility
- Size
- Performance
- Java programming language

History of Computing

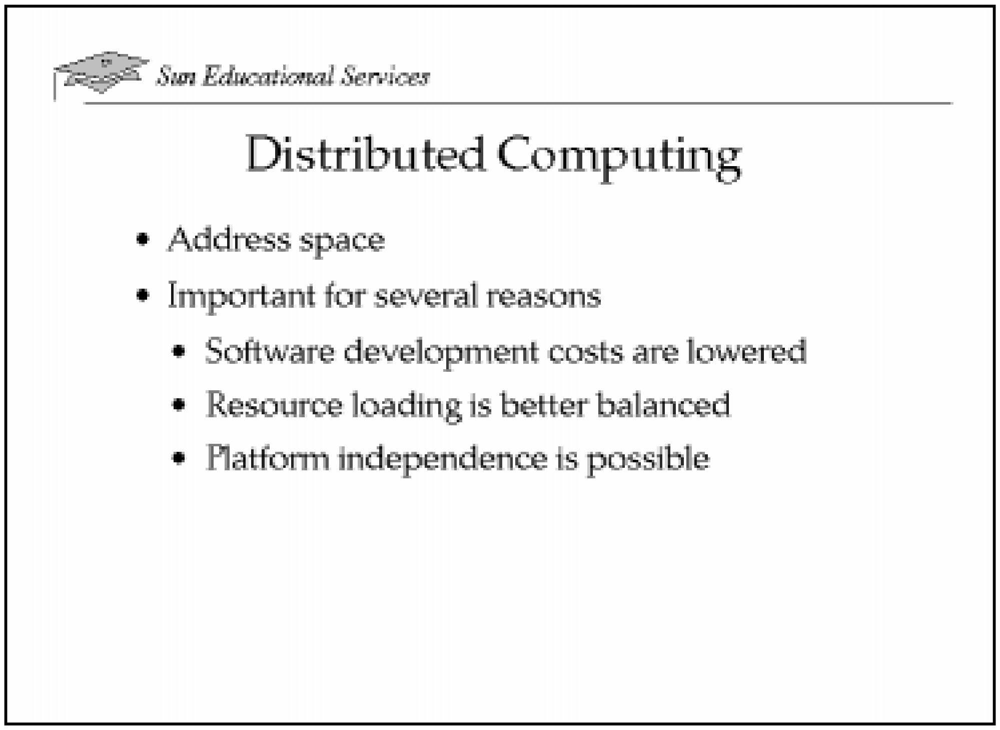
The history of computing began with the development of programs that ran within a single computer; users time-shared the central processing unit (CPU) resource of a mainframe system and accessed their "address space" from a terminal.

As computers became more powerful, smaller, and capable of executing a greater number of programs, the computer moved from the central location to the desktop. People soon realized there was an advantage to linking the distributed computers to a network and sharing common resources (applications) using servers on the network. With this approach, the client downloads or accesses an application from a common server and runs the application in the client's address space. Overall, this greatly improves the performance of the application and the client machine is the determining factor in the speed of the application.

However, the client-server environment uncovered issues with the sharing of common resources—compatibility and size. In a true homogeneous computing environment, where every client machine is running the latest operating system, and every machine is of the same architecture, a single program can be shared from the server. In real life, there are numerous versions of the operating system (OS), the hardware architectures are different (even from the same vendor), and the likelihood is small that a single version of a program can run across all client machines.

Additionally, applications have grown in size and processing data now moves down to the client, causing the client programs to be large and therefore slow to load and start up.

The Java programming language makes it possible to create applications that run on any platform that supports the Java virtual machine (JVM), which greatly increases the likelihood that the application downloaded from the server will run. However, it does not solve the problem of size; the client program is still large if processing occurs within the client's address space.



Sun Educational Services

Distributed Computing

- Address space
- Important for several reasons
  - Software development costs are lowered
  - Resource loading is better balanced
  - Platform independence is possible

Distributed Computing

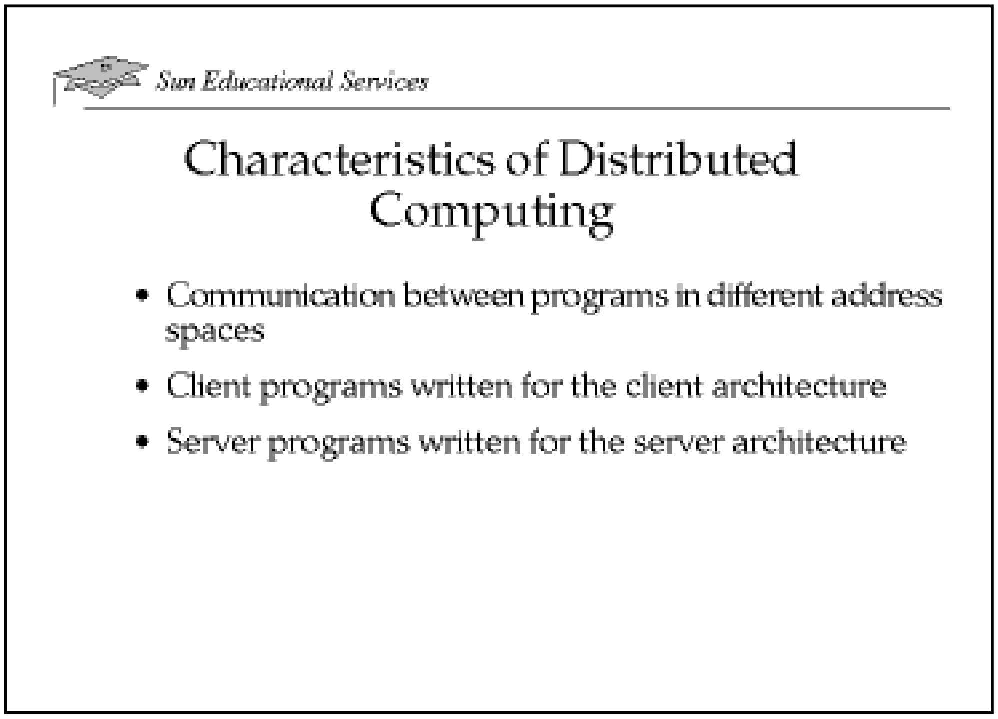
To solve the problem of program size, you need a way of moving the major processing pieces to another address space. This would allow the client to concentrate on the specific pieces required for the end-user of the application.

The solution is, ironically, somewhere between the single mainframe and the client-server model. The answer is to allow applications that are run on the client to access applications that are running on another machine—a server or another client. This technique is called *distributed computing.*

Importance of Distributed Computing

Distributed computing is important for several reasons:

- Software development costs are lowered – Many clients can access the same code; the client-side application development process is reduced because duplication of effort is reduced.
- Resource loading is better balanced – The whole client-server paradigm is based on the ability of many machines or processes to request the resources or services provided by a single or a few machines or processes, so that CPU, memory, and disk resources are more effectively allocated.
- Platform independence is possible – The client and server communicate through a known protocol that does not rely on hardware.

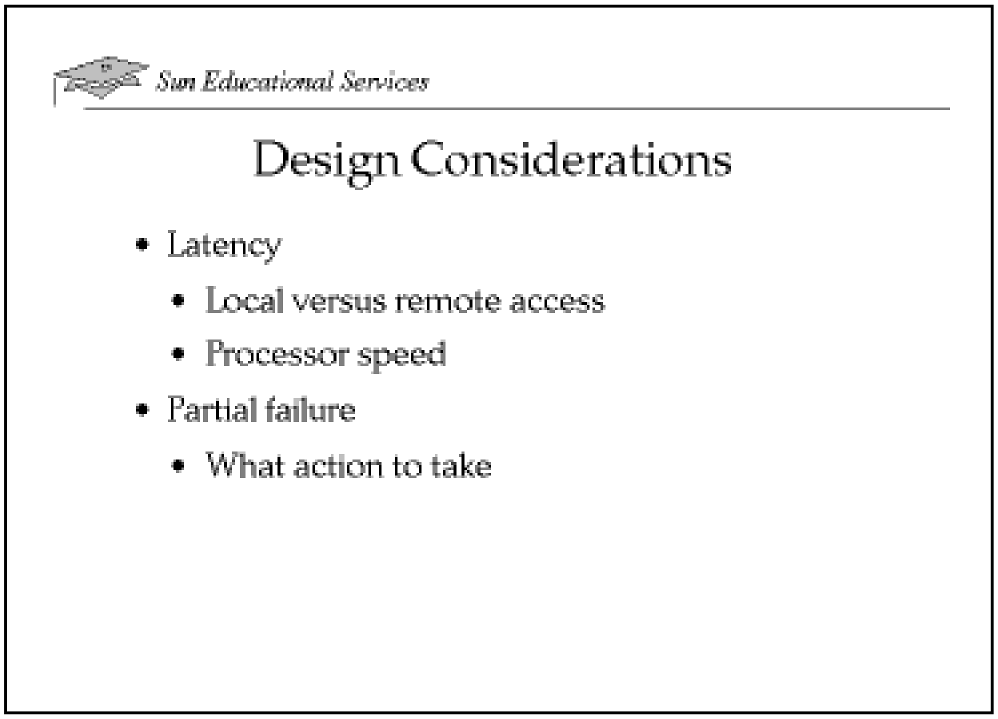

Sun Educational Services

Characteristics of Distributed Computing

- Communication between programs in different address spaces
- Client programs written for the client architecture
- Server programs written for the server architecture

Characteristics of Distributed Computing

In a distributed environment, programs that the client runs make calls to programs in other address spaces, either locally or remotely. This approach allows the client program to be written specifically for the client architecture and still allows the client program to make calls (through a known protocol) to server programs that potentially are running in a different architecture.



Sun Educational Services

Design Considerations

- Latency
  - Local versus remote access
  - Processor speed
- Partial failure
  - What action to take

Design Considerations

Latency

Proper analysis of what should reside locally and what can be accessed remotely is crucial to the performance of the distributed application. Obviously, a service that can be accessed over a local bus will be faster than one that must use the network. As the number and speed of processors that can be put in a system increases, the speed gap between local and remote access increases.

✓ ***"Note on Distributed Computing" states that latency can be as high as four to five orders of magnitude between local and remote access.***

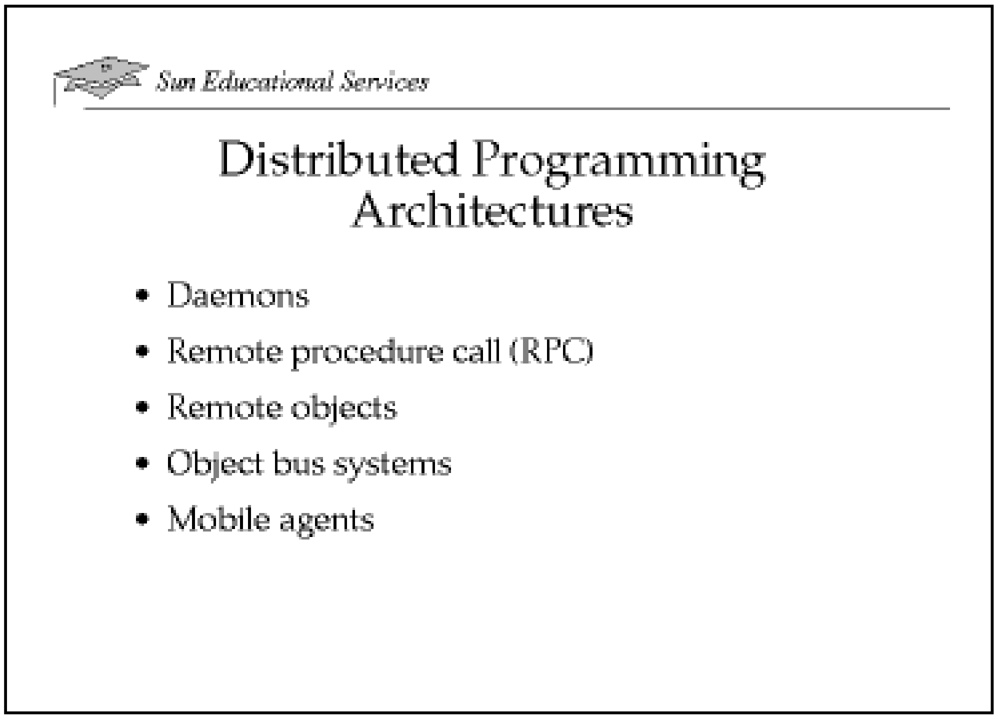
Partial Failure

In a local application, if a single component fails (a disk crashes, a CPU panics, or memory fills up), the application will fail to run. When writing a local application you do not have to take catastrophic failure into consideration, because there is little if anything that you can do to recover.

✓ ***Partial failure dictates that the programs deal with indeterminacy.***

However, in the distributed programming model, one component can fail (the client machine, the server machine, or the network), so partial failure detection and recovery procedures become important. For example, if a client application can no longer reach its server process, should it cease its execution? Should it retry the request? Should it wait a period of time before a retry? Should it attempt to request the service from somewhere else, and how many times should it retry before giving up? From the client side, failure of a processor on the server might not be distinguishable from a network failure.

✓ ***The following pages on the NFS™ environment are designed to get the students thinking about failure and recovery scenarios for distributed applications. Point out that the systems you will be learning about this week have many parts or components that can fail; that is, the database host, the database itself, the RMI registry, the Java technology runtime on the remote servers, the ORB, and so on.***


Sun Educational Services

Distributed Programming Architectures

- Daemons
- Remote procedure call (RPC)
- Remote objects
- Object bus systems
- Mobile agents

Distributed Programming Architectures

Over time, several approaches to distributed programming have been developed. While some of the older technologies might become obsolete (especially Remote Procedure Call), others still have a valid place today. The choice of a particular technology depends on what you want to achieve. At the end of this course you should not only know the different technologies available, but also be able to choose the most appropriate technology for the specific application that you want to build. The following approaches are examined in this module:

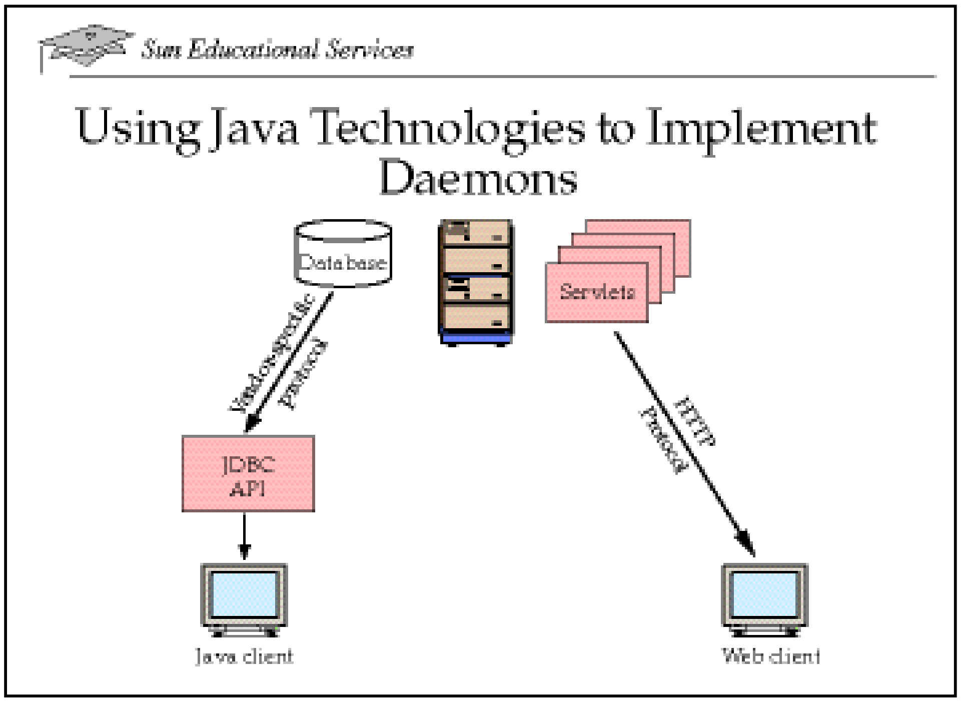
- Daemons
- Remote Procedure Call (RPC)
- Remote objects
- Object bus systems
- Mobile agents

Overview of Daemon Processes

A *daemon* is a process running on a machine that waits for service requests, and then services these requests. A local daemon could be a Java programming language thread (Java thread), which is normally suspended. The service request activates the thread. In a distributed environment, a daemon usually listens on an IP port for service requests, and processes these requests when they come in. Well-known examples of these types of daemons are:

- `ftpd`
- `telnetd`
- `httpd`
- `sendmail`

The communication between the client application and the daemon is usually performed using a daemon-specific protocol, such as Simple Message Transfer Protocol (SMTP) in the case of the `sendmail` daemon.



Sun Educational Services

Using Java Technologies to Implement Daemons

Using Java Technologies to Implement Daemons

You can develop daemon-like systems using nothing more than Java technology sockets (Java sockets). In fact, this is what you do when you develop a Java technology-based application that must interface with a legacy system. These legacy systems are usually daemons conceptually with their own proprietary communication protocol.

However, this course covers higher-level APIs, encapsulating the details of common, standardized types of daemons: database and hypertext transfer protocol (HTTP).

Do not be confused by the fact that these two daemons are not on the same side of a client-server relationship. JDBC allows you to build systems of Java technology clients (Java clients) that connect to potentially non-Java programming language servers. Servlets allow you to build Java technology elements to run on a Web server, interfacing with potentially non-Java programming language web clients.

Java Database Connectivity (JDBC) Application

The JDBC application encapsulates the specialities of different database server products. The API provides a set of generic interface classes that are implemented by a database-specific driver.

This means that the Java 2 SDK alone usually is not sufficient to test a JDBC application. To connect to a specific database, you need a JDBC driver from a vendor. However, there is one driver included with the Java 2 SDK: the JDBC-ODBC bridge driver. When you have a working open database connectivity (ODBC) driver on a client, you can make a JDBC connection to the database with only the Java 2 SDK.

Servlets

The Java Servlet API enables you to create Java technology-based servlets. This API is a standard extension to the Java 2 SDK and is delivered together with the Java 2 SDK. This means that you can develop servlets using only the Java 2 SDK. For testing purposes, a simple *servlet runner* is available with the Java 2 SDK. However, the servlet runner only supports HTTP servlets (see the next section) in single-thread mode.

The most common servlets today are HTTP servlets, which are modular extensions to a Web server. Common usages of HTTP servlets are:

- Dynamic hypertext markup language (HTML) page compilation; that is, getting the contents from other sources (for example, using the JDBC application from a database)

- Dynamic compilation of a table of contents or an index of an entire web site

- HTML forms processing, ranging from simple “register me please” to complex search engines processing query strings

- Dynamic compilation of embedded small parts of HTML pages, such as visitor counters, clocks, or mini calendars

Third-Party Java Technologies Using Daemons

You will find numerous JavaBeans™ components encapsulating Transmission Control Protocol/Internet Protocol (TCP/IP) protocols, such as File Transfer Protocol (FTP), Simple Mail Transfer Protocol (SMTP), Post Office Protocol3 (POP3), Network News Transfer Protocol (NNTP) and so on. These components save you the effort of coding the protocol yourself, using Java technology-based sockets. Also available are APIs or JavaBeans components for non-TCP/IP legacy protocols. You can even find complete VT100 or 3270 emulators or emulator Beans on the market.

Understanding Daemons

Discussion – Take a couple of minutes to read through the following questions, and prepare some answers on your own. Then discuss the questions in the class.

- List products or technologies you know that are implemented using daemons.

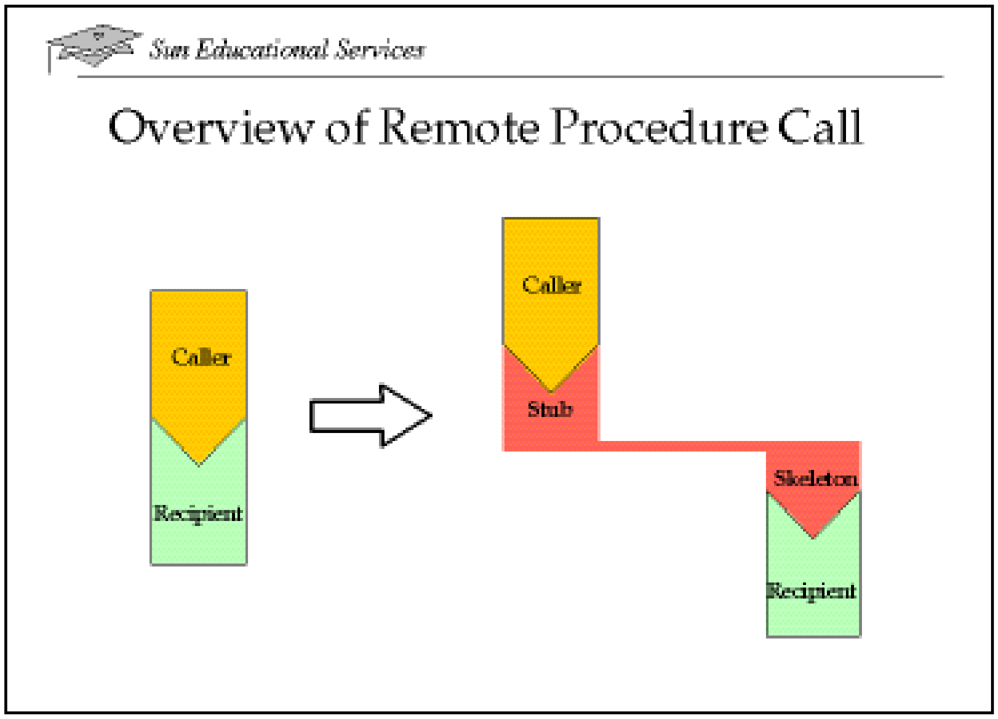
✓ ***Most of the TCP/IP standards, like*** `ftp`, `telnet`, `yp`***; but also a lot of "server"-products, like DB2, Oracle even ORBs use daemons for initial setup.***

- If you were to develop a distributed system using daemons, what would you do?

✓ ***Decide whether to use an existing protocol or develop a new one; implement a new daemon or use an existing one; develop an API set for client usage of the daemon; implement the client using the API set.***

- If you have an existing local application and want to distribute it now, what are the necessary steps to implement it with daemons?

✓ ***Decide which parts of your application are server, and which are client. Develop an API for the client so it can use the server. Rewrite the server so that it can function as a daemon. Rewrite the client so that it uses the API to access the server.***



Overview of Remote Procedure Call

With Remote Procedure Call (RPC), the client calls a local procedure as if the procedure were in the client's address space, but the procedure is actually represented by a proxy and the call is passed to a server.

If you are using a procedural language (such as C or Pascal), RPC allows you to split up your application onto several distributed systems without redesigning it too much. Conceptually, local and remote procedure calls look the same in program code.

It seems likely that RPC will become obsolete because procedural languages will become obsolete. It is not a good idea to do RPC with object-oriented programming languages like Java or C++, unless you have to communicate with a legacy system that happens to offer its services using RPC.

Third-Party Java Technologies Using RPC

The two major legacy RPC systems available today are Sun RPC and Open Software Foundation / Data Communication Equipment (OSF / DCE). At the time this course was developed, no Java technology implementations were available for any of them.

Understanding Remote Procedure Call (RPC)

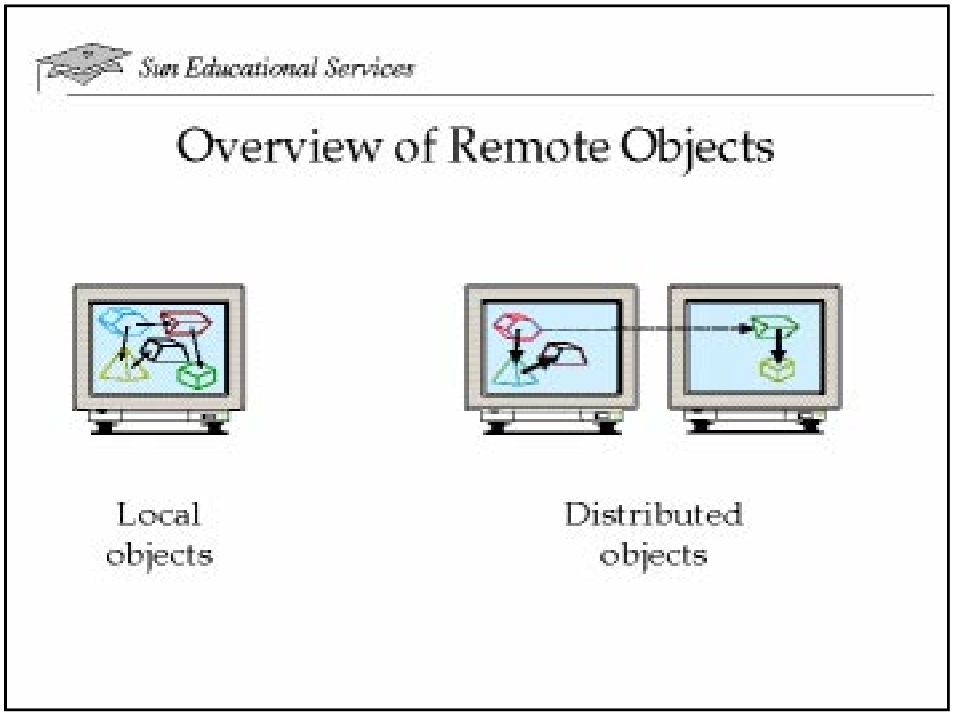
Discussion – Use a couple of minutes to read through the following questions, and prepare some answers on your own. Then discuss the questions in the class.

- List products or technologies you know that are implemented using RPC.

✓ ***There are few servers available that are accessed using RPC. RPC is mostly used internally in distributed applications, which are bought or developed as a whole. One example is Sun's NFS environment. A well-known toolkit to support the development of RPC applications is Open Software Foundation's Distributed Computing Environment (OSF/DCE).***

- Suppose you have an existing local application written in the Java programming language that you now want to distribute using RPC. What are the problems you will encounter?

✓ ***You have to leave the "thinking model" of your application. Creating instances of objects and calling methods on them is no longer possible, if they are not on the same machine. Therefore, the design of the application must be changed at a very fundamental level. As a guideline, everything remote must be accessible by static methods on one single remote object.***


Sun Educational Services

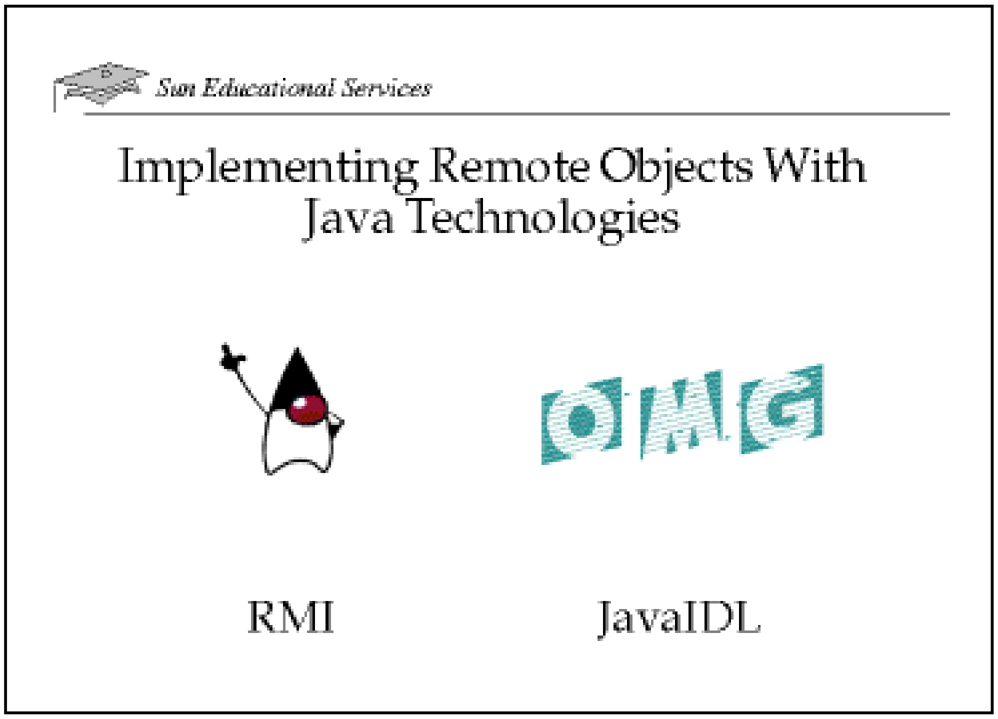
Overview of Remote Objects

Overview of Remote Objects

With remote objects, the client calls methods of a local object. The objects are represented by proxies, which pass the call request to the object implementations on other systems.

Therefore, in the remote objects model, programmers deal strictly with objects: calls to methods are always made through a representative object. Ideally, programmers treat every object as if it were local. The actual address space and location of the object are irrelevant. The programmer deals with the object calls through a set of well-defined interfaces, and each method call looks like a call through the local object representation.

In a way, remote objects are to object-oriented languages what RPC is to procedural languages. The local semantics are extended to distributed systems as seamlessly as possible.



Implementing Remote Objects With Java Technologies

The Java 2 SDK contains two technologies to do remote objects: the Java programming language's native Remote Method Invocation (RMI) technology and Java Interface Definition Language (JavaIDL) technology. This section does not compare them in detail. Both technologies have their strengths and weaknesses. At the end of the course you should be able to decide which technology to use under which circumstances.

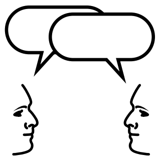
Remote Method Invocation Technology

RMI was introduced with the JDK™ software, Version 1.1. All you need to implement and test an RMI application is part of the Java 2 SDK. However, before you put an RMI application into production, consider carefully your requirements for fault tolerance and performance. The Java 2 SDK "RMI Registry" (a simple naming service, as well as a simple implementation repository) is usually too weak in production environments. As one result of the Enterprise JavaBeans initiative, robust RMI servers should be available soon.

JavaIDL Technology

JavaIDL is part of the Java 2 SDK. The JavaIDL technology allows you to build and test client and server objects conforming to the CORBA specification. The JavaIDL technology shares the limitations about production environments with the RMI technology. To put a JavaIDL application in production, you should consider using an extremely robust object request broker (ORB) to host your server objects.

Understanding Remote Objects



Discussion – Use a couple of minutes to read through the following questions, and prepare some answers on your own. Then discuss the questions in the class.

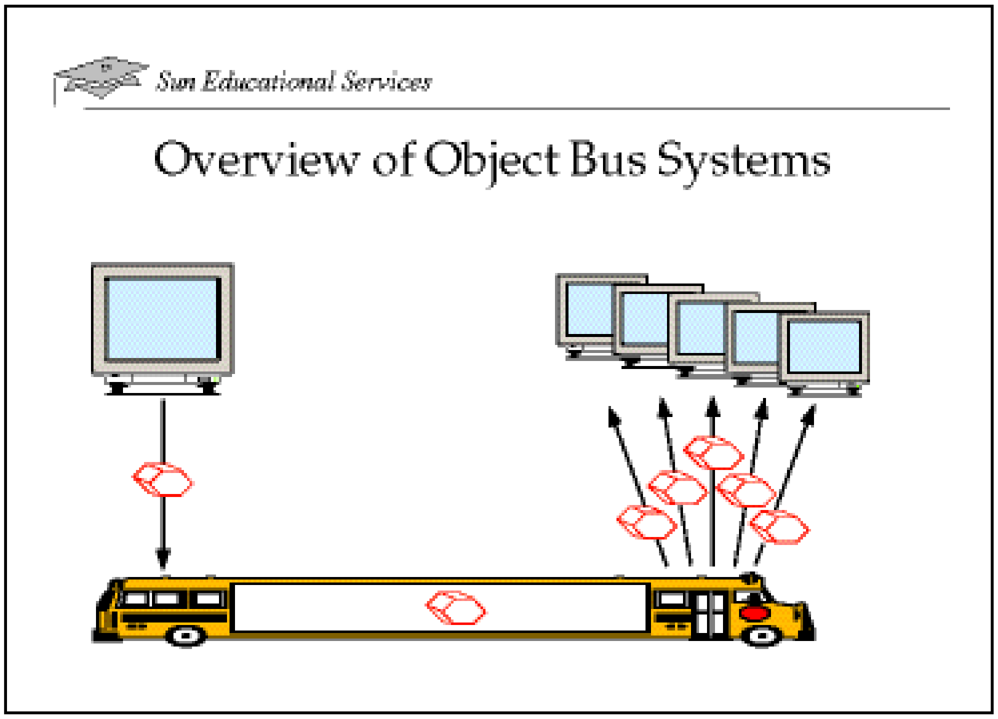
- List products or technologies you know that are implemented using remote objects.

✓ ***Again, servers that are used via remote objects are not widely used. However, from an application developer perspective, all the services used in a CORBA environment (naming, transaction and so on) are accessed using remote objects.***

- Remember how you do local event handling in the Java programming language. Can the same design be applied remotely? What would be different remotely? Is there a better solution?

✓ ***The complete answer to this question is covered in "Object Bus Systems." A quick summary is:***

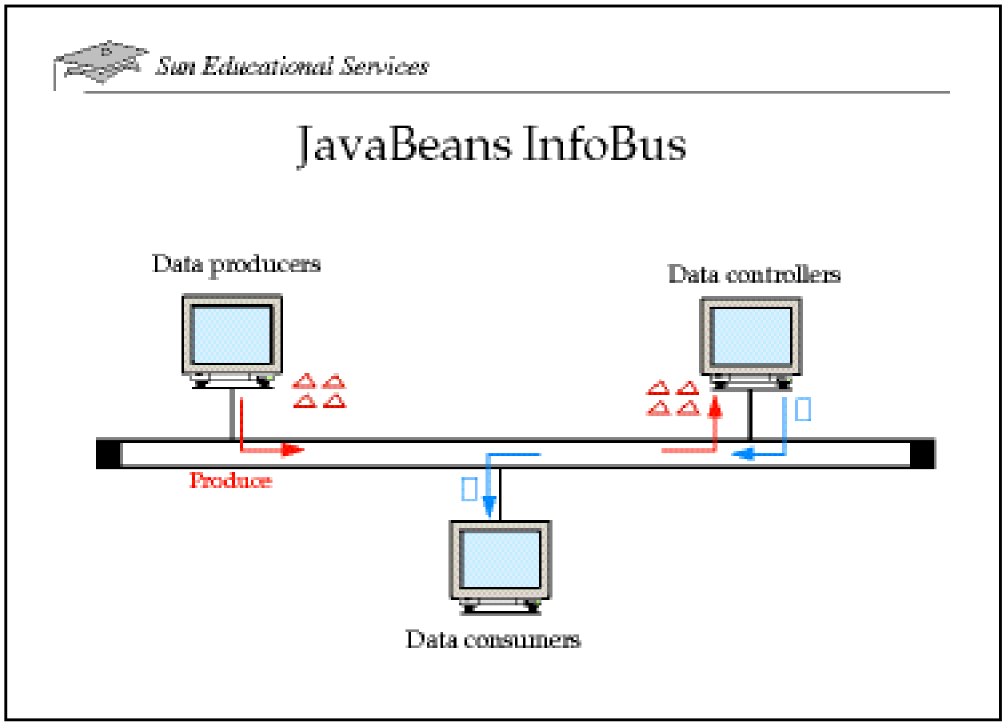
- ***Local event handling is done via a linked list (usually a Vector), holding the references to the registered event listener objects. The list is processed from top to bottom, and on each listener object the event handler is called, and the event object passed.***
- ***If the same setup is used for remote object references, partial failure must be taken into account: If the fifth reference points to a remote object with a currently broken connection, the event handling essentially stops at this point in time (at least until a timeout has expired). The rest of the list will get the event after a huge delay.***



Overview of Object Bus Systems

With remote objects, you cannot solve the problem of event distribution. Local event distribution is usually done serially. The event source goes through a list of registered event listeners and invokes their event handlers, handing the same event to every listener. You could do exactly the same process with remote objects; however, because of the network delays in distributed systems, going through a large list of event listeners and remotely calling event handlers is slow and generates a lot of traffic. It also seems unnecessary. The event being distributed is always the same, so why not post it *once* on the wire, and all the interested remote objects can catch the event object as it passes by?

Essentially an object bus system allows you to post an event to an abstraction of a wire called a software channel. Listeners interested in the channel register with the channel so they receive every posting made to the channel. This frees the event source from having to keep a list of interested listeners; they have to post only to a specific channel of registered listeners.



JavaBeans InfoBus

The JavaBeans Infobus is a type of object bus, but it has a different focus than “pure” object buses. It facilitates the distribution of data (rather than the flow of objects) between different JavaBeans objects. However, because the data to be exchanged is encapsulated in objects, the InfoBus is still considered to be an object bus. Because the InfoBus specializes in exchanging data, it supports the following items that are not supported by a “pure” object bus.

- Semantics of the data on the bus
- Membership protocols
- Security protocols
- Specialized events indicating items, such as “data ready to be received”

For more information on the InfoBus technology see the JavaBeans Web site: `http://www.java.sun.com/beans/`

Understanding Object Bus Systems

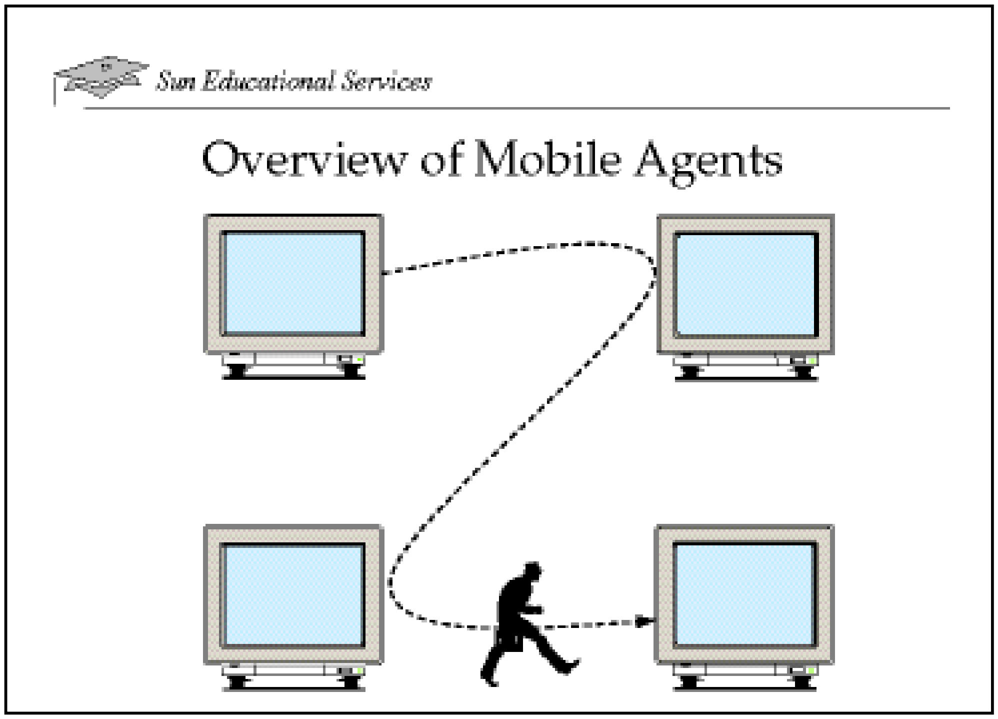
Discussion – Take a couple of minutes to read through the following questions, and prepare some answers on your own. Then discuss the questions in the class.

- Normal TCP/IP connections are point-to-point. If you have the channel abstraction, but must use normal TCP/IP connections, what are the consequences?

✓ ***Basically, you are back to having to send the event to each machine individually. Network traffic and the latency when sending an event are high. Partial failure, however, can be handled by the object bus system internally.***

- What would be better technologies to use as the transport?

✓ ***IP broadcast or IP multicast. If there are several different channels, with high volumes of data each, IP multicast generates a serious load on the clients, because they have to filter "their" events out. IP multicast delegates this problem to the Ethernet hardware, or at least to the Ethernet driver level.***


Sun Educational Services

Overview of Mobile Agents

Overview of Mobile Agents

The term *mobile agent* is used to refer to a variety of items. This section describes the concept of mobile agents in their broadest sense.

So far, when objects are created they remain on one machine. They use the network to communicate with each other, using remote method invocation. While they can exchange other objects with each other (remember, even a Java programming language String is an object), the objects being exchanged tend to represent pure data, with accessor methods to this data.

Mobile agents go one step further: The objects being exchanged carry *behavior* as well as data. For example, an object representing a bid to buy some stock. The "pure data" version contains a desired buying price and a desired amount. If, at the other end, the price is at your price *or lower*, you will make the deal with exactly your price, certainly not cheaper. Your application creating the bid object is therefore responsible for getting you a reasonable price.

However, the “agent” version can contain a price range and an amount range, as well as some intelligence to make a deal. It is now the responsibility of the bid object itself to get you the best deal possible, while the application creating the bid object merely fills in your parameters.

Exactly how intelligent these objects are, whether they can travel by themselves or have to be moved, and whether there is a strong security focus or no security focus, differentiates the mobile agent frameworks that are currently available or being researched.

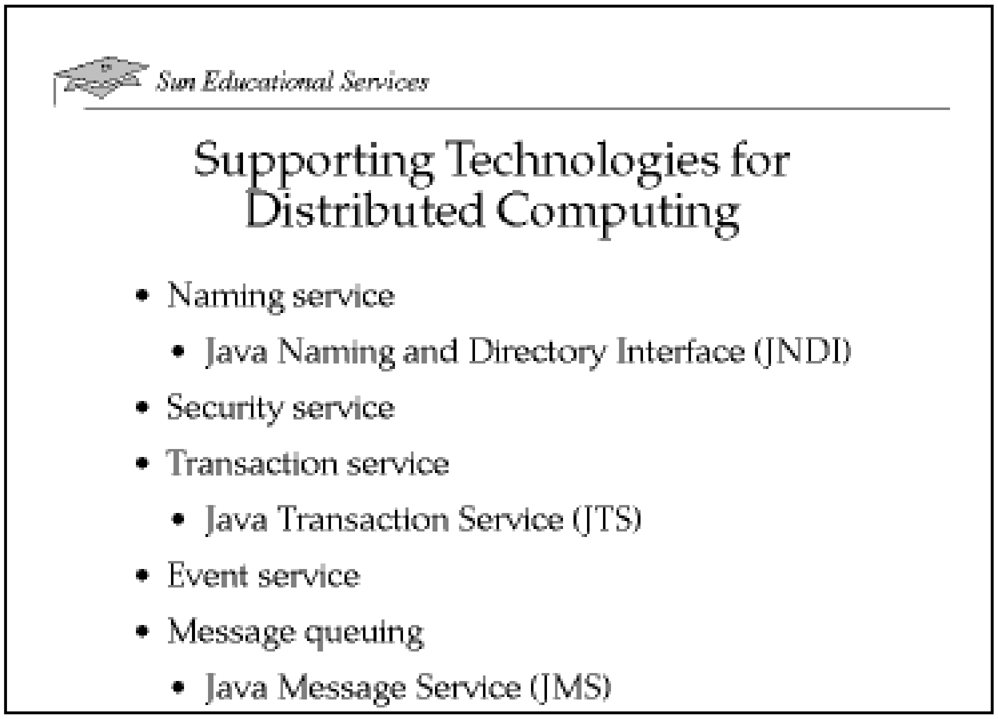
✓ ***One popular project is the IBM Aglet Workbench. According to the Aglets home page: “Aglets Workbench is a visual development environment for building aglets, a new breed of intelligent agent that can travel over a network and execute tasks at the same time. Aglets combine intelligent agent technology with network-savvy Java technology objects. They can go from one computer, or Internet host, to another while running and carrying data with them as they go.”***

Understanding Mobile Agents

Discussion – Take a couple of minutes to read through the following question, and prepare some answers on your own. Then discuss the question in the class.

- Imagine a scenario where a mobile agent could be used.

✓ ***In many ways, mobile agents are still "a solution seeking a problem to solve". Systems in development include network management agents, trading agents, or information-gathering agents.***


Sun Educational Services

Supporting Technologies for Distributed Computing

- Naming service
  - Java Naming and Directory Interface (JNDI)
- Security service
- Transaction service
  - Java Transaction Service (JTS)
- Event service
- Message queuing
  - Java Message Service (JMS)

Supporting Technologies for Distributed Computing

No matter which technology you decide to use for distributed computing, some tasks do not change. For example, you almost always need to locate server machines or objects, knowing nothing more than a service name. A naming service that resolves this name to an address comes in handy at this point. There is a whole range of such supporting technologies. One of the most comprehensive lists is given in the CORBA specification (`http://www.omg.org/`). The following pages describe the most important technologies, including:

- Naming service
- Security service
- Transaction service
- Event service
- Message queueing

Naming Service

In a broad sense, this service manages name-to-object associations. You can *bind* a name to an object, and you can *resolve* an object from a name. A specialized example in use daily is the Domain Name System (DNS), which resolves domain names, such as `java.sun.com` to Internet addresses, 204.160.241.19.

JNDI is intended to be a Java programming language standard extension, which means it comes free with the Java 2 SDK. The main goal of JNDI is to provide a standardized API to connect from the Java programming language to legacy enterprise naming services, such as lightweight directory access protocol (LDAP), network information system (NIS), or DNS.

Security Service

Security in large distributed systems, potentially multi-enterprise, is a difficult issue. For example, objects arriving on a system might claim to work "on behalf" of some person (remember the "bid" object). How do you guarantee that this claim is true? How do you prevent the introduction of malicious code into your system? With the Java 2 SDK, the Java programming language has most of the necessary basic mechanisms built in to construct a secure system. However, you still must construct the security system, which is error prone. A *security service* does this for you, giving you a whole system to work with, instead of several basic, less effective mechanisms.

Transaction Service

A transaction service helps you build transactions. A transaction is a *unit of work* that has several characteristics. The most important characteristic is that a transaction is *atomic*; this means if a transaction is interrupted by failure, all effects are undone (rolled back). It also means that you cannot see intermediate states, such as a temporarily inconsistent database. Transactions can be local to an address space, on a remote address space, or span multiple address spaces.

So far, not much is known about the Java Transaction Service. The following paragraph is cited from the "Java for the Enterprise" home page:

Java Transaction Service (JTS) is a low-level API used by sophisticated transactional application programs, resource managers, transaction processing monitors, transaction-aware communication managers, and transaction managers. Since these components are provided by different vendors, JTS's role is to ensure their interoperability in the Java environment.

Event Service

An event service enables you to send and receive events in distributed systems, in much the same way you do it locally with the regular Java programming language event mechanism.

Using an event service frees you from thinking about how events are optimally delivered in distributed systems; you just use the API. However, when purchasing an event service, be sure to keep the discussion in Module 6, ‘‘Object Bus Systems,” in mind. How the event service is implemented has a huge impact on performance and network traffic.

Message Queueing

A regular RMI call is synchronous. If object A calls object B remotely using RMI, object A’s calling thread is blocked until the call returns. This is exactly the same as in a local environment.

However, this is not always what you want. Imagine an application that lets a user fill out a credit application form. After the data is collected, an AppForm object is constructed, and, using RMI, a processAppForm method on the manager’s machine is called. This might work if the manager is at the desk at this moment and immediately gives approval. A better solution would be to send the AppForm object, and start filling out the next application (similar to if you had sent the application by email). Your client application can then pick up the processed AppForm object *asynchronously*, as soon as it is ready. A message queueing system supports you in building such an application.

Java Message Service

The Java Message Service (JMS) provides a standard API to existing message oriented middleware. It is the joint work of several enterprise messaging product vendors.

Supporting Technologies

Discussion – Take a couple of minutes to read through the following question, and prepare some answers on your own. Then discuss the question in the class.

- What other supporting technologies do you know or think should be available?

✓ ***The CORBA specification lists these CORBA services: Naming, Event Management, Persistent Object, Lifecycle, Concurrency, Externalization, Relationships, Transactions, Query, Licensing, Property, Time, Security Services, Trader Object Service, and Collection.***

Check Your Progress

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

- ❑ List the five supporting technologies for distributed computing
- ❑ Compare and contrast the different architectures for distributed computing
- ❑ List the distributed computing technologies available for the Java programming language

Think Beyond

Which of the distributed computing architectures could you use for a current development project?

Java Database Connectivity (JDBC) 2

Objectives

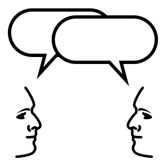
Upon completion of this module, you should be able to:

- Explain JDBC
- Describe the five major tasks involved in the JDBC programmer's interface
- State the requirements of a JDBC driver and its relationship to the JDBC driver manager
- Explain how to map database types into the Java programming language
- Describe how to use JDBC with a Java applet
- Compare and contrast two-tier and three-tier designs for JDBC drivers
- Create a JDBC application to solve a defined problem

The JDBC API enables developers to write code for interfacing with a database, without knowing the specifics of the database implementation.

Relevance

✓ ***Present the following questions to stimulate the students and get them thinking about the issues and topics presented in this module. They are not expected to know the answers to these questions. The answers to these questions should be of interest to the students, and inspire them to learn the content presented in this module.***



Discussion – Suppose you are tasked with writing an application that retrieves, updates, modifies, and deletes information from a database. Consider the following questions:

- What do you need to know about the database?

✓ ***You do not need to know very much about the database at all. By creating a generic set of interfaces that can be used to connect to any database vendor's product, the JDBC API does not restrict you to a specific database. Moreover, the JDBC API does not restrict you in the database design, because there are several design alternatives.***

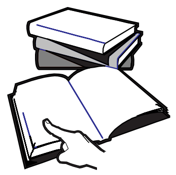
- What is the impact on the end-user (client) program if a different database is used?

✓ ***The developer writes an application using the interfaces described in the API as though they were actual class implementations. The driver vendor provides a class implementation of every interface in the API. If you move your code to a different database, you need to obtain a different driver implementation for the new database. You should not have to modify your code.***

- What considerations must be made to make the system flexible and responsive to changes in the business? For example, suppose that certain customers need to be denied service.

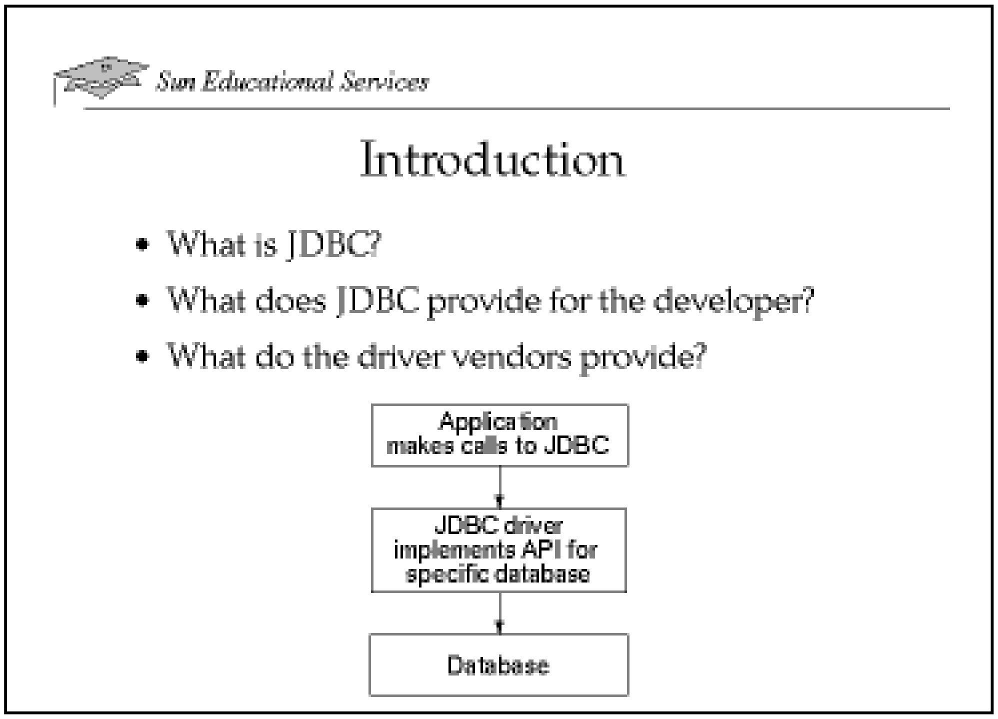
✓ ***This question gets at the development of three-tiered database systems, where the middle layer of software abstracts the data interaction from the end-user programs and provides a means to create business rules, where interactions with the database can be monitored.***

Additional Resources



The following resources can provide additional detail on the topics presented in this module:

- JDBC specification. [Online]. Available: `http://java.sun.com/products/jdbc`
- Bowman, Judith S., Sandra L. Emerson, Marcy Darnovsky. 1996. *The Practical SQL Handbook: Using Structured Query Language.* Addison-Wesley.



Sun Educational Services

Introduction

- What is JDBC?
- What does JDBC provide for the developer?
- What do the driver vendors provide?

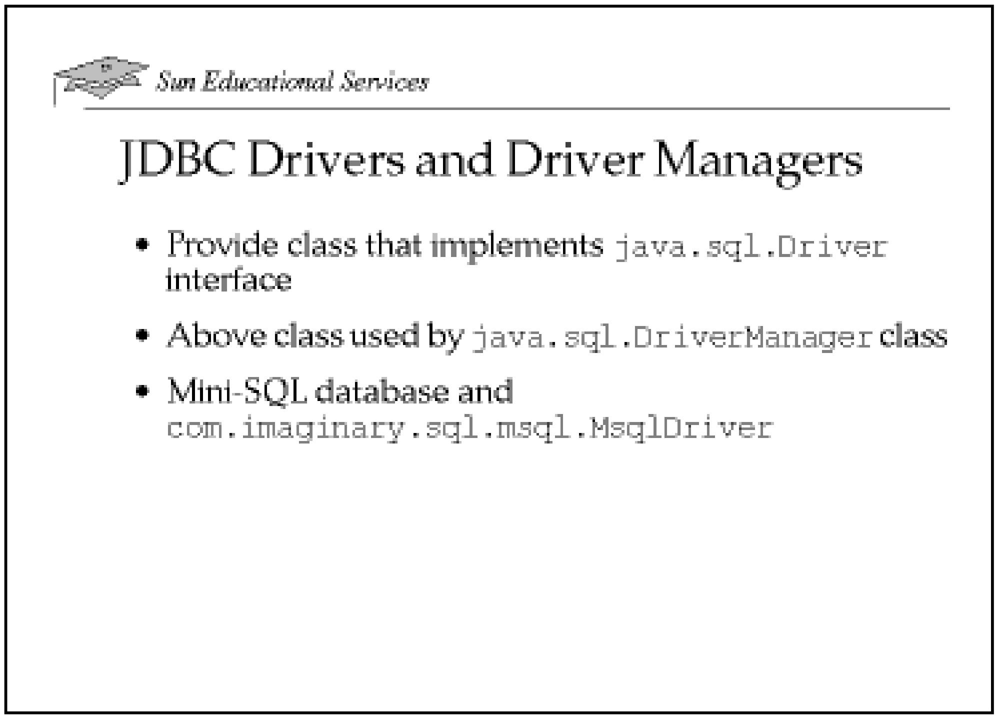
Application makes calls to JDBC
JDBC driver implements API for specific database
Database

Introduction

The JDBC API is a set of interfaces designed to insulate a database application developer from a specific database vendor. The JDBC API enables the developer to concentrate on writing the application—making sure that queries to the database are correct and that the data is manipulated as designed.

With the JDBC, the developer can write an application using the interface names and methods described in the API, regardless of how they were implemented in the driver. The developer writes an application using the interfaces described in the API as though they are actual class implementations. The driver vendor provides a class implementation of every interface in the API so that when an interface method is used, it is actually referring to an object instance of a class that implemented the interface.

JDBC API also enables developers to pass any string directly to the driver level. This makes it possible for developers to make use of custom features of their database without requiring that the application use only American National Standards Institute (ANSI) Structured Query Language (SQL).



Sun Educational Services

JDBC Drivers and Driver Managers

- Provide class that implements `java.sql.Driver` interface
- Above class used by `java.sql.DriverManager` class
- Mini-SQL database and `com.imaginary.sql.msql.MsqlDriver`

JDBC Drivers and Driver Managers

A JDBC "driver" is a collection of classes that implement the JDBC interfaces required to connect a Java program to a database. Each database driver must provide a class that has implemented the `java.sql.Driver` interface. The database driver class is used by the generic `java.sql.DriverManager` class when it needs a driver to connect to a particular database using a Uniform Resource Locator (URL) string. JDBC is patterned after Open Database Connectivity (ODBC); this makes the task of providing a JDBC implementation on top of ODBC small and efficient (Figure 2-1 on page 2-6). ODBC is a standardized API for accessing databases. ODBC provides similar functionality as JDBC for traditional languages, such as C and C++.

In this class, you use the `com.imaginary.sql.msql.MsqlDriver`[1]; a JDBC driver written to connect to a Mini-SQL database[2]. The JDBC implementation is actually written on top of the `msql` package[3]—an API written to connect Java applets and applications to a Mini-SQL database through a Java socket connection.

1. mSQL-JDBC API is provided courtesy of George Reese.

2. Mini SQL is provided with the courtesy of Hughes Technologies Pty Ltd, Australia (see `http://www.Hughes.com.au`).

3. MsqlJava API is provided courtesy of Darryl Collins.

Figure 2-1 illustrates how a single Java application (or applet) can access multiple database systems through one or more drivers.

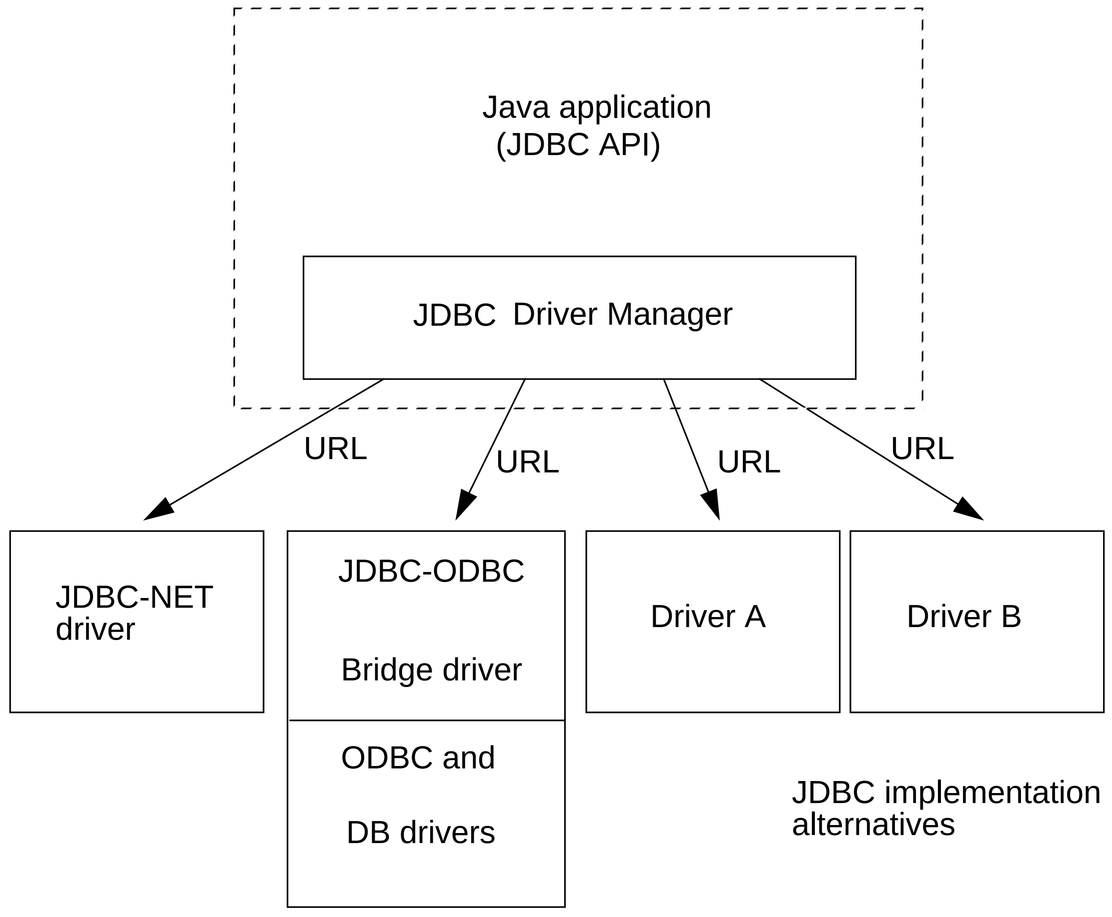
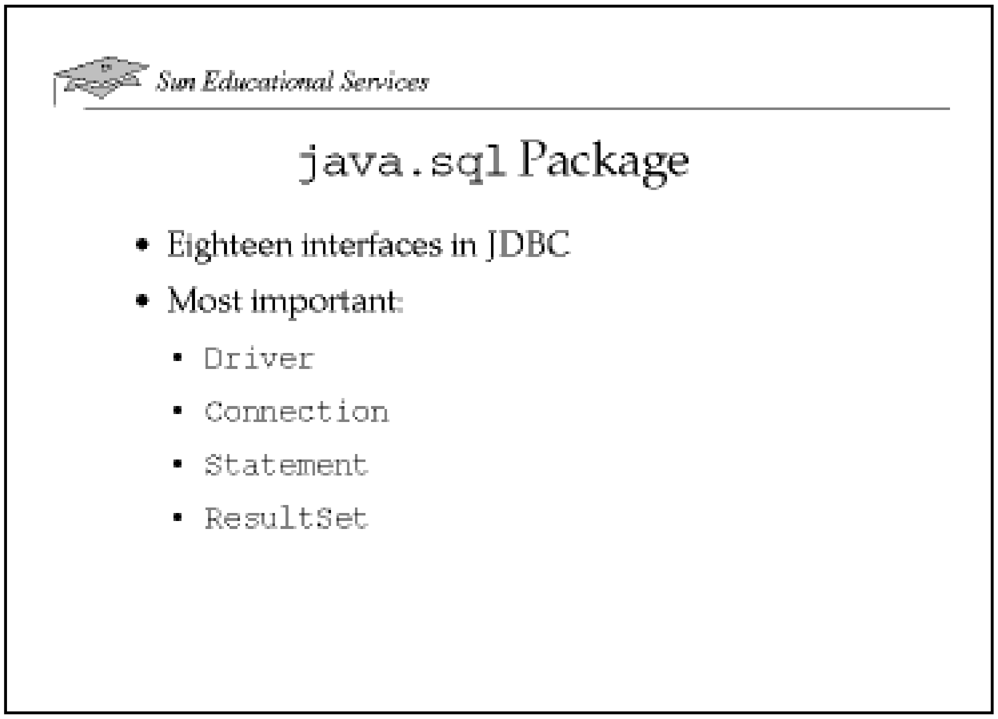


Figure 2-1 JDBC Drivers



Sun Educational Services

java.sql Package

- Eighteen interfaces in JDBC
- Most important:
  - Driver
  - Connection
  - Statement
  - ResultSet

java.sql *Package*

There are 18 interfaces associated with the JDBC. They are included in the following list and shown in Figure 2-2 on page 2-9:

- `Driver`
- `Connection`
- `Statement`
- `PreparedStatement`
- `CallableStatement`
- `ResultSet`
- `ResultSetMetaData`
- `DatabaseMetaData`
- `ArrayLocator` (new in JDBC 2.0)
- `Bloblocator` (new in JDBC 2.0)
- `ClobLocator` (new in JDBC 2.0)

● `Ref`	(new in JDBC 2.0)
● `SQLData`	(new in JDBC 2.0)
● `SQLInput`	(new in JDBC 2.0)
● `SQLOutput`	(new in JDBC 2.0)
● `SQLType`	(new in JDBC 2.0)
● `Struct`	(new in JDBC 2.0)
● `StructLoader`	(new in JDBC 2.0)

✓ ***All of these interfaces need to be implemented, but whether you do anything with the methods is up to you.***

`java.sql` *Class Hierarchy*

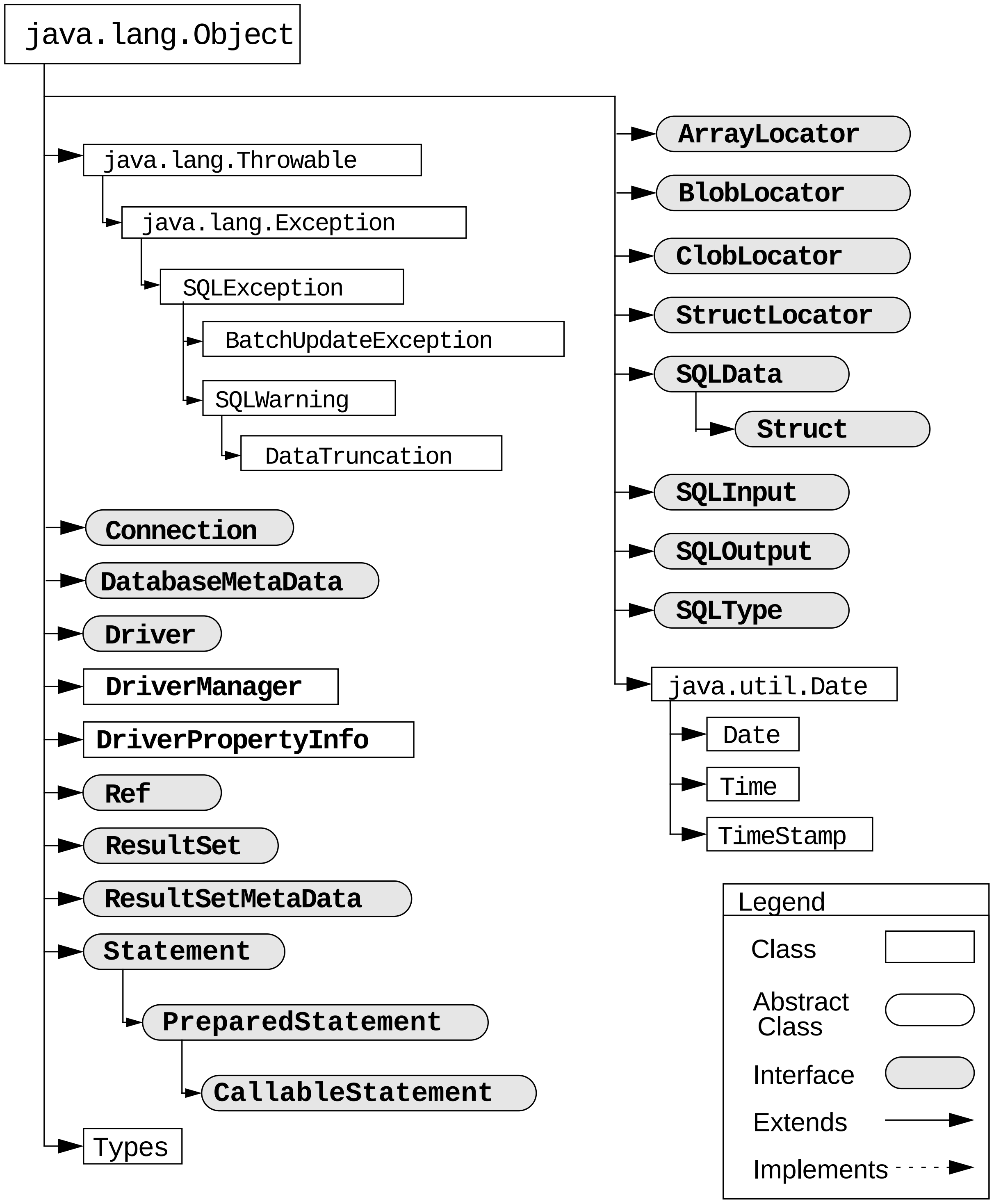
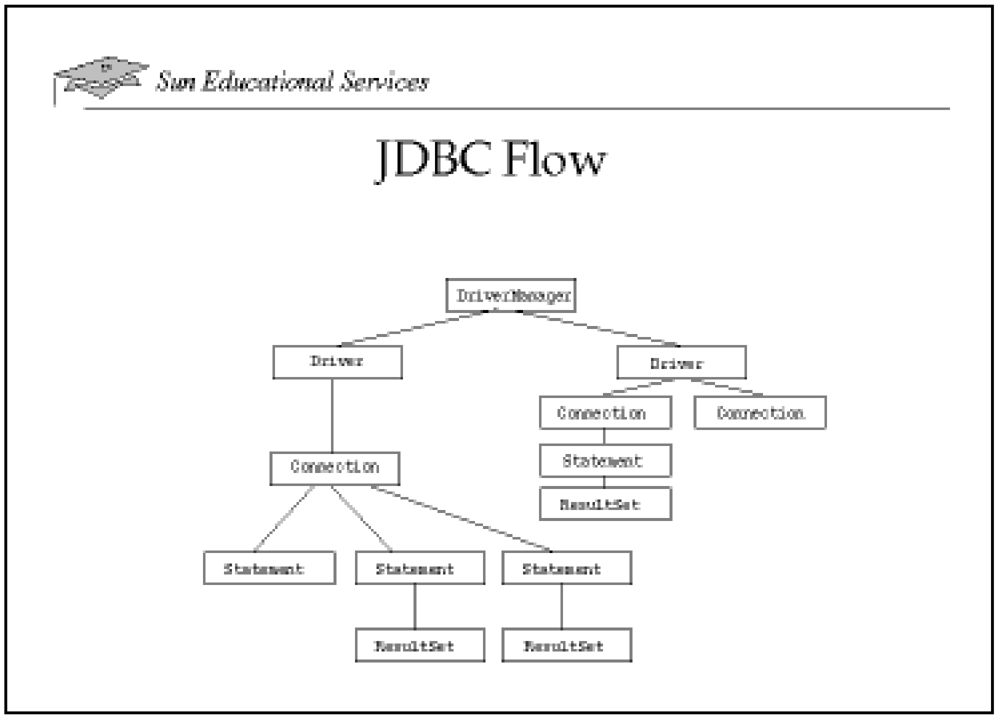


Figure 2-2 `java.sql` Class Hierarchy

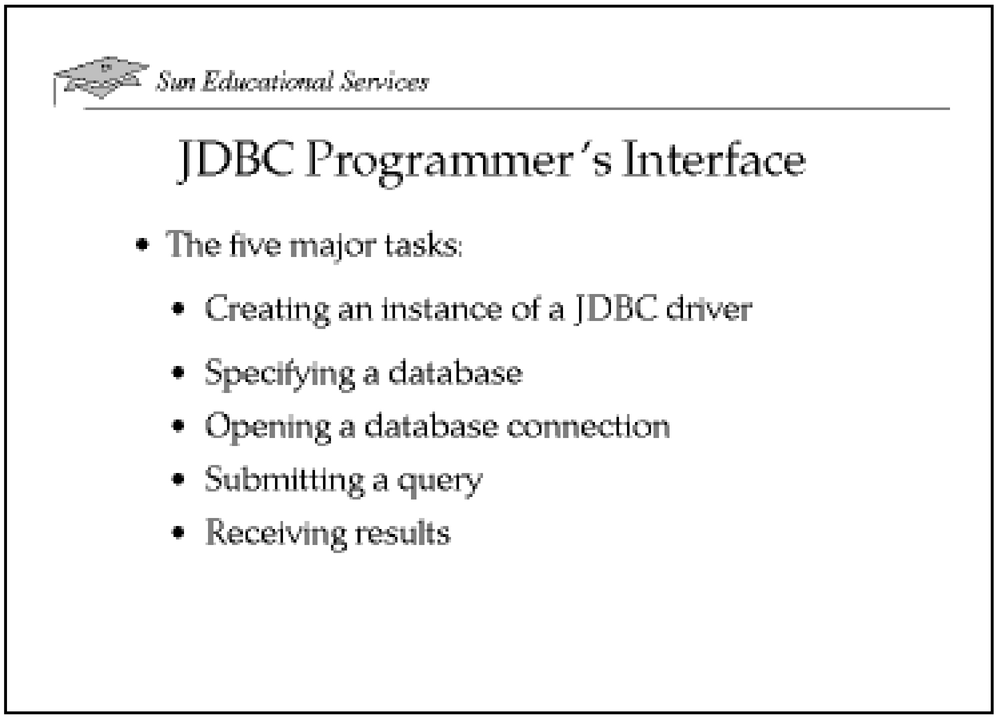


JDBC Flow

Each of the interfaces shown in Figure 2-2 enable you to open connections to particular databases, execute SQL statements, and process the results.

- A URL string is passed to the `getConnection` method of the `DriverManager`, which in turn locates a `Driver` interface.
- With a `Driver` interface, 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.

✓ ***A `ResultSet` object is always returned, but may not contain data when an update or insert query is executed.***



Sun Educational Services

JDBC Programmer's Interface

- The five major tasks:
  - Creating an instance of a JDBC driver
  - Specifying a database
  - Opening a database connection
  - Submitting a query
  - Receiving results

JDBC Programmer's Interface

This section covers some of the common tasks you perform using the JDBC programmer's interface. The following discussion assumes that you communicate with an mSQL database driver through the JDBC API. The information is divided into the following parts, which are separated by lessons:

- Understanding a JDBC example
- Using JDBC drivers (including how to create an instance and specify a database)
- Opening a database connection
- Using JDBC statements (including how to submit a query and receive results)

JDBC Example

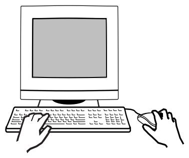
The following is a simple example illustrating the common tasks you perform with JDBC. This example uses the Mini-SQL database in the lab and shows how to create a `Driver` instance, a `Connection` object, and a `Statement` object; execute a query; and process the returning `ResultSet` object.

```
import java.sql.*;

public class JDBCExample {

    public static void main (String [] args) throws Exception {
        if (args.length < 1) {
            System.err.println("Usage:");
            System.err.println("JDBCExample jdbc-url");
            System.exit(1);
        }

        // Create a JDBC url
        String url = args[0];

        // The JDBC driver to use
        String driver = "com.imaginary.sql.msql.MsqlDriver";

        // The query to execute on the database
        String query = "select * from COFFEES";

        // Load the jdbc driver
        Class.forName(driver);

        // Use the driver manager to get a connection to the db
        Connection con = DriverManager.getConnection(url);

// Use the connection to create a statement
        Statement stmt = con.createStatement();

        // Execute a query using the statement
        ResultSet rs = stmt.executeQuery(query);


        // Print the result (row by row)
        while (rs.next()) {
            System.out.println();
            System.out.println("Coffee Name: " + rs.getString(1));
            System.out.println("Supplier Id: " + rs.getInt(2));
            System.out.println("Price      : " + rs.getFloat(3));
            System.out.println("Sales      : " + rs.getInt(4));
        }
    }
}

```

When the code is run, the contents of the Coffee table in the database are printed.

```
java JDBCExample jdbc:msql://<host name>:4333/le-shop

Coffee Name: Colombian
Supplier Id: 101
Price      : 7.99
Sales      : 0

Coffee Name: French_Roast
Supplier Id: 49
Price      : 8.99
Sales      : 0

Coffee Name: Espresso
Supplier Id: 150
Price      : 9.99
Sales      : 0

Coffee Name: Colombian_Decaf
Supplier Id: 101
Price      : 8.99
Sales      : 0
```

Exercise: Compiling a JDBC Application



Exercise objective – Compile and run a simple JDBC application.

Preparation

You need to know the following to complete this exercise:

- How to edit and compile Java technology source code ("Java source code") in the environment in which you are working
- The database host location and name
- What JDBC driver you are using and how the URL is called

✓ ***Assign each student a different database to work with. The databases are named*** `le-shop1` ***to*** `le-shop9`***. The URL for the exercise is*** `jdbc:msql://<host name>:4333/<database>`***, where*** `<host name>` ***is the machine name designated as the database server and*** `<database>` ***is replaced with*** `le-shop1` ***to*** `le-shop9`***.***

Tasks

Compile the JDBC Example

Complete the following steps:

1. Change the directory to `labfiles/mod2-jdbc/lab`.
2. Compile `JDBCExample.java`.

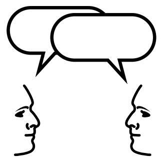
Run the JDBC Example

1. Run the example by typing the following in a shell:
   `java JDBCExample URL`
2. Study the results.

Study the Source Code (Optional)

1. Study the source code and get an idea of how JDBC works.

Exercise Summary



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

✓ ***Manage the discussion here based on the time allowed for this module, which was given in the "About This Course" module. If you find you do not have time to spend on discussion, then just highlight the key concepts students should have learned from the lab exercise.***

- Experiences

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

- Interpretations

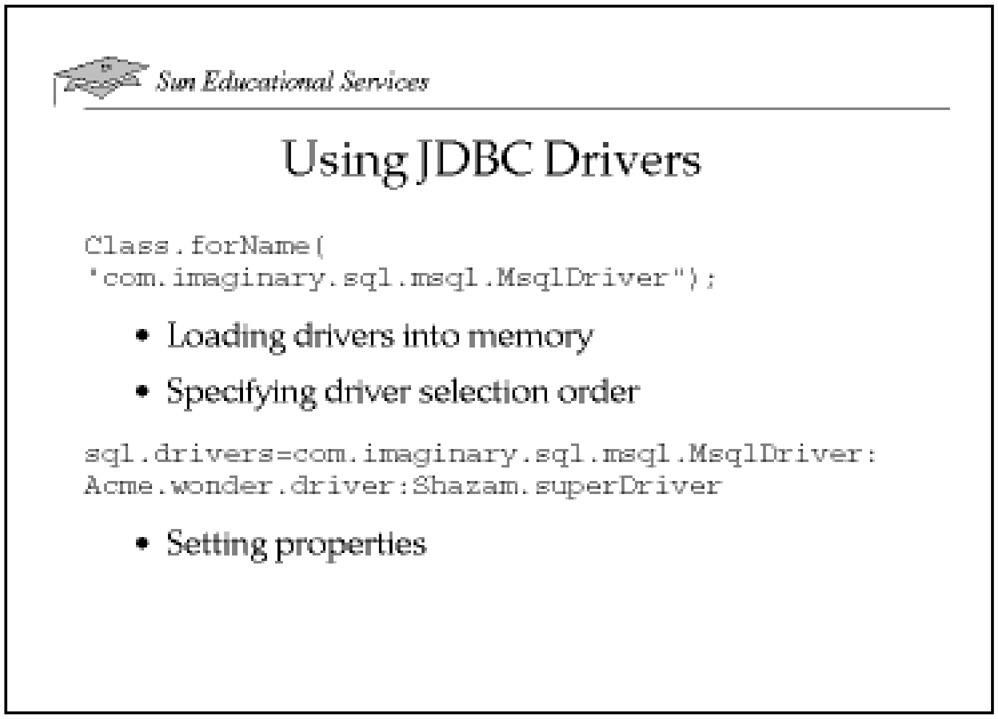
✓ ***Ask students to interpret what they observed during any aspects of this exercise.***

- Conclusions

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

- Applications

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



Sun Educational Services

Using JDBC Drivers

```
Class.forName(
'com.imaginary.sql.msql.MsqlDriver");
```

- Loading drivers into memory
- Specifying driver selection order

```
sql.drivers=com.imaginary.sql.msql.MsqlDriver:
Acme.wonder.driver:Shazam.superDriver
```

- Setting properties

Using JDBC Drivers

There are two ways to create a JDBC Driver instance: explicitly, or through the `jdbc.drivers` property.

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. The driver has a static initializer that registers an instance of itself with the driver manager. This driver remains behind the scenes, handling any requests of that type of database.

```
// Load the driver class
Class.forName("com.imaginary.sql.msql.MsqlDriver");
```

To load and register a driver, you can also create an instance of the driver yourself.

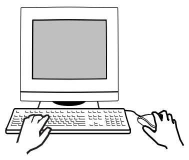
```
// Create an instance of the JDBC Driver by Imaginary
new com.imaginary.sql.msql.MsqlDriver();
```

It is not necessary to associate this driver with a variable; the driver exists after it is instantiated and successfully loaded into memory.

✓ ***The first method is the preferred way to load and register a driver.***

✓ ***No variable is needed with*** `new com.imaginary.sql.msql.MsqlDriver` ***because the constructor creates a static instance of itself that remains in memory because of a system reference. Also note that the driver is responsible for calling the*** `DriverManager.registerDriver(this)` ***method.***

Exercise: Loading a Driver



Exercise objective – Learn how to register a driver with the driver manager.

Tasks

Write a Program to Load and Register a Driver With the Driver Manager

Complete the following steps:

1. Change the directory to `labfiles/mod2-jdbc/lab.`
2. Write a simple class `LoadDriver`, which loads the mSQL driver. Use the methods defined by the `Driver` interface to print information, such as the driver's name and version on the console. Use `LoadDriver.java` as a template.
3. Compile and run your application. Use the same JDBC URL as in the previous exercise.

Experiment With the Driver

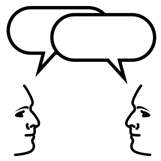
1. The LoadDriver program never accesses any data in the database, so try to shorten the provided JDBC URL until the program does not work anymore. What is the shortest URL?

✓ ***The minimal URL for the mSQL driver is*** `jdbc:msql:`***. The hostname, table, username and password are not needed by the driver manager to find the mSQL driver.***

2. Find out why the mSQL driver is not JDBC compliant (optional).

✓ ***The mSQL database does not implement all features needed for JDBC compliance. Therefore the driver is not compliant either.***

Exercise Summary



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

✓ ***Manage the discussion here based on the time allowed for this module, which was given in the "About This Course" module. If you find you do not have time to spend on discussion, then just highlight the key concepts students should have learned from the lab exercise.***

- Experiences

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

- Interpretations

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

- Conclusions

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

- Applications

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

Loading JDBC Drivers Through `jdbc.drivers`

You can have more than one database driver loaded into memory. You can also have more than one of the drivers loaded into memory, ODBC or a JDBC generic network protocol is capable of connecting to the same database. If this is the case, JDBC allows you to specify a list of drivers in a specified order. The order of selection is specified by a Java programming language properties tag, `jdbc.drivers`. The `jdbc.drivers` property should be defined as a colon-separated list of driver class names:

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

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.

Registering a Driver

In either case, when a driver is loaded, it is the responsibility of the driver implementation to register itself with the driver manager. For example, the Mini-SQL driver, `com.imaginary.sql.MsqlDriver` constructor looks like the following:

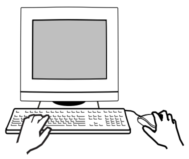
```
1  public class MsqlDriver implements java.sql.Driver {
2    static {
3      try {
4        new MsqlDriver();
5      }
6      catch(SQLException e) {
7        e.printStackTrace();
8      }
9    }
10   /**
11    * Constructs a new driver and registers it with
12    * java.sql.DriverManager.registerDriver() as specified by the JDBC
13    * draft protocol.
14    */
15   public MsqlDriver() throws SQLException {
16     java.sql.DriverManager.registerDriver(this);
17   }
18 ...
19 }
```

The JDBC driver by Imaginary creates an instance of itself in the static code block (lines 2–9).

When the constructor is called, either explicitly or implicitly, the driver registers itself with the driver manager (line 16).

✓ ***Note that as soon as the class is loaded, an instance of the driver is registered with the driver manager. Therefore use*** `Class.forName(<class name>)` ***to register the driver.***

Exercise: Loading Driver 2 (Optional)



Exercise objective - Learn how JDBC drivers are registered implicitly with the driver manager.

Tasks

Remove the Explicit Driver Registration From LoadDriver

Complete the following steps:

1. Change the directory to `labfiles/mod2-jdbc/lab.`
2. Remove the explicit driver registration from your `LoadDriver.java` (or use the already complete `LoadDriver2.java`).
3. Compile and run the program. Set the name of the drivers class (`com.imaginary.sql.msql.MsqlDriver`) in the `jdbc.drivers` property.

   ```
   java -Djdbc.drivers=com.imaginary.sql.msql.MsqlDriver
   LoadDriver2 jdbc:msql://<host name>:4333/le-shop
   ```

✓ ***Note – The property `jdbc.drivers` does not determine the sequence in which JDBC drivers are checked. It eliminates the need for any `class.forName(“...”)` statement.***

Exercise Summary

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

✓ ***Manage the discussion here based on the time allowed for this module, which was given in the "About This Course" module. If you find you do not have time to spend on discussion, then just highlight the key concepts students should have learned from the lab exercise.***

- Experiences

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

- Interpretations

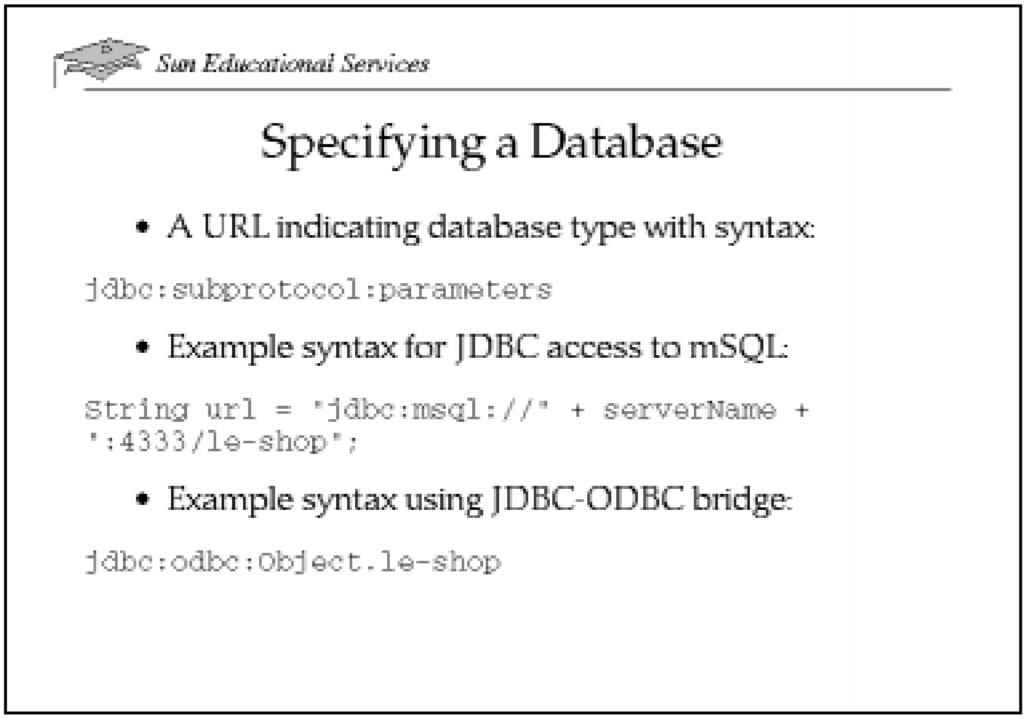
✓ ***Ask students to interpret what they observed during any aspects of this exercise.***

- Conclusions

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

- Applications

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



Sun Educational Services

Specifying a Database

- A URL indicating database type with syntax:

```
jdbc:subprotocol:parameters
```

- Example syntax for JDBC access to mSQL:

```
String url = "jdbc:msql://" + serverName +
":4333/le-shop";
```

- Example syntax using JDBC-ODBC bridge:

```
jdbc:odbc:Object.le-shop
```

Specifying a Database

Now that you have created the instance of the JDBC driver, you need to name the database to which you want to connect.

In JDBC, you do this by specifying a URL that indicates the database type. The proposed URL string syntax for a JDBC database is the following:

✓ ***Note –*** ***This is a URL-like string,*** **not** ***a Java URL object.***

```
jdbc:subprotocol:parameters
```

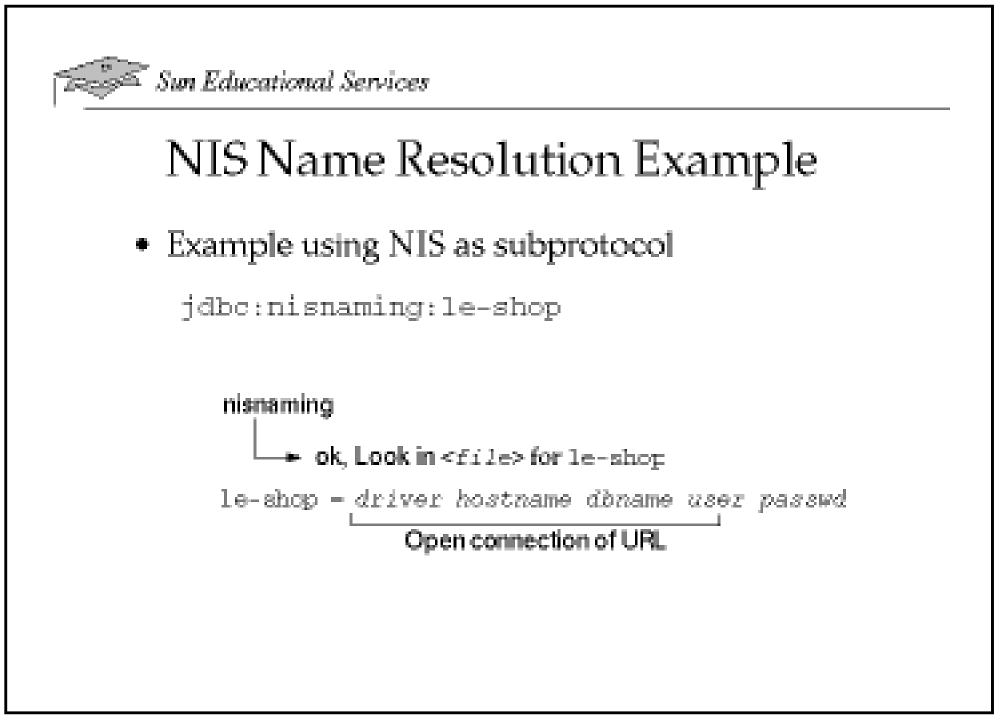
where `subprotocol` names a particular kind of database connectivity mechanism that can be supported by one or more drivers. The contents and syntax of the `parameters` depends on the subprotocol.

```
// Construct the URL for JDBC access
String url = args[0];
```

This is the URL for the JDBC access to the mSQL Ticketing database you will connect to in the lab. `serverName` is a variable set to the host name of the database server.

The subprotocol in the previous instance is `msql`. It could have been any other type of protocol that is accessed through a JDBC-ODBC bridge; for example:

```
jdbc:odbc:Object.le-shop
```



Sun Educational Services

NIS Name Resolution Example

- Example using NIS as subprotocol

```
jdbc:nisnaming:le-shop
```

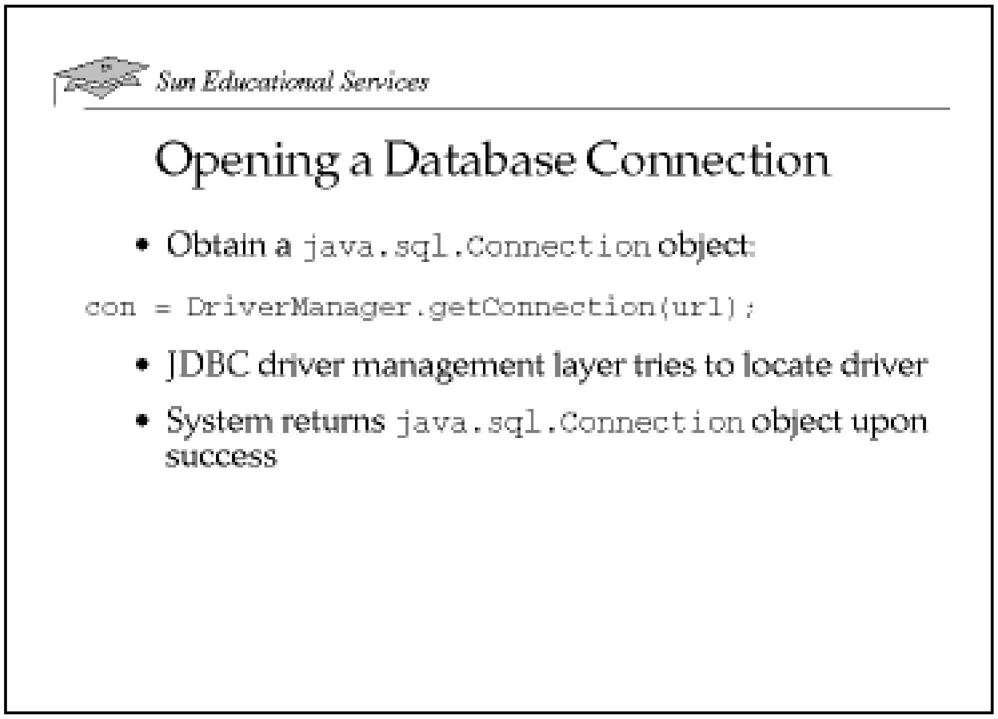
nisnaming
ok, Look in <file> for le-shop
le-shop = driver hostname dbname user passwd
Open connection of URL

NIS Name Resolution Example

You can use a subprotocol, such as NIS to resolve the name of a specified database:

```
jdbc:nisnaming:le-shop
```

The JDBC URL mechanism is intended to provide a framework so that different drivers can use various naming systems that are appropriate to their needs. Each driver only has to understand one URL naming syntax to enable it to reject other types of URLs, thus providing another layer of security.


Sun Educational Services

Opening a Database Connection

- Obtain a `java.sql.Connection` object:

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

- JDBC driver management layer tries to locate driver
- System returns `java.sql.Connection` object upon success

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.

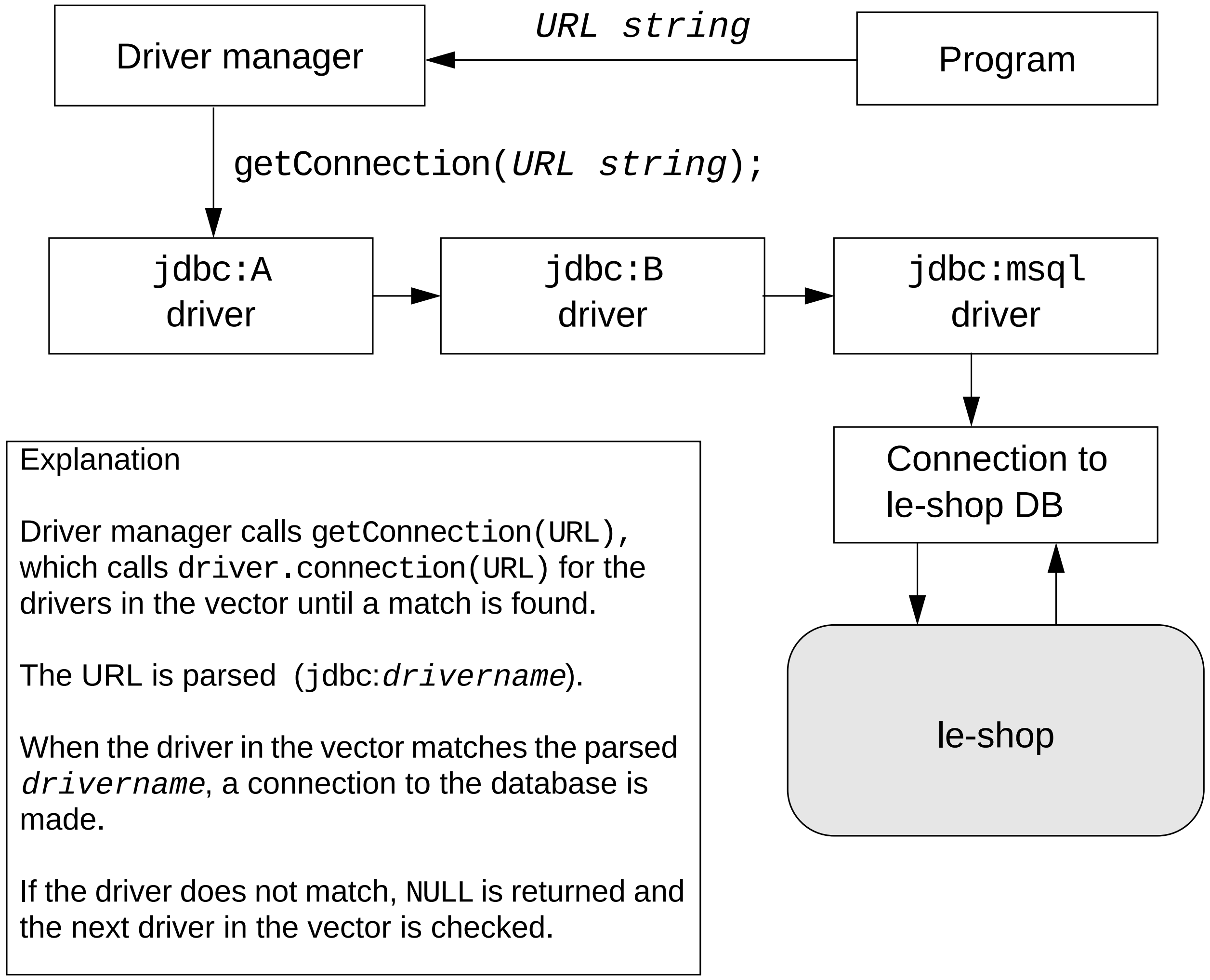
To do this, obtain a `java.sql.Connection` object by calling the JDBC driver's `java.sql.DriverManager.getConnection` method.

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

The following describes the process:

- The driver manager calls the `Driver.getConnection` method of every driver registered, passing the URL string as the parameter.
- If the driver "knows" the subprotocol name, then the driver returns an instance of a `Connection` object; otherwise it returns null.

Figure 2-3 illustrates how a driver manager resolves a URL string passed in the `getConnection` method. 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.



Driver manager calls `getConnection(URL)`, which calls `driver.connection(URL)` for the drivers in the vector until a match is found.

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

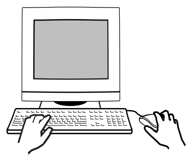
When the driver in the vector matches the parsed *`drivername`*, a connection to the database is made.

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

Figure 2-3 Example of Database Resolution

✓ ***Note that the driver does not have a "connection" with the database—this is the job of the*** `Connection` ***interface implementation.***

Exercise: Connecting to a Database



Exercise objective – Learn how to connect to a given database and how to gather meta data.

Tasks

Write a Simple Program to Connect to a Database

Complete the following steps:

1. Change the directory to `labfiles/mod2-jdbc/lab.`
2. Write a simple class `FirstConnection`, which loads the mSQL driver and connects to the given mSQL database. Use `FirstConnection.java` as a template.
3. Use the `getMetaData` method and the `DatabaseMetaData` class to gather information about the connection. Print the database name and version number on the console.
4. Compile and run the program.

Study How the Tables Are Listed (Optional)

1. The last lines of `FirstConnection.java` shows how the tables in a database are listed. Study the code and get an idea of how the `ResultSet` class works.

✓ ***The `getTables` method uses a parameter to narrow the list of tables to certain types. mSQL does not support any of the possible parameters.***

Exercise Summary

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

✓ ***Manage the discussion here based on the time allowed for this module, which was given in the "About This Course" module. If you find you do not have time to spend on discussion, then just highlight the key concepts students should have learned from the lab exercise.***

- Experiences

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

- Interpretations

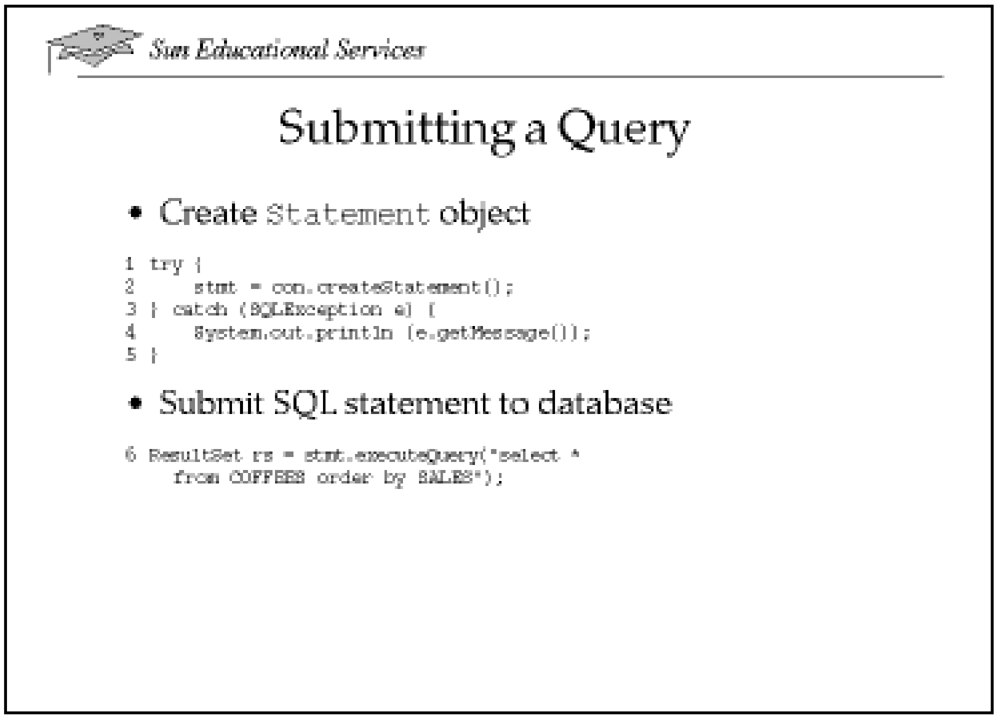
✓ ***Ask students to interpret what they observed during any aspects of this exercise.***

- Conclusions

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

- Applications

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



Sun Educational Services

Submitting a Query

- Create Statement object

```
try {
    stmt = con.createStatement();
} catch (SQLException e) {
    System.out.println (e.getMessage());
}
```

- Submit SQL statement to database

```
6 ResultSet rs = stmt.executeQuery("select *
    from COFFEES order by SALES");
```

Submitting a Query

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

```
try {
    stmt = con.createStatement();
} catch (SQLException e) {
    System.out.println (e.getMessage());
}
```

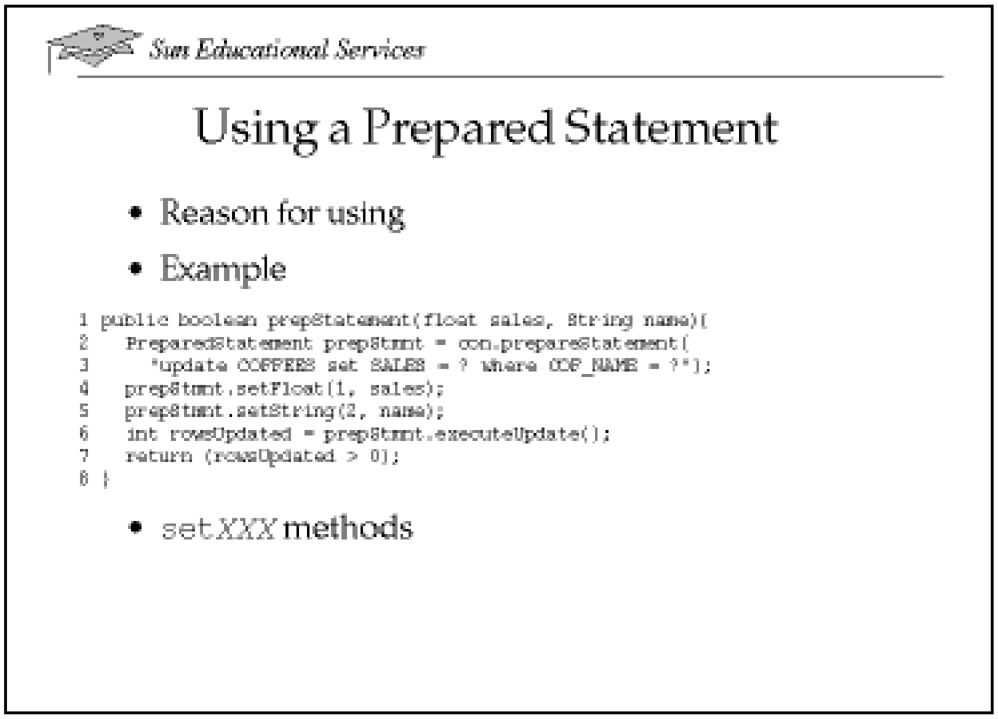
✓ ***SQL exceptions occur when there is a database access error; that is, a connection is broken or the database server went down. SQL exceptions provide information for debugging, such as***

- ***A string describing the error***
- ***An SQL state string following XopenSQL state conventions***
- ***A vendor-specific integer error code***

Use the `Statement.executeQuery` method to submit the SQL statement to the database. JDBC passes the SQL statement to the underlying database connection unaltered. JDBC does not attempt to interpret queries.

```
ResultSet rs = stmt.executeQuery("select *
from COFFEES order by SALES");
```

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



Sun Educational Services

Using a Prepared Statement

- Reason for using
- Example

```
public boolean prepStatement(float sales, String name){
  PreparedStatement prepStmt = con.prepareStatement(
    "update COFFEES set SALES = ? where COF_NAME = ?");
  prepStmt.setFloat(1, sales);
  prepStmt.setString(2, name);
  int rowsUpdated = prepStmt.executeUpdate();
  return (rowsUpdated > 0);
}
```

- `set`*`XXX`* methods

Using a Prepared Statement

If the same SQL statements are going to be executed multiple times, you should 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. An example of using a prepared statement is shown in the following code.

✓ ***Pre-compilation of multiple statements occurs at the database. The database (if it supports it) can take the entire command string and execute it against the database in a single execution and then return the results. The Imaginary JDBC driver for mSQL does not support pre-compiled statements, so each one is executed as an individual transaction.***

Note – Index numbers in JDBC start from 1, not 0.

```
public boolean prepStatement(float sales, String name){
   PreparedStatement prepStmnt = con.prepareStatement(
      "update COFFEES set SALES = ? where COF_NAME = ?");
   prepStmnt.setFloat(1, sales);
   prepStmnt.setString(2, name);
   int rowsUpdated = prepStmnt.executeUpdate();
   return rowsUpdated > 0;
}
```

The `set`*`XXX`* methods in Table 2-1 on page 2-34 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 `setInt`.

- ✓ `IN` ***parameters are passed by value into the operation; the value of the parameter is not expected to be changed.***
- ✓ `OUT` ***parameters are passed by reference from the operation; the operation is to set the value of the reference.***
- ✓ `INOUT` ***parameters are passed by reference into an operation; the value of the parameter passed into the operation is expected to be changed by the operation.***
- ✓ ***JDBC only supports*** `IN` ***and*** `OUT`, ***but JavaIDL (CORBA) supports all three.***

The `set`*`XXX`* *Methods*

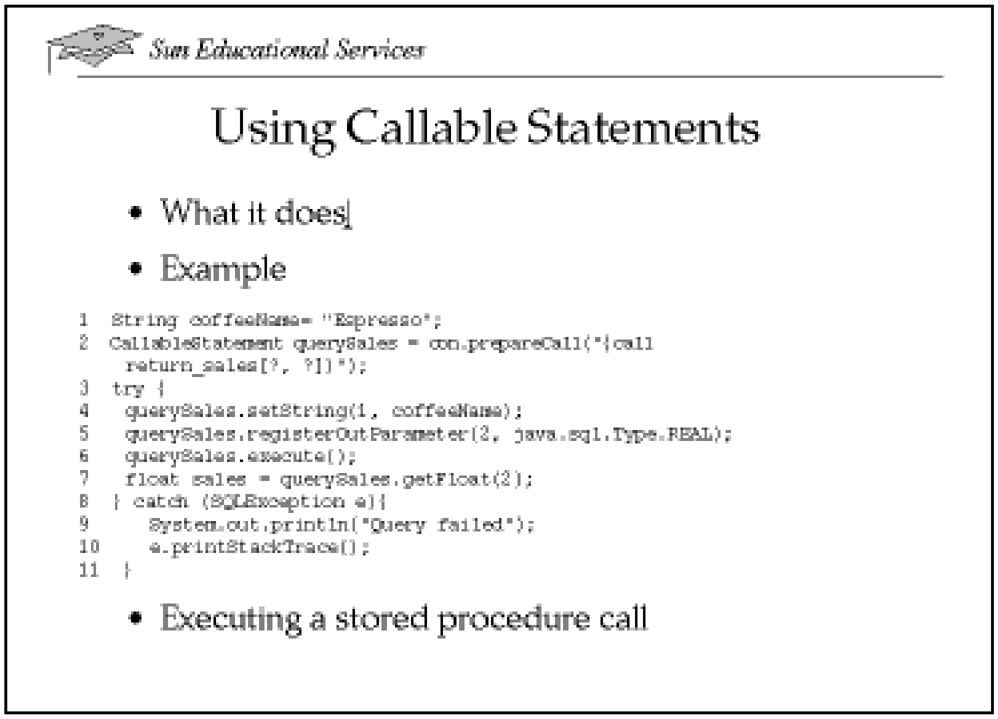
Table 2-1 contains the `setXXX` methods and SQL types.

Table 2-1 `set`*`XXX`* Methods and SQL Types

Method	SQL Type(s)
`setArrayLocator`	`LOCATOR(<array>)`
`setASCIIStream`	Uses an American Standard Code for Information Exchange (ASCII) stream to produce a `LONGVARCHAR`
`setBigDecimal`	`NUMERIC`
`setBinaryStream`	Uses a binary stream to produce a `LONGVARBINARY`
`setBlobLocator`	`LOCATOR(BLOB)`
`setBoolean`	`BIT`

Table 2-1 `set`*`XXX`* Methods and SQL Types

Method	**SQL Type(s)**
`setByte`	`TINYINT`
`setBytes`	`VARBINARY` or `LONGVARBINARY` (depending upon the size relative to the limits on `VARBINARY`)
`setCharacterStream`	Uses a `java.io.Reader` to produce a `LONGVARCHAR`
`setClobLocator`	`LOCATOR(CLOB)`
`setDate`	`DATE`
`setDouble`	`DOUBLE`
`setFloat`	`FLOAT`
`setInt`	`INTEGER`
`setLong`	`BIGINT`
`setNull`	`NULL`
`setObject`	The given Java technology object (“Java object”) is converted to the target SQL Type before being sent.
`setShort`	`SMALLINT`
`setString`	`VARCHAR` or `LONGVARCHAR` (depending upon the size relative to the driver’s limits on `VARCHAR`)
`setStructLocator`	`LOCATOR`(<structured-type>)
`setTime`	`TIME`
`setTimestamp`	`TIMESTAMP`
`setUnicodeStream`	Uses a Unicode stream to produce a `LONGVARCHAR`



Sun Educational Services

Using Callable Statements

- What it does
- Example

```
String coffeeName= "Espresso";
CallableStatement querySales = con.prepareCall("{call
  return_sales[?, ?]}");
try {
  querySales.setString(1, coffeeName);
  querySales.registerOutParameter(2, java.sql.Type.REAL);
  querySales.execute();
  float sales = querySales.getFloat(2);
} catch (SQLException e){
    System.out.println("Query failed");
    e.printStackTrace();
 }
```

- Executing a stored procedure call

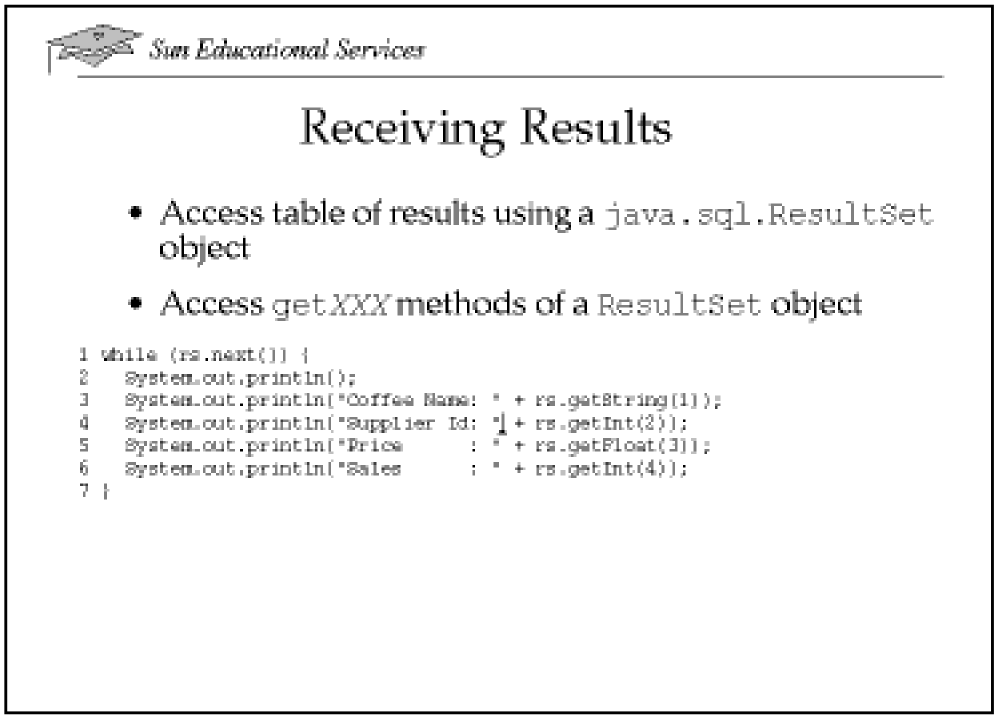
Using Callable Statements

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, methods for retrieving multiple results with a stored procedure are supported with the `Statement.getMoreResults` method.

For example, you could use a `CallableStatement` if you wanted to store a precompiled SQL statement to query a database containing the coffee inventory or suppliers.

```
String coffeeName= "Espresso";
CallableStatement querySales = con.prepareCall("{call
return_sales[?, ?]}");
try {
   querySales.setString(1, coffeeName);
   querySales.registerOutParameter(2, java.sql.Type.REAL);
   querySales.execute();
   float sales = querySales.getFloat(2);
} catch (SQLException e){
        System.out.println("Query failed");
        e.printStackTrace();
}
```

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



Sun Educational Services

Receiving Results

- Access table of results using a `java.sql.ResultSet` object
- Access `get`*`XXX`* methods of a `ResultSet` object

```
while (rs.next()) {
  System.out.println();
  System.out.println("Coffee Name: " + rs.getString(1));
  System.out.println("Supplier Id: " + rs.getInt(2));
  System.out.println("Price      : " + rs.getFloat(3));
  System.out.println("Sales      : " + rs.getInt(4));
}
```

Receiving Results

The result of executing a statement can be a table of data that is accessible using a `java.sql.ResultSet` object. The table consists of a series of rows and columns. The rows are received in order. A `ResultSet` keeps a cursor pointing to the current row of data and is initially positioned before its first row. The first call to `next` makes the first row the current row, the second call makes the second row the current row, and so forth.

The `ResultSet` object provides a set of `get` methods that enable access to the various column values of the current row. These values can be retrieved using either a column name or an index. You should use an index when referencing a column. Column indexes start at 1. When using a name to reference a column, it is possible that more than one column will have the same name, thus causing a conflict.

Note – The name used to reference a column is case sensitive.

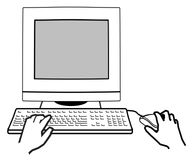
```
while (rs.next()) {
   System.out.println();
   System.out.println("Coffee Name: " + rs.getString(1));
   System.out.println("Supplier Id: " + rs.getInt(2));
   System.out.println("Price      : " + rs.getFloat(3));
   System.out.println("Sales      : " + rs.getInt(4));
}
```

The various `get`*`XXX`* methods access columns within the result sets table. The columns can be accessed in a random order, within the specified row.

To retrieve data from the `ResultSet` object, you must be familiar with the columns returned and their data types. A table mapping Java technology types (“Java types”) to SQL data types is provided in Table 2-3 on page 2-43.

Note – Once you have read the `ResultSet` object, the results are cleared; you can read the results only once.

Exercise: Executing an SQL Query



Exercise objective – Learn how to send SQL queries to the database and how to process the results.

Tasks

Write a Program to Send a SQL Query to a Database

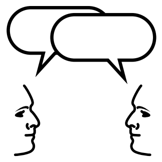
Using the template `Exec.java`, complete the `run` method. The query is stored in `queryString`, send it to the database and display the result on the console. Complete the following steps:

1. Change the directory to `labfiles/mod2-jdbc/lab`.
2. Get a `Statement` object from the connection.
3. Send the query stored in `queryString` to the database. Store the results in a `ResultSet` object.
4. Process the `ResultSet` and display the result on the console. (`getString(...)` can be used for every member in the column.)
5. Compile and run the program. Possible queries are:
   - `select * from COFFEES`
   - `select * from SUPPLIERS`
   - `select COF_NAME from COFFEES where SUP_ID=101`

Experiment Further (Optional)

1. Use metadata to add the column's label to the output.
2. Use SQL queries that do not return `ResultSet` objects, such as:
   - `insert into COFFEES values('My Coffee',150,5.55,0)`
3. Did you use `executeQuery` or `executeUpdate` to execute the query? Why?

Exercise Summary



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

✓ ***Manage the discussion here based on the time allowed for this module, which was given in the "About This Course" module. If you find you do not have time to spend on discussion, then just highlight the key concepts students should have learned from the lab exercise.***

- Experiences

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

- Interpretations

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

- Conclusions

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

- Applications

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

Using JDBC Statements

Table 2-2 contains the getXXX methods and the returned Java type.

Table 2-2 get*XXX* Methods and the Java Type Returned

Method	Java Type Returned
getArrayLocator	LOCATOR(<array>)
getASCIIStream	java.io.InputStream
getBigDecimal	java.math.BigDecimal
getBinaryStream	java.io.InputStream
getBlobLocator	LOCATOR(BLOB)
getBoolean	boolean
getByte	byte
getBytes	byte[]
getCharacterStream	java.io.Reader
getClobLocator	LOCATOR(CLOB)
getDate	java.sql.Date
getDouble	double
getFloat	float
getInt	int
getLong	long
getObject	Object
getShort	short
getString	java.lang.String
getStructLocator	LOCATOR(<structured-type>)
getTime	java.sql.Time
getTimestamp	java.sql.Timestamp
getUnicodeStream	java.io.InputStream or Unicode characters

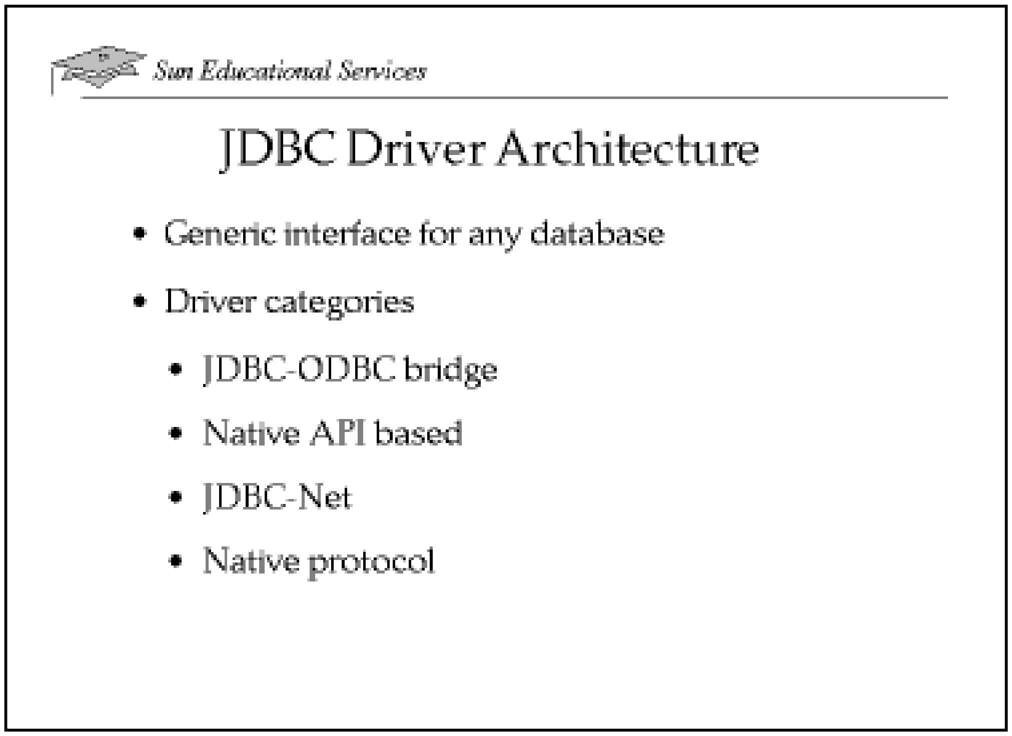
✓ ***Obviously,* getNull *makes no sense, but if the* get*XXX* *method encounters an SQL* NULL *type, the result of the method is null.***

Mapping SQL Types to Java Types

Table 2-3 shows the standard Java types for mapping various common SQL types.

Table 2-3 Mapping SQL Types to Java 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

JDBC Driver Architecture

- Generic interface for any database
- Driver categories
  - JDBC-ODBC bridge
  - Native API based
  - JDBC-Net
  - Native protocol

JDBC Driver Architecture

By allowing you to create a generic set of interfaces that you can use to connect to any database vendor's product, the JDBC API does not restrict you to a specific database. In addition, the JDBC API also does not restrict you in the application design, because you use either a two- or three-tier model to build your application.

JDBC Driver Categories

Today's JDBC drivers generally fit into one of four categories:

1. *JDBC-ODBC bridge plus ODBC driver.* A bridge provides JDBC access to the database using an existing ODBC driver. The ODBC driver uses native code, which must be installed on each database client. Therefore, this driver is most appropriate for corporate networks where client installations are not a major problem, or for an application server in a three-tier architecture.

2. *Native-API partly-Java technology driver ("Java driver").* This kind of driver is built on top of a native database client library (usually C or C++). The driver translates the JDBC calls into calls to the API of the database client library. The library is distributed by the provider of the database management system (DBMS). This driver is similar to the ODBC bridge, because it needs some native code installed on each database client.

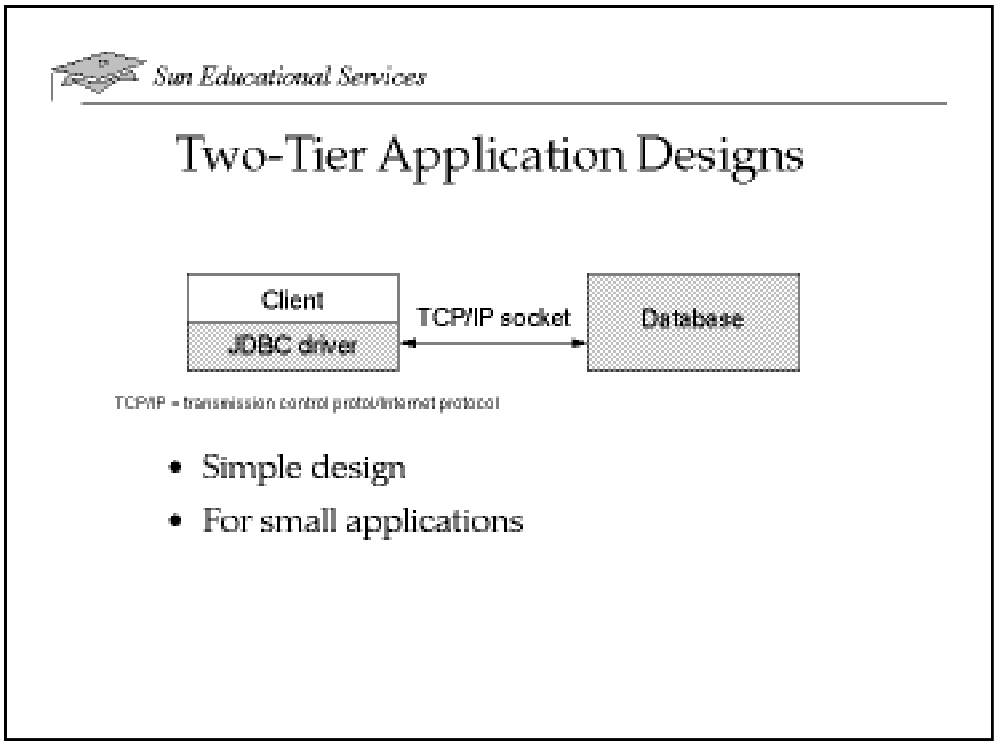
3. *JDBC-Net pure Java driver.* This kind of driver communicates with an intermediary server seated between the client and the database, using a network protocol specific to the intermediary. These calls are translated by the intermediary to DBMS-specific calls and forwarded to the database. No native code needs to be installed on the database client machines, because the driver can be implemented in pure Java technology. If the intermediary can talk to different DBMSs, you need only one JDBC driver.

4. *Native-protocol pure Java driver.* This kind of driver translates JDBC calls directly into the network protocol used by the DBMS. This allows direct calls from the client to the database. Because these protocols are proprietary, the database provider is the primary source for the driver. Several vendors have these in progress.

Drivers from categories 3 and 4 are the preferred way to access databases from JDBC, because they are pure Java technology. Drivers from categories 1 and 2 are considered as interim solutions until pure Java drivers are available. Table 2-4 shows the four categories and their properties.

Table 2-4 Driver Categories

Driver Category	Pure Java	Net Protocol
JDBC-ODBC Bridge	No	Direct
Native API based	No	Direct
JDBC-Net	Yes	Connector
Native Protocol	Yes	Direct



Sun Educational Services

Two-Tier Application Designs

Client
JDBC driver
TCP/IP socket
Database

TCP/IP = transmission control protol/internet protocol

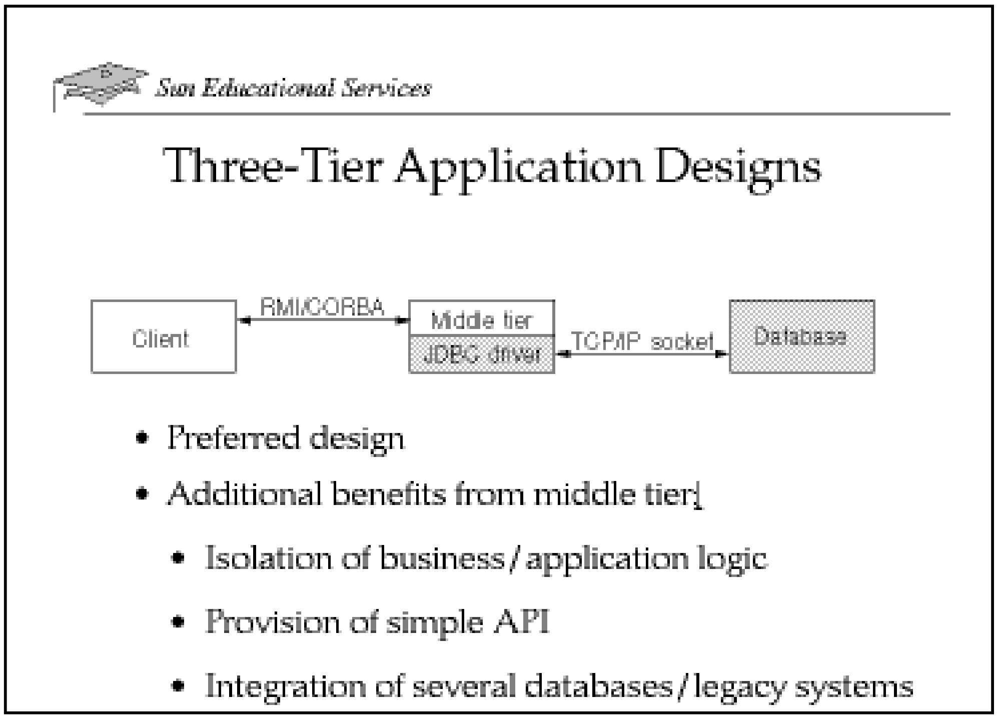
- Simple design
- For small applications

Application Designs

By creating a generic set of interfaces that can be used to connect to any database vendor's product, the JDBC API does not restrict you to a specific database. In addition, the JDBC API does not enforce a special application design. This section discusses two- and three-tier models as guidelines to database application design with the Java programming language and the JDBC program. An application consists of a database client, a database server, and some business or application logic.

Two-Tier Application Designs

An application designed after the two-tier model consists of a client as the top tier and a database server as the bottom tier. The client usually has a graphical user interface (GUI) to interact with the user and includes all the business or application logic. Furthermore, the client talks directly to the database using an appropriate JDBC driver. This design is straightforward and intuitive.


Sun Educational Services

Three-Tier Application Designs

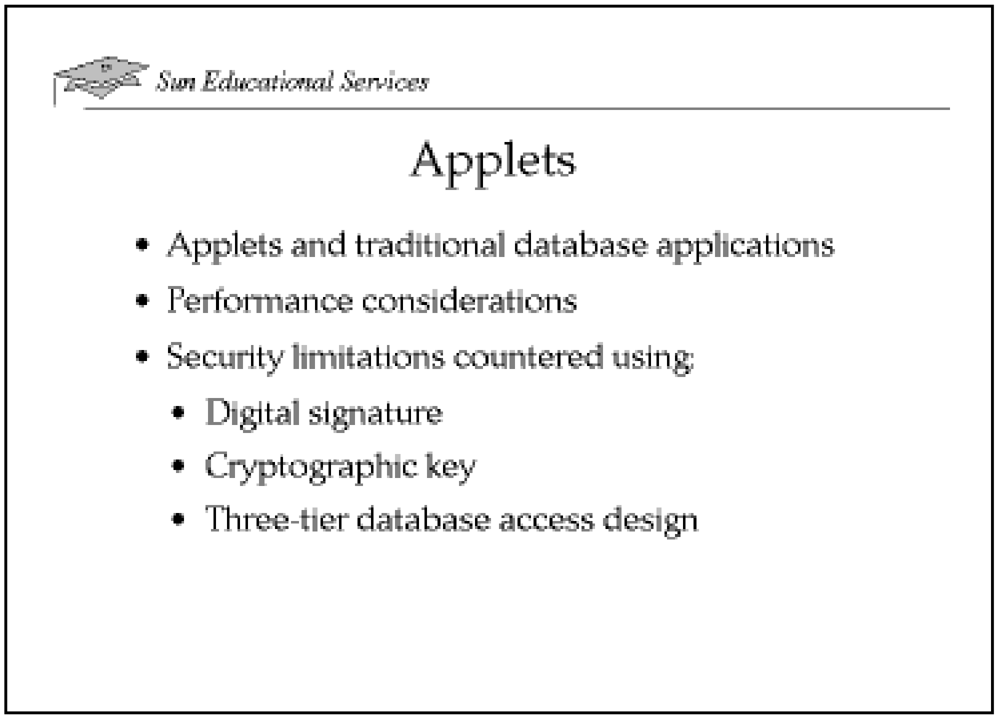
Client
RMI/CORBA
Middle tier
JDBC driver
TCP/IP socket
Database

- Preferred design
- Additional benefits from middle tier:
  - Isolation of business/application logic
  - Provision of simple API
  - Integration of several databases/legacy systems

Three-Tier Application Designs

An application designed after the three-tier model has a middle tier between the client and the database. Within this model, the client is much thinner and essentially consists of the GUI. Most of the business or application logic is moved to the middle tier. The middle tier offers a more abstract API to the client and they use any protocol, such as RMI, or CORBA to communicate. The middle tier uses JDBC to communicate with the database server, which plays the same role as in the two-tier model.

The three-tier model is the preferred model to design database applications. The added level of abstraction eases development and maintenance of the whole application. Furthermore, it is easier to integrate several databases or even legacy systems in an application.



Sun Educational Services

Applets

- Applets and traditional database applications
- Performance considerations
- Security limitations countered using:
  - Digital signature
  - Cryptographic key
  - Three-tier database access design

Applets

The use of Java applets has been the most publicized use of the Java platform. JDBC can be incorporated into applets to bring database access to the World Wide Web (WWW). For example, you can download a Java applet that can display the available flights for a specified date to and from specified destinations. This applet could access a relational database over the Internet, enabling a client to inquire about seat availability, book a reservation, or update the database.

You can also use applets in an intranet to provide access to company databases, like a corporate directory, where many divisions are working on different hardware platforms, but need a common database interface.

Applets and Traditional Database Applications

Applets differ from traditional database applications in a number of ways:

- Untrusted applets are severely constrained in the operations they are allowed to perform. Specifically, untrusted applets are prevented from having access to local files and the ability to open a network connection to any host other than the host that provided the applet.
- Applets running across the Internet cannot depend on a local database location or the database driver being in a `.INI` file or local registry on the client's machine, as in ODBC.

Performance Considerations

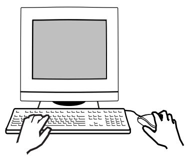
Performance considerations for database connectivity implementations differ when the database might be halfway around the world. The response time for an Internet-based database applet will be considerably slower than a database applet running on a local network.

Security Limitations

You can avoid some of the security limitations encountered with untrusted applets by using a digital signature or cryptographic key scenario. In these circumstances, an applet is treated like an application in the security sense, but there still would be problems interacting with client-side databases because of the difficulty locating the directory structure of the database or database driver.

As discussed earlier, a three-tier database access design can provide a middle tier of service on the network. The middle tier implementations can access databases on multiple networked hosts. These calls might be made through remote procedure call (RPC) or through an object request broker (ORB). In either case, the middle tier is best defined using an object paradigm; for example, customer objects with operations for customer invoicing, booking reservations, and other transactions.

Exercise: Building a JDBC Application (Optional)



Exercise objective – Combine all of the aspects shown earlier and use JDBC to build a larger application.

Preparation

LeShop is a small coffee shop that uses a database (le-shop) for its business. The example's focus is on JDBC, so there is a minimum of supporting functions, no GUI, and a simple two-tier design. It consists of five classes: `LeShop` is the main application. It uses `ConsoleMenu` and `ConsoleDialog` for user interaction on the console. The classes `Coffee` and `Supplier` interact with the database using JDBC. They both have a main method, so you can use them as standalone programs to test the code. Use `FirstConnection` and `Exec` from the previous exercises to verify that your code manipulates the database correctly.

Tasks

Set up the LeShop Tables

The classes `Coffee` and `Supplier` offer methods to create, initialize, and remove the necessary tables.

1. Change the directory to `labfiles/mod2-jdbc/lab.`
2. Complete the `insert(...)` method in `Coffee.java`. Compile and test it. Use `FirstConnection` or `Exec` from previous exercises to verify the results.

   Did you use `executeQuery` or `executeUpdate` to execute the query? Why?

✓ `ExecuteUpdate` ***would be the correct method to use, because neither insert nor drop table return a ResultSet. Of course, executeQuery works too.***

3. Complete the `removeTables` method in `Coffee.java`. Use `Exec` to test your SQL queries prior to coding them.

4. Compile and test the `removeTables` method in `Coffee.java`. Use `FirstConnection` or `Exec` from previous exercises to verify the results.

5. Repeat steps 2–5 using the `Supplier` class. (Optional – If the `Supplier` class is left out, copy the `Supplier1.java` from the `solutions` directory and rename it to `Supplier.java`.)

6. Compile `LeShop.java` and test the functioning parts.

Complete Sales Total

Complete the `getSalesTotal` methods in `Supplier` and `Coffee` and run `LeShop` again. The application is almost complete now.

1. Complete `getSalesTotal` in `Coffee.java`.

2. Compile `Coffee.java` and run `LeShop` to test the code.

3. Repeat steps 7–8 for `Supplier.java`.

Delete Single Entries (Optional)

Implement the `delete(...)` methods from `Coffee` and `Supplier`. The application will be complete afterwards.

1. Complete the `delete(...)` method in `Coffee.java`. You can use `Exec` to test your SQL queries prior to coding them.

2. Compile `Coffee.java` and run `LeShop` to test the code.

3. Repeat steps 10–11 for `Supplier.java`.

Tasks

Have Further Discussion (Optional)

The `initTable` method from `Supplier` and `Coffee` could be much better designed using a `PreparedStatement`. What would the code look like?

Beside the GUI, would it be difficult to turn the `LeShop` application into an applet?

Exercise Summary

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

✓ ***Manage the discussion here based on the time allowed for this module, which was given in the "About This Course" module. If you find you do not have time to spend on discussion, then just highlight the key concepts students should have learned from the lab exercise.***

- Experiences

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

- Interpretations

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

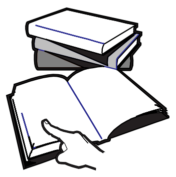
- Conclusions

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

- Applications

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

More Information



For more information on JDBC, refer to the latest revision of the JDBC specification. This specification is available at the `http://java.sun.com/products/jdbc` web site. It specifies the interface both from the points of view of the application programmer and the writer of the vendor driver.

Check Your Progress

Before continuing on to the next module, check that you were able to accomplish the following in this module:

- ❑ Explain JDBC
- ❑ Describe the five major tasks involved in the JDBC programmer's interface
- ❑ State the requirements of a JDBC driver and its relationship to the JDBC driver manager
- ❑ Explain how to map database types into the Java programming language
- ❑ Describe how to use JDBC with a Java applet
- ❑ Compare and contrast two-tier and three-tier designs for JDBC drivers
- ❑ Create a JDBC application to solve a defined problem

Think Beyond

To what applications will you add JDBC?

Remote Method Invocation (RMI) 3

Objectives

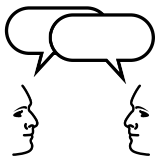
Upon completion of this module, you should be able to:

- Describe the RMI architecture, its layers and garbage collection
- Implement an RMI server and client in the Java programming language
- Generate client stubs and skeletons for remote services using the RMI stub compiler
- Define the RMI registry and describe how it works
- Explain the security issues related to RMI

The Java Remote Method Invocation (RMI) is a simple and powerful framework for distributed object computing that extends the pure Java object model to the network.

Relevance

✓ ***Present the following questions to stimulate the students and get them thinking about the issues and topics presented in this module. They are not expected to know the answers to these questions. The answers to these questions should be of interest to the students, and inspire them to learn the content presented in this module.***



Discussion – Consider the following questions:

- How many of the programs that you write could take advantage of existing programs/classes that you have already written in the Java programming language, but need to run on another machine elsewhere on the network?

✓ ***The RMI API may be viewed as a communication mechanism that takes care of TCP socket connectivity and putting objects "on the wire" (using the Java programming language's Object Serialization) between separate JVM invocations. The developer is removed one level by not having to write that functionality into the program each time.***

- As you survey the distributed programming technologies available to you, how important is it that the chosen technology supports the ability to pass Java objects "across the wire" as method parameters and/or return values?

✓ ***It should be*** *very* ***important to anyone who has taken*** *Java Programming Workshop* ***(SL-285), or anyone who had to pass complex data structures from one machine to another before the Java programming language supported object serialization. At that time, you had to write your own protocol to pass the data states across the wire, and a protocol handler to receive the data and instantiate a new object of a particular type, once the data has been received.***

✓ ***Keep in mind that CORBA does not allow objects to be passed across the wire, just primitive types, predefined composite types, and CORBA references.***

Relevance

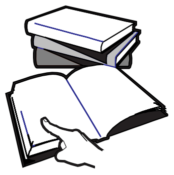
- How important is it that the distributed technology that you implement supports the ability to pass complex data types, for example, a variable-size `Vector` of Java objects? How about the capability to pass or return a subclass or subtype of the declared parameter or return value?

✓ ***Again, this is an inherent advantage to using RMI over CORBA, because CORBA returns must be strictly predefined in size and type, and are limited to primitives. The subclass, though, is incredibly powerful, in that you could declare your method to accept a*** `Command`[1] ***object. You could then subclass*** `Command` ***into*** `Deposit` ***and*** `Withdraw`***, and RMI would allow you to pass either*** `Deposit` ***or*** `Withdraw` ***to your method. CORBA does not allow you the flexibility of passing subtypes.***

✓ ***Once you understand this, imagine that the*** `Command` ***object implements*** `runnable`***, so from your method, you can invoke the*** `run` ***method of the object that you have received. Its behavior will be different, depending on whether it is a*** `Deposit` ***or a*** `Withdraw` ***object. So you are passing*** *`behavior`* ***rather than just data, which is the process of implementing an agent.***

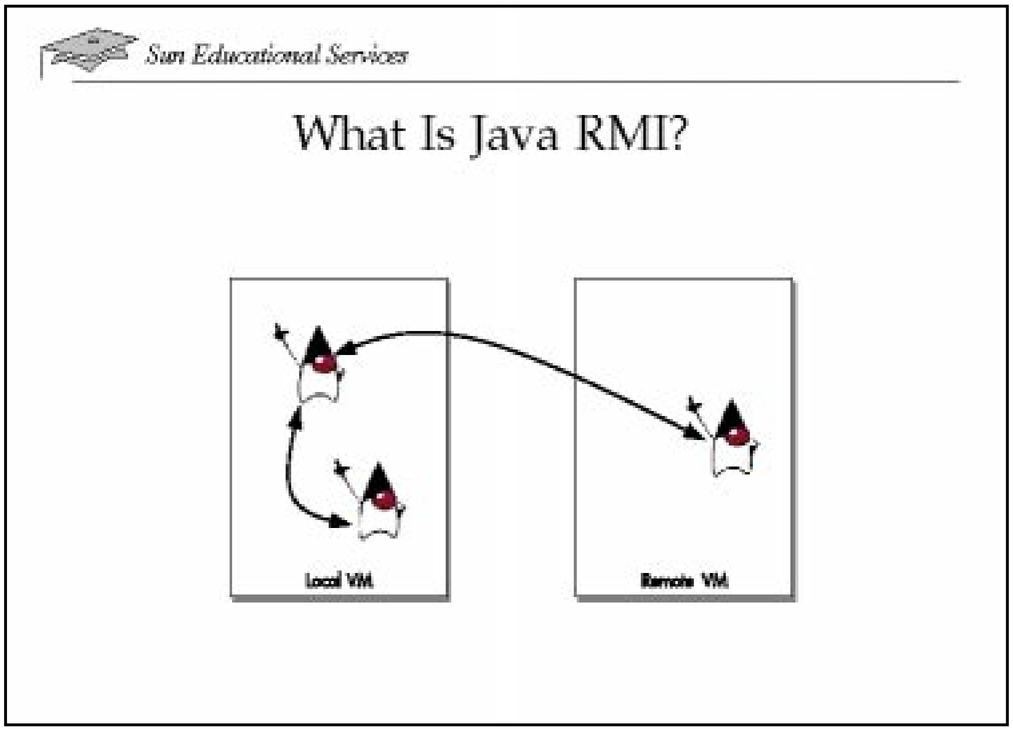
1. See "Command pattern," pp. 233–242 of *Design Patterns – Elements of Reusable Object-Oriented Software* by Gamma, Helm, Johnson, and Vlissedes (Addison-Wesley, 1995).

Additional Resources

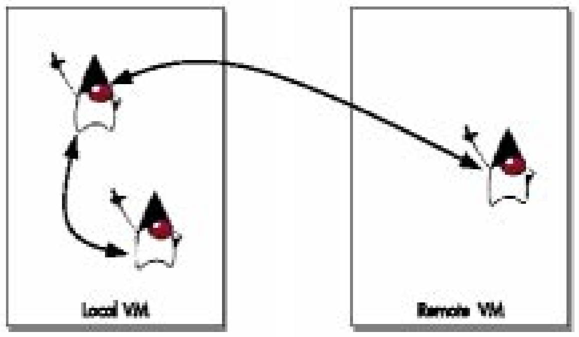


Additional resources – The following resources can provide additional details on the topics presented in this module.

- The Java Remote Method Invocation Specification. [Online]. Available: `http://java.sun.com/products/jdk/1.2/docs/guide/rmi/spec/rmiTOC.doc.html`
- Getting started using RMI. [Online]. Available: `http://java.sun.com/products/jdk/1.2/docs/guide/rmi/getstart.doc.html`
- Frequently Asked Questions, RMI and Object Serialization. [Online]. Available: `http://java.sun.com/products/jdk/rmi/faq.html`
- Java Remote Method Invocation – Distributed Computing for Java. [Online]. Available: `http://java.sun.com/marketing/collateral/javarmi.html`
- Object Serialization. [Online]. Available: `http://java.sun.com/products/jdk/rmi/serial/index.html`


Sun Educational Services

What Is Java RMI?



What Is Java RMI?

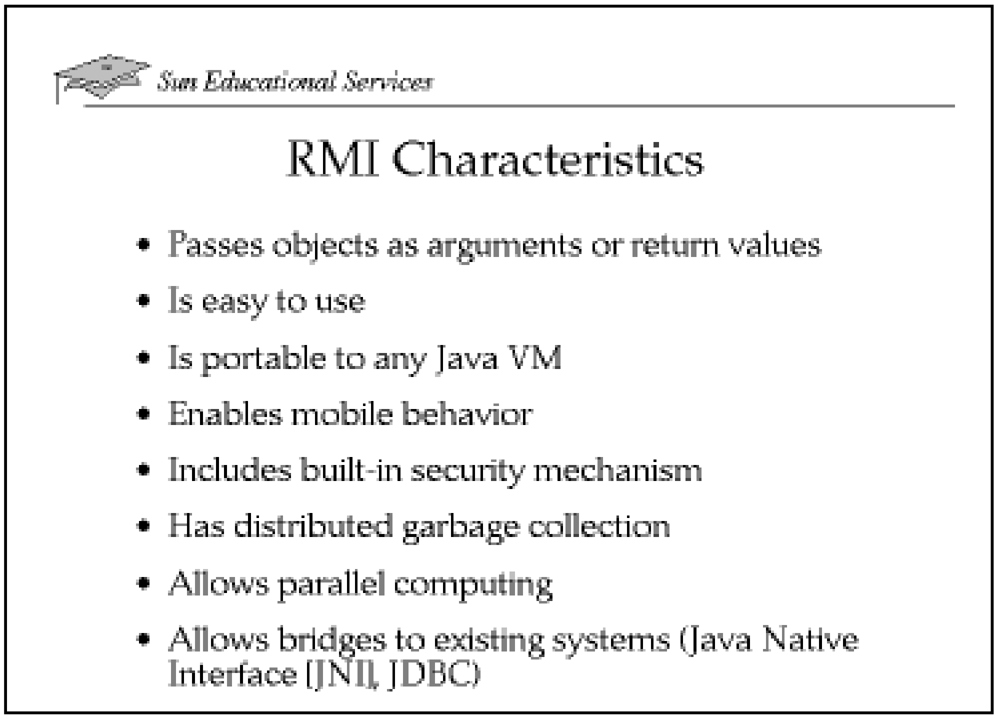
Java Remote Method Invocation (RMI) is the method preferred by Sun Microsystems to write distributed objects using the Java environment. RMI extends the Java slogan “Write Once, Run Anywhere”™ with “Run Everywhere.” The distributed object model is in fact seamlessly integrated into the Java programming language.

Until the release of the Java Remote Method Invocation API, sockets were the only facility built into the Java programming language that provided direct communication between machines. In a fashion similar to Remote Procedure Calls (RPC), RMI abstracts the socket connectivity and data streaming, enabling developers to write code that accesses remote objects in the same fashion as objects instantiated from within the local runtime system.

RMI supports traditional distribution designs, such as client/server or peer-to-peer, and enriches these models with new kinds of agent-based distributed applications that move behavior between clients and servers.

RMI enables calls to remote objects that exist in the runtime of a different Java Virtual Machine (JVM) invocation. This “remote” or “server” JVM can be executing on the same machine or on an entirely different machine from the RMI “client.” A Java program can make a call on a remote object once it obtains a reference to the object, either by looking it up in the registry (a simple naming facility provided by RMI) or by receiving the reference as an argument or return value.

Unlike CORBA, your application must be written entirely in the Java programming language and is therefore a pure Java-to-Java mechanism providing a consistent environment for distributed applications. RMI is in fact an extension to the language itself, and was designed to ease the design and implementation of distributed applications, at the price of staying within the Java environment. No separate interface definition language (IDL) or special language-neutral environment is used. Everything has to be written in the Java programming language.


Sun Educational Services

RMI Characteristics

- Passes objects as arguments or return values
- Is easy to use
- Is portable to any Java VM
- Enables mobile behavior
- Includes built-in security mechanism
- Has distributed garbage collection
- Allows parallel computing
- Allows bridges to existing systems (Java Native Interface [JNI], JDBC)

RMI Characteristics

The characteristics of RMI include the following:

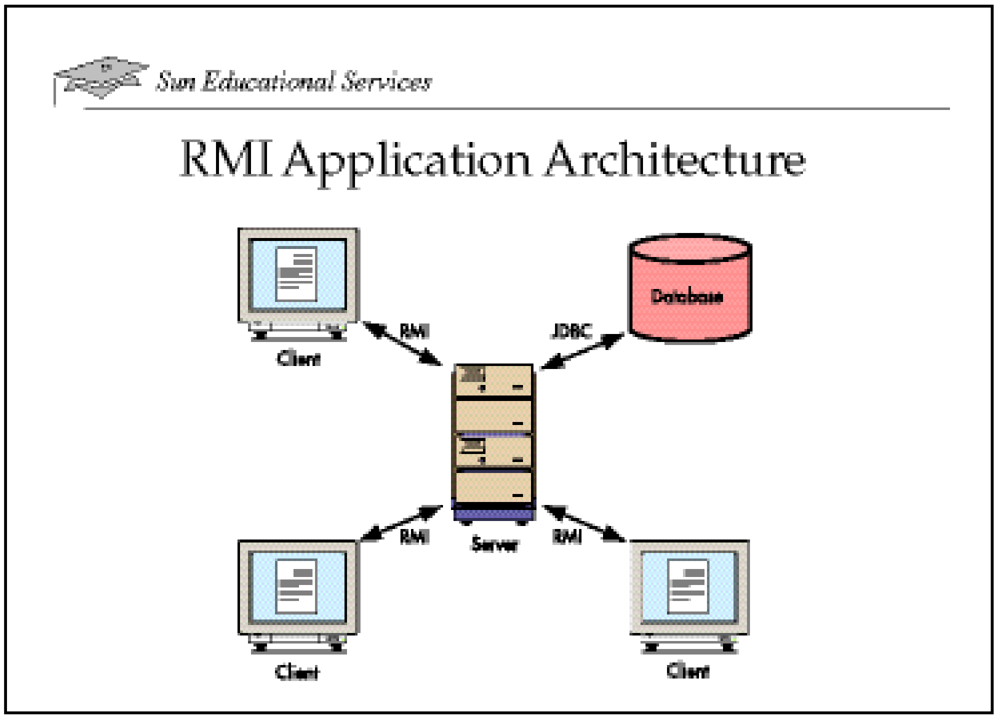
- RMI is object-oriented. It passes objects as arguments or return values and not some predefined data types. It allows you to handle even the most complex objects as a single argument. No composing or decomposing into primitive data types is needed.

- RMI is a simple but powerful framework. It allows you to build the core part of server and client applications with just a few lines of code.

- RMI benefits directly from the platform independence of the Java programming language. Any RMI-based application is portable to any Java virtual machine.

- RMI allows you to move behavior, such as agents or business logic code to the part of your network where it makes the most sense for your application. With the ability to pass behavior comes the ability to directly implement object-oriented design patterns that rely on different behavior schemes.

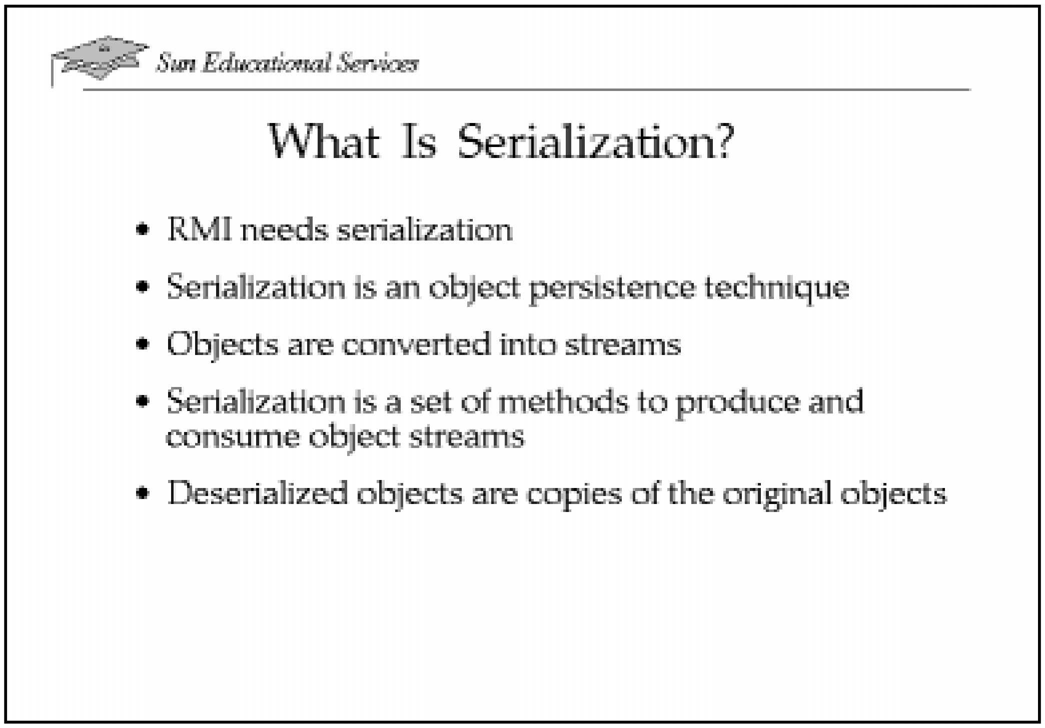
- The Java programming language provides built-in security mechanisms that are also used by RMI, such as a security manager protecting the system from hostile code.
- RMI has built-in distributed garbage collection that removes remote server objects that are no longer referenced by any clients in the network.
- RMI is inherently multithreaded and therefore allows you to create sophisticated servers using concurrent threads to process client requests.
- RMI also allows a connection to existing server systems using the Java native interface (JNI) or to standard relational databases with the JDBC classes.



RMI Application Architecture

✓ ***Do not spend too much time on this overhead. Everything on it will be covered in detail later on. The purpose of this overhead is to give an overview, which should not take more than five minutes.***

RMI allows you to move behavior from server to client and from client to server. Imagine an application for an insurance company allowing you to calculate some installments. The interface for this application certainly includes some methods that check whether the entered figures conform to values that are set by the company rules. To fill out an insurance form, clients get an object from the server that implements these rules. When these rules change, clients get the updated implementation the next time they request the insurance form object. The clients' interface does not need to be touched at all. The checking takes place at the client side with the advantage of immediate feedback to the user and reduced load on the server. You can move behavior in both directions: imagine, in the design in the overhead, a server handling asynchronous computation-intensive requests from some of the client applications.



Sun Educational Services

What Is Serialization?

- RMI needs serialization
- Serialization is an object persistence technique
- Objects are converted into streams
- Serialization is a set of methods to produce and consume object streams
- Deserialized objects are copies of the original objects

What Is Serialization?

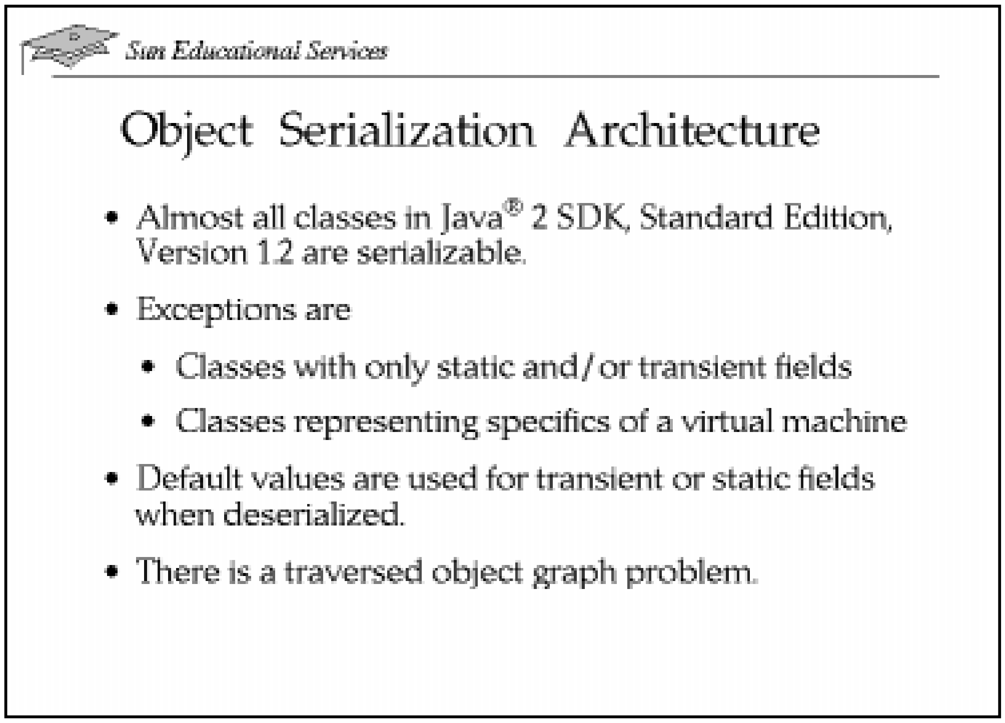
Serialization allows you to save and restore the state of an object. Without serialization, parameters and return values would have to be simple primitive data types. The RMI architecture would not work as seamlessly as it does without mechanisms for sending objects over the network. This would limit drastically the usefulness of a distributed design, and make the power of remote objects applicable only to trivial applications.

Almost every application requires some way of keeping data. Most applications use a database for the storage or persistence of data. However, databases are not typically used to store objects, particularly Java objects. A process is required to keep the state of a Java object in such a way that the object can be easily stored and retrieved, and returned to its original state.

Java object serialization is a lightweight object persistence technique that allows you to convert objects into streams. The Java object serialization framework consists of a set of methods that allow you to duplicate an object by converting its field values into a stream that can be easily reconverted to its original object representation by the receiver of the stream.

The deserialized object is a copy of the original object and not the original itself. To send an object reference to another object and not a copy of the original, that is, pass the object by reference rather than by value, use a `Remote` object, which means that the respective class must implement the `Remote` interface.

Objects that act as containers to be serialized implement the interface `Serializable` that allows the contents of the object to be saved or restored in a single stream.



Sun Educational Services

Object Serialization Architecture

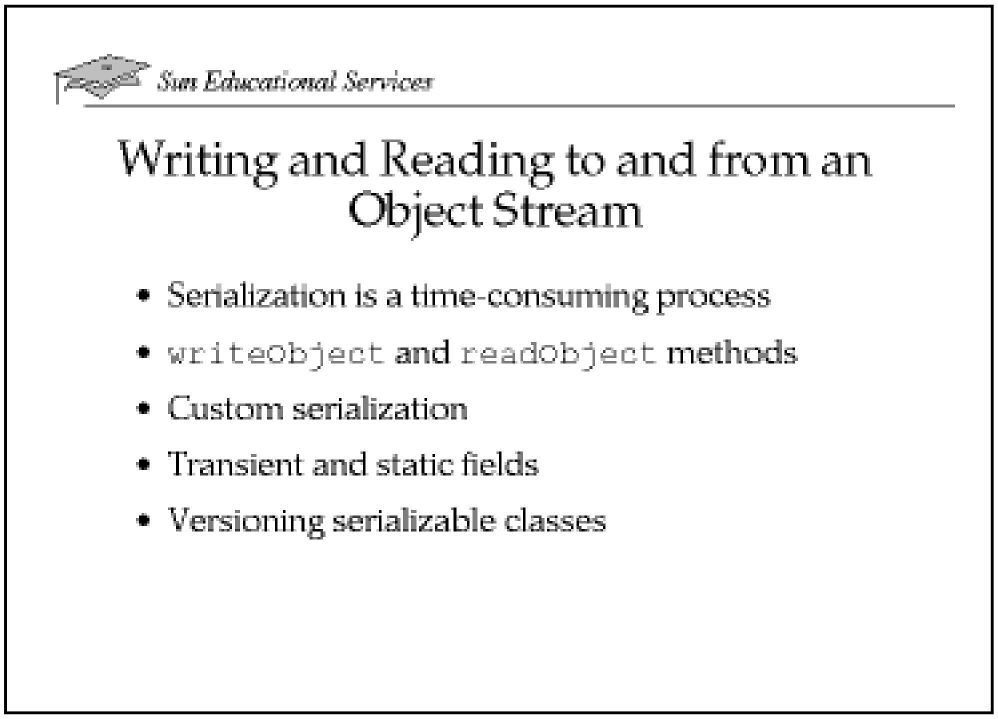
- Almost all classes in Java® 2 SDK, Standard Edition, Version 1.2 are serializable.
- Exceptions are
  - Classes with only static and/or transient fields
  - Classes representing specifics of a virtual machine
- Default values are used for transient or static fields when deserialized.
- There is a traversed object graph problem.

Object Serialization Architecture

You can serialize almost all classes in the Java 2 SDK. The exceptions are classes containing only static or transient fields, or both fields and classes that represent specific characteristics of the local virtual machine. There is no reason for these classes to be sent to other objects.

Classes that are made up of fields other than static or transient fields, you can serialize, but the default mechanism does not make values of the static or transient fields persistent. When you serialize the object, the default values for the transient and static fields are set as if the object would have been instantiated from scratch. For most RMI applications, the default built-in serialization is all you need.

Object streams do not transmit an object's bytecode; that is, the compiled class files, instead, they transmit a representation of its structure and related data.


Sun Educational Services

Writing and Reading to and from an Object Stream

- Serialization is a time-consuming process
- `writeObject` and `readObject` methods
- Custom serialization
- Transient and static fields
- Versioning serializable classes

Writing and Reading to and from an Object Stream

Writing to an Object Stream

Writing and reading an object to and from a file stream is a simple process. For example, the following is a simple `Point` class:

```
class Point implements java.io.Serializable {
   int x;
   int y;
   Point( int x, int y){
      this.x = x;
      this.y = y;
   }
}
```

The class Point implements the Serializable interface and can therefore be written to an object stream with code that looks like the following:

```
Point myPoint = new Point(1,2);
FileOutputStream fos = new FileOutputStream(myfile);
ObjectOutputStream oos = new ObjectOutputStream(fos);
oos.writeObject(myPoint);
oos.close();
```

You must add some exception handling, and the upper code in a try-catch clause must be included as usual when dealing with input/output (I/O) methods. The constructor for ObjectOutputStream expects an OutputStream object. FileOutputStream is a valid subclass allowing you to write the object into a file stream.

Reading From an Object Stream

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 before methods on that class can be executed.

```
Point serialPoint;
FileInputStream fis = new FileInputStream(myFile);
ObjectInputStream ois = new ObjectInputStream(fis);
serialPoint = (Point)ois.readObject();
```

Time-Consuming Process

Serialization can be a time-consuming process. When an object is serialized, the system is building up a graph representing the object with all its contained references that can be traversed at the time of deserialization. Complex objects result in complex graphs that take time to construct.

Customizing `readObject` *and* `writeObject`

For special cases, the default serialization might not fulfill all needs. Then you must customize the serialization according to your requirements. You should be aware of the following rules:

- You must implement the interface `java.io.Serializable`.
- You must define a default constructor with no arguments.
- You must implement the `readObject` and the `writeObject` methods.

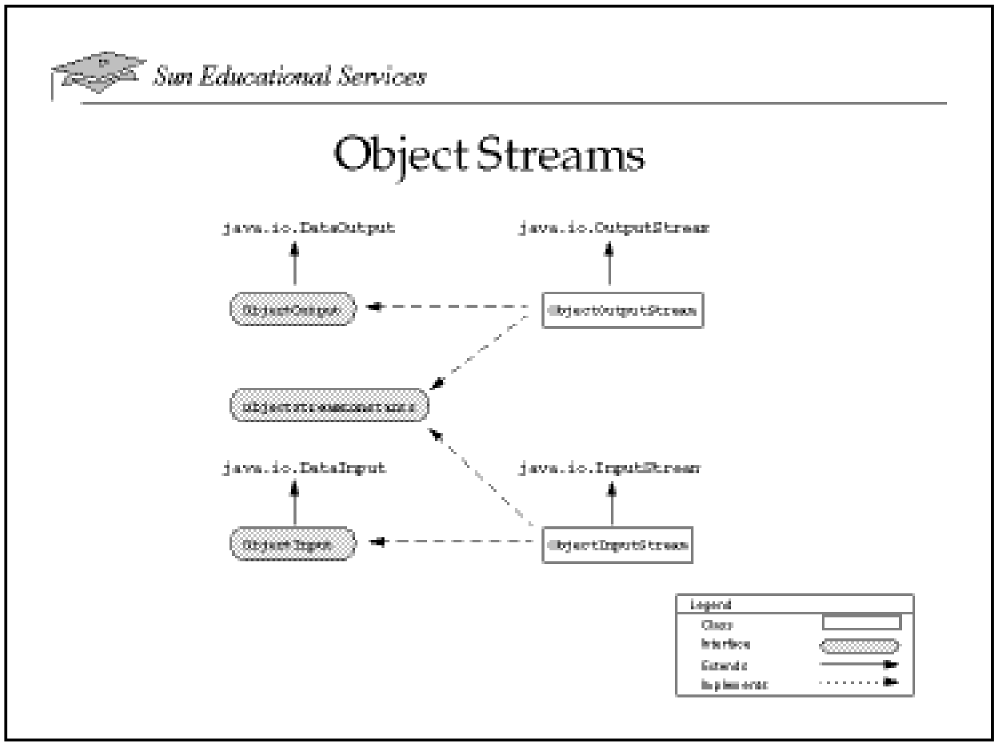
Customizing the default methods could, for example, consist of calculating values for special fields of the object. You can customize the serialization only when the object is deserialized.

Transient and Static Fields

You cannot serialize fields that are marked transient or static. This offers an easy way to prevent fields from being serialized. Changing a non-static field to static also changes the semantics of this class. The preferred way to protect a field from serialization without affecting its behavior is to mark it as `transient`. Providing the methods `writeObject` or `readObject` offers a way to set non-default values for transient fields after deserializing the respective object.

Versioning Serializable Classes

Serializable classes define a version ID that helps to find out which version of a class has been serialized and which version is expected for deserialization. You can use the `serialver` utility to determine the version ID. This utility comes with the Java 2 SDK, and returns the `serialVersionUID` for one or more classes.



Object Streams

Java object serialization produces just one stream format for the encoding and storing of objects. Each object acting as a container implements one of two interfaces: `ObjectOutput` or `ObjectInput.` They define some methods for the reading or writing of objects and primitives and some basic stream operations, such as `close` or `flush`.

All the work to be done with the serialization or deserialization of objects is taken care of by the stream objects that implement either the `ObjectOutput` or the `ObjectInput` interface.

`ObjectOutputStream` *Class*

The `ObjectOutputStream` implements the `ObjectOutput` interface, which itself extends the `DataOutput` interface to write primitives. The `ObjectOutput` provides an interface for object storage. The `writeObject` method is used to write objects to a stream. An object is checked first to see if it implements the `Serializable` interface before it is written to the stream using the `writeObject` method. Exceptions can occur while accessing the object or its fields, or when attempting to write to the storage stream.

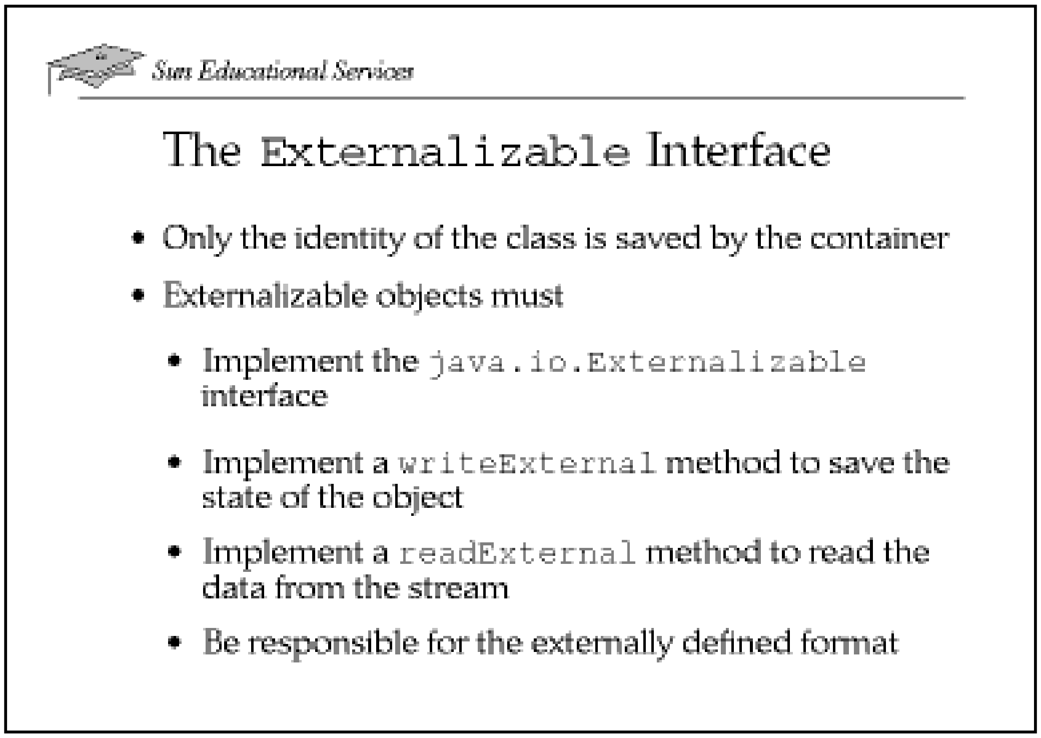
ObjectInputStream *Class*

The ObjectInputStream class is used for accessing objects written to an ObjectOutputStream, which implements the interface ObjectInput. This in turn extends DataInput to read primitives. The readObject method defined in the interface is used for object retrieval. It reads from the storage stream and returns an object. The readObject first gets an identification header from the stream. The next step is to read the bytecode and to create an instance with the default constructor before the readObject method fills in the object's value fields. Exceptions are thrown when attempting to read the storage stream, or if the class name of the serialized object cannot be found.

Serializable *Interface*

Any object that is serializable must implement the Serializable interface or implement a class that extends the Serializable interface. This interface is in fact empty and serves merely to flag objects that use serialization.

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. Again, only the data (and class name) of the referenced objects are stored. Methods are not written, because methods are assumed to be unchanged and can be accessed from the restored object state.



Sun Educational Services

The Externalizable Interface

- Only the identity of the class is saved by the container
- Externalizable objects must
  - Implement the `java.io.Externalizable` interface
  - Implement a `writeExternal` method to save the state of the object
  - Implement a `readExternal` method to read the data from the stream
  - Be responsible for the externally defined format

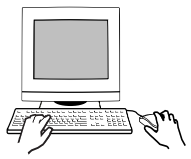
The `Externalizable` *Interface*

For classes that implement the `Externalizable` interface, the container saves only the identity of the class. In this case, the storage and retrieval of the object's state become the responsibility of the object itself.

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.
- Be responsible for the externally defined format. The `writeExternal` and `readExternal` methods are solely responsible for this format.

Exercise: Serialization



Exercise objective – Serialize and deserialize a class, and test the versioning mechanism of serialization.

Tasks

Complete Templates to Deserialize and Reserialize a Given Class

In this exercise you complete two templates: `Serialize.java` and `Deserialize.java`. These classes are used to serialize and deserialize the class `Name.java`, which already exists.

1. Change the directory to `labfiles/mod3-rmi/lab/serial.`

2. Modify the template `Serialize.java` to serialize the class `Name`, and write the output into a file stream. The class `Name` already exists in the directory. Compile your class (pay close attention to the package structure).

3. Start `Serialize` with the following command:

   ```
   java Serial.Serialize test.ser Firstname Lastname
   ```

   By doing this, you provide three String parameters: the name of the file with the serialized object (test.ser), a first name, and a second name.

4. Modify the template `Deserialize.java` so that it reads the serialized `Name` class from the file `test.ser` and displays the two strings on the console.

✓ ***The deserializing command should be*** `java Serial.Deserialize test.ser`***.***

Task

Find the Version of a Class and Test the Versioning Mechanism

Find the version of `Name.class`, by using the `serialver` utility provided with the Java 2 SDK.

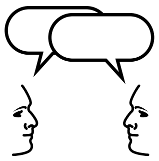
1. Show the version number of the class `Name.class` with the `serialver` utility.

✓ ***Remember the package. The correct command for step 5 is*** `serialver Serial.Name`***.***

2. (Optional) Modify `Name.java` in a way that changes its `serialver`. Then try to deserialize an "old" `test.ser` file.

3. (Optional) Modify your "new" `Name.java` by setting its `serialver` manually to the `serialver` of the "old" `Name.java`. (Use the output of the `serialver` utility from step 5.) Then try again to deserialize an "old" `test.ser` file.

Exercise Summary



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

✓ ***Manage the discussion here based on the time allowed for this module, which was given in the "About This Course" module. If you find you do not have time to spend on discussion, then just highlight the key concepts students should have learned from the lab exercise.***

- Experiences

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

- Interpretations

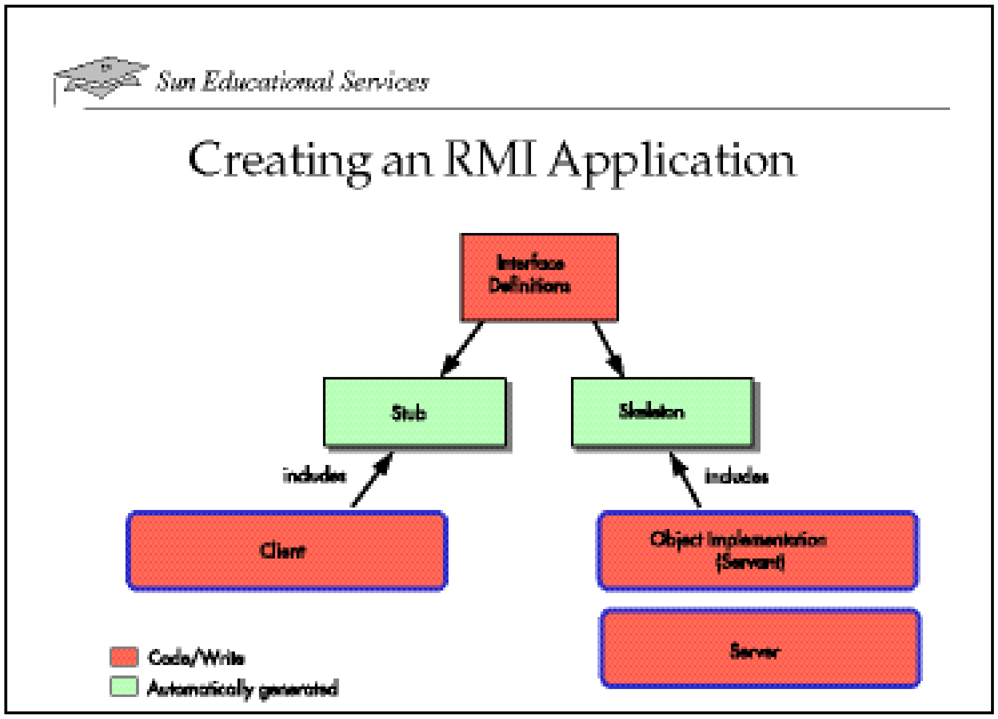
✓ ***Ask students to interpret what they observed during any aspects of this exercise.***

- Conclusions

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

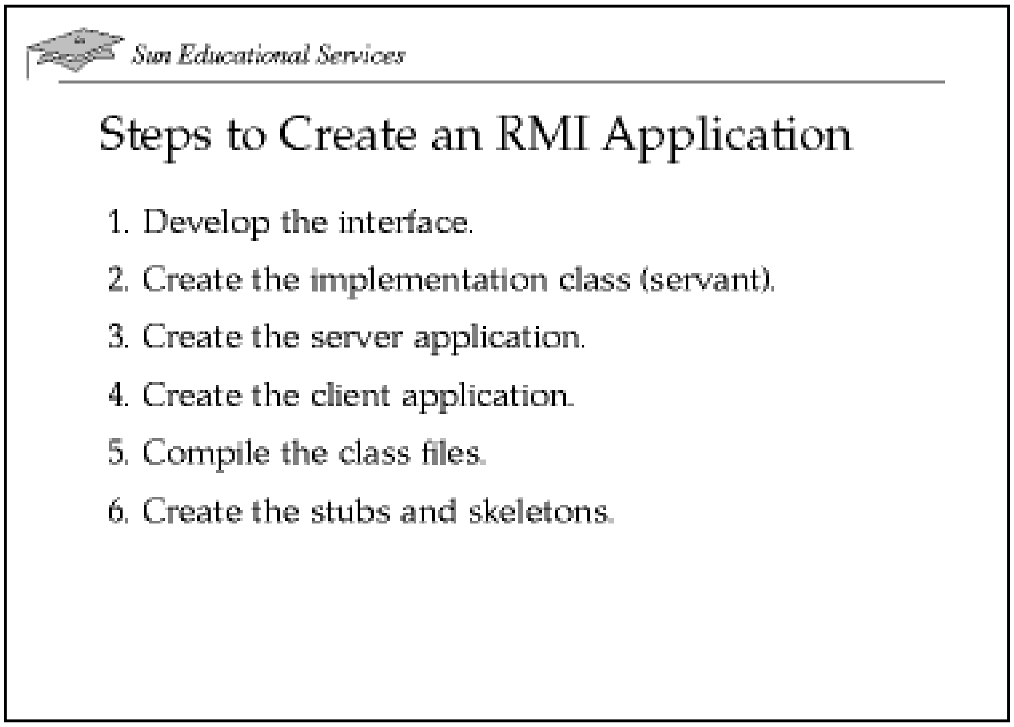
- Applications

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



Creating an RMI Application

The above overhead shows graphically the necessary steps to create an RMI application. The red (or dark) parts represent code that you must write, and the green (or light) parts represent code that is automatically generated with the RMI compiler utility.


Sun Educational Services

Steps to Create an RMI Application

1. Develop the interface.
2. Create the implementation class (servant).
3. Create the server application.
4. Create the client application.
5. Compile the class files.
6. Create the stubs and skeletons.

Steps to Create an RMI Application

There are six major steps involved in creating the RMI application:

1. Develop the Java technology interface (“Java interface”) that defines the remote object.
2. Create the implementation class (the *servant*) for the interface.
3. Create the *server*, which manages the *servant* instances.
4. Create the *client*, which uses the remote object (in the end, the *client* uses the *servant*).
5. Use the Java compiler to create the Java class files.
6. Run the RMI compiler (`rmic`) to create the helper classes (stubs and skeletons).

Step 1: Develop the Java Interface

The Java interface describes the server object. This interface has two differences, as compared to a "regular" Java interface: the `Echo` interface must extend `java.rmi.Remote`, and every method must throw `java.rmi.RemoteException`.

The following is an example of the interface describing an Echo object:

```
package EchoApp;
   import java.rmi.Remote;
   import java.rmi.RemoteException;

public interface Echo extends Remote {
      public String sayEcho(String myName)
      throws RemoteException;
}
```

Step 2: Create the Implementation Class (Servant)

For each interface, you must write a servant class. Instances of the servant class implement remote objects. Each instance implements a single remote object, and each remote object is implemented by a single servant. The servant must extend the class `java.rmi.server.UnicastRemoteObject`

The servant provides an implementation for all of the methods declared in the interface. In this case it is just one: `sayEcho`. In addition, if the class does not have a constructor, you must explicitly provide a modified default constructor. If you do not, the compiler does not "like" the class and issues an error message.

Despite the fact that the implementation classes are called *servants*, the usual naming convention is to append `Impl` to the Java interface name: `EchoImpl.java`.

```
package EchoApp;

import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class EchoImpl
   extends UnicastRemoteObject
   implements Echo {

   public EchoImpl() throws RemoteException {};

   public String sayEcho(String myName) {
      return "\nHello " + myName + "!!\n";
   }
}
```

✓ ***This*** `sayEcho` ***method does not throw*** `RemoteException`***, even if the declaration in the interface*** `Echo.java` ***says so. Why? First, the compiler chooses not to throw an exception, since doing this does not break the contract between the interface implementation class and the interface user class. Second, the servant never throws a*** `RemoteException` ***on its own, so why should there be a declaration which suggests that it will? The*** `RemoteException` ***is, in fact, generated and thrown by the client side stub if something goes wrong in one of the communication layers.***

Step 3: Create the Server

The RMI runtime uses the server to instantiate and publish the servant. In its simplest form, the server pre-instantiates all the required servants and keeps them alive forever (or at least as long as the server lives). In a production system this would usually not be good enough. To keep thousands of servants alive, even when they are not actively used by clients, is a waste of resources. In this case, more complex servers must be developed and used. They must be able to instantiate servants on demand, and maybe even persist and discard them after a period of non-usage. The server, `EchoServer.java`, is a simple server:

```
package EchoApp;

import java.rmi.Naming;

public class EchoServer {
   public static void main(String args[])
      throws Exception{

        // Create the servant
        // instance for registration
        EchoImpl echoRef = new EchoImpl();

        // Bind the object to the rmiregistry
        Naming.rebind("Echo", echoRef);

        System.out.println
        ("Echo object ready and bound
        to the name 'Echo'!");
    }
}
```

You have to instantiate the servant and register it with the name server (which is part of `rmiregistry`). After the servant is registered, its object reference can be retrieved and used by the client application. The following section describes how this works.

Step 4: Create the Client Application

The following describes how to create the client application.

```
package EchoApp;

import java.rmi.Naming;

public class EchoClient {
   public static void main(String args[])
      throws Exception {

      //Check the argument count
      if (args.length != 2) {
      System.err.println("Usage:
      EchoClient <server> <your name>");
      } else {
      //Get the Hello instance,
      //cast it to the Hello interface
      String url = new
      String("rmi://"+args[0]+"/Echo");
      Echo echoRef = (Echo)Naming.lookup(url);

      // call the Echo server object
      // and print results
      String reply = echoRef.sayEcho(args[1]);
      System.out.println(reply);
      }
   }
}
```

`Echo` was used as a name to bind the servant into the name server. Now use the same name to look it up. The result is an object of type `java.lang.Object`, which you have to cast to an `Echo` to be able to use its methods.

Step 5: Compile the Java Class Files

Use the following command to compile the files:

```
javac -d . *.java
```

✓ ***Run this command from the*** `/mod3-rmi/solutions/echo/` ***directory.***

Step 6: Create the Stubs and Skeletons

To complete the setup, you need the stub and the skeleton class. Provide a stub and a skeleton for every class that implements the `java.rmi.Remote` interface. In this case, this is `EchoImpl.java`, which implements `Echo.java`, which extends `Remote`. The stub compiler runs against the previously compiled class file.

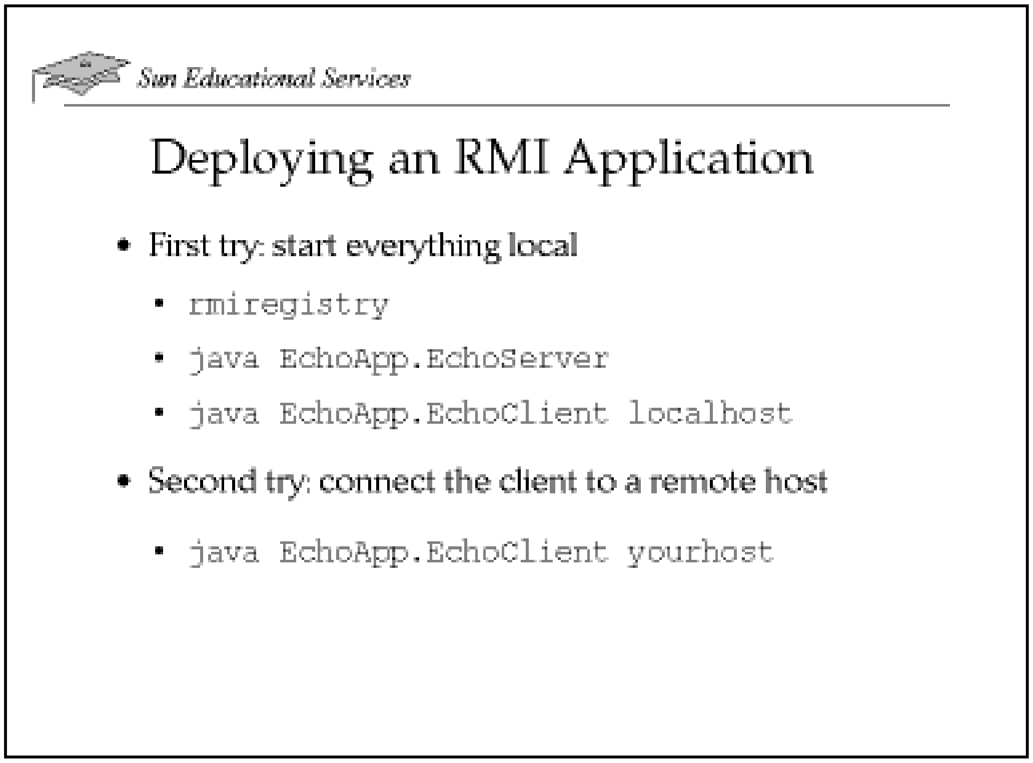
The utility to create the stub and skeleton is called the RMI compiler or `rmic`. The syntax is similar to the `javac` compiler. The stub and skeleton are compiled by typing the following:

```
rmic -d . EchoApp.EchoImpl
```

In your `echo` directory, you should now have an additional `EchoApp` directory, which contains the following files:

```
Echo.class
EchoClient.class
EchoImpl.class
EchoImpl_Skel.class
EchoImpl_Stub.class
EchoServer.class
```

If this is not the case on your system, redo steps 1–6 or ask your instructor what could have gone wrong.



Sun Educational Services

Deploying an RMI Application

- First try: start everything local
  - `rmiregistry`
  - `java EchoApp.EchoServer`
  - `java EchoApp.EchoClient localhost`
- Second try: connect the client to a remote host
  - `java EchoApp.EchoClient yourhost`

Deploying an RMI Application

First Try: Start Everything Local

Assuming there are no compilation problems, the RMI registry is started first, then the server, and finally the client.

```
rmiregistry (Windows: start rmiregistry)
```

The RMI registry does not produce any output, so you can safely start it in the background.

The server, which creates the servant and registers it with the RMI registry, is started next.

```
java EchoApp.EchoServer
```

The server does produce some output, so it is better to start it in its own console or command prompt.

In the last step, start the client.

```
java EchoApp.EchoClient localhost <your name>
```

Second Try: Connect to a Remote System

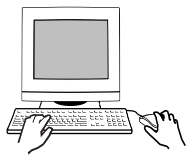
To connect your client to a different machine, provide the name of the machine running the RMI registry you want to use.

```
java EchoApp.EchoClient <host name> <your name>
```

This line connects you to host *`<host name>`*. Because a server can bind servants only to an RMI registry running on the same host as the server itself, *`<host name>`* also runs the servant.

Right now, all the files necessary to start up the server, servant, and client are accessible in the current directory or through the class path. You will see in a later section how to download the necessary code on demand.

Exercise: Compiling a Basic RMI Application



Exercise objective – Compile and run the Echo application. In addition, test the robustness of the RMI system by breaking the code of the Echo application intentionally.

Tasks

Compile the Echo *Application*

Work through the six steps described in the handout to compile the Echo application.

1. Change the directory to `labfiles/mod3-rmi/lab/echo`.
2. Compile the entire Echo application using `javac`.
3. Compile the stub using `rmic`.

✓ ***You can find more detailed instructions in the README file included in*** `/mod3-rmi/solutions/echo/`.

Run the Echo *Application*

Work through the steps described in "Deploying an RMI Application" to get the Echo application running.

1. Run the Echo application locally on your system.
2. Team up with somebody, and distribute the client and the server parts of Echo on two different machines.

Break the Echo *Application to Test the Robustness of the RMI System*

After Echo runs successfully, try to break the code in several ways to test the fault tolerance of RMI. Try out the following in the servant code (`EchoImpl.java`):

3. Do not implement a method that is in the interface (or fail to implement the correct method signature).

✓ ***The code does not compile successfully.***

4. Omit `extends UnicastRemoteObject`.

✓ ***The code can be compiled and rmic runs without error. However, at runtime a*** `MarshalException` ***gets thrown.***

5. Omit `implements Echo`.

✓ ***The code can be compiled successfully, but*** `rmic` ***complains.*** `Class EchoApp.EchoImpl` ***does not directly implement a*** `Remote` ***interface. This is true, since*** `Echo.java` ***extends*** `Remote`***, but*** `Echo.java` ***was no longer implemented.***

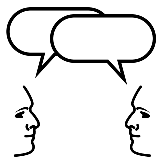
Work With the RMI Online Documentation (Optional)

Use the online RMI documentation to solve a specific problem.

1. Sometimes, you have your own class hierarchy, and cannot extend `UnicastRemoteObject` with your servant implementation. Look at the documentation for `java.rmi.server.UnicastRemoteObject` to find out how you can subclass from an arbitrary object, and still create a valid remote object.

2. Change `HelloImpl.java` to subclass `Object` instead of `UnicastRemoteObject`.

Exercise Summary



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

✓ ***Manage the discussion here based on the time allowed for this module, which was given in the "About This Course" module. If you find you do not have time to spend on discussion, then just highlight the key concepts students should have learned from the lab exercise.***

- Experiences

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

- Interpretations

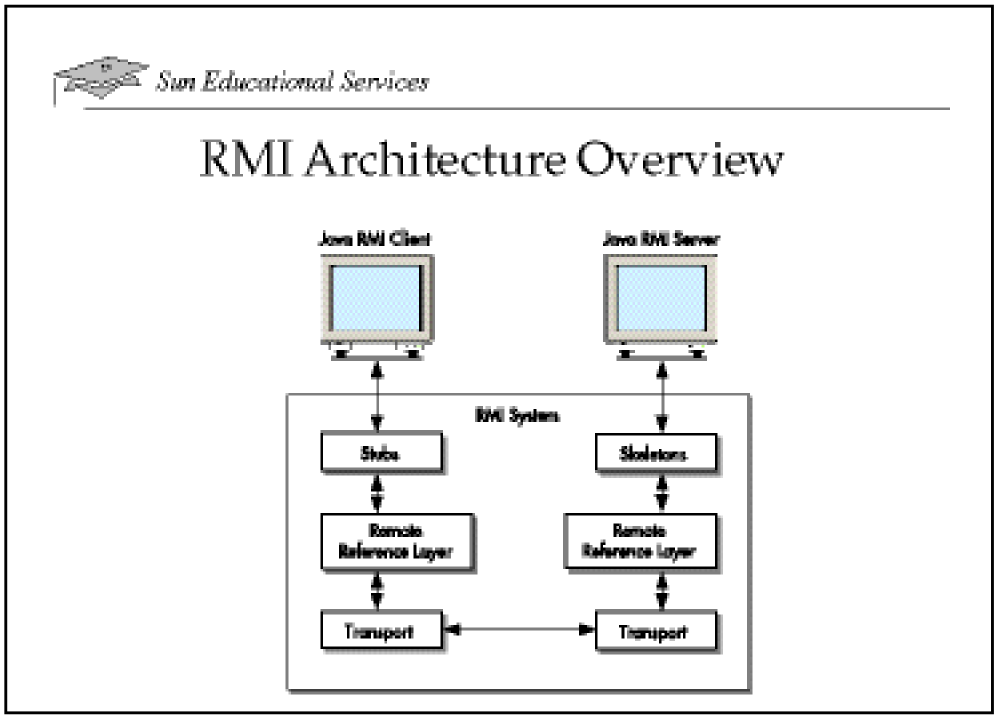
✓ ***Ask students to interpret what they observed during any aspects of this exercise.***

- Conclusions

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

- Applications

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



Sun Educational Services

RMI Architecture Overview

RMI Architecture

RMI Architecture Overview

An RMI application consists of four layers:

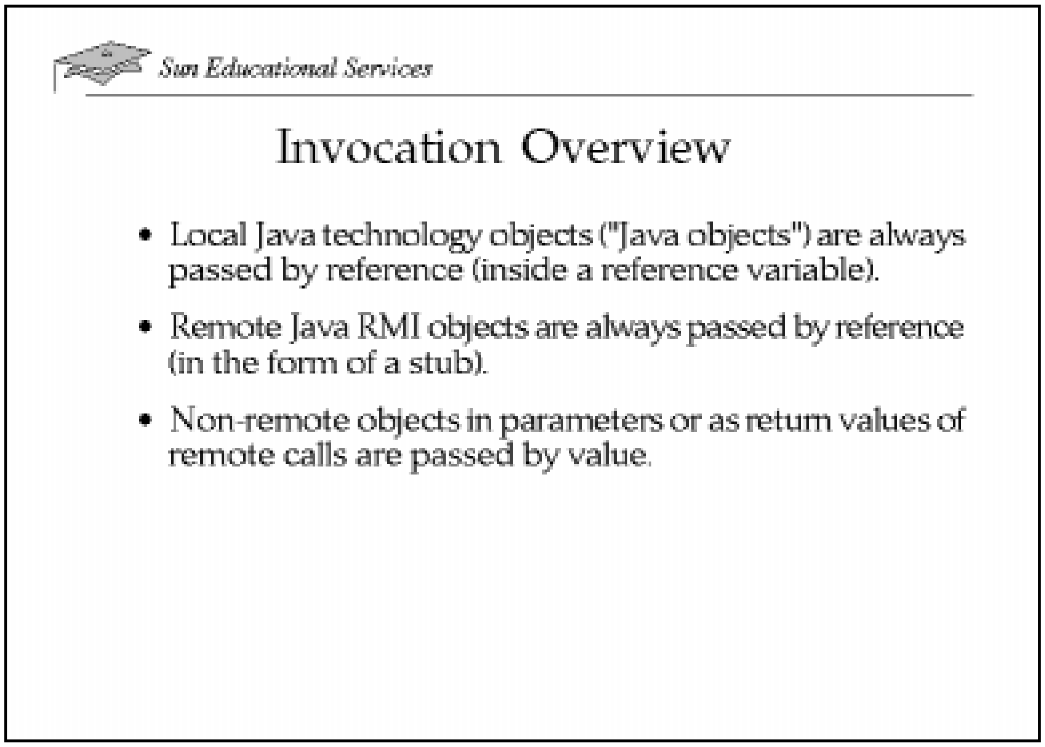
- Application
- Stub/skeleton
- Remote reference
- Transport

The client code first talks to its stub, which then sends the message to the remote reference layer (RRL). The RRL then passes it through the transport layer to the server machine.

At the server, the message goes all the way up again. The transport layer passes it to the RRL, which in turn retranslates it to the skeleton where it finally appears at the server's object implementation.

As a developer, you are responsible for creating only the Java RMI interface definitions and the implementation classes and for the generation of the stub and skeleton classes. The RRL and transport layer implementations are taken care of for you.

✓ ***Skeletons are not required by Java 2. They are required for the Java 1.1 RMI model. Skeletons are required if compatibility between Java 2 and Java 1.1 is desired (Java 2, The Complete Reference, Patrick Naughton and Herbert Schildt, McGraw-Hill, 1999, p. 836).***



Sun Educational Services

Invocation Overview

- Local Java technology objects ("Java objects") are always passed by reference (inside a reference variable).
- Remote Java RMI objects are always passed by reference (in the form of a stub).
- Non-remote objects in parameters or as return values of remote calls are passed by value.

Invocation Overview

Local Java objects are invoked by reference; however, the RMI system is passing objects in parameters or return values by copy and not by reference, because a reference to a local object makes sense only within one virtual machine. RMI uses the Java programming language serialization mechanism to flatten a local object into a serial stream and send it over the wire to the receiving instance.

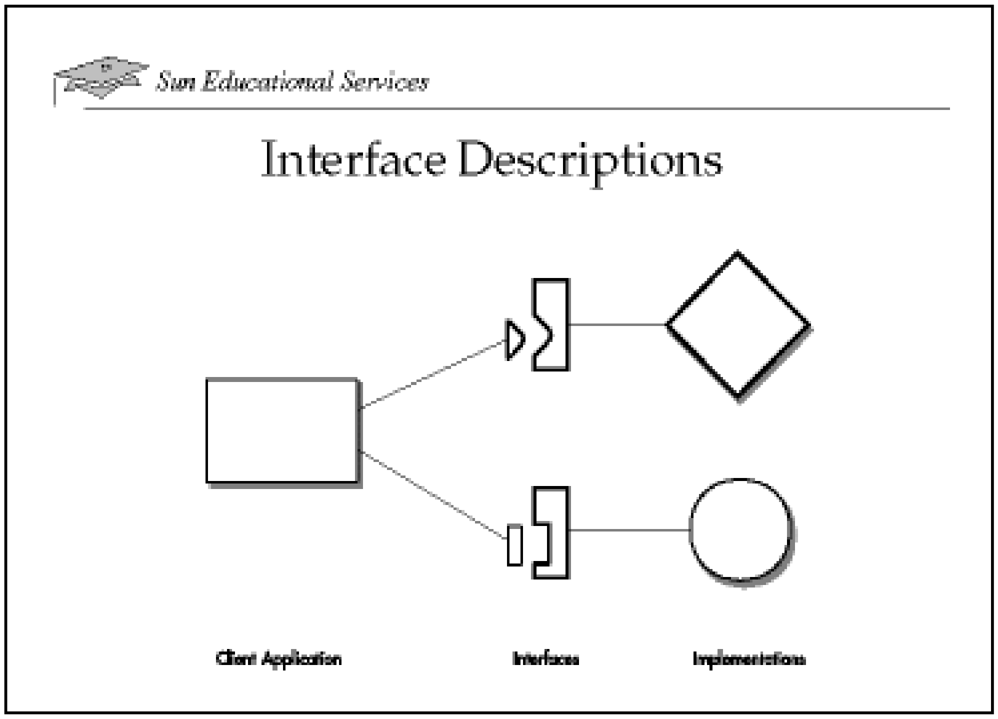
The remote methods are described by interfaces that are in turn implemented by remote objects called *servant objects*. At least one interface that extends the `java.rmi.Remote` interface must be implemented by the remote object. The remote object is accessible only to other objects through the methods defined in its interface.

RMI clients interact with remote objects using their published interfaces. The actual object is not sent to the client. The client gets a proxy or stub; that is, a handle to the remote object, by looking up the registry or by the return value of another remote method call. It is this stub the client is interacting with when remote methods are invoked. A single remote object can have many clients, which in turn all have their own stubs representing the remote object. On the server, a similar structure exists. A server-side proxy called a skeleton is responsible for passing the method calls to the referenced object.

The Java 2 SDK can get rid of the skeleton class. This *does not* imply that a skeleton is no longer needed. The `RMIRemoteObject` class has been modified and now has the additional job of being a universal skeleton, in combination with the Java programming language introspection mechanism.

The invocation of a method on a remote object is definitely not the same as calling a local object, but the code looks similar and can easily be understood. Nevertheless, keep the following differences in mind:

- Objects that are passed as parameters to remote methods and objects that are returned from a method are passed by value rather than by reference. A reference to the remote object is used only if a remote object is passed to or returned by a remote object.
- Objects that are passed as parameters to remote methods, and objects that are returned from a method, must be serializable.
- Remote objects are referred to by clients through the implemented remote interface. Casting the remote object to any of the implemented interfaces is therefore possible.
- The `toString` method has been changed to include information about the used network protocol and the originating host name and port number.
- The `hashCode` method has been modified to return identical keys for any references to the same remote object.
- The `equals` method checks whether the object references of remote objects are equal and not the contents of remote objects.


Sun Educational Services

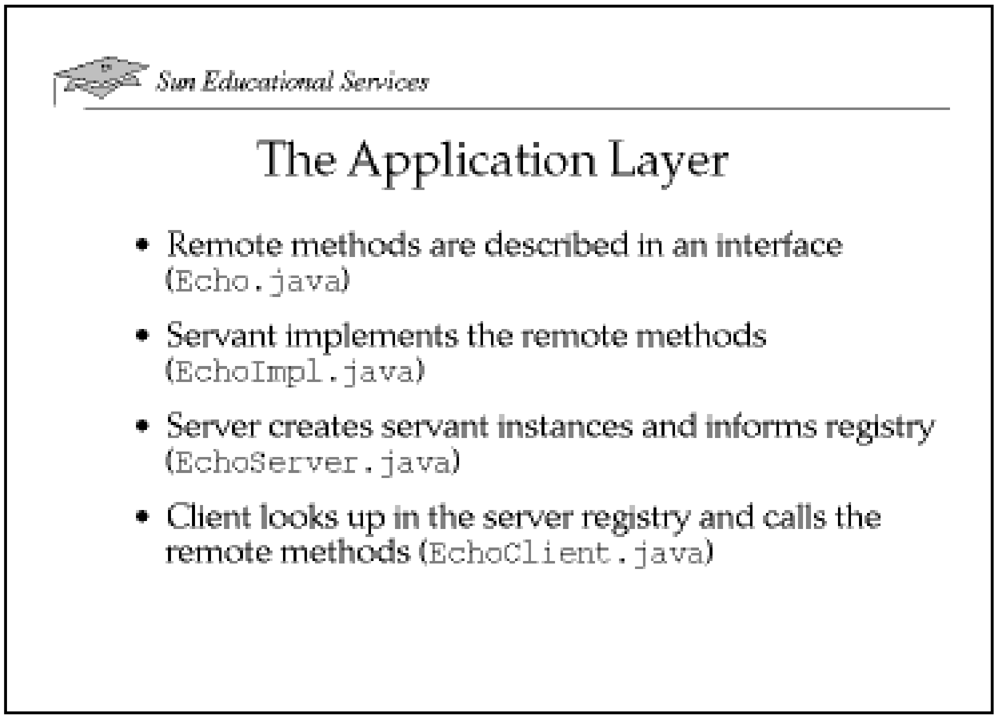
Interface Descriptions

Interface Descriptions

You can use RMI between virtual machines on a single system or over a network between virtual machines (VMs) on multiple machines. Using interfaces allow you to create a highly modular application design.

An application component is designed as a set of well defined interfaces that objects use to communicate with each other. That is, the interaction with an object is entirely described by the interface. The actual design and implementation of these components is invisible outside of the individual object and can be replaced anytime without any restrictions for the rest of the system.

In RMI, all interfaces are described in pure Java code. With the RMI stub compiler (`rmic`) the stub and skeleton files are generated directly from the compiled code that implements the interfaces.



Sun Educational Services

The Application Layer

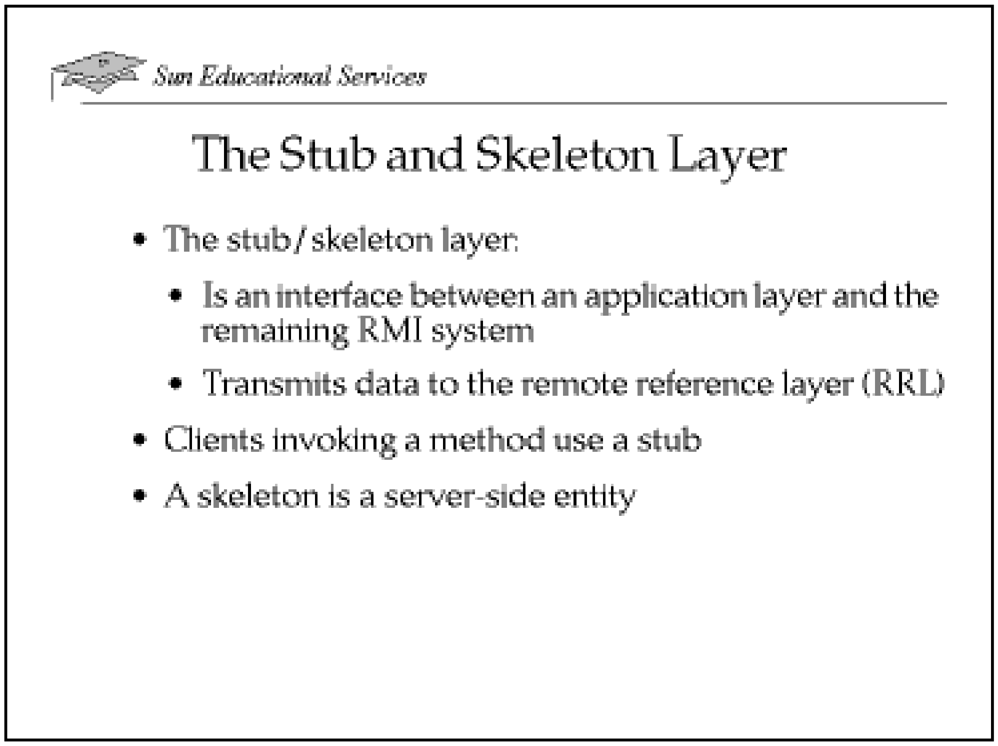
- Remote methods are described in an interface (`Echo.java`)
- Servant implements the remote methods (`EchoImpl.java`)
- Server creates servant instances and informs registry (`EchoServer.java`)
- Client looks up in the server registry and calls the remote methods (`EchoClient.java`)

The Application Layer

The application layer consists of the actual implementation of the client and server applications. The methods that are made available for remote clients are described in an interface that extends `java.rmi.Remote`. The `java.rmi.Remote` interface does not contain any methods and is used only to mark objects as remotely accessible. Such interfaces are implemented as usual; only some additional code for dealing with `RemoteExceptions` is needed.

As a subclass of the class `UnicastRemoteObject,` the export of the remote object is handled by the superclass. You can also call the `exportObject` method explicitly.

The final part for an RMI application is to register the remote object with a name server to be available for the first contact of a requesting client.



Sun Educational Services

The Stub and Skeleton Layer

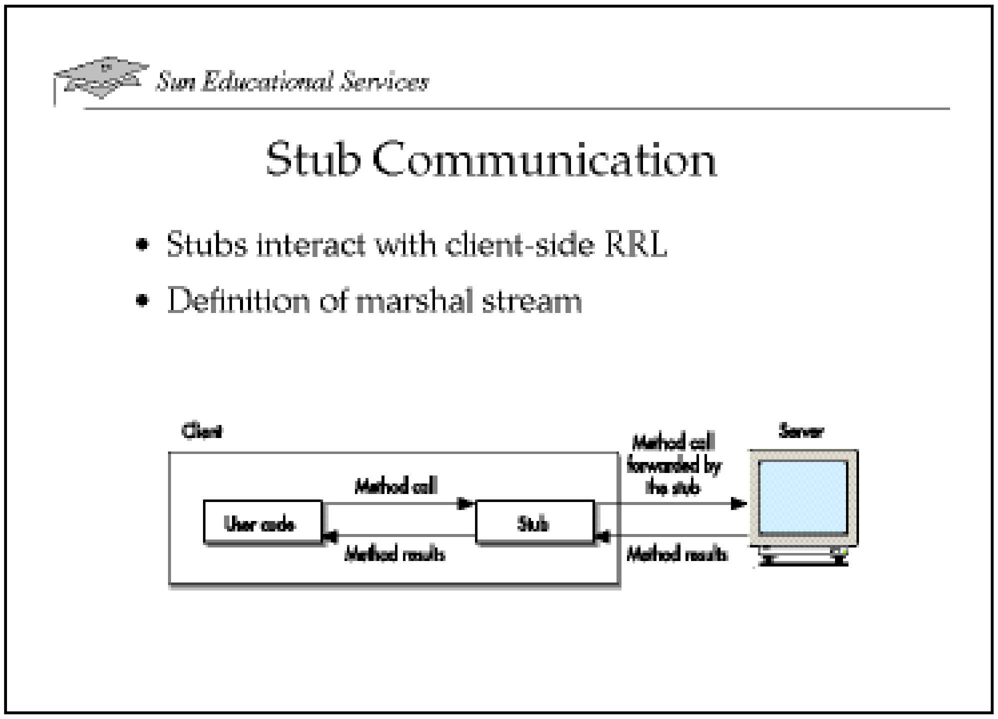
- The stub/skeleton layer:
  - Is an interface between an application layer and the remaining RMI system
  - Transmits data to the remote reference layer (RRL)
- Clients invoking a method use a stub
- A skeleton is a server-side entity

The Stub and Skeleton Layer

The stub as the client-side and the skeleton as the server-side representation are class files that are generated by the RMI stub compiler (`rmic`). This layer does not deal with any of the specifics of any transport, but transmits data to the remote reference layer.

The stub is responsible for the initiation of calls to the remote object it represents. It is also the stub that marshals method arguments to a stream object that contains parameters, errors, and exceptions, and unmarshals (receives and interprets) streams returned by the remote object.

All interfaces of a remote object are also implemented by the stub class that represents this object at the client. A stub class is type equivalent to any of the represented remote interfaces and can therefore be cast as any of those. A test with the `instanceof` operator answers whether a certain interface is implemented by a remote object.



Sun Educational Services

Stub Communication

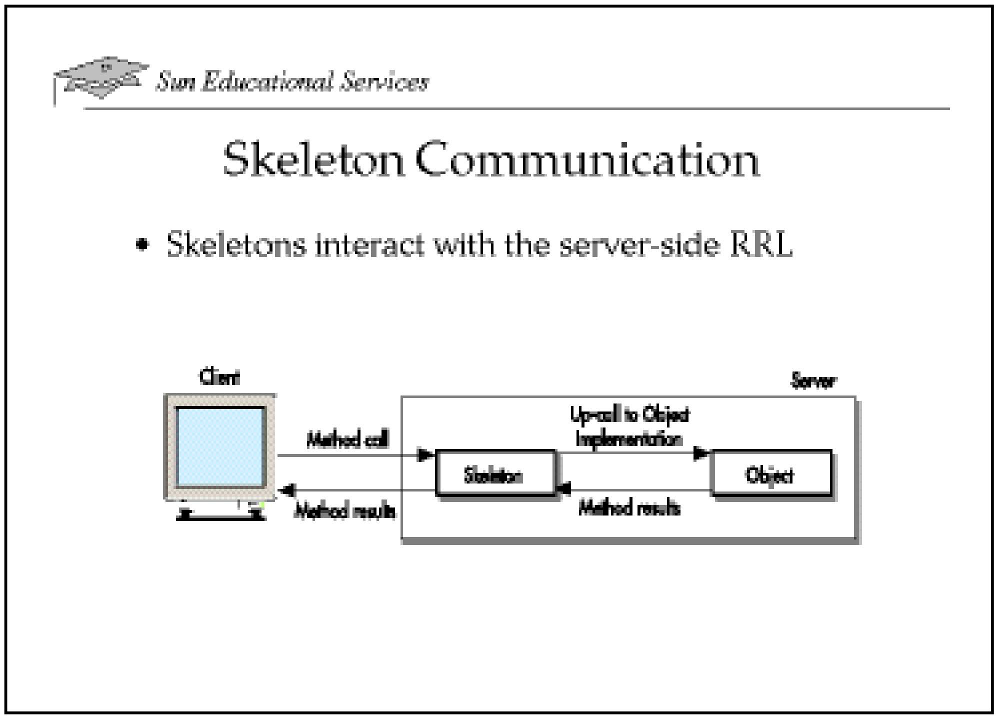
- Stubs interact with client-side RRL
- Definition of marshal stream

Client
Method call
User code
Stub
Method results
Method call forwarded by the stub
Server
Method results

Stub Communication

Stubs interact with the client-side RRL in the following ways:

- The stub receives the remote method invocation and initiates a call to the remote object.
- The RRL returns a *marshal stream*, which is used to communicate with the server's RRL.
- The stub makes the remote method call, passing any arguments to the stream.
- The RRL passes the method's return value to the stub.
- The stub acknowledges to the RRL that the method call is complete.



Sun Educational Services

Skeleton Communication

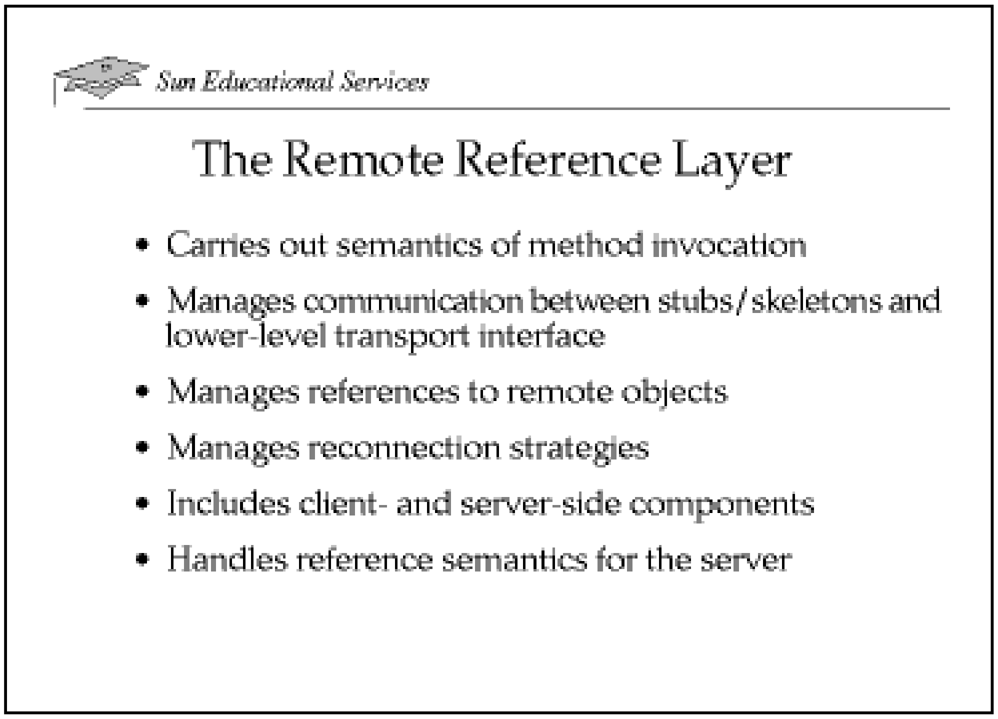
- Skeletons interact with the server-side RRL

Skeleton Communication

The skeleton class receives the method calls, marshals the parameters, and dispatches the methods to be exported. Skeletons interact with the server-side RRL in the following ways:

- The skeleton unmarshals (receives and interprets) any arguments from the I/O stream, established by the RRL.
- The skeleton makes the up-call to the actual remote object implementation.
- The skeleton marshals (interprets and sends) the return value of the call (or an exception, if one occurred) onto the I/O stream.

As mentioned before, the skeleton class is no longer needed with the Java 2 SDK, its function is integrated in `UnicastRemoteObject`.



Sun Educational Services

The Remote Reference Layer

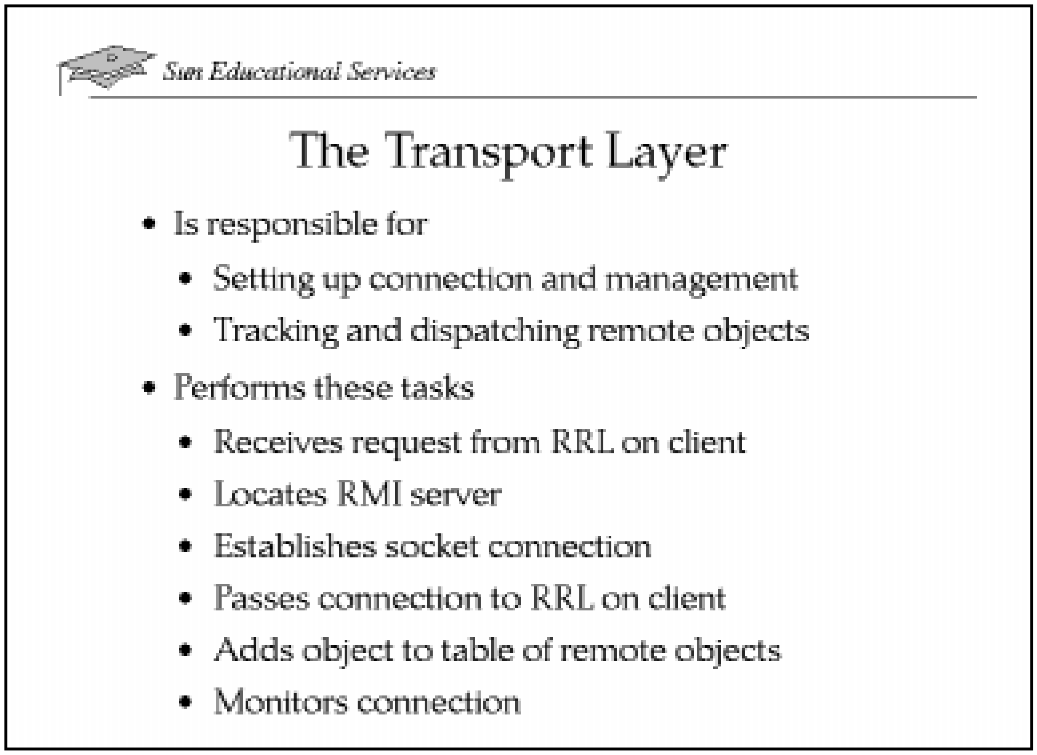
- Carries out semantics of method invocation
- Manages communication between stubs/skeletons and lower-level transport interface
- Manages references to remote objects
- Manages reconnection strategies
- Includes client- and server-side components
- Handles reference semantics for the server

The Remote Reference Layer

The remote reference layer (RRL) bridges the gap between stub and skeleton code and the network communication.

The RRL is responsible for carrying out the semantics of the method invocation. It manages communication between the stubs/skeletons and the lower-level transport interface using a specific remote reference protocol, which is independent of the client stubs and server skeletons. The RRL's responsibilities include managing references to remote objects and reconnection strategies if an object should become unavailable.

The RRL has two cooperating components: the client side and the server side. The client-side component contains information specific to the remote server, and communicates using the transport layer to the server-side component. The server-side component implements the specific remote reference semantics prior to delivering a remote method invocation to the skeleton.



Sun Educational Services

The Transport Layer

- Is responsible for
  - Setting up connection and management
  - Tracking and dispatching remote objects
- Performs these tasks
  - Receives request from RRL on client
  - Locates RMI server
  - Establishes socket connection
  - Passes connection to RRL on client
  - Adds object to table of remote objects
  - Monitors connection

The Transport Layer

The transport layer is responsible for connection setup, connection management, and keeping track of and dispatching remote objects (the targets of remote calls) residing in the transport's address space.

The transport layer performs the following tasks:

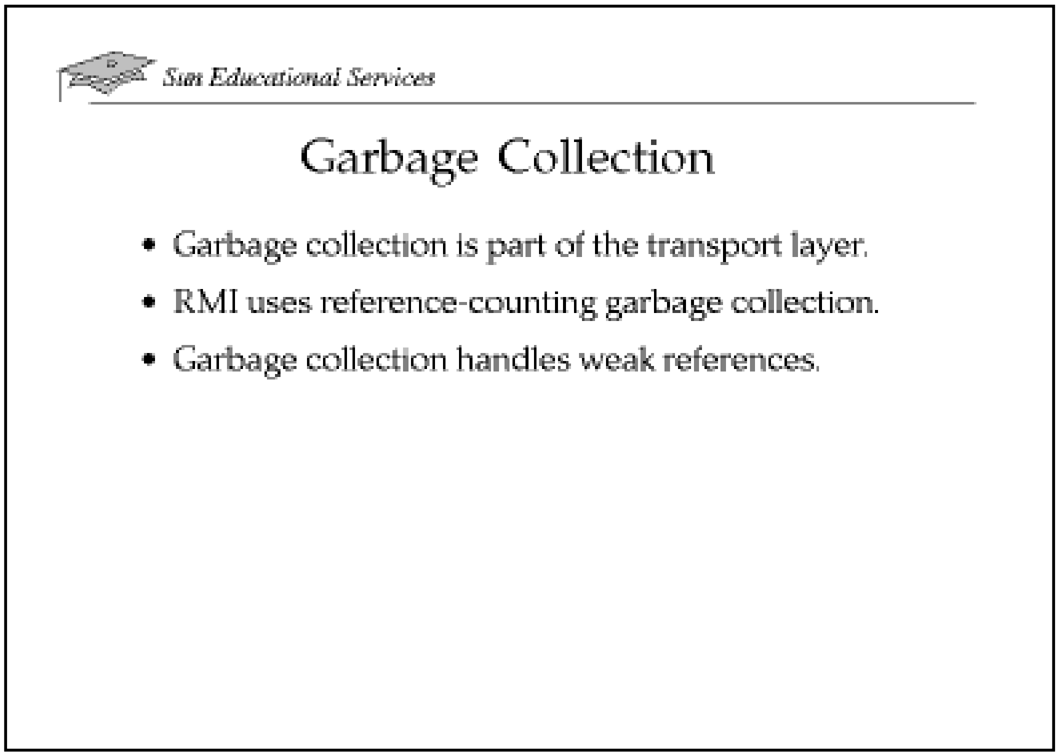
- Receives a request from the client-side remote reference layer
- Locates the RMI server for the remote object requested
- Establishes a socket connection to the server

✓ ***Currently, Java RMI uses TCP sockets. Typically, there are two socket connections between the server and client: one for method calls and one for the DGC. RMI will attempt to re-use existing socket connections for multiple objects from the same server, but if an existing socket is in use when the attempt is made, a new socket will be created.***

- Passes that connection back up the client-side remote reference layer
- Adds this remote object to a table of remote objects with which it knows how to communicate
- Monitors connection “liveness”

✓ ***A client cannot close a connection to an RMI server because it is handled at this layer, so 1.1 connections time out if they are unused for a period of time. This period defaults to 10 minutes, and is set by the*** `java.rmi.dgc.leaseValue` ***property.***

You can adjust the transport layer to deal with special requirements, such as encryption or data compression.


Sun Educational Services

Garbage Collection

- Garbage collection is part of the transport layer.
- RMI uses reference-counting garbage collection.
- Garbage collection handles weak references.

Garbage Collection

RMI uses a reference-counting garbage collection scheme that keeps track of external live references to remote objects within a local virtual machine. A live reference is, in this case, an active connection over a TCP/IP session.

Distributed garbage collection is a tricky field. For example, a remote object on the server is prematurely collected as garbage because of a link failure. The client tries to reconnect with a reference to an object that does not exist anymore. This causes a `RemoteException` that must be handled by the calling object. There is no automatic rebinding.

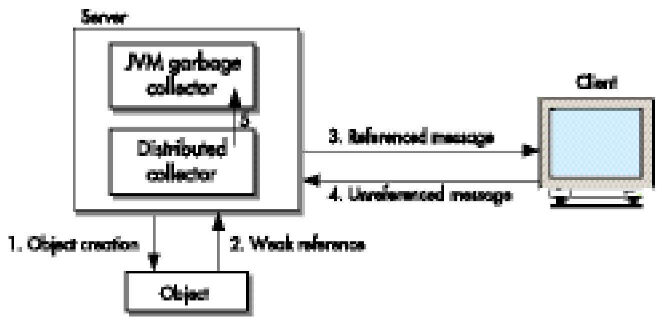
When a live reference enters a JVM, its reference count is incremented. When an object is found to be unreferenced, the reference count is decremented as the object is finalized. When the last reference has been discarded, an "unreferenced" message is sent to the server.

A remote object that is no longer referenced by any client is considered to have a *weak reference*. The weak reference allows the server's garbage collector to discard the object if no other local references to the object exist. As long as a local reference to a remote object exists, it cannot be collected as garbage.

A reference to a remote object is in fact leased by the client and ends after a certain time. This period defaults to 10 minutes and can be set by the `java.rmi.dgc.leaseValue` property. The client starts leasing with a call and is responsible for extending the leasing periodically by additional calls. If the client fails to make the additional calls, the leasing of the remote reference expires and the garbage collector will do its job. For local references, the garbage collector calls the `finalize` method of the object before removing it from memory, which holds a chance for a last notification.

Sun Educational Services

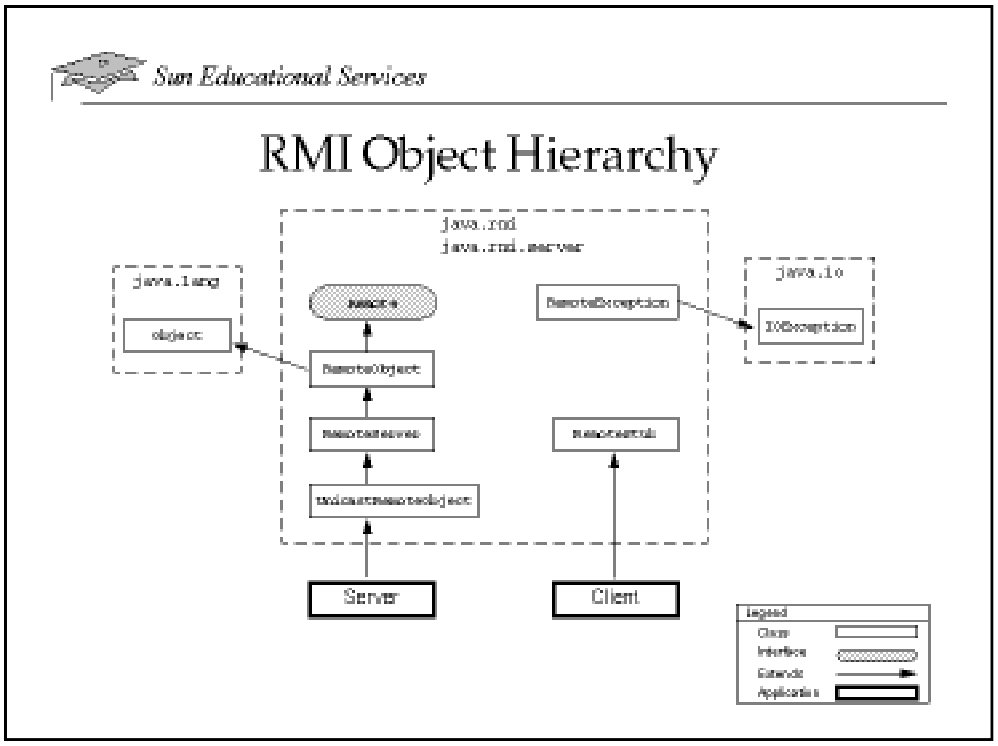
Garbage Collection



Garbage Collection

The steps involved in garbage collection are:

1. The server (implementation) starts and creates an object that is referenced remotely. This establishes a weak reference to the object.
2. When the client requests the object, the client's JVM creates a live reference, and the first reference to the object sends a "referenced" message to the server.
3. When the object goes out of scope on the client, an "unreferenced" message is sent to the server.
4. When the count on the object goes to 0, and there are no local references to the object, the object reference can be passed to the server's local garbage collector for collection.



RMI Object Hierarchy

The `Remote` interface identifies remote objects. Every remote object must implement this interface, which itself does not contain any methods. Its purpose is only to flag remote objects.

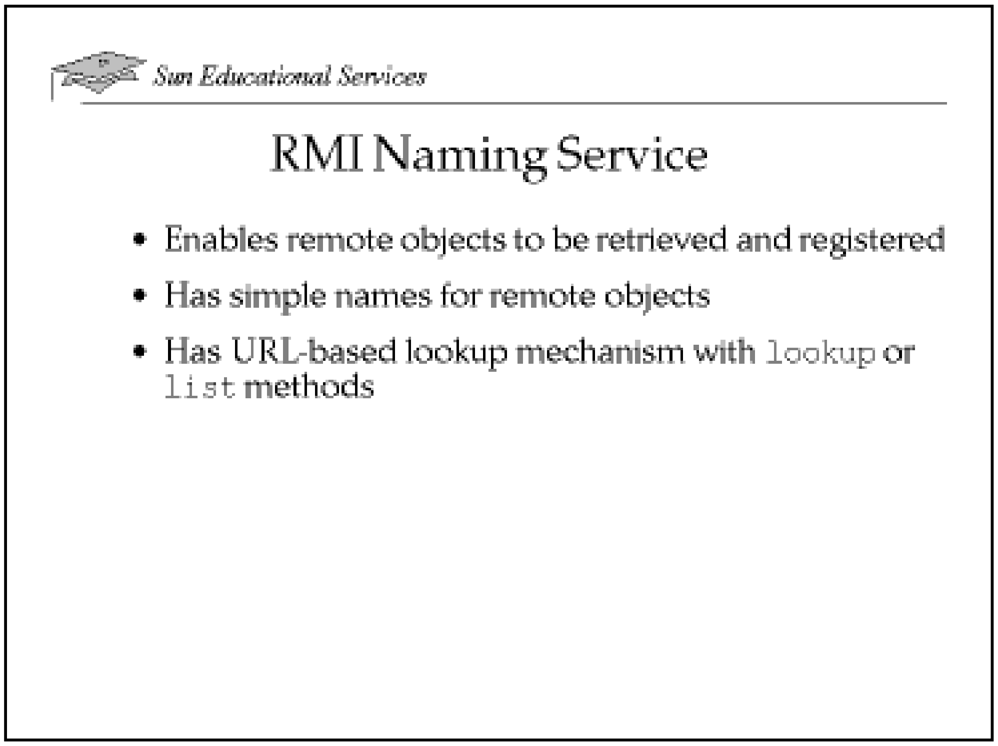
The `RemoteObject` class acts like a remote representative for the Java root class `Object` and provides some overridden methods in their distributed version, such as `hashCode`, `equals`, and `toString`. The `getClass` method works for local as well as for remote objects. Used with a remote object, it returns the stub's type.

The `RemoteServer` class is used for the creation and export of server objects and is therefore the superclass of all RMI servers. The `getClientHost` method returns the IP address of the requesting client. The `getLog` and `setLog` methods allow logging.

Most servants are subclassing `UnicastRemoteObject`. References to a server object are valid only during the lifetime of this object. The `exportObject` method allows a stub object to be exported explicitly.

The `RemoteStub` class is the superclass for all client stubs. It is the client's representative of an implementation class.

The RMI system currently does not support multicast remote objects and replication.



Sun Educational Services

RMI Naming Service

- Enables remote objects to be retrieved and registered
- Has simple names for remote objects
- Has URL-based lookup mechanism with `lookup` or `list` methods

RMI Naming Service

The RMI registry enables remote objects to be retrieved and registered using simple names. Each server process can either maintain its own registry or be part of a single registry that supports all virtual machines on a local system.

Remote objects that are to be exported have to register their names to be found by a requesting client. The server application uses the static `bind`, `unbind`, and `rebind` methods of the `java.rmi.Naming` class to register its object implementations with the RMI registry. A URL-based lookup by the client that also uses the `Naming` class to achieve this can then be performed when the client requests a reference to a remote object.

The normal registration process is usually handled by maintaining the RMI registry on a defined port number allowing a client to query the service with the `lookup` or `list` methods. An alternative design would be to integrate the naming services into the RMI application to gain full control over the registry.

The fact that remote objects can return other remote objects means that the initial contact to the registry service—the bootstrapping of the naming service—is performed only once. Further references to remote objects on the respective server can be obtained by calling the already referenced object.

In the `Echo` application, the naming service is reflected in the RMI client class (`EchoClient.java`) with its `lookup` method and in the implementation class

```
// EchoClient.java:
...
String url = new String("rmi://"+args[0]+"/Echo");
Echo EchoRef = (Echo)Naming.lookup(url);
...
```

and in the RMI server class with the binding to the Naming class.

```
// ;
// EchoServer.java:
...
// Create the servant instance for registration
EchoImpl echoRef = new EchoImpl();

// Bind the object to the rmiregistry
Naming.rebind("Echo", echoRef);
```

The `rebind` method associates or “binds” the name `Echo` to the `echoRef` object, removing any object previously bound with this name from the registry.

The `Naming` class provides two static methods that enable the developer to register an implementation, `bind` and `rebind`. The only difference is that the `bind` method throws a `java.rmi.AlreadyBoundException` if another object has already been registered on this server, using the name passed as the first argument to the method.

The arguments to bind and rebind are a URL-like `String` and the instance name of the object implementation. The `String` is expected to be in the format:

```
rmi://host:port/name
```

where `rmi` is the protocol, *`host`* is the name of the RMI server (which might need to be fully DNS or NIS+ qualified), *`port`* is the port number that the server should be listening to for requests, and *`name`* is the exact name that clients should use in `Naming.lookup` requests for this object. If not specified, the protocol defaults to `rmi`, *`host`* defaults to the local host, and *`port`* defaults to `1099`.

✓ ***For security reasons, an application can bind only to a registry running on the same host.***

The `rmiregistry` *Application*

The `rmiregistry` is an application that provides a simple non-persistent naming lookup service that helps a client to locate remote objects. The `EchoServer` provides `rmiregistry` with the object reference and a `String` *`name`* through the `rebind` method call.

✓ ***The registry is also responsible for polling the table of remote objects, and providing reference information (if each object has a live reference or a weak reference) to the distributed garbage collector.***

The `rmiregistry` must be running before the `EchoServer` application attempts to bind.

```
    rmiregistry
```

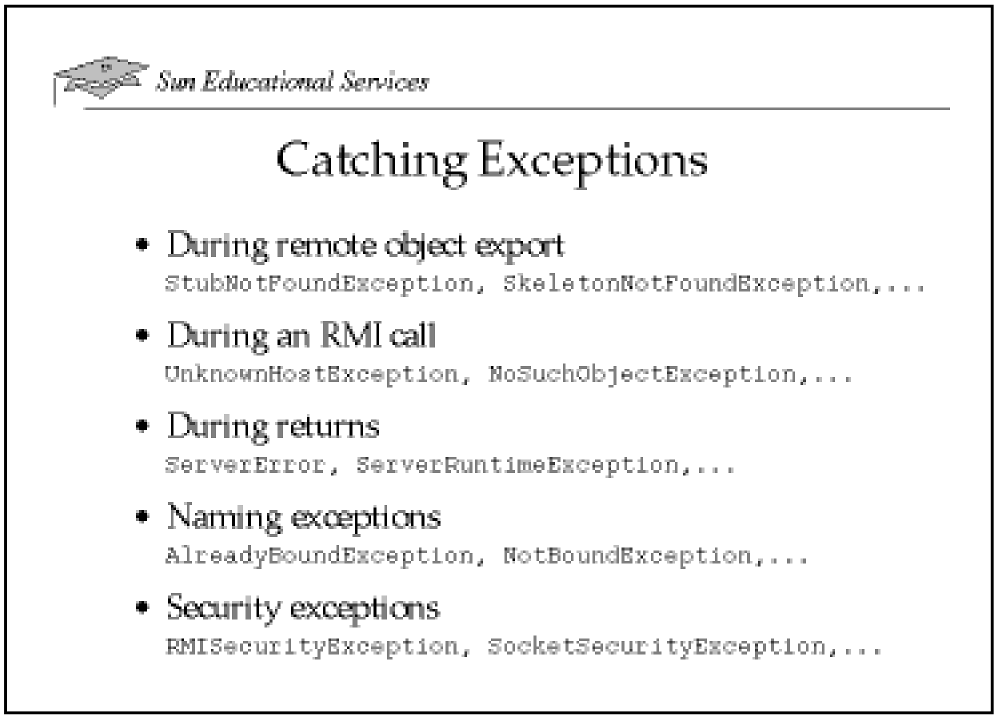
You can set properties for the RMI server JVM from the command line:

- `java.rmi.server.codebase` – A URL that indicates from where clients can download classes.
- `java.rmi.server.logCalls` – If set to true, the server logs calls to `stderr`. By default it is set to false.

```
 java -Djava.rmi.server.logCalls=true EchoApp.EchoServer
&
```

Once the implementation is exported by the registry, the client can send a URL string to request that the `rmiregistry` provide a reference to the remote object. The lookup is accomplished through a client call to `Naming.lookup`, passing in a URL string as the argument:

```
rmi://host:port/name
```



Sun Educational Services

Catching Exceptions

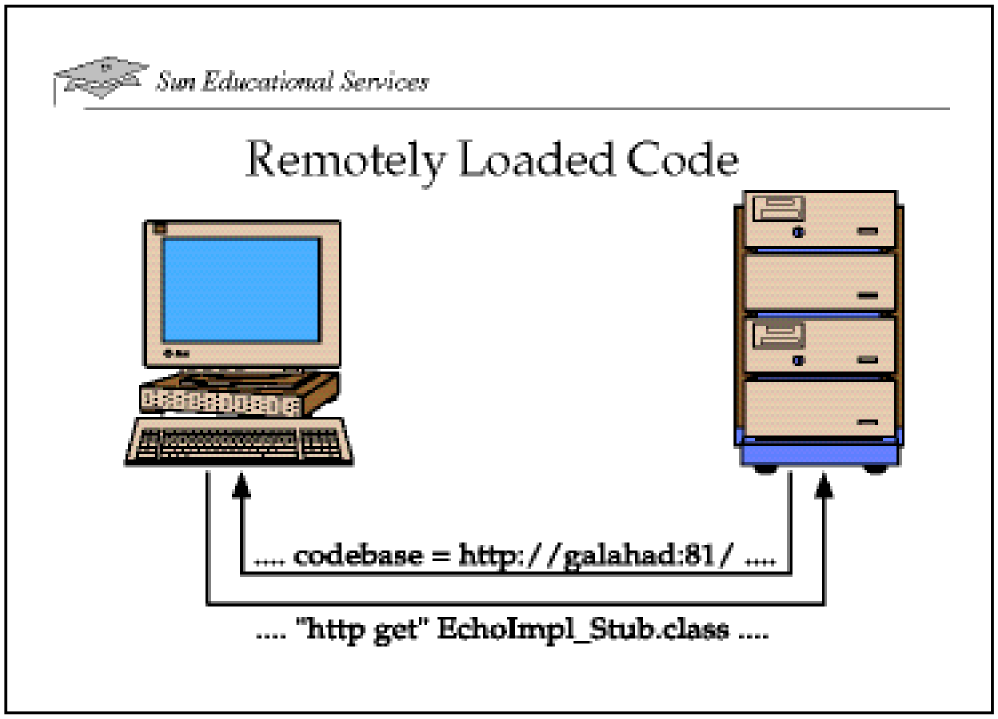
- During remote object export
  `StubNotFoundException, SkeletonNotFoundException,...`
- During an RMI call
  `UnknownHostException, NoSuchObjectException,...`
- During returns
  `ServerError, ServerRuntimeException,...`
- Naming exceptions
  `AlreadyBoundException, NotBoundException,...`
- Security exceptions
  `RMISecurityException, SocketSecurityException,...`

Catching Exceptions

There are many more ways for distributed applications to fail than there are for their local counterparts. Consequently, the RMI systems extend the Java exception classes to deal with the additional complexity. The exception handling is therefore not bound to a local virtual machine.

There are various reasons for exceptions to be thrown during a remote method invocation. Consequently, there are various exceptions defined in the RMI system to reflect this. The RemoteException class is the superclass of all RMI exceptions.

In the `Echo` application, there are try-catch clauses in the client class embracing the `lookup` method, as well as in the server class around the code that binds the object to the registry.


Sun Educational Services
Remotely Loaded Code
.... codebase = http://galahad:81/ ....
.... "http get" EchoImpl_Stub.class ....

Remotely Loaded Code

The Java programming language allows you to download code from a remote system, such as from a Web server. This is also true for applets doing RMI. Keep in mind that an RMI applet needs more than just the applet class, there is also the stub and the interface. These classes also must be available on the HTTP server.

With RMI, you can download some of the required classes for an RMI *application* from a Web server. There is a special class called `RMIClassLoader`, which is part of the `rmi.server` package. This class loader is used by the RMI runtime system transparently if it cannot find a required class in the local class path. The most important class that does not have to be on the client is the stub. Other new classes could be acquired as a return value from a remote call. If all the client “knows” about a return value is the interface, the actual class that implements this interface is known only at runtime.

The `java.rmi.server.codebase` *Property*

While the server gets all its needed classes from the class path, the client has to get them from a Web server. How does the client know which URL to use? You could hardcode the URL in the client code, or deliver it as a startup argument on the client side. The problem with these approaches is that they make it difficult to relocate a server. All clients or client startup scripts would have to be updated.

RMI offers a better approach. The URL is set on the server side, and RMI clients learn about the URL automatically; it is embedded in the RMI wire protocol. The URL is set as a system property when the server is started. For example:

```
java -Djava.rmi.server.codebase=http://myhost:8080/
EchoApp.EchoServer
```

This line tells the clients to get their class files from the host *galahad* on port *81*. The trailing "`/`" is mandatory, because the URL always denotes a directory, and never a single file.

Note – The `rmiregistry` application also gets the URL from the server. At the moment the server binds the servant into the registry, the `rmiregistry` needs the class file. If the `rmiregistry` is able, at this moment, to locate the class using the class path, instead of using the server supplied URL, the URL system ceases to work. A client cannot get the class using the URL anymore. It is unclear why this is the case, but this behavior causes a lot of confusion about the `java.rmi.server.codebase` property. If you start the `rmiregistry`, be sure to start it with an empty class path.

Security Aspects

Security on the Client

It is potentially dangerous to download remote code. Therefore, to use the `RMIClassLoader`, you must have a `SecurityManager` in place to ensure that classes loaded from the network meet the desired security restrictions. If there is no security manager running, the application can load classes only from the class path. You do not have to develop a `SecurityManager` on your own, RMI provides one: the `RMISecurityManager`. This security manager behaves much like the AppletSecurityManager that is present in most browsers.

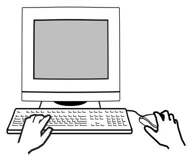
Security on the Server

The typical closed-system scenario has the server configured to load no remote classes. The services it provides are defined by remote interfaces that are all local to the server machine. The server has no security manager and cannot load classes even if clients send along the URL. If clients send remote objects for which the server does not have stub classes, those method invocations fail when the request is unmarshalled, and the client receives an exception.

The more open server system defines its `java.rmi.server.codebase` so that classes for the remote objects it exports can be loaded by clients, and so that the server can load classes when needed for remote objects supplied by clients. The server has both a security manager and RMI class loader, which protects the server. A somewhat more cautious server can use the property `java.rmi.server.useCodebaseOnly` to disable the loading of classes from client-supplied URLs.

A client can supply a URL in the same way a server does, by setting the `java.rmi.server.codebase` property *on the client*.

Exercise: Remotely Loaded Code



Exercise objective – So far, all the code needed to run the Echo client and server was loaded from the current directory or from the class path. Change this so that the client receives its stub automatically using a Web server.

Preparation

With the lab setup comes a mini Web server that is precompiled. Start the mini Web server, and pass the port number where it should serve the files, and the path where the EchoApp package is found after you complete it.

```
java httpsrv.ClassFileServer 8080 ~/labfiles/echoremote
```

✓ ***Compile from*** ~/SL301_revC_August_1999/SL301_XXX_LF ***using the following command*** javac httpsrv/*.java ***, then launch the server*** java httpsrv.ClassFileServer 8080 ~/SL301_revC_August_1999/SL301_XXX_LF/labfiles/echoremote

Tasks

Modify the Client so That it Can Receive Code From a Web Server

The Java programming language allows you to load code only from a remote location if a security manager is in place. The first task is to modify the client to instantiate and install a security manager.

1. Change the directory to `labfiles/mod3-rmi/lab/echoremote.`
2. Modify the client to instantiate and install an `RMISecurityManager`. Use the Java 2 SDK documentation to find out how to accomplish this (look specifically at the classes `java.rmi.RMISecurityManager` and at `java.lang.System`).

Move the Compiled Stub out of the Client's Class Path

The class path is always searched before any remote location is tried. Therefore, you must move the client code so that the stub is no longer in the class path.

1. In the `echoremote` directory, you will find an empty directory, `client`. Use this directory as a starting point to launch the client. Copy the client (`EchoClient.java`) and the interface (`Echo.java`) into this directory. (Be sure to preserve the package structure.)

2. Start the `rmiregistry` in such a way so that none of the `Echo` classes are in the class path.

3. Launch the server and pass the location of the Web server in the `java.rmi.server.codebase` property.

   ```
   java -Djava.rmi.server.codebase=http://yourhost:8080/
   EchoApp.EchoServer
   ```

   There is a blank just before `EchoApp.EchoServer`.

4. Now you are ready to test the client. Type:

   `java EchoApp.EchoClient` ***`yourhost`*** **`Emerald`**

   The client should still work, even if the stub is no longer available from the class path.

Test the Robustness of the Remote Code Loading Mechanism (Optional)

Once you are convinced the download of code works, try to break it to find out how robust the system is. Try the following:

1. Do not load the `RMISecurityManager.`

2. Start the `rmiregistry.`

3. Do not set the `java.rmi.server.codebase` property.

4. Set the property on the client instead of on the server.

Exercise Summary

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

✓ ***Manage the discussion here based on the time allowed for this module, which was given in the "About This Course" module. If you find you do not have time to spend on discussion, then just highlight the key concepts students should have learned from the lab exercise.***

- Experiences

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

- Interpretations

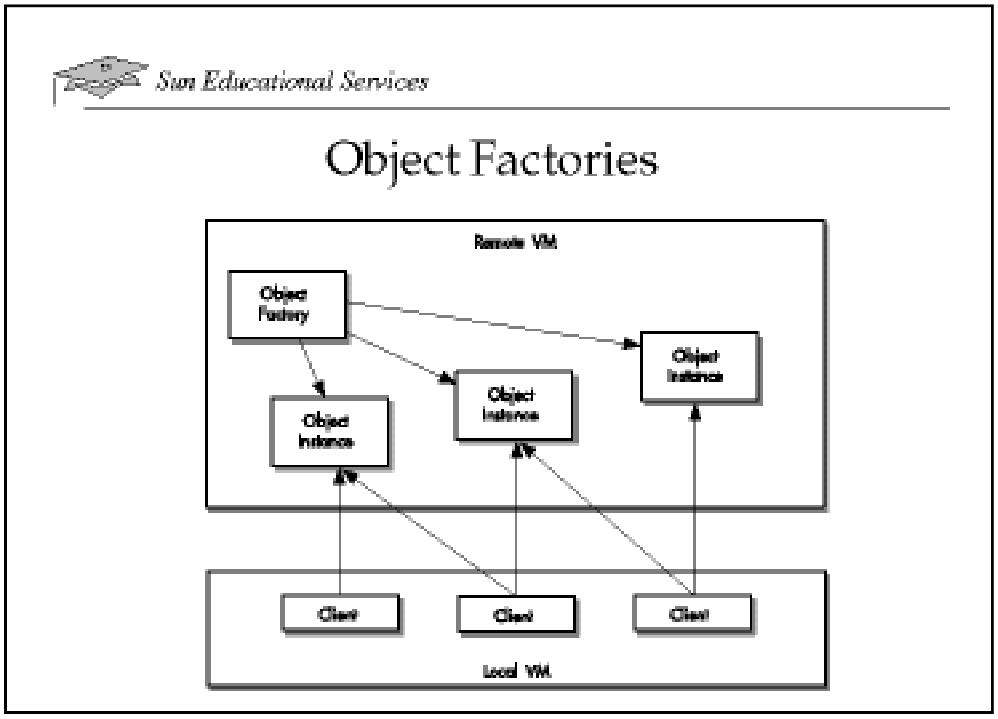
✓ ***Ask students to interpret what they observed during any aspects of this exercise.***

- Conclusions

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

- Applications

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



Advanced RMI

Object Factories

A servant must be running to be used by its clients, but a client cannot start a servant directly (this is the job of the server). The technique that defines this limitation is called *object factory* and allows the creation of multiple instances of remote objects on the fly, controlled by the client application.

The basic idea of object factories is to have distributed constructors to create the needed objects. An object factory is therefore a class that has one goal: create instances of objects with which clients need to interact. Normally, object factories only have a single "create a new object" method that is called by a client. The calling object gets a remote reference back to this newly created instance and can then call methods on it.

The following sections describe how the Echo application is converted to use an object factory.

Creating the Factory Interface

There is no need to change the original Echo interface, because the sayEcho method of the original servant is still needed. Instead, you must develop a new interface: the interface for the object factory. It describes the methods a factory server must implement to create other server objects. There must be at least one method doing two jobs: create a new servant instance, and return an object reference to this new instance.

In the Echo example, the factory looks like the following:

```
1  public interface EchoFactory extends Remote
2     Echo getEcho(String greeting) throws
3        RemoteException;
4  }
```

The getEcho method returns—as its name suggests—an instance of Echo. The getEcho method also takes a greeting string which is used to customize the different remote Echo instances.

Creating the Factory

The object factory is a simple RMI server class extending UnicastRemoteObject and implementing the EchoFactory interface; that is, it must provide the body for the getEcho method.

```
1  public Echo getEcho(String message)throws
2     RemoteException {
3        EchoImpl echoRef = new EchoImpl(message);
4        return (Echo)echoRef;
5  }
```

The cast from EchoImp to Echo in line 3, which is needed to fulfill the return value defined in the interface definition. With this factory implementation, a client can create different remote Echos dynamically.

Adjusting the Echo Implementation File

Theoretically, the EchoImpl could have been left unchanged, because the Echo interface did not change. But the problem with the current EchoImpl is that one EchoImpl instance is indistinguishable from another. There is no point in having more than one instance if all the instances do the same task. This is why EchoImpl is changed to support the customization of its greeting message. This way, different EchoImpl instances can be constructed, each one greeting in a different language. To accomplish this, you add a new constructor. The new constructor takes the greeting string and stores it internally. Not shown are the changes to the sayEcho method, which replaces the hardcoded message with the contents of the greeting variable.

```
public EchoImpl(String greeting) throws
   RemoteException {
      this.greeting = greeting;
}
```

Changing the Server

The Echo server does the same task. It creates a servant that is up and running and waiting to be called. However, the servant is the factory. The factory then is in charge of instantiating instances of Echo.

```
EchoFactory factoryRef = new EchoFactoryImpl();
```

The EchoFactory is bound to the registry as usual:

```
Naming.rebind("EchoFactory",factoryRef);
```

Changing the Client

The client now has the additional job of requesting one or more Echo instances. But first, it needs a reference to the remote factory object from the registry:

```
EchoFactory remoteFactory =
(EchoFactory)Naming.lookup(url);
```

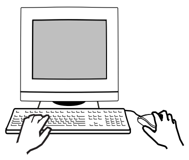
It then applies the getEcho method several times using different greeting strings and stores the references it gets back:

```
Echo echoRef1 = remoteFactory.getEcho("Hello");
Echo echoRef2 = remoteFactory.getEcho("Hi");
Echo echoRef3 = remoteFactory.getEcho("Howdy");
```

The sayEcho method is then called on these instances and the resulting string is displayed on the client console:

```
String reply = echoRef2.sayEcho(args[1]);
   System.out.println(reply);
reply = echoRef1.sayEcho(args[1]);
   System.out.println(reply);
reply = echoRef3.sayEcho(args[1]);
   System.out.println(reply);
```

Exercise: Object Factory



Exercise objective – Implement the changes described in the Object Factory section.

Tasks

Modify the Echo Application to Match the Setup Described in the Object Factory Section

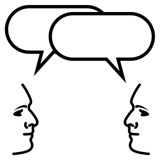
Complete the following steps:

1. Change the directory to `labfiles/mod3-rmi/lab/factory.`
2. The goal is to build an object factory that delivers `Echo` instances doing the same thing as the original `Echo` servants: they complete a string and send it back to the client. To achieve this, some of the existing files you must modify, and write two additional files: a factory interface (`EchoFactory.java`) and its implementation (`EchoFactoryImpl.java`). Consider which of the original files you must modify.
3. Create the factory interface (`EchoFactory.java`). It should consist of single method `getEcho(String greeting)` that returns a variable of type `Echo` and takes as parameter a `String` variable `greeting`. Do not forget to throw the `RemoteException`. Later, this method is called with different greetings to get specialized remote `Echo` objects.
4. Create the implementation for the new factory interface. Take the `EchoImpl.java` as a model. Name the new file `EchoFactoryImpl.java.` The goal of the `getEcho` method is to create new instances of objects that implement the original `Echo` interface and to return a handle to this new instance.
5. Add another constructor to the `EchoImpl.java` file that accepts the greeting string when an instance is created by the `getEcho` method of the object factory.

6. Change the old `EchoServer.java`. It should now start the factory, and no longer directly start an `Echo`.

7. Enable `EchoClient.java` to get the remote factory object from the registry and then apply the `getEcho` method several times using different strings, such as "Hello" and "Ciao" to have the factory create some objects. As in the original `EchoClient` code, the `sayEcho` method is called on these instances and the resulting string is displayed on the console.

Exercise Summary



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

✓ ***Manage the discussion here based on the time allowed for this module, which was given in the "About This Course" module. If you find you do not have time to spend on discussion, then just highlight the key concepts students should have learned from the lab exercise.***

- Experiences

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

- Interpretations

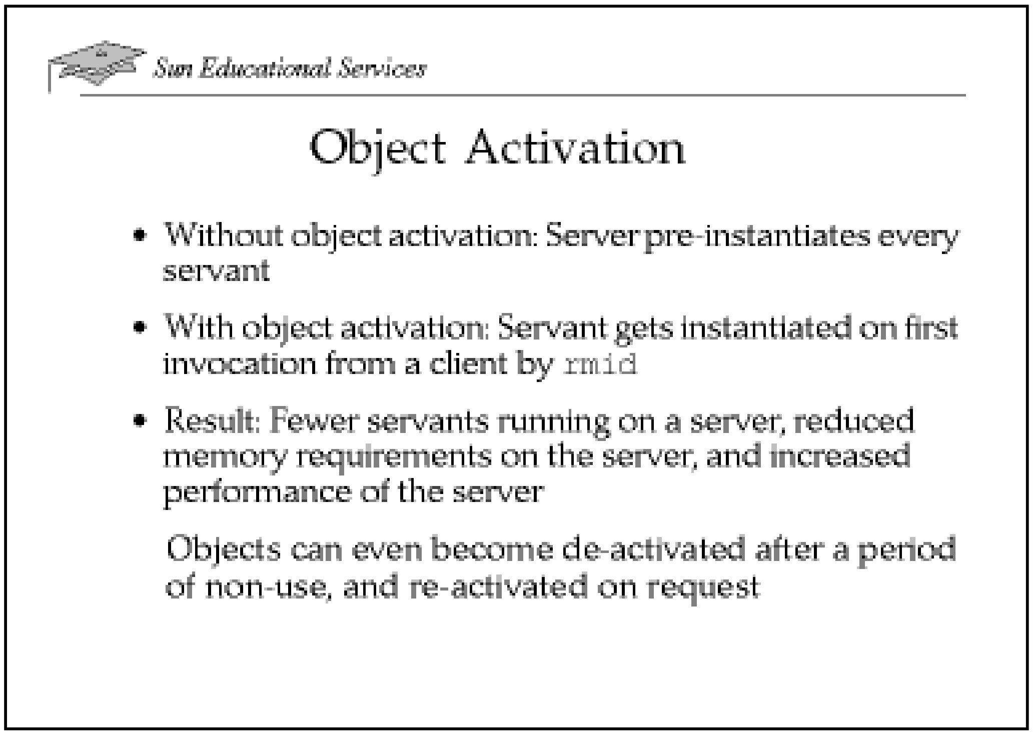
✓ ***Ask students to interpret what they observed during any aspects of this exercise.***

- Conclusions

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

- Applications

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


Sun Educational Services

Object Activation

- Without object activation: Server pre-instantiates every servant
- With object activation: Servant gets instantiated on first invocation from a client by `rmid`
- Result: Fewer servants running on a server, reduced memory requirements on the server, and increased performance of the server

  Objects can even become de-activated after a period of non-use, and re-activated on request

Object Activation

Before the release of the Java 2 SDK, a remote object had to be instantiated and running on the server before it could be used. This can be inefficient, because not all objects are used all the time. For example, a remote object that is used only on Sundays should not be required to be running seven days per week. The Java 2 SDK changes this. There is a new remote object class, which complements `UnicastRemoteObject`: `java.rmi.activation.Activatable`. An object of this type can be activated (instantiated), deactivated (potentially serialized), and reactivated (deserialized) on demand. All this is completely transparent to the client: as far as the client is concerned, all objects are constantly active, just like before.

Before activation, a server was required to instantiate and register the remote object (servant). With activation, the remote object is no longer instantiated right away. Instead, *information* about the remote object is generated and registered. After doing this, the server can exit. Because the server is not actually a server anymore (it does not serve remote requests, it just sets everything up), it is usually called `XxxSetup`, not `XxxServer`. The activation, deactivation, and reactivation of remote objects is handled by the new RMI daemon, `rmid`.

The Activatable Version of `Echo`

To convert the `UnicastRemoteObject` version of `Echo` to an `Activatable` version, you must change the servant (`EchoImpl`) class. The server (`EchoServer`) in its present form is not needed anymore. Instead, you need a class to set up the activation system. To indicate the new role of the server, rename it `EchoSetup`. The client (`EchoClient`) does not have to be changed, because activation is a server-side issue.

Modifying `EchoImpl`

To modify `EchoImpl`, complete the following steps:

1. Make the appropriate imports in the implementation class.

```
import java.rmi.*;
import java.rmi.activation.*;
```

2. Modify the class declaration so that the class now extends from `java.rmi.activation.Activatable.`

```
public class EchoImpl
extends Activatable implements Echo {
```

3. Remove or comment out the old no-argument constructor.

```
//public EchoImpl throws RemoteException {};
```

4. Declare a two-argument constructor in the implementation class.

```
public EchoImpl(ActivationID id, MarshalledObject data)
throws RemoteException {
// Register the object with the activation system
// then export it on an anonymous port
super(id, 0);
}
```

Modifying EchoServer *to* EchoSetup

Unlike the RMI server class that must stay alive as long as the implementation needs to be made available, the job of the “setup” class is to create all the information necessary for the activatable class, without creating an instance of the remote object.

The setup class passes information about the activatable class to `rmid`, and registers a remote reference (an instance of the activatable class's stub class) and an identifier with the `rmiregistry`. The setup class then exits. There are six steps to creating a setup class:

1. Make the appropriate imports in the setup class.

```
import java.rmi.*;
import java.rmi.activation.*;
import java.util.Properties;
```

2. Install a SecurityManager.

```
System.setSecurityManager(new RMISecurityManager());
```

3. Create an `ActivationGroup` instance.

```
Properties props =
(Properties)System.getProperties().clone();
ActivationGroupDesc agd = new
ActivationGroupDesc(props,null);

ActivationGroupID agi =
ActivationGroup.getSystem().registerGroup(adg);
ActivationGroup.createGroup(agi,agd,0);
```

4. Create an `ActivationDesc` instance.

The job of the `ActivationDesc` is to provide all the information that `rmid` requires to create a new instance of the implementation class. This information consists of several parts:

- The class name passed as a String argument.
- The location of remote code as described in a String, which takes on the form of a URL (this is not a URL class).
- A `MarshalledObject` instance. You can use the instance of `MarshalledObject` to pass configuration data to the servant. (Look at the constructor of EchoImpl to see how this is accomplished.) In the example, no special configuration data is needed, so this argument is going to be `null`.

This is all that is needed to create an `ActivationDesc`. The following is the code segment that creates this object:

```
String location = new String(“http://localhost:8080/”);
MarshalledObject data = null;

ActivationDesc desc =
new ActivationDesc("EchoApp.EchoImpl", location, data);
```

5. Remove the reference to the implementation class creation, declare an instance of your remote interface, and register the activation descriptor with `rmid`.

```
Echo echoRef = (Echo)Activatable.register(desc);
```

6. Bind the stub that was returned by the Activatable.register method to a name in the `rmiregistry`.

```
Naming.rebind("Echo", echoRef);
```

Running the Activatable Version of Echo

To run the activatable version of `Echo`, complete the following steps:

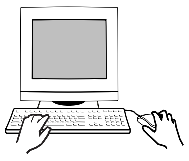
1. Start the `rmiregistry`. Ensure that the registry is started with no class path or that the class path does not include a path to any of the classes the client will download.

2. Start the activation daemon, `rmid`.

3. Run the setup program.

4. Run the client program.

Exercise: Object Activation



Exercise objective - Modify the server side of the `Echo` application to activate the `Echo` servant on demand, instead of at server startup.

Tasks

Modify the `Echo` *Application (*`EchoServer` *and* `EchoImpl`*)*

Complete the following steps:

1. Change the directory to `labfiles/mod3-rmi/lab/activate.`

2. In this directory, you will find the well-known `Echo` application. Change `EchoImpl` to be activatable, and change `EchoServer` to `EchoSetup`.

Start the Modified Echo Application

1. Start the activation daemon, `rmid`.

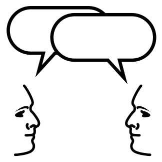
2. Run the ClassFileServer.

3. Run the setup program.

4. Start the client.

✓ ***See the README file in*** `/labfiles/mod3-rmi/solutions/activate`, ***which lists the necessary commands in greater detail.***

Exercise Summary



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

✓ ***Manage the discussion here based on the time allowed for this module, which was given in the "About This Course" module. If you find you do not have time to spend on discussion, then just highlight the key concepts students should have learned from the lab exercise.***

- Experiences

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

- Interpretations

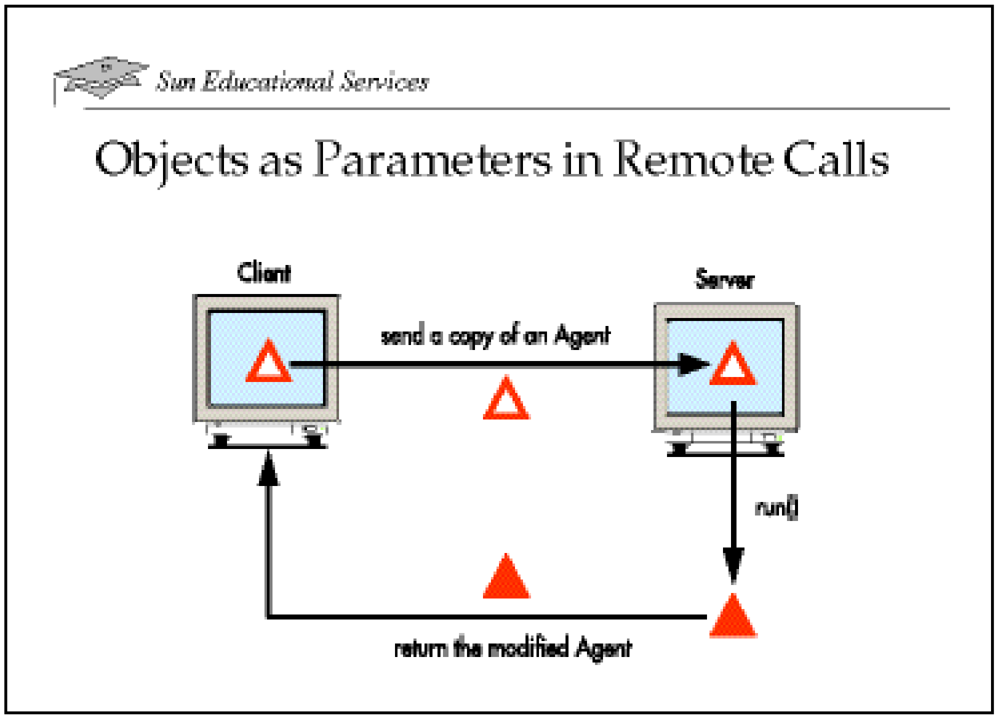
✓ ***Ask students to interpret what they observed during any aspects of this exercise.***

- Conclusions

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

- Applications

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



Objects as Parameters in Remote Calls

One of the unique features of RMI is that it can pass full Java objects to and from the server. Because RMI is 100 percent Java-to-Java, RMI can pass objects that are subtypes (or subclasses) of a declared parameter or return type. This allows RMI client applications to pass objects to the server that have not been previously defined.

RMI "Agents"

By exploiting this capability of leaving the actual implementation of an object until runtime, you can design an RMI system in which the server receives objects from clients and executes those objects on the client's behalf.

This approach to dynamically executing an object passed from one address space to another is known as the "command" design pattern.

To make an agent known to the server, declare something. A Java interface named `Agent.java` looks like the following:

```
package AgentApp;

public interface Agent {
     void run();
}
```

This interface describes a class that can do “something.”

To support the environment of the Agent, you must develop, in the usual way, an RMI system. First, examine the interface of the remote object (the servant), which is responsible for accepting the Agent, calculating it, and sending it back:

```
package WorkerApp;
import java.rmi.*;
import AgentApp.Agent;

public interface Worker extends Remote {
   Agent accept (Agent agent) throws RemoteException;
}
```

Not surprisingly, you will have a `WorkerServer`, a `WorkerImpl`, and a `WorkerClient` together with the worker interface. In addition, you will have one example of a class that implements the Agent interface: `CalcFactorial`.

The `WorkerServer` is not listed here. It is analogous to the `EchoServer` you already know. The `WorkerImpl` is a little different; it must trigger the calculation in the Agent it receives. The following is the code segment that deals with the Agent:

```
public Agent accept (Agent agent) {
   agent.run();
   return agent;
}
```

The servant calls the Agent’s `run` method and passes the modified `Agent` back as the return argument.

The `WorkerClient` method creates a new Agent (a `CalcFactorial`, which implements the Agent interface), and then calls the `accept` method of the servant.

```
CalcFactorial agent = new CalcFactorial(100);
agent = (CalcFactorial)workerRef.accept(agent);
System.out.println("Got the following
   result: " + agent.getResult());
```

The following is an example of the `CalcFactorial` class:

```
package AgentApp;

public class CalcFactorial implements Agent,
      java.io.Serializable {
   private int value;
   private double result;

   public CalcFactorial() { }

   public CalcFactorial(int value)
      { this.value = value; }

   public void run()
   {
      result = 1;
      for (int i=1; i<=value; i++)
      {
      result *= i;
      }
      //The text appears on the virtual
      //machine the agent is really
      //calculating on.
      System.out.println("\nCalculated the value "
      + result + " on this (virtual) machine!\n");
   }

   public double getResult () {
      return result;
   }
}
```

As the name indicates, the `CalcFactorial` class calculates the factorial of an `integer` and passes the result back in a value of type `double`.

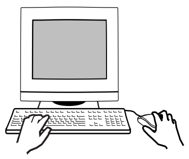
The following are the major steps that occur when you start up everything and run the client:

1. A new `CalcFactorial` class is created on the client.
2. The `CalcFactorial` class travels over the wire to the server (it is serialized on the client, sent to the server, and deserialized on arrival). The server is now in possession of an exact copy of the `CalcFactorial` class the client created.
3. On the server's copy of the `CalcFactorial` class, the method `run` is called. This call changes the state of the server's `CalcFactorial`: Its private `result` field is updated. The `CalcFactorial` on the server is no longer equal to the `CalcFactorial` on the client.
4. The modified `CalcFactorial` is sent back to the client, as the return argument. Again, the object is serialized, sent over the wire, and deserialized on the client. On the client, throw away the old `CalcFactorial`, assign the new one to the variable `agent`, and then print its `result`.

What is the outcome of this setup? You can to move a compute-intensive task to the server, and get the result back to the client.

You might wonder if, instead, it would have been simpler to make the Agent a remote interface, and create an `AgentImpl` class. But this method's setup is powerful enough to create additional classes that implement the Agent interface. Because a client takes the name of the Agent from the command line, you do not have to change anything to support the new agents. You do not have to recompile any of the `Worker` classes, stop and restart the RMI system, or install modified code on the server. The name `Worker` is appropriate; it is a generic service to "work" on compute-intensive tasks.

Exercise: Objects as Parameters in Remote Calls



Exercise objective – Use the provided Agent code and test whether anything can be gained by letting the Agent run on somebody else's machine.

Preparation

Team up with somebody else (preferably somebody with either a slower or a faster machine than you have).

Tasks

Compile and Run the Provided Agent Application

Complete the following steps:

1. Change the directory to `labfiles/mod3-rmi/agent`.

2. Compile and run the provided application locally.

✓ ***Do not forget to start the httpsrv.ClassFileServer.***

✓ ***The directory*** `labfiles/mod3-rmi/agent` ***contains a README file with more explicit instructions and troubleshooting information.***

Run the Agent on Another Machine

Complete the following steps:

1. Let your agent run on the machine of your classmate.

2. Modify your *client* to produce an indication of the time spent sending the Agent off and let it calculate (use `System.currentTimeMillis` to produce a timestamp). You probably have to modify your agent, or all you measure is the network delay (the calculation of the factorial is too fast). For example, you can repeat the calculation a couple of times in a secondary loop, to increase the time the `Agent` spends on the remote system.

   Can you see a performance gain by sending the Agent to somebody else's machine?

✓ ***The calculation had to be repeated 100,000 times to get a measurable result.Think About the Agent Interface***

3. Think about the `Agent` interface for a minute. Is it a good abstraction?

✓ ***It depends on how you look at it. If all you care about is the server, then the interface is okay. It describes everything a server must know about an Agent. If you want to build a universal client as well, the interface is not good enough. Right now, the client has to "know" the intimate details of*** `CalcFactorial` ***to get at its result. A universal client would rely only to an interface as well. An expanded Agent interface would include methods for data retrieval, such as String*** `getResultAsText`***, Class*** `getResult`***, or maybe even a callback method such as*** `void paintResult(Graphics gfx, Rectangle rect)`***. An even better solution would be to develop two interfaces: one for usage on the server, and one for usage on the client. An Agent implementation would then have to implement both interfaces.***

   This exercise was about a "compute" server. Are there other possibilities for generic server types that accept objects of which they know only the interface?

✓ ***This goes back to the generic agent discussion of Module 1.***

Exercise Summary

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

✓ ***Manage the discussion here based on the time allowed for this module, which was given in the "About This Course" module. If you find you do not have time to spend on discussion, then just highlight the key concepts students should have learned from the lab exercise.***

- Experiences

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

- Interpretations

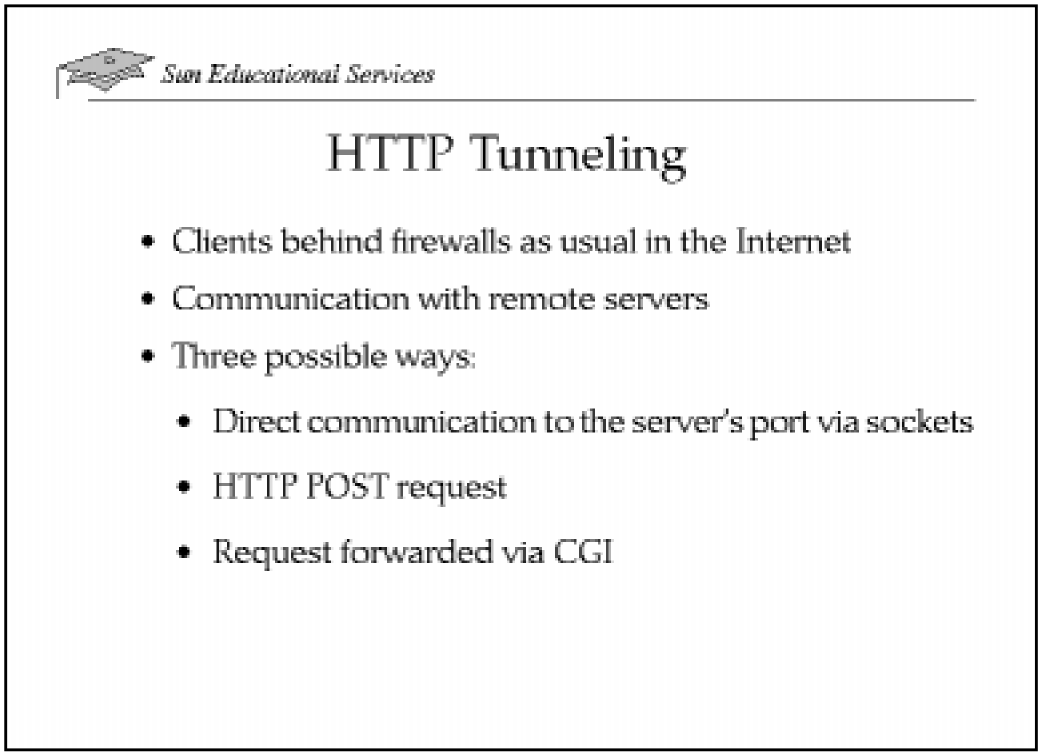
✓ ***Ask students to interpret what they observed during any aspects of this exercise.***

- Conclusions

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

- Applications

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


Sun Educational Services

HTTP Tunneling

- Clients behind firewalls as usual in the Internet
- Communication with remote servers
- Three possible ways:
  - Direct communication to the server's port via sockets
  - HTTP POST request
  - Request forwarded via CGI

HTTP Tunneling

RMI provides a means for clients behind firewalls to communicate with remote servers. This allows you to use RMI to deploy clients on the Internet, such as in applets available on the World Wide Web.

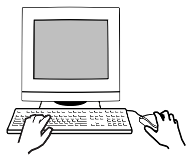
Traversing the client's firewall can slow down communication, so RMI uses the fastest successful technique to connect between client and server. The technique is discovered by the reference to `UnicastRemoteObject` on the first attempt the client makes to communicate with the server. It tries each of the following three possibilities:

- Communicate directly to the server's port using sockets. If this fails, build a URL to the server's host and port and use an HTTP POST request on that URL, sending the information to the skeleton as the body of the POST. If successful, the results of the post are the skeleton's response to the stub.
- If this also fails, build a URL to the server's host using port 80 (the standard HTTP port) using a CGI script that forwards the posted RMI request to the server.

✓ ***The client application may disable the packaging of RMI calls as HTTP requests by setting the*** `java.rmi.server.disableHTTP` ***property to be true.***

✓ ***By default only port 80 is used.***

Exercise: Developing an RMI Application From Scratch (Optional)



Exercise objective – Develop an entire RMI application from scratch. No specific templates are provided, but you can use the `Echo` files as a guideline. Only the interface of the remote object is provided.

Tasks

Implement and Run a Remote Counter Application

Take the `Echo` files as templates to build a complete RMI application based on the interface named `RemoteCount.java`. The client will call a remote method a given number of times and measure the time between invoking a call and getting back an answer. The average time is then displayed on the console.

1. Change the directory to `labfiles/mod3-rmi/remotecount.`

2. Implement the server and the servant, based on the interface `RemoteCount.java`. The methods in the interface have the following purposes:

   - `sum` returns an `int` value showing how many times the remote method `increment` was invoked. It should be called after the iterations have finished.

   - `setZero` does not return any value. It sets the increment variable back to 0 and should be called before a new series of loops start.

   - `increment` increments a counter on the server by 1.

3. Develop a client that you can start with a command, such as:

```
java Count.RemoteCountClient localhost 1000
```

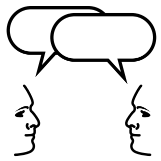
This calls the remote method 1000 times. The client console should then look like the following:

```
sum = 0
Now calling the remote method
Average Time for a method call = 1.7 milliseconds
```

The client measures the time at the beginning of the loop and at the end. Dividing the time measured by the number of loops returns the average time a remote call takes. The method for getting the current time is `System.currentTimeMillis`.

4. Team up with your neighbor to make the remote calls on another machine and to see if there are differences in time.

Exercise Summary



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

✓ ***Manage the discussion here based on the time allowed for this module, which was given in the "About This Course" module. If you find you do not have time to spend on discussion, then just highlight the key concepts students should have learned from the lab exercise.***

- Experiences

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

- Interpretations

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

- Conclusions

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

- Applications

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

Check Your Progress

Before continuing on to the next module, check that you were able to accomplish the following in this module:

❑ Describe the RMI architecture, its layers and garbage collection

❑ Implement an RMI server and client in the Java programming language

❑ Generate client stubs and skeletons for remote services using the RMI stub compiler

❑ Define the RMI registry and describe how it works

❑ Explain the security issues related to RMI

Think Beyond

What applications do you have that could take advantage of RMI?

Java Interface Definition Language (JavaIDL) 4

Objectives

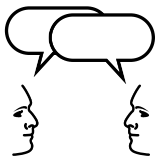
Upon completion of this module, you should be able to:

- Describe the basic CORBA object management architecture
- Describe the role of JavaIDL in relation to other commercial Java CORBA products
- Create and deploy a JavaIDL server object and a JavaIDL client application
- Describe how the JavaIDL bootstrapping process works
- Describe how IDL is mapped to the Java programming language
- Explain why the RMI Agent example does not work with the current CORBA

JavaIDL adds CORBA (Common Object Request Broker Architecture) capability to the Java platform, providing standards-based interoperability and connectivity. Runtime components include a fully compliant Java ORB for distributed computing using Internet inter-operability protocol (IIOP) communication.

Relevance

✓ ***Present the following questions to stimulate the students and get them thinking about the issues and topics presented in this module. They are not expected to know the answers to these questions. The answers to these questions should be of interest to the students, and inspire them to learn the content presented in this module.***



Discussion – Consider the following questions:

- How would you wrap existing legacy code (C, C++, COBOL)?

✓ ***This is the primary function of CORBA. This technology is exemplified by JavaIDL. CORBA allows problems to be modeled in a single language (IDL) and then compiled to other languages.***

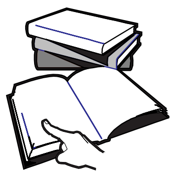
- How does a system publish its services in such a way that any client can request any service?

✓ ***This is the job of the ORB—to provide a central naming service for all services.***

✓ ***This module covers JavaIDL. All of the IDL mapping and features described are supported by JavaIDL. There are other features that are supported by other implementations of CORBA—this document is intended only to introduce students to JavaIDL.***

✓ ***JavaIDL supports IIOP, a generic ORB core that communicates via IIOP; an idltojava "compiler" that generates both stub and skeleton code; and an implementation of the CORBA Object Service (COS) Naming service called tnameserv.***

Additional Resources



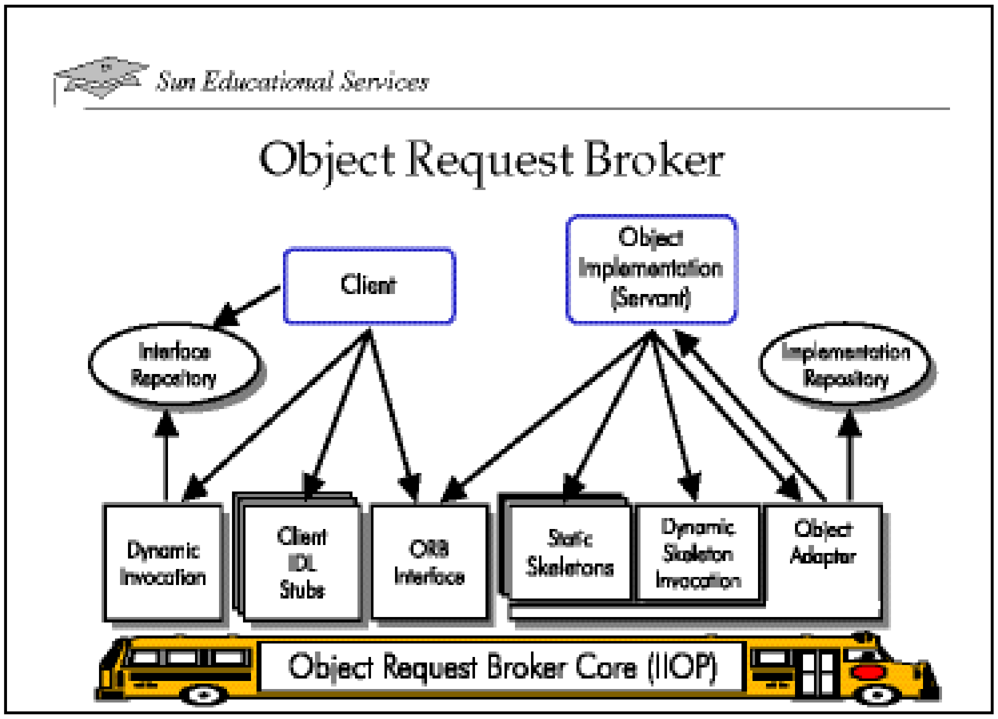
Additional resources – The following resources can provide additional detail on the topics presented in this module:

- The Object Management Group. [Online]. Available: `http://www.omg.org/`
- *Mapping of OMG IDL to Java*. [Online]. Object Management Group. Available: `ftp://www.omg.org/pub/docs/formal/98-02-29.pdf`
- *ORB Portability Java Language Mapping*. [Online]. Object Management Group. Available: `ftp://ftp.omg.org/pub/docs/orbos/97-11-12.pdf`
- *Revised ORB Portability IDL/Java Language Mapping*. [Online]. Object Management Group. Available: `ftp://ftp.omg.org/pub/docs/orbos/98-01-06.pdf`
- *Java to IDL Mapping*. [Online]. Object Management Group. Available: `ftp://ftp.omg.org/pub/docs/orbos/98-02-01.pdf` `ftp://ftp.omg.org/pub/docs/orbos/98-03-08.pdf` (errata to the above document)
- *Objects By Value*. [Online]. Object Management Group. Available: `ftp://ftp.omg.org/pub/docs/orbos/98-01-01.pdf`
- *The Portable Object Adaptor*. [Online]. Object Management Group. Part of the complete CORBA 2.2 specification. Available: `ftp://www.omg.org/pub/docs/formal/98-02-14.pdf`
- RMI over IIOP: JavaOne session slides. [Online]. Sun Microsystems, Inc. Available: `http://java.sun.com/javaone/javaone98/sessions/T406/kgh1.htm`
- Orafali, Robert and Dan Harkey. *Client/Server Programming with Java and CORBA Second Edition*. 1998. Wiley Computer Publishing.
- Lewis, Geoff, Steven Barber, and Ellen Siegel. *Programming with JavaIDL*. 1997. Wiley Computer Publishing.

Module Overview

CORBA enables you to bridge the gaps among different programming languages, hardware architectures, operating systems, and locations. In this module, you will look at how to build a Java CORBA server object, and a Java CORBA client application that talks to the server object, using JavaIDL. In theory, the client application could also talk to an equivalent C++ CORBA server object, and the server object could talk to the client application. However, doing that is not included in this course.

This module includes a brief look at the core of the Object Management Architecture (OMA): the Object Request Broker (ORB), and JavaIDL.



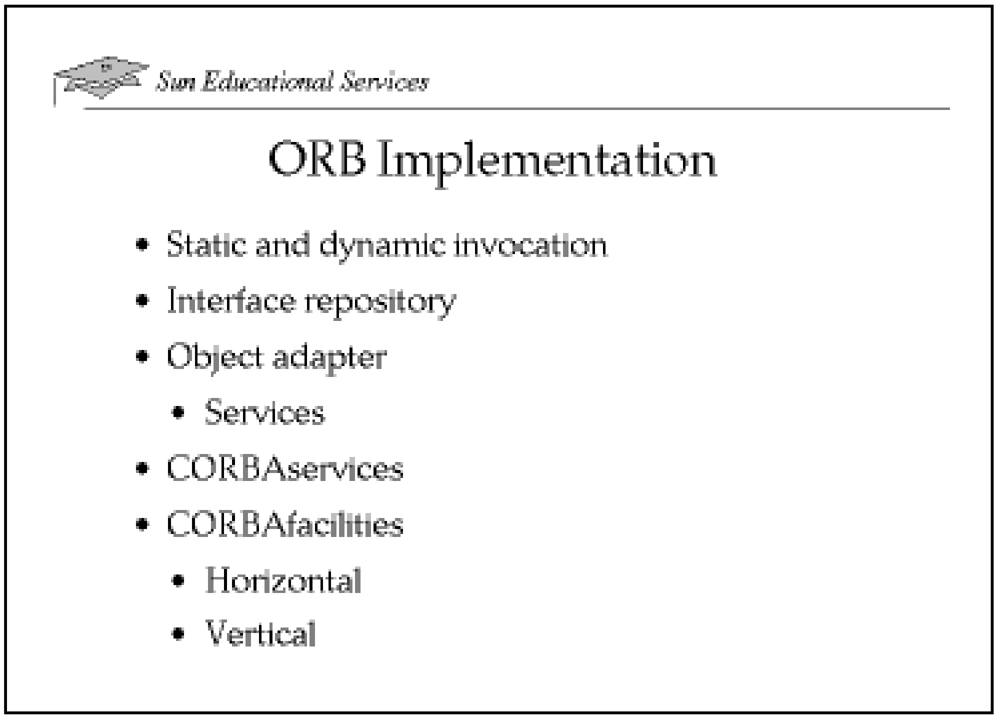
Object Request Broker

The ORB is the core of the reference model. It provides the communications infrastructure that enables objects to transparently make and receive requests and responses in a distributed environment. It is the foundation for building applications from distributed objects and for achieving interoperability between applications in heterogeneous environments.

The ORB forms a bridge that eliminates differences in location, platform, and programming language. ORBs use a standard communications protocol specified by the General Interoperability Protocol (GIOP). However, this course does not define the transport mechanism, so the OMG also created a specification for communications over TCP/IP, the Internet Inter-ORB Protocol (IIOP).

Consistent with its focus on interfaces, CORBA does not specify how the ORB is to be implemented; it can be a daemon process, library, or combination thereof.

Note – The actual degree of involvement of the ORB in service requests and responses is not specified by the standard. In some implementations the ORB is involved only in the initial establishment of the communications link and not in subsequent method invocations.


Sun Educational Services

ORB Implementation

- Static and dynamic invocation
- Interface repository
- Object adapter
  - Services
- CORBAservices
- CORBAfacilities
  - Horizontal
  - Vertical

ORB Implementation

Static and Dynamic Invocation

CORBA defines two mechanisms for invoking object methods: static invocation and dynamic invocation. Static invocation is limited to the interfaces known when the client code is compiled. Dynamic invocation allows the client to discover interfaces at runtime by interrogating the ORB. This is a powerful feature and is vital for building flexible systems that can respond to frequent changes in the real world.

Interface Repository

The key feature that makes dynamic invocation possible is the interface repository. In its simplest form, this is a database of interfaces populated by the IDL compiler. Architecturally speaking, it is one of the most critical components in CORBA, as it contains most of the "metadata" for the whole federation of objects. It is also involved in type-checking of method invocations, as well as referencing interfaces and methods across ORBs.

✓ ***Dynamic invocation relies on the use of the Any type. These parameters are type-checked at runtime by interrogating the interface repository. The current JavaIDL does not provide support for an interface repository, and the Any type is beyond the scope of this course.***

Object Adapter

In general, an adapter allows objects with incompatible interfaces to communicate. CORBA defines the object adapter as an ORB component that provides object reference, activation, and state-related services to an object implementation. These include:

- Registering implementation objects with the ORB
- Interpreting and translating object references
- Locating implementation objects
- Activating and deactivating implementation objects
- Invoking methods

✓ ***Explain that the object adapter, represents a basic design pattern of OO software engineering. Like an electrical adapter, it is used to connect interfaces which do not communicate directly.***

To perform these tasks, the object adapter maintains an implementation repository for storing information that describes object implementations.

While an ORB must have, at a minimum, the basic object adapter (BOA) specifically defined by CORBA, it can also have special-purpose object adapters. You can use a specialized object adapter, for example, to implement persistence.

CORBAservices

CORBAservices support basic functions for using and implementing distributed applications and are independent of application domains. For example, the Lifecycle Service defines interfaces for creating, deleting, copying, and moving objects; it does not dictate how the objects are implemented in an application. Refer to the OMG document *CORBAservices* for discussion of the services defined by CORBA.

CORBAfacilities

This is a collection of shared services at a higher architectural level than the CORBAservices.

In general:

- Horizontal facilities can be shared across many applications, such as printing and electronic mail.
- Vertical facilities apply to a specific business domain, such as finance or manufacturing.

A number of facilities are defined by OMG (see the document *CORBAfacilities*).

Horizontal Facilities

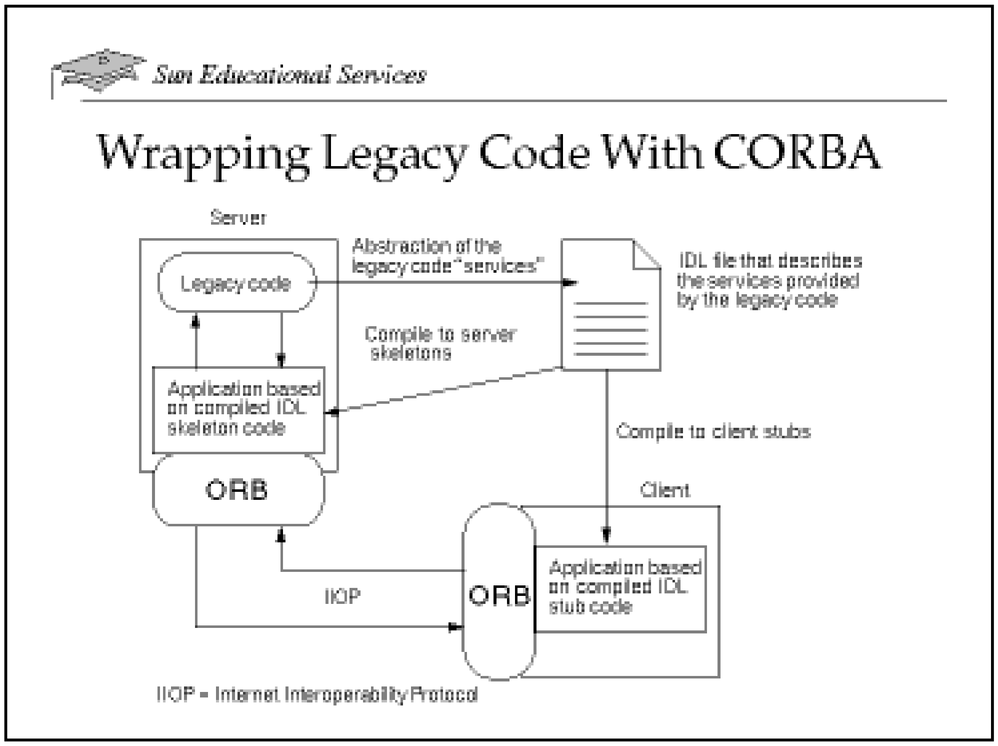
The following lists the horizontal, or generic, facility categories:

- **User interface facilities** include Compound Presentation, Desktop Management, Rendering Management, Scripting, and User Support Facilities.
- **Information management facilities** include Compound Interchange, Data Encoding and Representation, Data Interchange, Information Exchange, Information Modeling, Information Storage and Retrieval, and Time Operations Facilities.
- **System management facilities** include Collection Management, Consistency, Customization, Data Collection, Event Management, Instance Management, Instrumentation, Policy Management, Process Launch, Quality of Service Management, Scheduling Management, and Security Facilities.
- **Task management facilities** include Agent, Automation, Rule Management, and Workflow Facilities.

Vertical Facilities

The following lists the vertical CORBAfacilities that are currently identified by OMG in the following areas:

- Accounting
- Application development
- Computer-integrated manufacturing
- Currency
- Distributed simulation
- Information superhighways
- Internationalization
- Mapping
- Oil and gas exploration and production
- Security
- Telecommunications


Sun Educational Services
Wrapping Legacy Code With CORBA
Server
Legacy code
Abstraction of the legacy code "services"
IDL file that describes the services provided by the legacy code
Compile to server skeletons
Application based on compiled IDL skeleton code
Compile to client stubs
ORB
Client
Application based on compiled IDL stub code
ORB
IIOP
IIOP = Internet Interoperability Protocol

Wrapping Legacy Code With CORBA

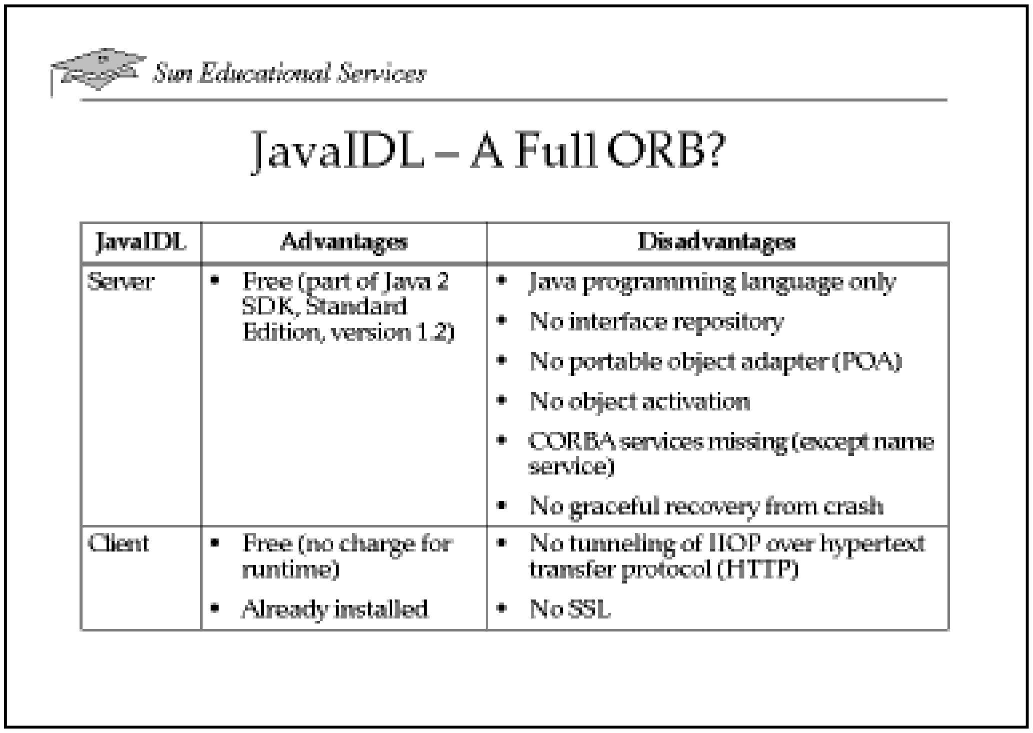
One of the primary features of CORBA is the support for multiple programming languages across multiple platforms. CORBA vendors have had time to create IDL compilers for a variety of platforms and languages, and have sold a number of solutions for "wrapping" a large body of existing code with a CORBA implementation.

- Legacy code is modeled using IDL. This file represents the "contract" of the services the server implementation (legacy code) provides. Some compromises might have to be made to make the IDL work.

- On the server side, the IDL is compiled and "linked" into the legacy code. This can happen by calling legacy libraries, or, in some cases, compiling the legacy code directly into the skeletons created by the IDL compiler.

- On the client side, the IDL is compiled to the programming language desired, such as Java, and the client application can make calls to the methods that represent the legacy "services."

- On both sides, the communication between the skeleton code and stub code and the ORB is transparent. The ORB code is designed to insulate the client/server from the specifics of the object implementation.


Sun Educational Services

JavaIDL – A Full ORB?

JavaIDL	Advantages	Disadvantages
Server	• Free (part of Java 2 SDK, Standard Edition, version 1.2)	• Java programming language only • No interface repository • No portable object adapter (POA) • No object activation • CORBA services missing (except name service) • No graceful recovery from crash
Client	• Free (no charge for runtime) • Already installed	• No tunneling of IIOP over hypertext transfer protocol (HTTP) • No SSL

JavaIDL in Relation to CORBA

JavaIDL – A Full ORB?

JavaIDL is an Object Request Broker (ORB) provided with the Java 2 SDK. You can use it to define, implement, and access CORBA objects from the Java programming language. JavaIDL is compliant with the CORBA/IIOP 2.0 specification and the IDL-to-Java programming language mapping.

The JavaIDL ORB supports transient CORBA objects (objects whose lifetimes are limited by their server process's lifetime). JavaIDL also provides a transient nameserver to organize objects into a tree-directory structure. The nameserver is compliant with the *Naming Service Specification* described in CORBAservices.

Is a Commercial ORB Necessary?

At first glance, JavaIDL seems to be a complete ORB, together with an implementation of the *OMG Naming Service*. The obvious question is therefore: Is JavaIDL all that is necessary or do you still need a commercial ORB? The answer is not simple. You need to compare JavaIDL with the current Java programming language-compliant commercial ORBs. The following paragraphs look at JavaIDL, both the client and the server parts.

JavaIDL on the Client

JavaIDL is a good choice on the client, as long as you do not need any special features: tunnelling of IIOP through HTTP (to circumvent firewalls), and using SSL to encrypt the IIOP datastream. If you need these features, you must switch to another ORB on the client. RMI supports both features (SSL since Java 2 SDK), so there is a chance that JavaIDL will support tunnelling and SSL as well.

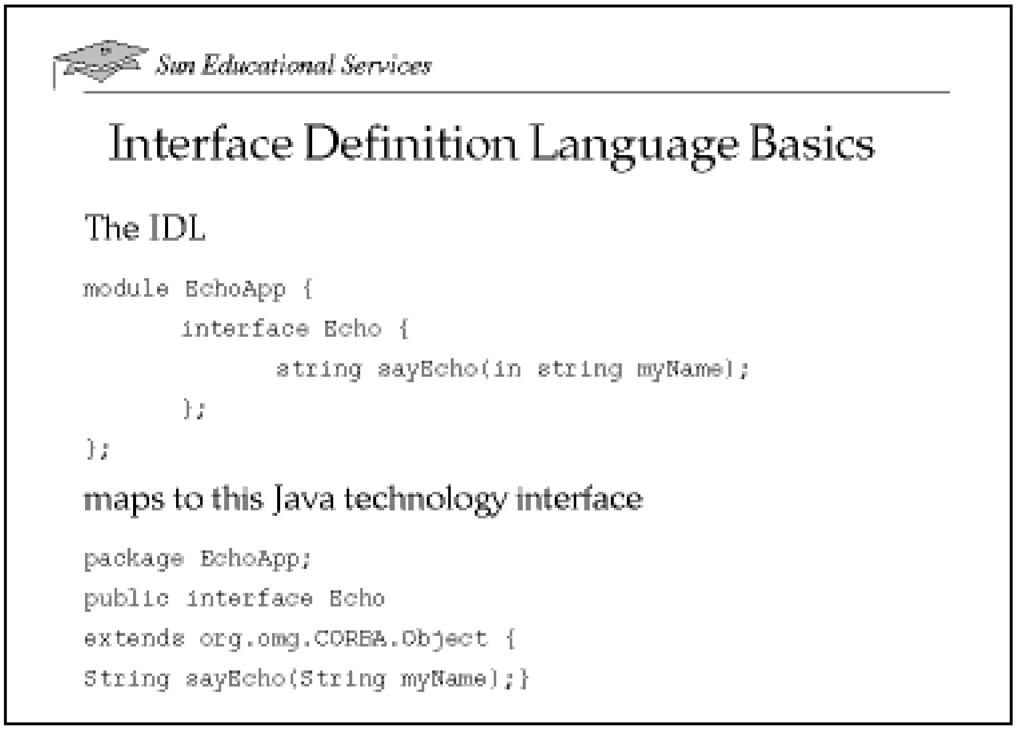
JavaIDL on the Server

For developing, and many simple purposes, JavaIDL can be a good choice on the server. However, the list of unsupported features is significantly longer on the server than it was on the client. It includes the following:

- Objects written in a programming language other than Java
- An interface repository
- Portable object adapter (POA)
- Transparent object activation
- Any CORBAservice besides the naming service
- A persistent naming service
- A more robust server, which can gracefully recover from a crash (by periodically persisting the state of all servants running, and by supporting transactions)

In a nutshell, you should use a commercial ORB on the server for production. On the client, JavaIDL is good enough most of the time, and has the additional advantage of being already installed (if the clients are on Java 1.2.)

✓ ***An experienced programmer who does not use the Java programming language and is a CORBA developer might ask if one can deploy an applet that was developed using Visibroker onto a JavaIDL client. With C++, this is not possible: an application that is intended to run on a Orbix runtime cannot be delivered with Visibroker stubs. With the Java programming language, the same thing is usually possible. A "binary compatible stub" standard was agreed on. You will only encounter problems if your applet uses services that are not available on the client runtime; for example, the COS naming service of JavaIDL is not available on the Netscape Communicator built-in CORBA runtime. For such a case, it is possible to download and replace the ORB itself at runtime, using an applet parameter.***



Sun Educational Services

Interface Definition Language Basics

The IDL

```
module EchoApp {
      interface Echo {
            string sayEcho(in string myName);
      };
};
```

maps to this Java technology interface

```
package EchoApp;
public interface Echo
extends org.omg.CORBA.Object {
String sayEcho(String myName);}
```

Interface Definition Language Basics

After learning the basics, you can start with the first step of creating your own CORBA application: declaring the interface of your server object. With RMI, you used the Java programming language to do this; you declared a Java interface that described the server object.

CORBA uses a specially designed language to accomplish the same task: Interface Definition Language (IDL). A compiler (`idltojava`) later translates IDL-to-Java code, generating (among other items) a Java interface that you can use on the client in much the same way you used the Java interface with RMI.

The IDL standard, like a Java interface, declares only the *interface* of the server objects, abstracting all of the implementation details.

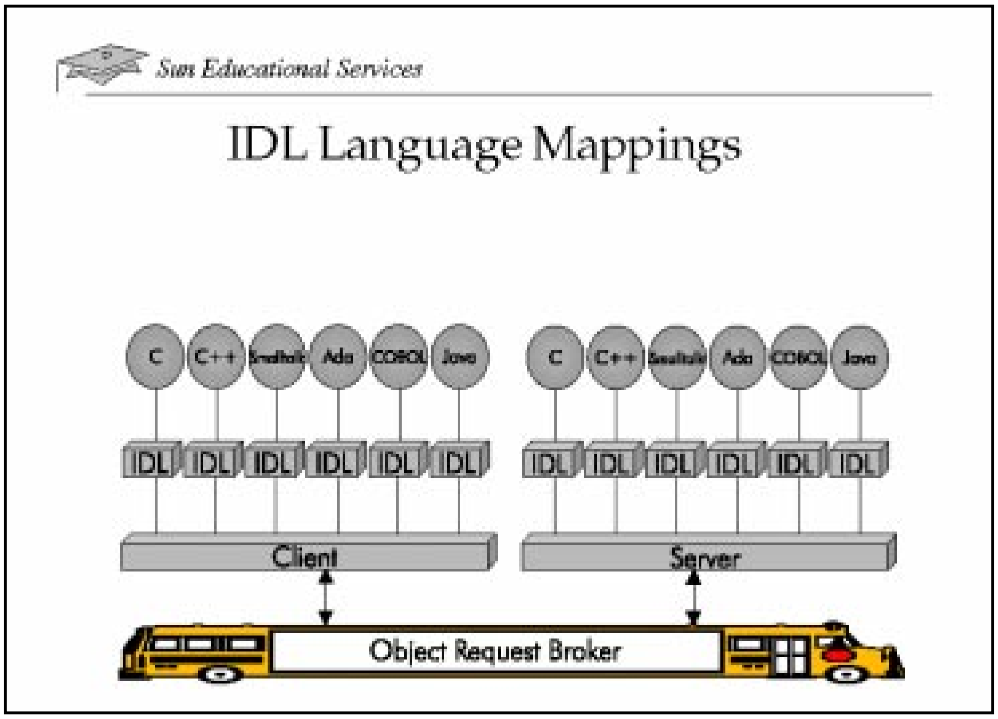
For example:

```
module EchoApp {
   interface Echo {
      string sayEcho(in string myName);
   };
};
```

This interface definition is passed to the IDL compiler, which generates the following:

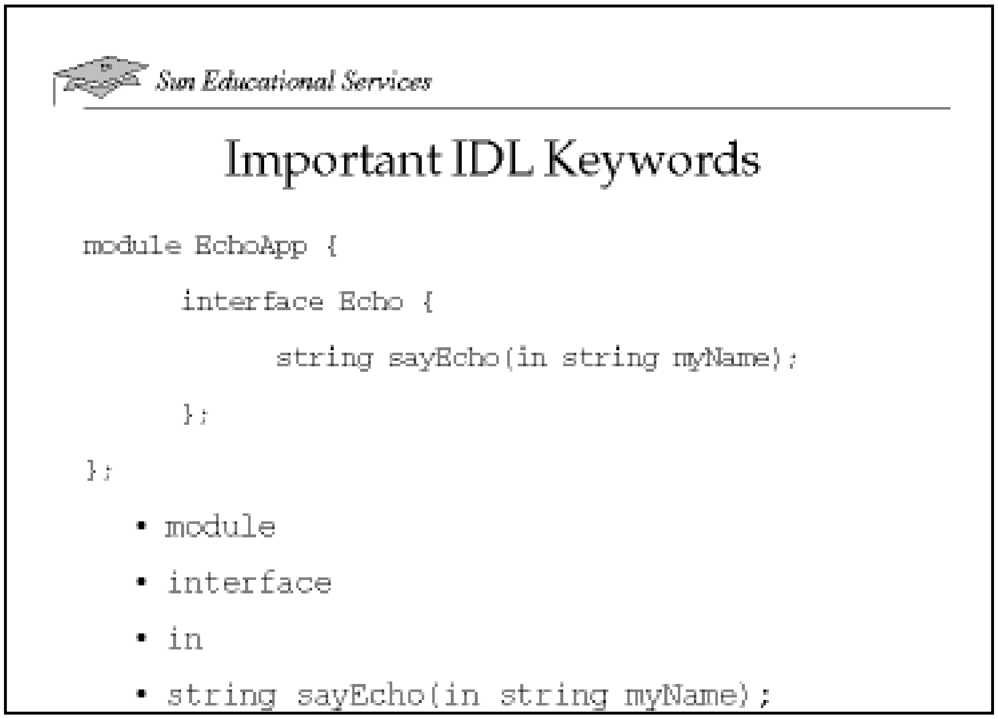
- `Echo.java` – This is the Java interface mirroring the IDL interface using the Java programming language syntax.
- `EchoHelper.java` – This object is primarily used to cast a generic `org.omg.CORBA.Object` to the `Echo` type. (You will see later why you cannot cast the usual way.)
- `EchoHolder.java` – This object is used if an `Echo` object is being passed as an `out` or `inout` parameter in a method call. You will see more about holders later on.
- `_EchoImplBase.java` – This object is extended by the actual `Echo` implementation. `_EchoImplBase.java` provides code for the server side to manage interaction between the ORB and the actual `Echo` object.
- `_EchoStub.java` – This object is implicitly used by the client. It implements the interface `Echo`. It provides code for the client to allow it to invoke the method `sayEcho` using local Java programming language semantics.

All of the networking code required for making and responding to requests is automatically generated by the IDL compiler, and the client and server code can be *independently* targeted for any language for which the ORB vendor supplies an IDL mapping**.**



IDL Language Mappings

Currently there are several language mappings available and more will be added when the need arises. The OMG has standard mappings for the following programming languages: C, C++, Smalltalk, COBOL, Ada '95, and Java. Most of the existing commercial ORBs support generation of both client and server code.



Sun Educational Services

Important IDL Keywords

```
module EchoApp {
      interface Echo {
            string sayEcho(in string myName);
      };
};
```

- `module`
- `interface`
- `in`
- `string sayEcho(in string myName);`

Important IDL Keywords

So that you can start coding soon, this section includes only the most important IDL keywords. More IDL details are covered after you have deployed your first JavaIDL application.

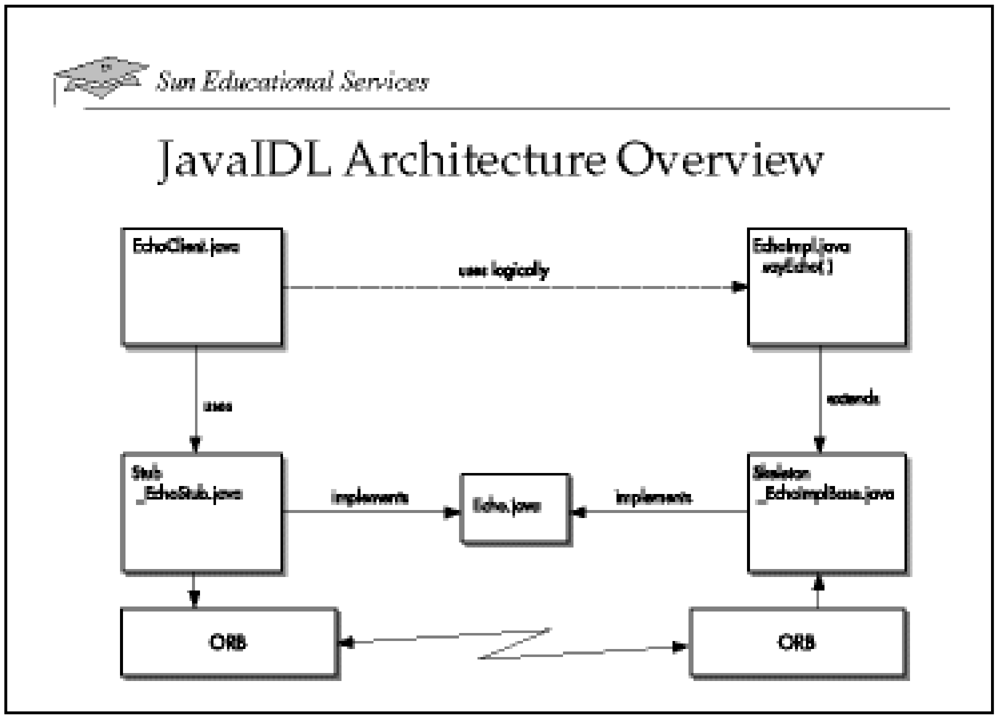
The following is the simple object again:

```
module EchoApp {
   interface Echo {
      string sayEcho(in string myName);
   };
};
```

Look at these keywords:

- `module` – This keyword describes a naming context, in much the same way a Java programming language package ("Java package") does. The `idltojava` compiler translates a module to a Java package.
- `interface` – This keyword describes the interface of a server object. It maps directly to a Java interface with the same name.
- `string` – IDL, like the Java programming language, has a range of primitive datatypes. In IDL, `string` is one of them (unlike Java, where String is a class). Nonetheless, an IDL string maps to a Java String.
- `in` – This keyword denotes that the parameter *`myName`* travels one way from the client to the server.
- `string sayEcho(in string myName)` – While not a keyword, this whole sentence is the declaration of a method named `sayEcho`. This declaration is mapped to a Java method with the same name. Because string translates to String, the full Java programming language method ("Java method") declaration is `String sayEcho(String myName)`.

This quick IDL introduction is enough to enable you to create a JavaIDL server and client.


Sun Educational Services

JavaIDL Architecture Overview

JavaIDL Architecture Overview

JavaSoft's JavaIDL generates both stub and skeleton code from an IDL file by compiling the IDL file with `idltojava`. The IDL compiler for JavaIDL generates a set of stubs and skeletons that provide functionality for static invocation on CORBA objects.

The following describes the operation of a client/server CORBA system:

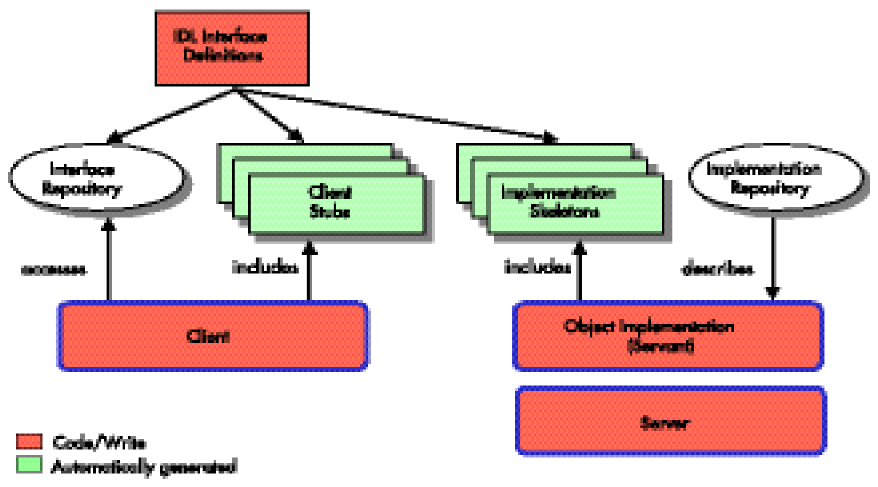
- On the client side, the client invokes a method on an object that is "stubbed" to appear local to the client. The stub tells the client's ORB what arguments to marshal and tells the ORB to invoke the CORBA object identified by the remote reference. The ORB "knows" how to marshal arguments, but not which arguments to marshal for a particular method.
- The ORB uses the object reference to determine which remote ORB should receive the request, passes the marshalled parameters over the wire to the ORB, and waits for a result.

- On the server side, the ORB receives the request, inspects the object reference to determine which server contains the object, and invokes the skeleton for that object. The skeleton tells the ORB how to unmarshal the parameters, and invokes the server implementation of the method requested.

- The server method returns a result to the skeleton, which arranges how to marshal the result with the server-side ORB and returns it to the stub. The stub contacts its ORB to unmarshal the result and return it to the client.

Sun Educational Services

Creating a JavaIDL Application



Creating and Deploying a JavaIDL Application

Creating a JavaIDL Application

There are six major steps involved in creating the JavaIDL application (client and server) from the IDL specification:

1. Develop or acquire the IDL.
2. Compile the IDL using the JavaIDL `idltojava` compiler.
3. Create the implementation class (the *servant*) for the interface defined in the IDL.
4. Create the *server*, which manages the *servant* instances.
5. Create the *client*, which uses the remote object (in the end, the *client* uses the *servant*).
6. Create the Java class files. Use the Java technology compiler ("Java compiler") `javac` to create the Java technology class ("Java class") files for all of the Java technology files ("Java files").

Step 1: Develop or Acquire the IDL

The IDL file is the contract between the server object and the client application. It is usually developed at the time a new server object is developed. Therefore, in a productive CORBA environment, the IDL for existing server objects can usually be acquired, if you have to develop a matching Java client.

Step 2: Compiling the IDL

Assuming the JavaIDL directory is in your path, compile the IDL file by specifying what you want `idltojava` to do with the file. For example, the simplest use of the compiler is to generate both client code (stubs) and server code (skeletons), and not to use a preprocessor first (CPP).

```
idltojava -fno-cpp Echo.IDL
```

By default, `idltojava` generates the following Java files, in the `EchoApp` package:

```
_EchoImplBase.java
_EchoStub.java
Echo.java
EchoHelper.java
EchoHolder.java
```

Step 3: Create the Implementation Class (Servant)

For each interface, you must write a servant class. Instances of the servant class implement ORB objects. Each instance implements a single ORB object, and each ORB object is implemented by a single servant.

`idltojava` generates a base class `_<`*`interface_name`*`>ImplBase` for every IDL interface declared in the IDL file. This base class must be extended to create the servant.

The servant code therefore extends the `_EchoImplBase` class, and provides an implementation for all of the methods declared in the IDL file. In this case this is just one: `sayEcho`.

Despite the fact that the implementation classes are called *servants*, the usual naming convention is to append `Impl` to the Java interface name: `EchoImpl.java`.

```
package EchoApp;

class EchoImpl extends _EchoImplBase {
   public String sayEcho(String myName) {
      return "\nHello " + myName + "!!\n";
   }
}
```

Step 4: Create the Server

The server is what the ORB uses to instantiate and publish the servant. In its simplest form, the server pre-instantiates all the needed servants and keeps them alive forever (or at least as long as the server lives). In a production system this is usually not good enough. To keep thousands of servants alive, even when they are not actively used by clients, is a waste of resources. In this case, more complex servers must be developed and used. They must be able to instantiate servants on demand, and maybe even persist and discard them after a period of non-usage. The server, `EchoServer.java`, is a simple one:

```
package EchoApp;

import org.omg.CosNaming.*;
import org.omg.CORBA.*;


public class EchoServer {
   public static void main(String args[]) {
      try{
      // create and initialize the ORB
      // pass the command line arguments to it
      ORB orb = ORB.init(args, null);

      // create servant and register it with the ORB
      // it is now a CORBA object,
      // but not yet retrievable
      // via the COS name server.
      EchoImpl echoRef = new EchoImpl();
      orb.connect(echoRef);

      // get the remote reference to the
      // COS name server
```

```
     // Narrow it to the correct type.
     org.omg.CORBA.Object objRef =
     orb.resolve_initial_references("NameService");
     NamingContext ncRef =
     NamingContextHelper.narrow(objRef);

     // bind the object reference
     // to the name "Echo"
     NameComponent nc =
     new NameComponent("Echo", "");
     NameComponent path[] = {nc};
     ncRef.rebind(path, echoRef);
     System.out.println("Echo object
     ready and bound to the name 'Echo'!");
     System.out.println("It's object reference
     is: " + orb.object_to_string(echoRef));

     // now you have to stop, or the virtual machine
     // will exit, killing with it the newly
     // created Servant object.
     // Stop in a non CPU intensive way
     // by waiting forever
     java.lang.Object sync =
     new java.lang.Object();
     synchronized (sync) {
     sync.wait();
     }

     } catch (Exception e) {
     System.err.println("ERROR: " + e);
     e.printStackTrace(System.out);
     }
   }
}
```

As you can see, you not only have to instantiate the servant and register it with the ORB, but also register it with the name server. After the servant is registered, the client application can retrieve and use the servant's object reference. You will see how this works in the following section.

✓ ***Experienced CORBA developers might ask why this example does not have to use an Object adapter (BOA or POA). With a C++ server, you would have been forced to use one. The answer lies in the new IDL to Java technology binding standard: it declares in the ORB interface two new methods:*** `connect` ***and*** `disconnect`***. These two are used to connect and disconnect a transient servant to the ORB, without an object adapter in between.***

✓ ***This is different from RMI. You have to explicitly wait at the end of the*** `main` ***thread to avoid exiting the virtual machine. While the source code of RMI and JavaIDL has not been looked at, it is believed that the difference must be in the way the two systems use threads. RMI uses "regular" threads, while JavaIDL uses "daemon" threads. A Java virtual machine exits as soon as the last non-daemon thread finishes.***

Step 5: Create the Client Application

The following is an example of how to create a client application.

```
package EchoApp;
import org.omg.CosNaming.*;
import org.omg.CORBA.*;

public class EchoClient {
   public static void main(String args[]) {
      try {
      // create and initialize the ORB
      // pass the command line arguments to it
      ORB orb = ORB.init(args, null);

      // get the remote reference to the COS name
      // server. Narrow it to the correct type.
      org.omg.CORBA.Object objRef =
      orb.resolve_initial_references("NameService");
      NamingContext ncRef =
      NamingContextHelper.narrow(objRef);

      // resolve the Object Reference in Naming
      NameComponent nc = new NameComponent
      ("Echo", "");
      NameComponent path[] = {nc};
      org.omg.CORBA.Object tempEchoRef =
      ncRef.resolve(path);
      // we have an org.omg.CORBA.Object,
      // but we need an Echo
```

```
        Echo echoRef = EchoHelper.narrow(tempEchoRef);

        // call the Echo server object and print results
        String reply = echoRef.sayEcho("Martin");
        System.out.println(reply);

        } catch (Exception e) {
        System.out.println("ERROR : " + e) ;
        e.printStackTrace(System.out);
        }
    }
}
```

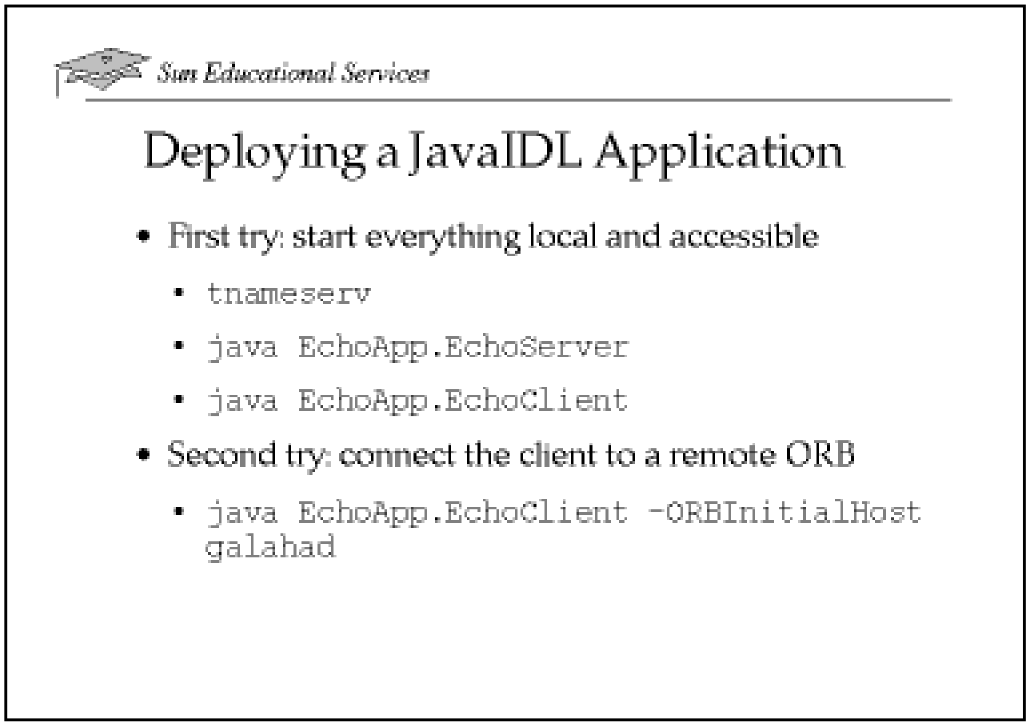
`Echo` was used as a name to bind the servant into the name server. Now use the same name to look it up. The result is an object of type `org.omg.CORBA.Object`, but you need an object of type `Echo`. A simple cast, such as`(Echo)ncRef.resolve(path)` does not always work; it depends on the ORB. The CORBA way to do casting is to use the `narrow`-method of the generated helper class `EchoHelper.narrow`.

Step 6: Create the Java Class Files

To compile the files, type the following:

```
javac -d . *.java
```

The files in the `EchoApp` packages are compiled automatically, because both `EchoServer` and `EchoClient` are dependent on them.



Sun Educational Services

Deploying a JavaIDL Application

- First try: start everything local and accessible
  - `tnameserv`
  - `java EchoApp.EchoServer`
  - `java EchoApp.EchoClient`
- Second try: connect the client to a remote ORB
  - `java EchoApp.EchoClient -ORBInitialHost galahad`

Deploying a JavaIDL Application

First Try: Start Everything Local and Accessible

Assuming there are no compilation problems, you start the name server first, then the server, and finally the client.

```
tnameserv
```

The default port is 900, which can be problematic. The UNIX® environment hosts do not let you connect to ports below 1024 if you are not `root`. Fortunately, you can change the port. Use `tnameserv -ORBInitialPort 1050` to connect using port 1050, for example.

You start the server next, which creates the servant and registers it with the name server.

```
java EchoApp.EchoServer
```

Again, add `-ORBInitialPort 1050` as argument to connect to port 1050.

In the last step, start the client.

```
java EchoApp.EchoClient
```

`-ORBInitialPort 1050` uses port 1050 to connect to the server ORB and name server.

Second Try: Connect The Client to a Remote ORB

To connect your client to a different machine, add a parameter on startup.

```
java EchoApp.EchoClient -ORBInitialHost yourhost
```

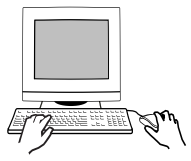
This line connects you to host *`yourhost`*. `-ORBInitialPort` and `-ORBInitialHost` can be combined.

This might be a good time to find out why passing arguments works: nothing was done in the code to support this. You should be able to find this out by yourself, by looking through your `EchoClient` source code and the Java documentation.

✓ ***The line*** `ORB orb = ORB.init(args, null);` ***passes all the command line arguments unchanged to the ORB. Unfortunately, the documentation of the valid parameters is a little hard to find: The ORB.init method documentation reveals nothing. Currently, the description of the valid parameters can be found in the Java 2 SDK documentation, JavaIDL part, "Programming Guide", section "initialization". Part of this section is also a description on how to set the parameters for an Applet.***

In Module 3, "Remote Method Invocation (RMI)," you were able to download stubs on demand, by using the RMIClassLoader on the client and an HTTP-based code server. Because there is no IDLClassLoader available, the same thing is not easily possible with JavaIDL. Usually, the stubs are packaged together with the rest of the client code and the generated helper classes, before the whole package is deployed onto the client.

Exercise: Compiling an Application



Exercise objective – Compile and run the Echo application. In addition, test the robustness of the JavaIDL system by intentionally breaking the code of the Echo application.

Tasks

Compile the Echo *Application*

Work through the six steps described in this module to compile the Echo application.

1. Change the directory to `labfiles/mod4-jidl/lab/echo.`

2. Compile the Echo server and client.

Run the Echo *Application*

Work through the steps described in "Deploying a JavaIDL Application" to get the Echo application running.

1. Run the Echo application locally on your system.

2. Team up with somebody and distribute the client and the server part of Echo on two different machines.

Break the Echo *Application to Test the Robustness of the JavaIDL System*

After Echo runs successfully, try to break the code in several ways to test the fault tolerance of the JavaIDL.

1. In the servant code:

   ▼ Do not implement a method that is in the IDL (or fail to implement the correct method signature).

   ▼ Leave out "`extends _EchoImplBase`".

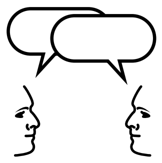
2. In the server code:

   ▼ Comment out `orb.connect(echoRef)`.

   ▼ Replace `NamingContextHelper.narrow(objRef)` with an explicit casting.

   ▼ Comment out the part at the end, where the server stops "forever."

Exercise Summary



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

✓ ***Manage the discussion here based on the time allowed for this module, which was given in the "About This Course" module. If you find you do not have time to spend on discussion, then just highlight the key concepts students should have learned from the lab exercise.***

- Experiences

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

- Interpretations

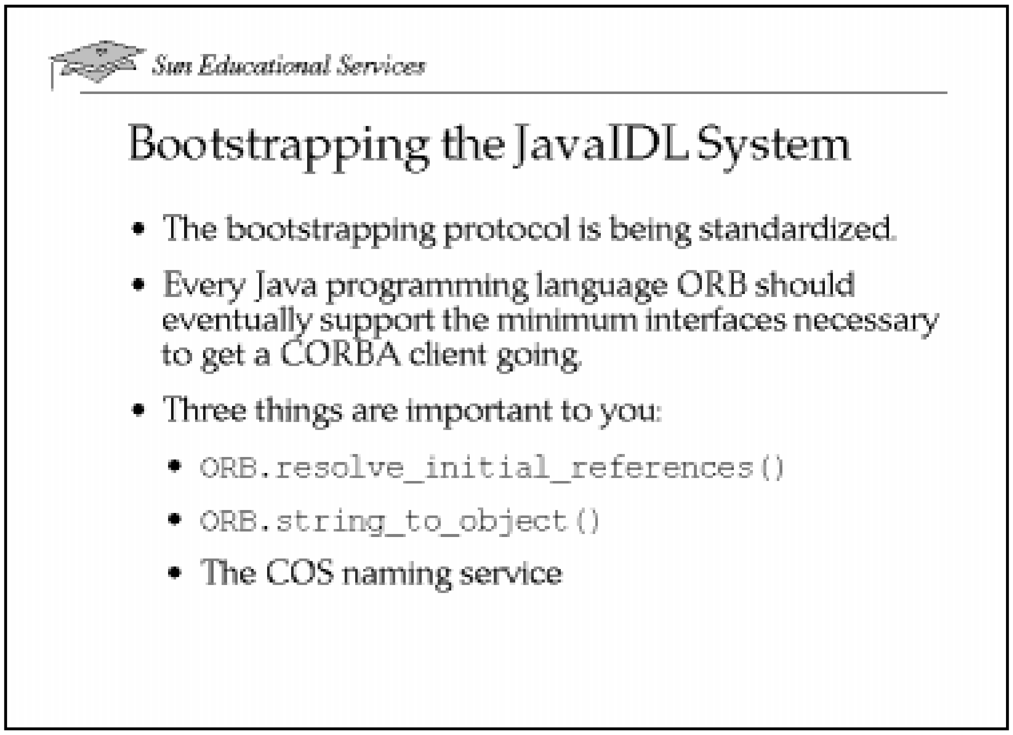
✓ ***Ask students to interpret what they observed during any aspects of this exercise.***

- Conclusions

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

- Applications

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



Sun Educational Services

Bootstrapping the JavaIDL System

- The bootstrapping protocol is being standardized.
- Every Java programming language ORB should eventually support the minimum interfaces necessary to get a CORBA client going.
- Three things are important to you:
  - `ORB.resolve_initial_references()`
  - `ORB.string_to_object()`
  - The COS naming service

Bootstrapping the JavaIDL System

Most of the bootstrapping code was shown in the `EchoClient` and `EchoServer`. This section reviews some of the details.

What Is Bootstrapping?

Using remote objects in a client application is easy and transparent, once you have the references. The bootstrapping protocol describes how you can get at the initial remote object references in a standardized way.

If this protocol were not standardized, then every ORB vendor would do it differently. Consequently, an applet written for Visibroker would not work if run on a client with only JavaIDL installed. You would have to download not only the applet, but also the Visibroker ORB to the client, which wastes bandwidth.

A protocol standardization is in the works. An initial submission was delivered to OMG in April, 1998. The document is titled “Interoperable Naming Service” and can be found on `ftp://ftp.omg.org/pub/docs/orbos/98-03-04.pdf`. The proposed standard is not specific to the Java programming language. Other languages will profit from this standard as well, in the form of increased source code level portability of CORBA client applications.

JavaIDL conforms to the proposed standard. Because the proposal was jointly submitted by several OMG members, it is unlikely that the document will change much before it is adopted.

Bootstrapping the Client Application

Bootstrapping consists of three steps:

1. Obtain a reference to the local ORB.
2. Use the ORB to retrieve the reference to a COS naming service.
3. Use the COS naming service to retrieve the remote object references.

Alternatively, you can use the ORB to resolve a named object reference to an actual reference. This might be necessary if no naming service is available.

Obtain a Reference to the Local ORB

The following line retrieves the ORB, when called from a Java application:

```
ORB orb = ORB.init(args, null);
```

On startup, the ORB checks the following properties:

```
org.omg.CORBA.ORBClass
org.omg.CORBA.ORBSingletonClass
org.omg.CORBA.ORBInitialHost
org.omg.CORBA.ORBInitialPort
```

You can set properties in three places:

- On the command line (this is why `ORB.init` gets a reference to the command-line arguments)
- In the form of a Java `Properties` object (which is passed instead of `null` as the second argument in `ORB.init`)
- As a Java system property

If the property is passed on the command line, you must omit `org.omg.CORBA`.

An applet does not have command-line arguments. Instead, an applet has parameters encoded in the HTML file that loads the applet. Instead of `args`, pass a reference to the applet to the ORB:

```
orb = ORB.init(this, null);
```

Of the four parameters, only the first two are standardized. `ORBInitialHost` and `ORBInitialPort` are JavaIDL-specific. They are used to specify the host and IP port number of the machine providing the initial services (for instance, the naming service).

By setting `ORBClass` to a specific value, the ORB can be replaced at runtime. The following applet parameter, for example, will run the applet using Visibroker 3.0 instead of JavaIDL:

```
<param name=org.omg.CORBA.ORBClass
value=com.visigenic.vbroker.orb.ORB>
```

Of course, the specified ORB must be accessible to the applet, either from the local machine or from the Web server.

Bootstrapping the Client Application

Use the ORB to Retrieve the Reference to a COS Naming Service

To retrieve initial references, the bootstrapping protocol specifies two calls that must be supported by the ORB:

- `ORB.list_initial_services()`
- `ORB.resolve_initial_references()`

The first call delivers an array of strings, describing the services available on the ORB. Usually, at least the name service ("NameService") is available. However, the array might be empty. One of these strings is passed into the second call, to get back an object reference to the desired service. The following code segment finds the reference for the naming service, and narrows it down to the correct type:

```
org.omg.CORBA.Object objRef =
orb.resolve_initial_references("NameService");

NamingContext ncRef =
NamingContextHelper.narrow(objRef);
```

Bootstrapping the Client Application

Use the COS Naming Service to Retrieve the Remote Object References

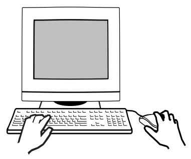
The following code segment retrieves a named object reference from the naming service, and narrows it to the correct type:

```
NameComponent nc = new NameComponent("Echo", "");
NameComponent path[] = {nc};
Echo echoRef =
 EchoHelper.narrow(ncRef.resolve(path));
```

A complete name consists of one or more `NameComponent` elements, arranged in an array. This whole name is `resolved` into an object reference. The actual object name is the last element in the array. The other elements form the name context. For example, imagine the elements of the array printed out and separated by a period: `acme.switzerland.zurich.headoffice.floor2.printer21`. In this example, the name is bound to the "root" context: there is only one element in the array.

The second parameter in the construction of a `NameComponent` can be used freely to add a description to the name, such as "executable" or "object_code". There is no standard on how to use this property.

Exercise: Bootstrapping/COS Name Server



Exercise objective – Find out which initial services are available in JavaIDL and (if available) in Netscape Communicator. In addition, explore the COS naming service.

Preparation

Copy `EchoApplet.java` and `EchoApplet.html` to `ServicesApplet.java` and `ServicesApplet.html`.

Tasks

Develop an Applet That Lists the Initial Services Available

Use `ServicesApplet.html` and `ServicesApplet.java` as templates to develop an applet that lists the initial services available to an ORB.

1. Change the directory to `labfiles/mod4-jidl/lab/boot.`
2. Develop the applet.
3. Try this applet in appletviewer and (if available) in Netscape Communicator. What services did you find in which environment?

✓ ***JavaIDL supports NameService, Netscape Communicator so far (v4.51) does not support any services.***

Explore Some Specifics of the COS Naming Service

1. See what happens if you use the second field in naming (*kind*)
   - When binding the name (modify `EchoServer.java`)
   - When looking up the object (modify `EchoClient.java`)
   - When looking up the object, but use a wildcard (“*”) as the *kind*

✓ ***Surprisingly, both the name and the*** kind ***have to match, and wildcards do not work. This is surprising because the*** kind ***is only a further description of the object, and not really a part of the name.***

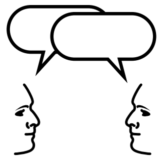
Tasks

Explore Some Specifics of the COS Naming Service (Continued)

- ▼ When binding the object a second time, with the name differing only in the *kind* part (modify `EchoServer.java`; warning: use "`bind`", not "`rebind`", as the latter replaces already existing bindings)

✓ ***As the*** kind ***is only a description of the object, and not really part of the name, a second*** `bind` ***throws an AlreadyBound exception.***

Exercise Summary



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

✓ ***Manage the discussion here based on the time allowed for this module, which was given in the "About This Course" module. If you find you do not have time to spend on discussion, then just highlight the key concepts students should have learned from the lab exercise.***

- Experiences

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

- Interpretations

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

- Conclusions

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

- Applications

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

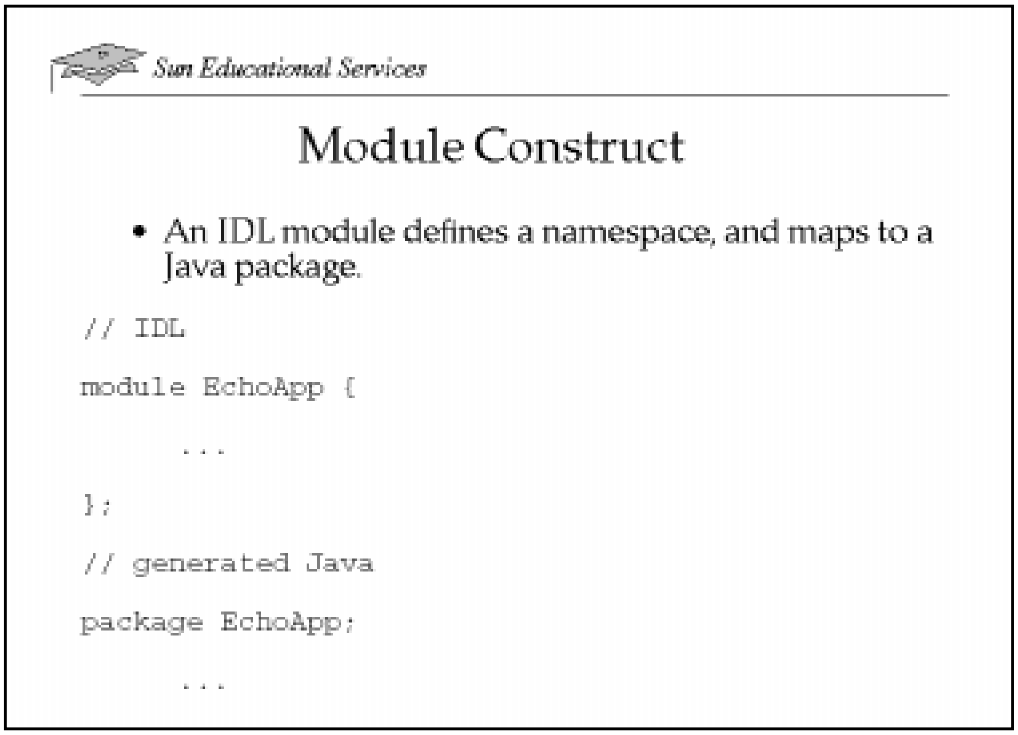
IDL-to-Java Programming Language Mapping Details

With IDL, you can declare items that are unknown to the Java programming language, such as constants, structs, or variable size arrays. Because there are several ways to implement these declarations using constructs known to the Java programming language, a standard on exactly how the IDL-to-Java programming language mapping is done is needed. This section covers the details of this standard, including how each IDL keyword maps to the Java programming language.

Note – This section is *not* a tutorial on IDL. To learn IDL, consult a book covering CORBA.

The following (artificially constructed) IDL contains all the keywords and constructs that are described on the next pages:

```
module EchoApp {
   interface Echo {
      attribute string name;
      readonly attribute string internalname;
      string sayEcho(in string myName)
      raises (IDLException);
   };

   exception IDLException {
      string reason;
   };

   struct Address {
      string name;
      boolean grownUp;
   };

   typedef sequence<Address, 5> MaxFiveAddresses;
   typedef sequence<Address> LotsOfAddresses;

   enum SpecialBool {yes, no, maybe};
};
```



Sun Educational Services

Module Construct

- An IDL module defines a namespace, and maps to a Java package.

```
// IDL
module EchoApp {
      ...
};
// generated Java
package EchoApp;
      ...
```

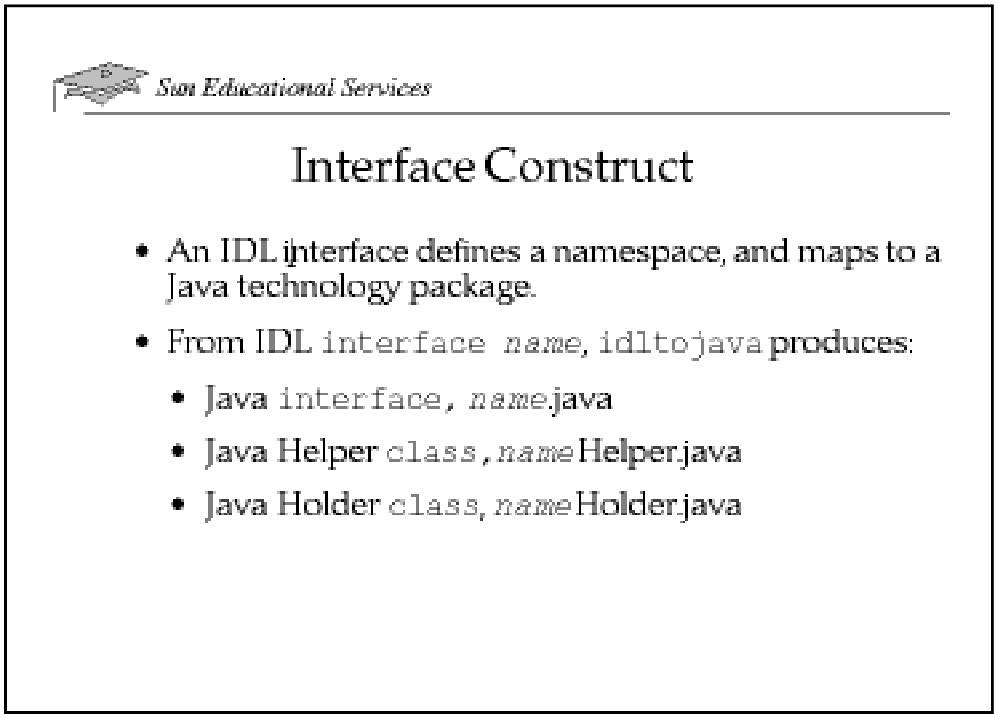
Module Construct

The *module* construct is used to scope IDL identifiers. Here EchoApp is the enclosing scope:

```
1  // IDL
            1  module EchoApp {
            2     ...
            3  };
```

In the Java programming language, a `module` maps to a package.

```
1  // generated Java
            4  package EchoApp;
            5     ...
```



Sun Educational Services

Interface Construct

- An IDL interface defines a namespace, and maps to a Java technology package.
- From IDL `interface` *`name`*, `idltojava` produces:
  - Java `interface,` *`name`*.java
  - Java Helper `class,` *`name`*Helper.java
  - Java Holder `class`, *`name`*Holder.java

Interface Construct

An *interface* defines the basic IDL service. Interfaces are a collection of *attributes*, *exceptions*, and *operations* that can be requested of an object by a client. JavaIDL maps IDL interfaces to a Java interface. The issue is that IDL supports multiple inheritance of interfaces only (as IDL defines interfaces only) while the Java programming language supports multiple inheritance of interfaces, it does not support multiple inheritance of classes.

Interface Example

Consider the following example of an IDL interface:

```
// IDL
module EchoApp {
   interface Echo {
   ...
   };
};
```

Compiled with `idltojava`, the following files are generated (on the client side):

- A Java interface file `Echo.java` that defines methods that correspond to the operations in the `Echo` interface:

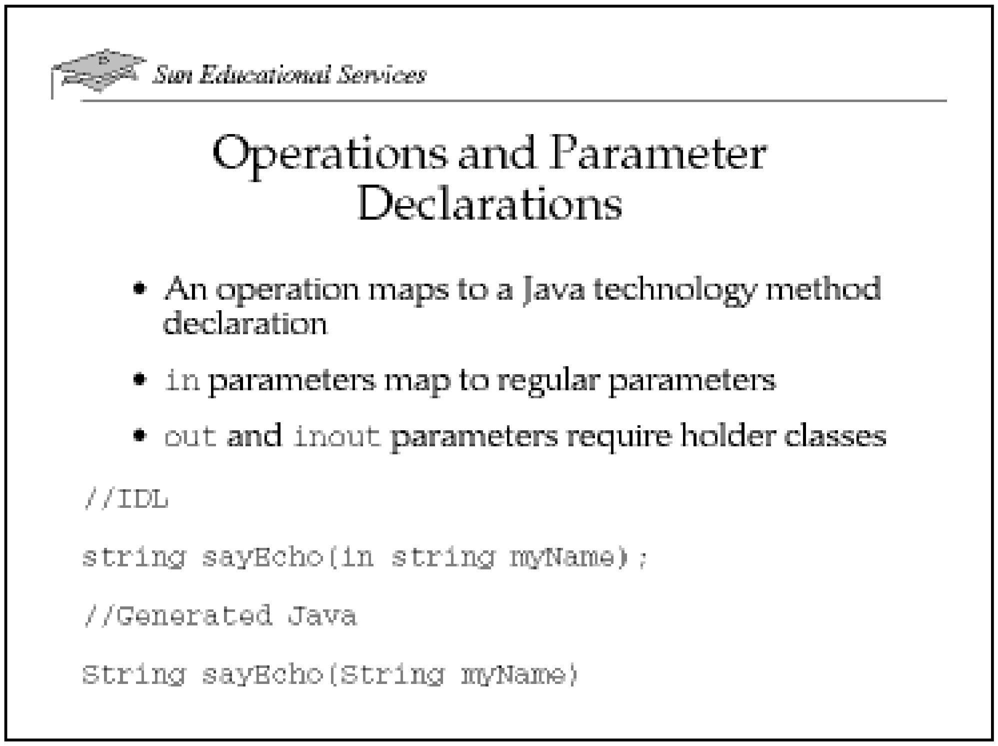
```
// generated Java
package EchoApp;
public interface Echo extends org.omg.CORBA.Object {
...
}
```

- A Java class file `EchoHelper.java` that defines methods used to assist the ORB. The most notable method is `narrow,` which is used to cast a CORBA Object type to its appropriate Java object type:

```
// generated Java
package EchoApp;
public final class EchoHelper {
...
public static EchoApp.Echo
narrow(org.omg.CORBA.Object that)
throws org.omg.CORBA.BAD_PARAM {
...
}
}
```

- A Java class file `EchoHolder.java,` which defines methods for setting and getting the values of an `Echo` object referred to through `out` and `inout` parameters:

```
// generated Java
package EchoApp;
public final class EchoHolder
     implements org.omg.CORBA.portable.Streamable{
...
}
```


Sun Educational Services

Operations and Parameter Declarations

- An operation maps to a Java technology method declaration
- `in` parameters map to regular parameters
- `out` and `inout` parameters require holder classes

```
//IDL

string sayEcho(in string myName);

//Generated Java

String sayEcho(String myName)
```

Operations and Parameter Declarations

Operations

Operation declarations in IDL are similar to C function declarations. An operation consists of the following:

- The return type of the operation or `void`
- An identifier that names the operation in its scope
- Zero or more parameters
- An optional `raises` clause

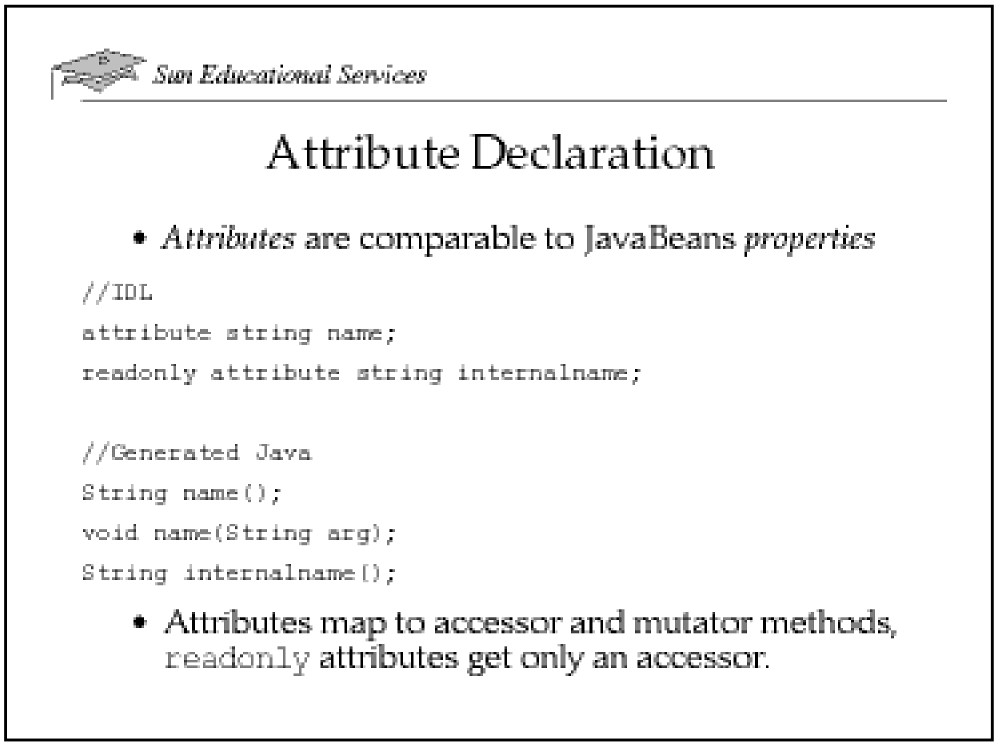
Parameters

An operation parameter can be of any IDL basic or user-defined type. It must have a directional attribute that informs the communications service in both the client and server of the direction in which the parameter is to be passed. Parameters consist of the following:

- `in` – The parameter is passed from client to server.
- `out` – The parameter is passed from server to client.
- `inout` – The parameter is passed in both directions.

The Java programming language does not support passing by value[1], so `out` and `inout` do not map directly for primitive types. The Holder classes are created to support all `out` and `inout` operations and return values.

1. Technically, if you pass an object by value in the Java programming language, you pass a *copy* of the reference variable. Therefore it is true that the Java programming language does not support passing by value.



Sun Educational Services

Attribute Declaration

- *Attributes* are comparable to JavaBeans *properties*

```
//IDL
attribute string name;
readonly attribute string internalname;

//Generated Java
String name();
void name(String arg);
String internalname();
```

- Attributes map to accessor and mutator methods, `readonly` attributes get only an accessor.

Attribute Declaration

An interface can have attributes as well as operations. IDL attributes are similar to JavaBeans *properties*: an attribute definition is logically equivalent to declaring an accessor and a mutator function. These, respectively, retrieve and set the value of the attribute. The optional `readonly` keyword indicates that there is only an accessor function.

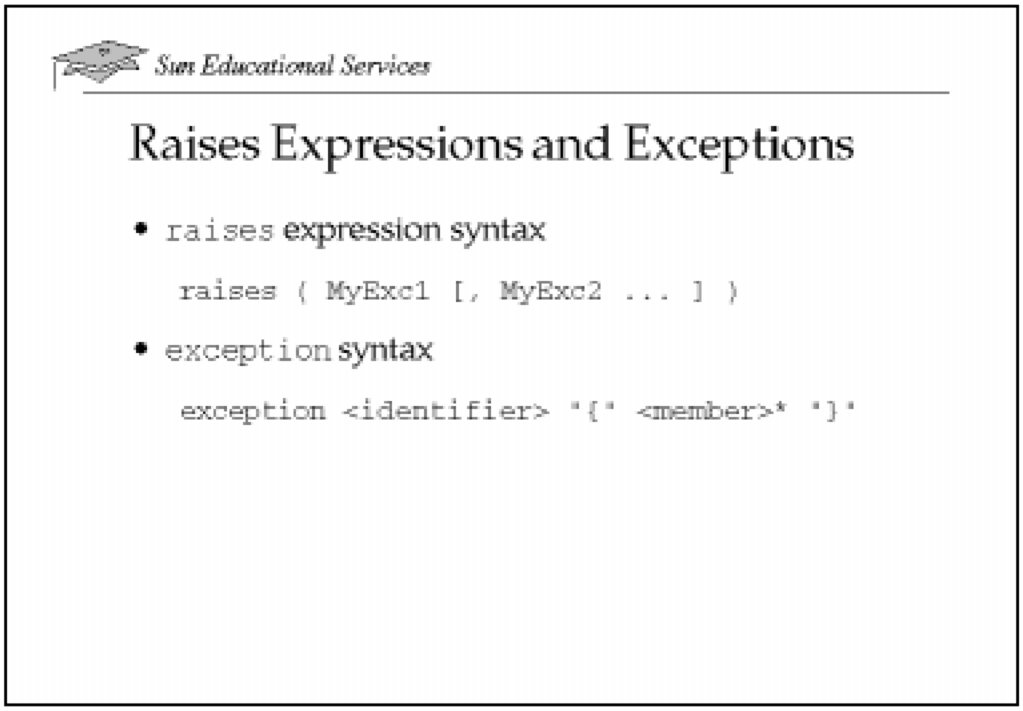
Consider the following IDL interface:

```
module EchoApp {
interface Echo {
attribute string name;
readonly attribute string internalname;
string sayEcho(in string myName);
};
};
```

This produces the following Java interface:

```
package EchoApp;
public interface Echo extends org.omg.CORBA.Object {
    String name();
    void name(String arg);
    String internalname();
    String sayEcho(String myName);
}
```

✓ ***Note that you will need to implement the actual “name” and “internalname” data field in the implementation class (servant) and fill in the methods to return the contents of the attributes.***



Sun Educational Services

Raises Expressions and Exceptions

- raises expression syntax

  raises (MyExc1 [, MyExc2 ...])

- exception syntax

  exception <identifier> "{" <member>* "}"

Raises Expressions and Exceptions

Raises Expressions

A `raises` expression specifies which exceptions can be raised as a result of an invocation of the operation. The syntax is:

```
raises ( MyExc1 [, MyExc2 ... ] )
```

The exceptions raised by an operation can be operation-specific or those from the set of standard exceptions. The latter might not always be listed in a `raises` expression (analogous to a subclass of a Java programming language `RuntimeException`, which does not have to be in the `throws` clause either).

All the CORBA runtime exceptions are unchecked. Compare this to RMI: every remote method call in RMI must be enclosed in a `try{} / catch (RemoteException){}` brace.

Exceptions

An `exception` declaration permits the declaration of a `struct`-like data structure that might be returned to indicate that an exceptional condition has occurred. Unlike the Java programming language, an exception is not a class, but a datatype. The syntax is:

```
exception <identifier> "{" <member>* "}"
```

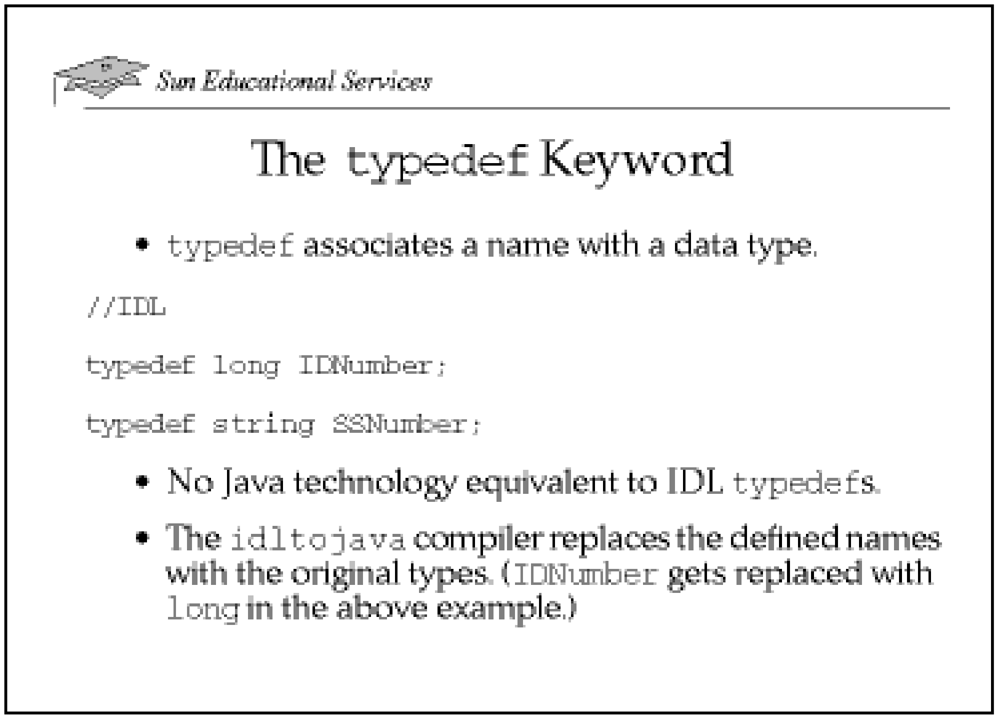
✓ ***What would happen if an exception were an object? The event object cannot be sent back with CORBA (no objects by value), only a remote reference to it. To make things worse: since CORBA does not have a remote garbage collector, and since exception objects tend to be very short lived, you would soon have a lot of no-longer-used exception objects clogging up the server. A datatype exception, on the other hand, can be sent back to the client by value, and can be immediately deleted on the server after sending it off.***

An exception has an identifier and zero or more returned member values. If an exception is returned from a request, the identifier is accessible; if members are declared, they are accessible. Otherwise, no further information is available.

```
module EchoApp {
...
exception IDLException {
string reason;
};
};
```

JavaIDL exceptions extend the `omg.org.CORBA.UserException` class:

```
package EchoApp;
public final class IDLException
    extends org.omg.CORBA.UserException {
    public String reason;
    public IDLException() {
        super();
    }
    public IDLException(String __reason) {
        super();
        reason = __reason;
    }
}
```



Sun Educational Services

The typedef Keyword

- typedef associates a name with a data type.

```
//IDL
typedef long IDNumber;
typedef string SSNumber;
```

- No Java technology equivalent to IDL typedefs.
- The idltojava compiler replaces the defined names with the original types. (IDNumber gets replaced with long in the above example.)

The `typedef` *Keyword*

IDL provides constructs for naming data types. The `typedef` keyword is used to associate a name with a data type.

```
typedef long IDNumber;
typedef string SSNumber;
```

The Java programming language has no construct equivalent to the IDL `typedef` statement. IDL `typedefs` for simple types do not map directly to the Java programming language. Instead, the original type is substituted for the new type everywhere the new type is encountered by the `idltojava` compiler.

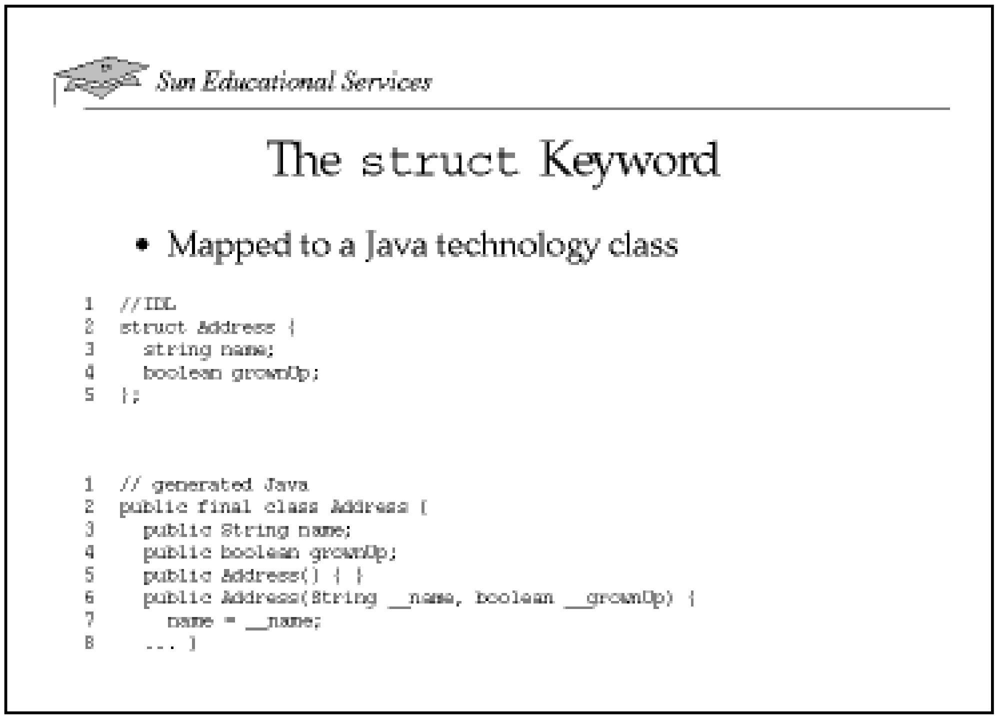
Basic Java Technology Types

IDL types map to Java types as shown in Table 4-1.

Table 4-1 IDL Types to Java Types Mappings

IDL Type	**Java Type**
`float`	`float`
`double`	`double`
`long, unsigned long`[a]	`int`
`long long, unsigned long long*`	long
`short, unsigned short*`	`short`
`unsigned long`	`int`
`unsigned short`	`int`
`char, wchar`	`char`
`boolean`	`boolean`
`octet`	`byte`
`string, wstring`	`java.lang.String`
`enum, struct, union`	`class`

a. It is up to the developer to maintain the accuracy of unsigned numbers because Java technology does not support unsigned types.


Sun Educational Services

The struct Keyword

- Mapped to a Java technology class

```
//IDL
struct Address {
  string name;
  boolean grownUp;
};
```

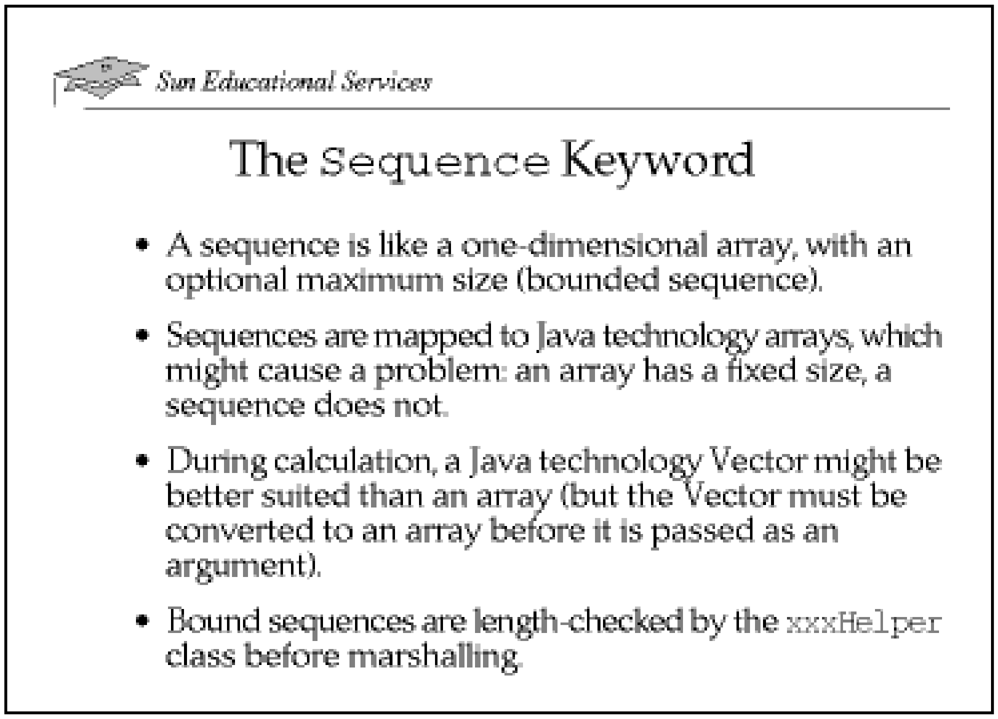
```
// generated Java
public final class Address {
  public String name;
  public boolean grownUp;
  public Address() { }
  public Address(String __name, boolean __grownUp) {
    name = __name;
  ... }
```

The `struct` *Keyword*

The IDL `struct` keyword is used to contain a collection of data that you would like to pass as a single item. The `struct` keyword is mapped to a class that provides instance variables for the fields and a constructor that takes a value for each variable.

```
//IDL
module EchoApp {
...
struct Address {
string name;
boolean grownUp;
};
};



// generated Java
package EchoApp;
public final class Address {
    //instance variables
    public String name;
    public boolean grownUp;
    //constructors
    public Address() { }
    public Address(String __name, boolean __grownUp) {
name = __name;
grownUp = __grownUp;
    }
}
```

✓ ***You have already heard several times that it is not possible to pass objects over the wire, only references; and now*** `structs` ***are mapped to objects. How can this Java object travel from a server to a client and vice versa? The Java object does*** not ***travel. What goes over the wire is a CORBA*** `struct`***, not a Java object. If JavaIDL "decides" to map the incoming CORBA struct to a local Java object, this is fine. All it has to do later, once the CORBA struct travels back, is to map the Java object to an outgoing CORBA*** `struct` ***again.***



Sun Educational Services

The Sequence Keyword

- A sequence is like a one-dimensional array, with an optional maximum size (bounded sequence).
- Sequences are mapped to Java technology arrays, which might cause a problem: an array has a fixed size, a sequence does not.
- During calculation, a Java technology Vector might be better suited than an array (but the Vector must be converted to an array before it is passed as an argument).
- Bound sequences are length-checked by the `xxxHelper` class before marshalling.

The `Sequence` *Keyword*

A sequence is like an array, with two key differences:

- It is one-dimensional.
- It does not have a fixed size. Optionally, it can have a maximum size (which is set at compile time).

A sequence can be unbounded or bounded. The bounded sequence is the one with a maximum size.

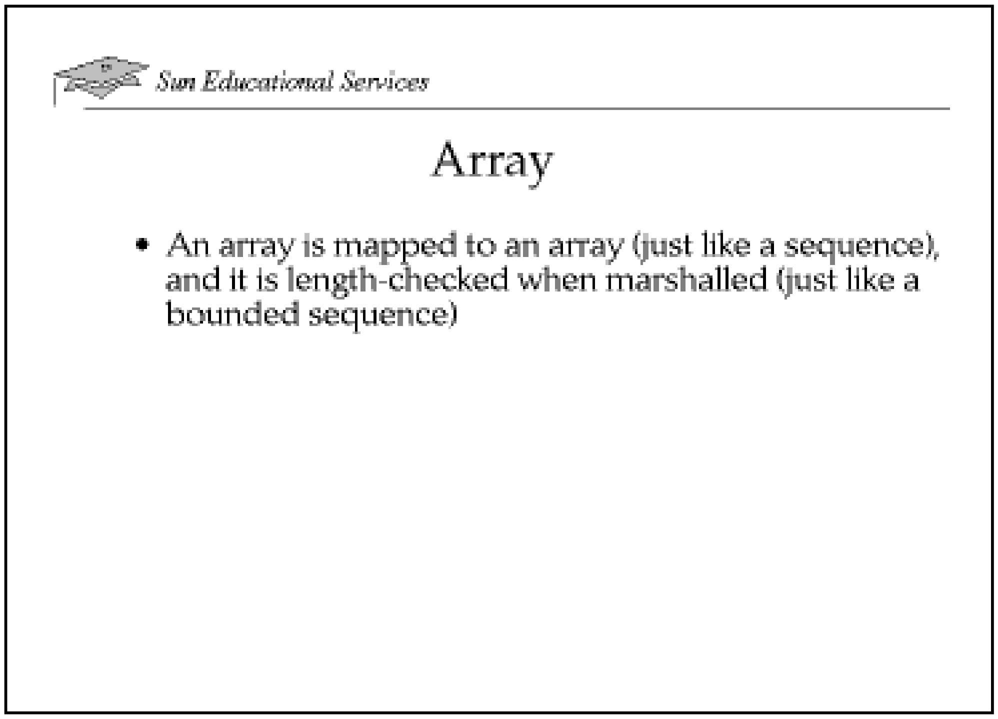
```
typedef sequence<Address, 5> MaxFiveAddresses;
typedef sequence<Address> LotsOfAddresses;
```

Sequence Mapping

Sequences are mapped to Java programming language arrays (“Java arrays”). The `idltojava` compiler generated a helper and a holder class for each sequence, but these classes are used only internally by the stub and skeleton. For the developer, IDL sequences look like regular Java arrays.

However, there is a distinct difference between a sequence and an array: An array does have a fixed size, which is set when the array is constructed. The sequence, on the other hand, does have a *length*, which is determined at runtime.

This difference is not a problem when a sequence is retrieved after a call. The sequence is unmarshalled into an array that has exactly the right size. When passing a sequence, you must pass an array that has the correct size. If the size cannot be determined at compile time, you can use Java programming language vectors, which can grow dynamically. At the moment of passing the sequence, the vector must be converted into a correctly sized array.



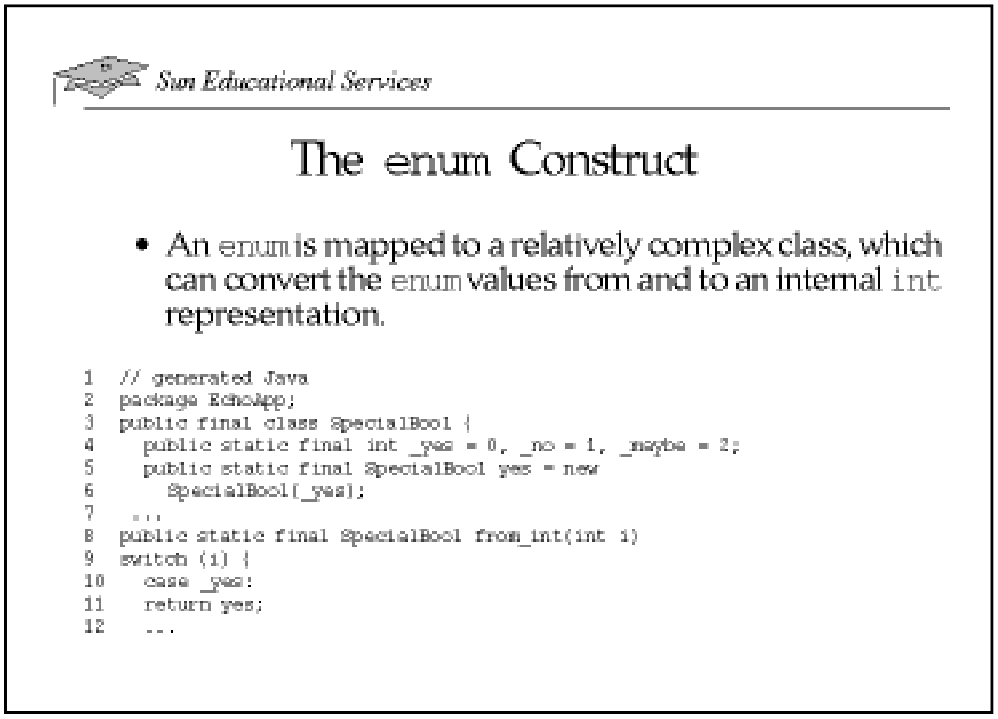
Sun Educational Services

Array

- An array is mapped to an array (just like a sequence), and it is length-checked when marshalled (just like a bounded sequence)

Array

IDL arrays are mapped in the Java programming language in the same way as the bounded sequence. The difference is that while a bounded sequence can accept arrays smaller than its bound, the array must be exactly the same size. The size is checked when the array is marshalled as an argument to an IDL operation.



Sun Educational Services

The enum Construct

- An enum is mapped to a relatively complex class, which can convert the enum values from and to an internal int representation.

```
// generated Java
package EchoApp;
public final class SpecialBool {
  public static final int _yes = 0, _no = 1, _maybe = 2;
  public static final SpecialBool yes = new
    SpecialBool(_yes);
 ...
public static final SpecialBool from_int(int i)
switch (i) {
  case _yes:
  return yes;
  ...
```

The enum *Construct*

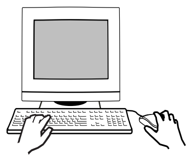
The IDL `enum` constructs are mapped to Java classes with a single `static final` variable for each member of the enumerated type.

```
//IDL
enum SpecialBool {yes, no, maybe};
```

The following is the Java code generated from the IDL:

```
package EchoApp;
public final class SpecialBool {
     public static final int _yes = 0,
       _no = 1,
       _maybe = 2;
     public static final SpecialBool yes = new
SpecialBool(_yes);
public static final SpecialBool no = new
SpecialBool(_no);
     public static final SpecialBool maybe = new
SpecialBool(_maybe);
     public int value() {
         return _value;
     }
     public static final SpecialBool from_int(int i)
throws  org.omg.CORBA.BAD_PARAM {
           switch (i) {
             case _yes:
                 return yes;
             case _no:
                 return no;
             case _maybe:
                 return maybe;
             default:
              throw new org.omg.CORBA.BAD_PARAM();
           }
     }
     private SpecialBool(int _value){
         this._value = _value;
     }
     private int _value;
}
```

Exercise: IDL-to-Java Programming Language Mapping Details



Exercise objective – Explore details of the IDL to Java programming language mapping; specifically, the difficulties with the IDL `sequence` to Java programming language `array` mapping.

Tasks

Convert a JDBC ResultSet to an IDL Sequence

A JDBC `ResultSet` is a datastructure with an unknown length (a typical Java programming language `Enumeration`). This matches perfectly to an IDL `sequence`, which does not have a fixed length either. However, the IDL `sequence` is mapped to a Java programming language array, not to some variable length Java programming language datastructure. Your job is to convert the JDBC `ResultSet` into a *correctly sized* array of `Coffee` (that is, the array must not contain null values).

1. Change the directory to `labfiles/mod4-jidl/lab/jdbc.`

2. Look at the IDL file `DBLookup.IDL`. It describes a server object that is capable of doing a database lookup for you and sending back the entire result set as a "sequence of coffee." The datastructure "coffee" is also declared in `DBLookup.IDL`.

   The IDL, the server, and the client are already written and do not need to be changed.

3. Compile the servant and run it. It does a DB lookup, but does not use the retrieved `ResultSet` to fill in the sequence. Compile and run the provided application to confirm that it is working.

4. Replace the existing code with code that converts the `ResultSet` into an array of `Coffee`! (Hint – Use a Java programming language `Vector` temporarily.)

5. Start the `tnameserv`, server, and client.

✓ ***When students get the database connection error on the server console, remind them of the JDBC exercises and have them change the host in*** `CoffeeHostImpl.java`***.***

Tasks

Explore Specifics of the IDL-to-Java Programming Language Mapping (Optional)

In this exercise, you use an IDL containing the most important IDL keywords: `MappingTest.IDL`. This IDL can be used to test the whole range of the IDL to Java programming language mapping specification.

1. Change the directory to `labfiles/mod4-jidl/lab/idl.`
2. Compile the IDL file, and look at the various generated files, especially `MInterface.java`. `MStruct.java`, `MSequenceHolder.java`, `MEnum.java` and `MEnumHolder.java`, because these are the classes that are used in the source code.
3. Compile the application provided and start it. There is an error with the attribute. Obviously an attribute declaration in IDL alone does not suffice.
4. Extend the servant to support the declared attribute (`MAttribute`).
5. Change the client so that the custom exception gets fired (look at the servant code to see how to trigger this). Even if you fill in the arguments on the servant *before* throwing the exception, they are not sent back to the client.
6. Change the client back to the previous state.
7. Try to break the servant in several ways:

   ▼ Instead of using the Holder you get, create a new one (that is, replace `theMSequenceHolder.value = new ...` with `theMSequenceHolder = new MSequenceHolder...`). What happens?

✓ ***A CORBA internal error is thrown. You are allowed to replace the value of a holder, but never the holder itself.***

   ▼ The sequence is bounded, as you can see in the IDL file. Try to give back too many elements. What happens?

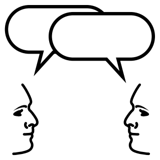
✓ ***A CORBA marshalling error, `org.omg.CORBA.MARSHAL`, is thrown.***

- ▼ Give back an incorrectly sized array. Use an array of size 2, but fill in only the first element; the second one is initialized with `null`. What happens?

✓ ***CORBA internal error,*** `org.omg.CORBA.UNKNOWN`***, gets thrown. A sequence, by definition, cannot contain null elements. This is analogous to a Java vector. Unfortunately, a Java array can contain null elements, so you have to pay close attention. Why the sequence got mapped to a Java array and not to a vector, is not known.***

- ▼ Try to overwrite the `in` argument in the servant. Does it work? Does the argument change on the client too?

Exercise Summary



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

✓ ***Manage the discussion here based on the time allowed for this module, which was given in the "About This Course" module. If you find you do not have time to spend on discussion, then just highlight the key concepts students should have learned from the lab exercise.***

- Experiences

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

- Interpretations

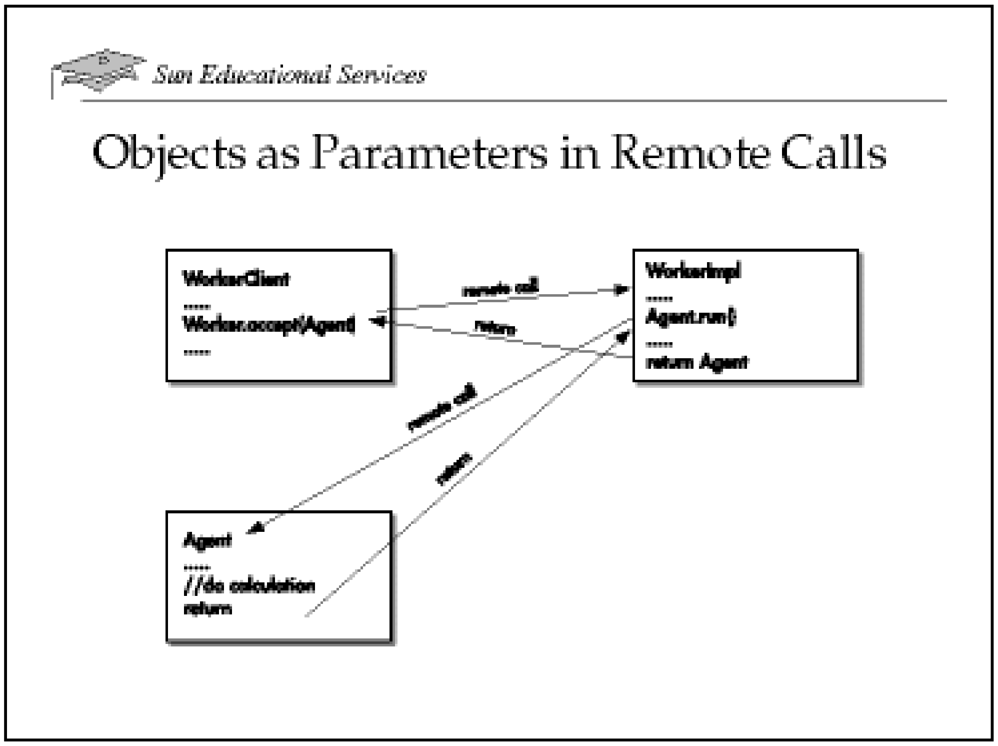
✓ ***Ask students to interpret what they observed during any aspects of this exercise.***

- Conclusions

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

- Applications

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


Sun Educational Services

Objects as Parameters in Remote Calls

Objects as Parameters in Remote Calls

So far, only primitive datatypes have been used as arguments in the remote calls. These arguments were passed by value to the server implementation, inside a holder class if necessary. In this section you will pass an object as an argument. Use the same setup as was used in Module 3, “Remote Method Invocation (RMI),” for the Agent example. To start, the following are the IDLs:

```
module AgentApp {
interface Agent {
void run();
long getResult();
};
};

module WorkerApp {
interface Worker {
void accept (inout AgentApp::Agent agent);
};
};
```

Two remote objects are declared: the Worker is the servant, and the Agent is the class that is passed as an argument. This is analogous to the two Java interface declarations, `Worker.java` and `Agent.java`, in Module 3.

The source code is derived from the RMI example, so it looks similar. One difference is in the IDL. It contains an additional method, `long getResult`. You will see later why you need this method. Another difference is the requirement for a holder class: `inout` arguments get mapped this way. The following is a section from `WorkerClient.java`:

```
Agent agent = new CalcFactorial(10);
AgentHolder holder = new AgentHolder(agent);
workerRef.accept(holder);
System.out.println("Returned from remote call\n");
agent = holder.value;
System.out.println("Result: " + agent.getResult());
```

Unexpectedly, this code seems to work. The output of the application is the same as with RMI. Did the Agent travel over the wire after all? That is, was it passed by value to the server? Find the answer yourself by doing the following exercise.

Exercise: Objects as Parameters in Remote Calls

Exercise objective – Confirm that objects by value do not work with JavaIDL by trying to duplicate the Agent example from Module 3 with JavaIDL.

Tasks

Solve the Agent Example From RMI With JavaIDL and Find the Differences From RMI

Complete the following steps:

1. Change the directory to `labfiles/mod4-jidl/lab/agent.`
2. Compile and run the application provided. Look at the source code and compare it to the RMI agent example. Locate the one crucial difference.

✓ ***In JavaIDL, the Agent is declared as a remote object; there is an IDL for it. idltojava was run against the IDL, and the actual implementation (`CalcFactorial`) extends _AgentImplBase. This means that whenever you have an Agent as a parameter in a remote call, a remote reference to the Agent is passed. In RMI, on the other hand, the Agent is not a remote object. The interface `Agent.java` does not extend Remote, and `rmic` was not run against `CalcFactorial.java` (which implements the Agent). This means that whenever there is an Agent as a parameter in a remote call (such as a `CalcFactorial`), a copy of the Agent is passed.***

3. Find the consequences of the difference. On which machine does the Agent execute? If you think you know the answer, prove it by modifying the source code.

✓ ***The Agent always runs on the (virtual) machine it was created on, since it cannot leave it. In step 2, it runs on the client, which is exactly what was not wanted.***

4. Modify the servant (`WorkerImpl.java`) to do the following:
   a. Create a new `CalcFactorial.`
   b. Calculate the result (call its `run` method).
   c. Assign the new object to the holder-class (`Agent.value = ...` in the source code).

Discussion

To test your knowledge of 'object by reference,' ask yourself the following questions:

- On which machine does the `CalcFactorial` run this time?

✓ ***As before, the*** `CalcFactorial` ***runs on the (virtual) machine it was created on, only this time, this machine is the server.***

- On what class does the holder "hold" the client *before* returning: `CalcFactorial.class` or `_AgentStub.class`?

✓ `CalcFactorial.class`***. After all, you just created this object and assigned it to the holder's value.***

- On what class does the holder "hold" the client *after* returning: `CalcFactorial.class` or `_AgentStub.class`?

✓ `_AgentStub.class`***. A remote reference to the newly created*** `CalcFactorial` ***on the server is passed back.***

Exercise Summary

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

✓ ***Manage the discussion here based on the time allowed for this module, which was given in the "About This Course" module. If you find you do not have time to spend on discussion, then just highlight the key concepts students should have learned from the lab exercise.***

- Experiences

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

- Interpretations

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

- Conclusions

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

- Applications

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

Explanation

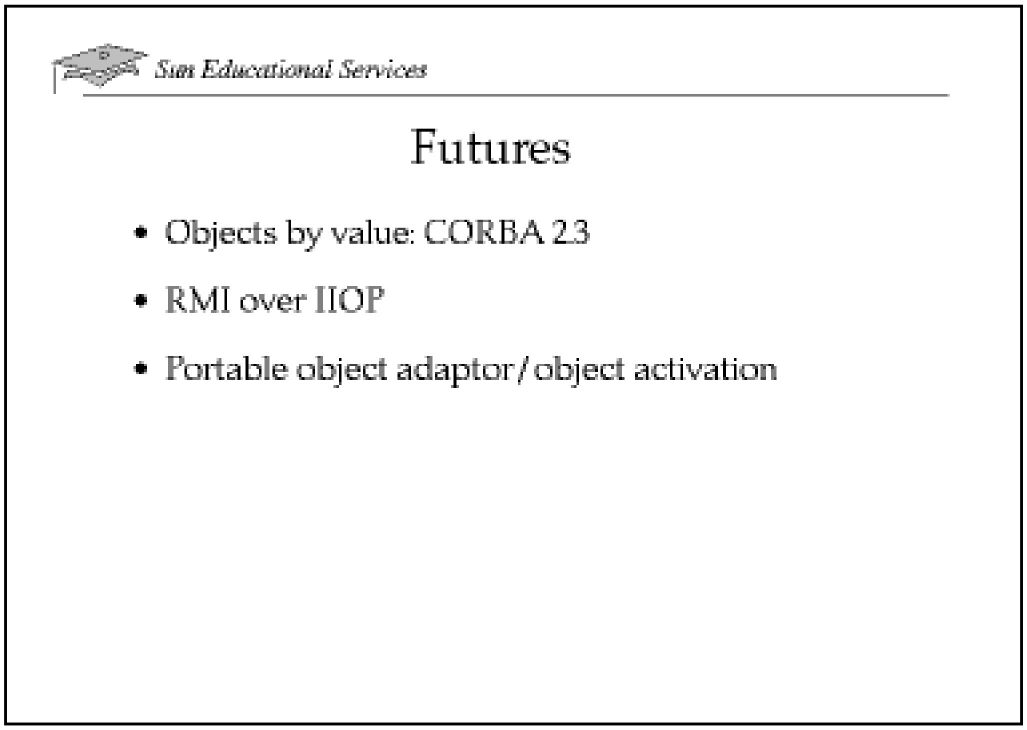
The preceding exercise is based on the following principle: The `WorkerServant` gets an `AgentHolder`, which contains an Agent. Remember, Agent is just an interface. What class does the `WorkerServant` really get? The answer is `_AgentStub`. Obviously, the `AgentServant` does not get a copy of the Agent, but a remote reference to the Agent residing in the client. This is the crucial difference from RMI. In the RMI example, the `WorkerServant` gets a copy of the object; that is, a `CalcFactorial`.

If you call `run` from the `WorkerServant` code, you will do a remote call, this time back to the client. This is what is known as a *callback*. The server calls back to the client, and the roles are reversed.

CORBA does not allow passing of objects by value. Whenever there is an object as a parameter in a method call, a remote reference is passed.

Now that you know this rule, what happens if you create a new `CalcFactorial` on the server, and pass this object back inside of the holder class? Again, the rule holds. You *cannot* send back a copy of the object to the client; it *has to be* a remote reference. An unusual situation now exists. You had a local object on the client before the `accept` call, and a remote reference right after the call. This gets confusing, for both you as the developer and for CORBA. The ORB on the server keeps a reference to the newly created CalcFactorial, because CalcFactorial is a CORBA object. Theoretically, it could be dropped once the server ORB "knows" that no remote references to this object exist anymore (which is the case as soon as the client exits). This behavior is known as *remote garbage collection*. However, JavaIDL does not implement remote garbage collection, and so the object stays on the server forever, wasting resources.

This callback is also the reason why you needed the additional method `getResult`. The server created a `CalcFactorial`, but the client does not "know" this. As far as the client is concerned, it gets back a remote reference to an Agent, not to a `CalcFactorial.` (You do not even have an IDL for a `CalcFactorial`.) If the Agent does not have a method `getResult`, the client cannot get at the result of the calculation. Compare this to the RMI case. There, you got back a *copy* of an Agent, which happened to be a `CalcFactorial`. Because you do have the *copy* back on the client, you can cast it to any class this Agent happens to be.



Sun Educational Services

Futures

- Objects by value: CORBA 2.3
- RMI over IIOP
- Portable object adaptor/object activation

Futures

Objects by Value

You have seen several times that you cannot transmit CORBA objects by value. This is going to change; the next version of CORBA, 3.0, is supposed to contain this enhancement. A draft specification is already under review. You can find the FTP address of the draft document in the "Additional Resources" section of this module.

✓ ***At*** `http://www.omg.org/news/pr98/compnent.html` ***you can also find more current information on the most current version of CORBA.***

RMI Over IIOP

Sun Microsystems is working with IBM on an implementation of RMI that uses the CORBA wire protocol (IIOP) instead of its own. This will allow interoperability of CORBA and RMI: an RMI server can be called from a CORBA client, and a CORBA server can be called from an RMI client. To support this as transparently as possible, IIOP must allow objects by value. This is the main reason why the *objects by value* proposal has been made.

The “Additional Resources” section at the beginning of this module cites a JavaOne session about RMI over IIOP.

Portable Object Adapter/Object Activation

So far, JavaIDL allows you to write only simple, transient servers. Object activation, as you have seen with RMI, is not possible with the current JavaIDL beta. Other commercial ORB products allow more feature-rich servers to be written. However, because there was no sufficiently exact standard on how to do this prior to CORBA 2.2, servers are not portable between different ORB products.

CORBA Version 2.2 includes the definition of a portable object adapter (POA). Using this adapter, you can write feature-rich servers in an ORB-independent fashion. If and when JavaIDL will include the POA is not known at this time.

Another possibility for writing better servers is to use RMI. Once the RMI over IIOP extension is available, you can write servers using RMI, with little CORBA knowledge. Because RMI supports object activation today, you would not have to wait for POA to arrive in JavaIDL.

Check Your Progress

Check that you were able to accomplish or answer the following questions related to content in this module:

- ❑ Describe the basic CORBA object management architecture
- ❑ Describe the role of JavaIDL in relation to other commercial Java CORBA products
- ❑ Create and deploy a JavaIDL server object and a JavaIDL client application
- ❑ Describe how the JavaIDL bootstrapping process works
- ❑ Describe how IDL is mapped to the Java programming language
- ❑ Explain why the RMI Agent example does not work with the current CORBA

Think Beyond

How could you use JavaIDL to distribute your applications?

Servlets 5

Objectives

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

- Understand the basic concepts of servlets
- Create your own servlets for Web-based applications
- Compare HTTP servlets to other technologies, such as common gateway interface (CGI)

Servlets are used to extend or implement server functionality. The Servlets API enables developers to implement servlets that can be used by any server enabled for the Java programming language.

Relevance

✓ ***Present the following questions to stimulate the students and get them thinking about the issues and topics presented in this module. They are not expected to know the answers to these questions. The answers to these questions should be of interest to the students, and inspire them to learn the content presented in this module.***

Discussion – Consider the following questions:

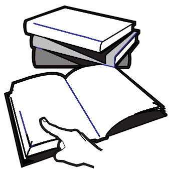
- What are the advantages of server-side components over classic mainframe servers?

✓ ***Servers become easier to understand, to maintain, and to extend when written in a modular way using components.***

- What makes the Java programming language an ideal platform for server applications?

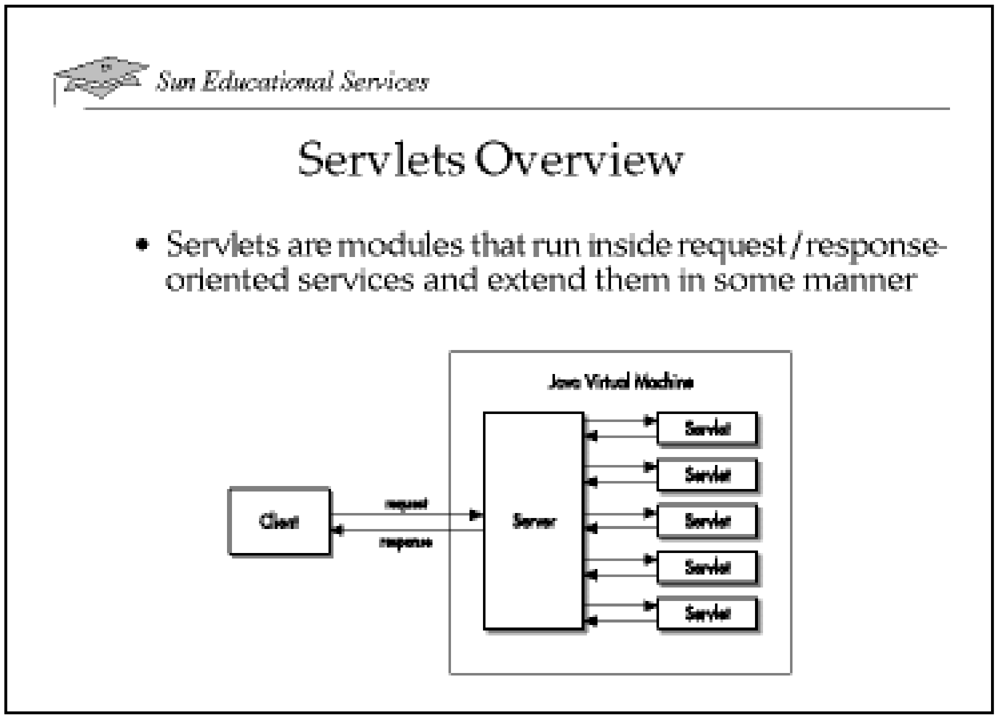
✓ ***Platform neutrality, integrated thread model, garbage collection, and its security model.***

Additional Resources



Additional resources – The following resources can provide additional detail on the topics presented in this module:

- Java Server Product. [Online]. Available: `http://java.sun.com/products/java-server`
- Java Servlets. [Online]. Available: `http://java.sun.com/products/java-server/servlets`
- Java Web Server. [Online]. Available: `http://java.sun.com/products/webserver/index.html`



Sun Educational Services

Servlets Overview

- Servlets are modules that run inside request / response-oriented services and extend them in some manner

Servlets Overview

Servlets are protocol- and platform-independent server-side components written in the Java programming language. Servlets are used in servers enabled for the Java programming language to extend the services the server provides. Because you can load servlets dynamically, they extend a server’s functionality at runtime.

Because servlets run inside a server, they are the server-side counterpart to applets. Servlets are Java application components that you can download on demand to the server that needs them.

Servlets were first developed for the Java Web Server™. The Servlet API is generic enough to implement any request-response type of server, although today’s main usage are Web-based services.

Servlets API

The Servlets API is a standard Java programming language extension and is bundled with the Java 2 SDK in the `javax.servlet` package. The package core consists of six interfaces, two stream classes, and two exception classes. A servlet must implement the `Servlet` interface. For easy and fast development, an implementation of the `Servlet` interface is provided within the package.

The server calls the servlet's `service` method to handle a request from a client. Servlets are running in a multithreaded environment and many requests to a servlet's service method can be made simultaneously. Therefore, servlets must be implemented in a thread-safe way. To simplify thread-safe servlets development, a servlet can implement the `SingleThreadModel` interface. A single-threaded servlet's service method is never called more then once at a time. The server ensures that the service is never called several times simultaneously.

Figure 5-1 on page 5-6 shows the class hierarchy for the `javax.servlet` package.

The `javax.servlet` *Package*

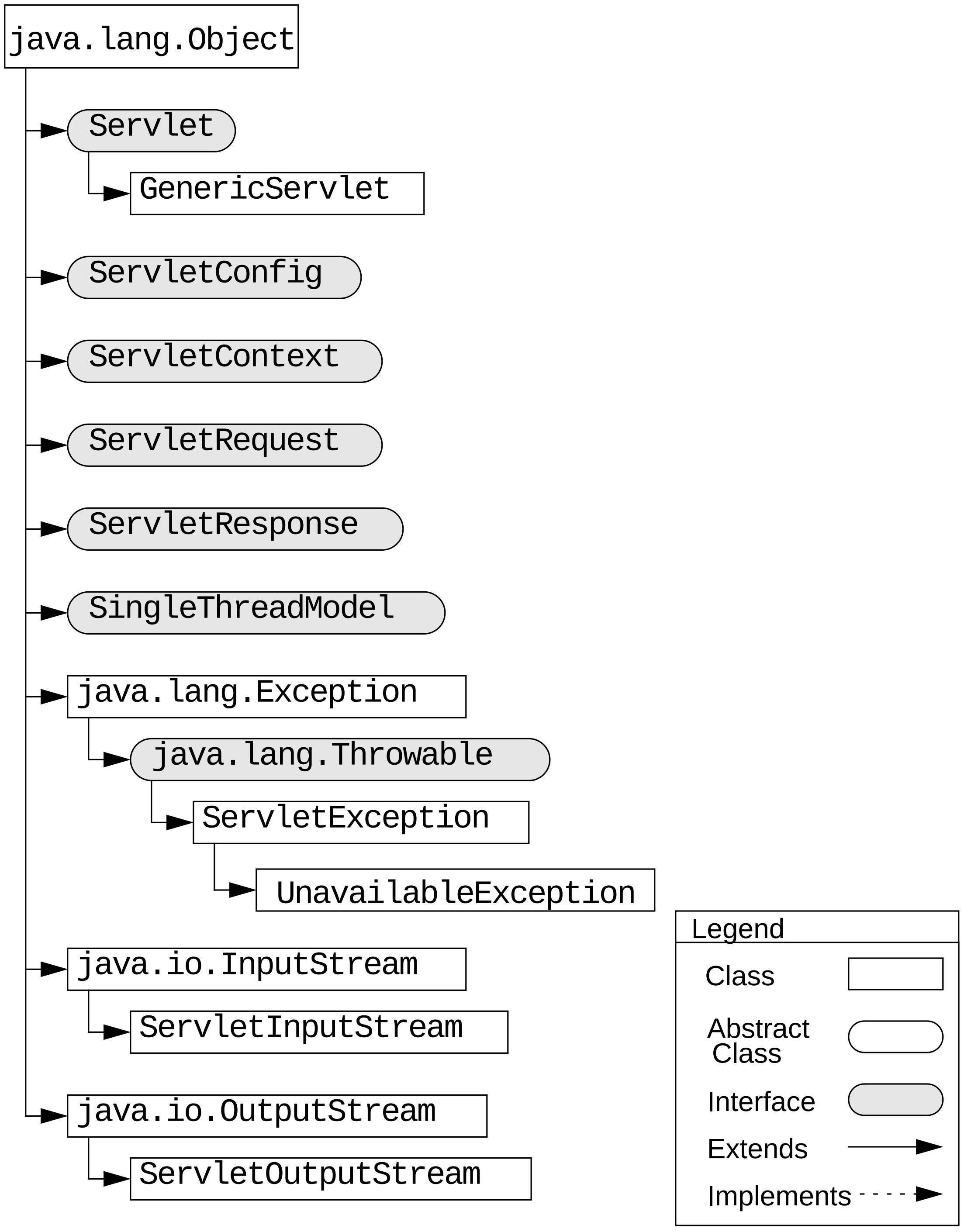
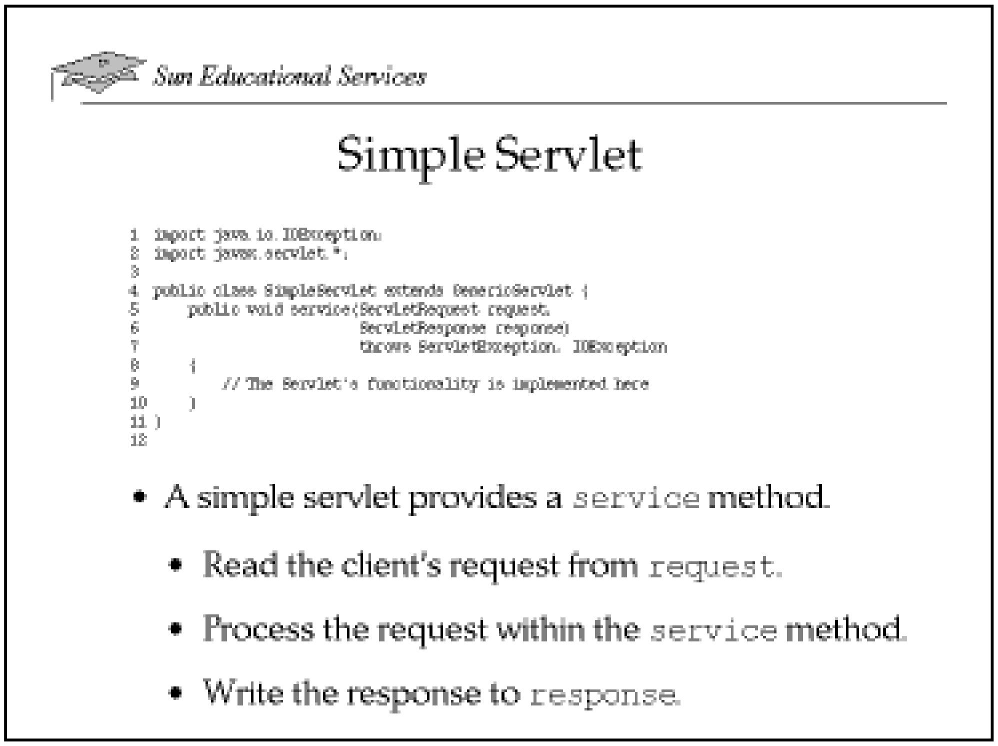


Figure 5-1 The `javax.servlet` Package



Sun Educational Services

Simple Servlet

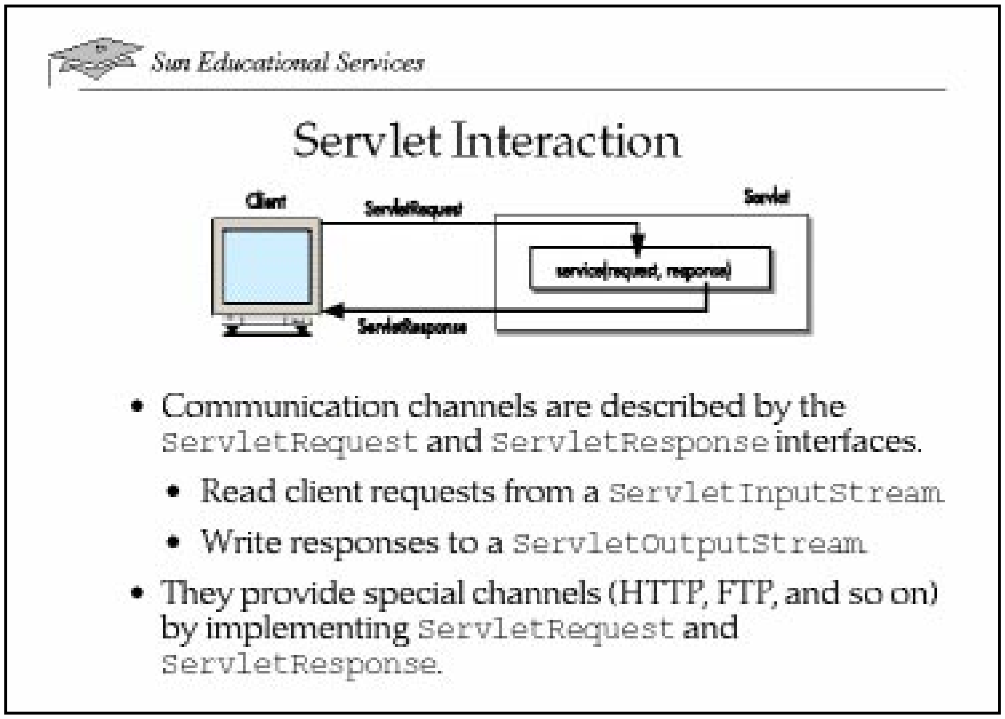
```
import java.io.IOException;
import javax.servlet.*;

public class SimpleServlet extends GenericServlet {
    public void service(ServletRequest request,
                        ServletResponse response)
                        throws ServletException, IOException
    {
        // The Servlet's functionality is implemented here
    }
}
```

- A simple servlet provides a `service` method.
  - Read the client's request from `request`.
  - Process the request within the `service` method.
  - Write the response to `response`.

Simple Servlet

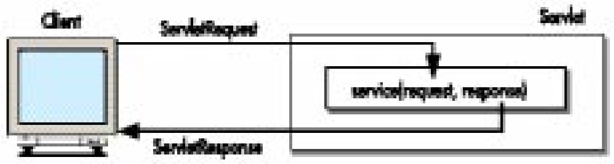
The implementation of a simple servlet is outlined in the following example. It extends the `GenericServlet` class included in the `Servlet` package. The `GenericServlet` implements the `Servlet` interface and eases development of servlets, because an implementation of every method defined by the interface is provided.

```
import java.io.IOException;
import javax.servlet.*;

public class SimpleServlet extends GenericServlet {
    public void service(
        ServletRequest  request,
        ServletResponse response)
        throws ServletException, IOException
      {
      // The Servlet's functionality is
      // implemented here.
      }
}
```



Sun Educational Services

Servlet Interaction



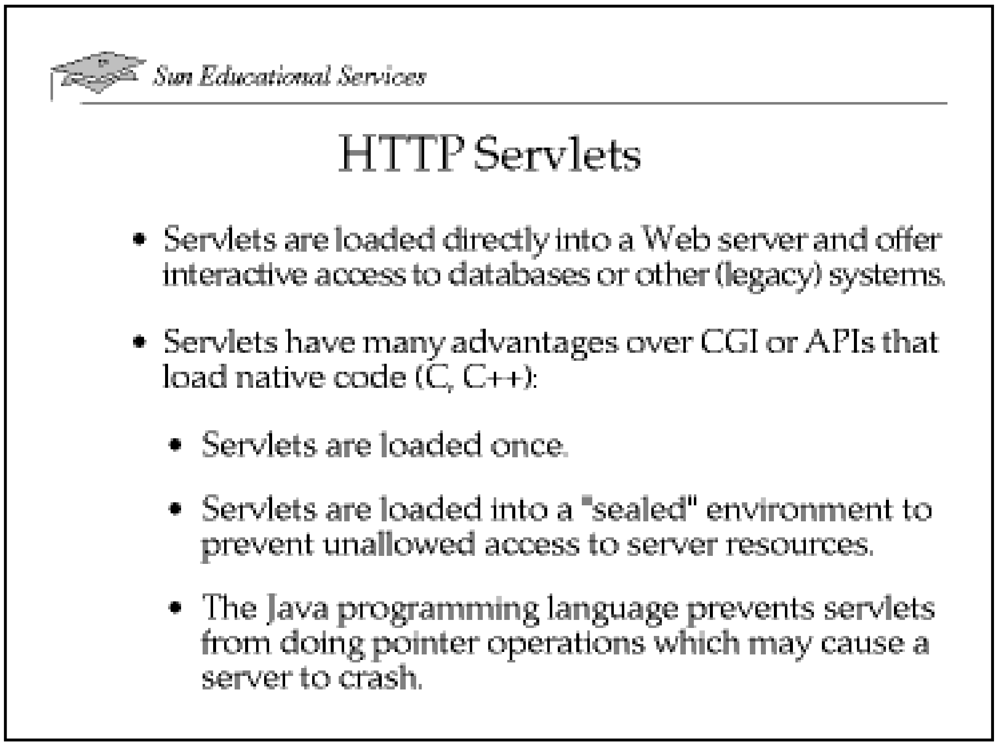
- Communication channels are described by the `ServletRequest` and `ServletResponse` interfaces.
  - Read client requests from a `ServletInputStream`
  - Write responses to a `ServletOutputStream`
- They provide special channels (HTTP, FTP, and so on) by implementing `ServletRequest` and `ServletResponse`.

Servlet Interaction

The service method is provided with two parameters, `Request` and `Response`, which encapsulate the interaction with clients or chained servlets. The request parameter contains all data sent by the client, including status and meta information. The response parameter is used to send a result back to the client. Usually the data is read from an `InputStream`, processed and written to an `OutputStream`. The `SimpleServlet`'s service method retrieves the two streams as follows:

```
ServletInputStream in =request.getInputStream();
ServletOutputStream out =response.getOutputStream();
```

The data format used to communicate is not specified within the API. It is up to the user to specify how requests and responses are encoded. The `javax.servlet.http` package defines new `Request` and `Response` interfaces, which are used to implement HTTP communication channels.



Sun Educational Services

HTTP Servlets

- Servlets are loaded directly into a Web server and offer interactive access to databases or other (legacy) systems.
- Servlets have many advantages over CGI or APIs that load native code (C, C++):
  - Servlets are loaded once.
  - Servlets are loaded into a "sealed" environment to prevent unallowed access to server resources.
  - The Java programming language prevents servlets from doing pointer operations which may cause a server to crash.

HTTP Servlets

HTTP servlets are used by Web servers enabled for the Java programming language to build interactive Web applications, generate dynamic Web pages, or integrate databases or other systems into the Web. The servlets are loaded upon request into the running Web server in response to client requests.

HTTP servlets are Java technology components (“Java components”) and therefore do not have memory leaks or illegal memory accesses that might occur with components written in C or C++. The HTTP servlet is loaded once into the server and can afterwards serve many requests, even simultaneously. This approach differs much from the Common Gateway Interface (CGI), where for each request a new process needs to be started, leading to delays and placing an excessive burden on the Web server.

The Java Web Server provided by Sun was the first Web server that used servlets to build complex interactive Web applications. Today, many other Web servers are enabled for the Java programming language servlets.

Note – You can extend Netscape, Microsoft, and other web server products with servlet support.

The `javax.servlet.http` *Package*

The `javax.servlet.http` package is a standard Java programming language extension and is bundled with the `javax.servlet` package. It contains interfaces and classes to handle HTTP- and HTML-specific communication. Furthermore the package offers HTTP session management and tracking using either cookies or automatically generated URLs. The `HTTPServletRequest` interface offers access to HTTP specific header and protocol information.

This section examines the `HttpServlet` class, which offers developers a good starting point to build servlets for Web servers.

The `javax.servlet.http` *Package*

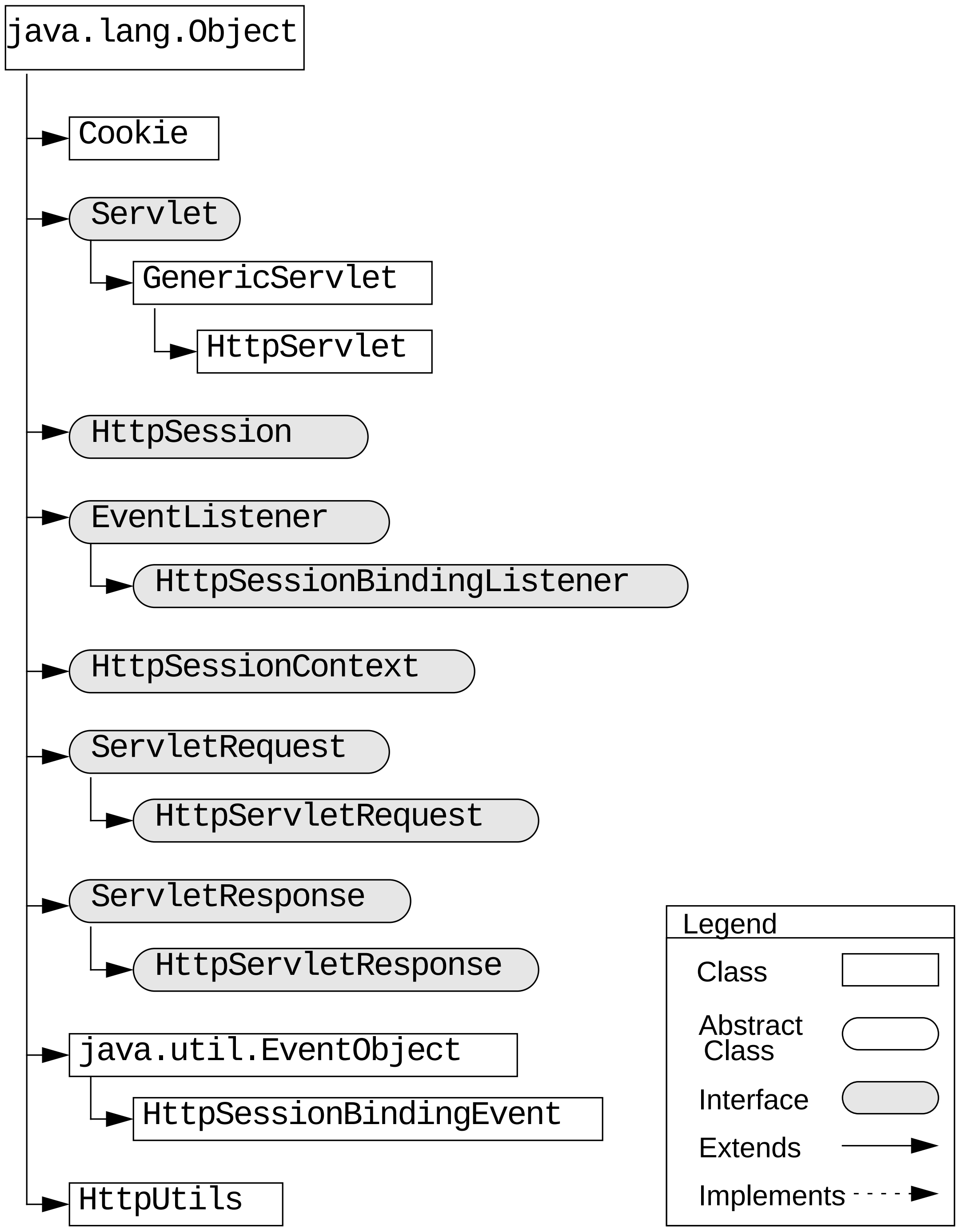


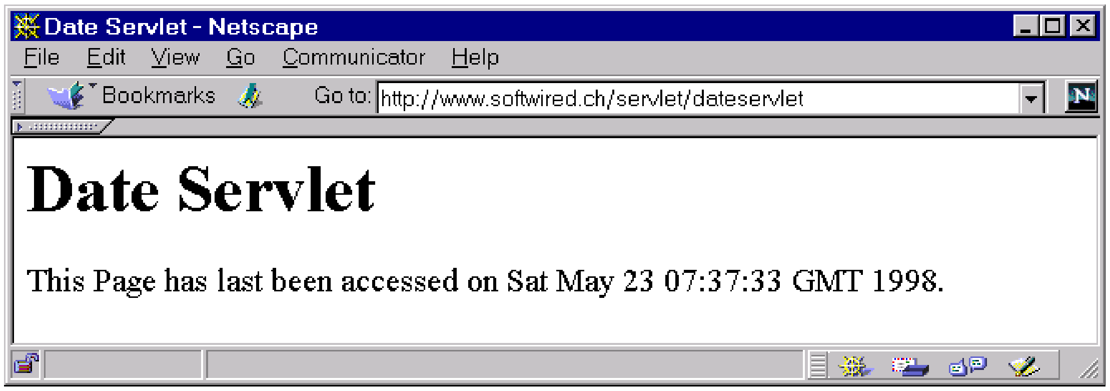
Figure 5-2 The `javax.servlet.http` Package

HTTP Servlet Example

The following example servlet prints the date and time formatted in HTML to its response stream. The result displayed by a Web browser looks similar to Figure 5-3.

```
import java.io.*;
import java.util.Date;
import javax.servlet.*;
import javax.servlet.http.*;

public class DateServlet extends HttpServlet
{
    public void doGet(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {

       // set response header fields first
       response.setContentType("text/html");

       // then write the data of the response
       PrintWriter out = response.getWriter();

       out.println("<HTML><HEAD><TITLE>");
       out.println(getServletInfo());
       out.println("</TITLE></HEAD><BODY>");
       out.println("<H1>"+getServletInfo()+"</H1>");
       out.println("<P>This Page has last been " +
                   "accessed on "+new Date()+".");
       out.println("</BODY></HTML>");
       out.close();
    }

    public String getServletInfo() {
        return "Date Servlet";
    }
}

```

.

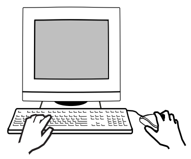

Date Servlet - Netscape
File Edit View Go Communicator Help
Bookmarks
Go to: http://www.softwired.ch/servlet/dateservlet

Date Servlet

This Page has last been accessed on Sat May 23 07:37:33 GMT 1998.

Figure 5-3 `DateServlet`'s output

Exercise: Creating Simple HTTP Servlets



Exercise objective – Running your first servlet called `DateServlet`.

✓ ***As of JSDK2.1,*** `servletrunner` ***has been superseded with the*** `startserver` ***command which contains a shell script for launching the 'runner'.***

Preparation

1. Change the directory to `labfiles/mod5-servlets/lab` and compile `DateServlet.java`.
2. Copy `DateServlet.class` to the Java Servlet Development Kit's `/servlets` directory, `/jsdk2.1/webpages/WEB-INF/servlets`.
3. Copy `DateServlet.html` to the `/webpages` directory.
4. Set the server name and port in `/jsdk2.1/default.cfg`, and execute the `startserver` command.

✓ ***More explicit instructions and troubleshooting information can be found in*** `SL301_SOL_LF/labfiles/mod5-servlets/lab/README`***.***

Tasks

Run `DateServlet` and test its functionality.

1. Open the `/jsdk2.1/webpages/DateServlet.html` page in Netscape and click on the link to the `DateServlet`.
2. Click on the `Reload` button several times and watch for the different response times, from the first to subsequent requests. Also check the date and time change for each request. Observe the different response times by shutting the `server` down and restarting it. Click on Reload in the browser again.

Tasks

Run `DateCounterServlet`

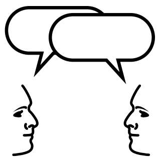
Servlets can save states between several requests. This is shown by adding a counter to the servlet and printing it on the Web page. You see how many times the page (the servlet) has been accessed.

Note – The servlet counts every request and not just requests from one person or machine. The exercise in this module shows how accesses from individual persons are counted.

---

1. Copy `DateServlet.java` to `DateCounterServlet.java`.
2. In the new file, change the class name from `DateServlet` to `DateCounterServlet`.
3. Add a counter as an instance variable to the servlet.
4. Add the counter's value to the HTML output.
5. Compile `DateCounterServlet` and test it using your browser and `DateCounterServlet.html`. (Note – You do not have to restart the `server`. The newly generated class file is loaded on demand.)

Exercise Summary



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

✓ ***Manage the discussion here based on the time allowed for this module, which was given in the "About This Course" module. If you find you do not have time to spend on discussion, then just highlight the key concepts students should have learned from the lab exercise.***

- Experiences

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

- Interpretations

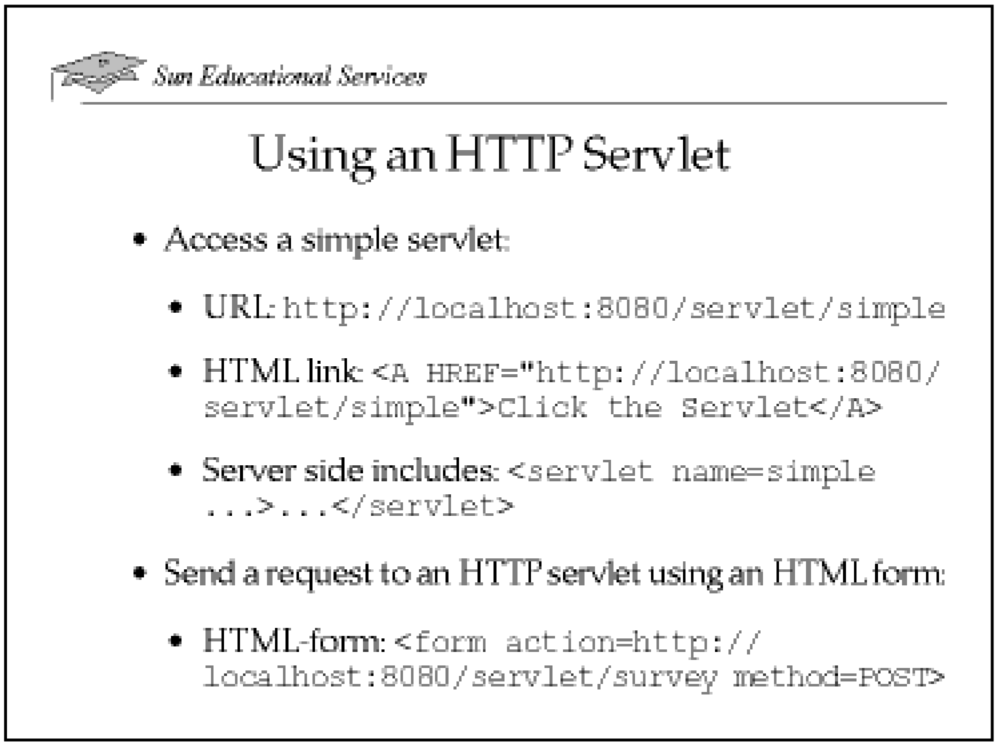
✓ ***Ask students to interpret what they observed during any aspects of this exercise.***

- Conclusions

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

- Applications

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



Sun Educational Services

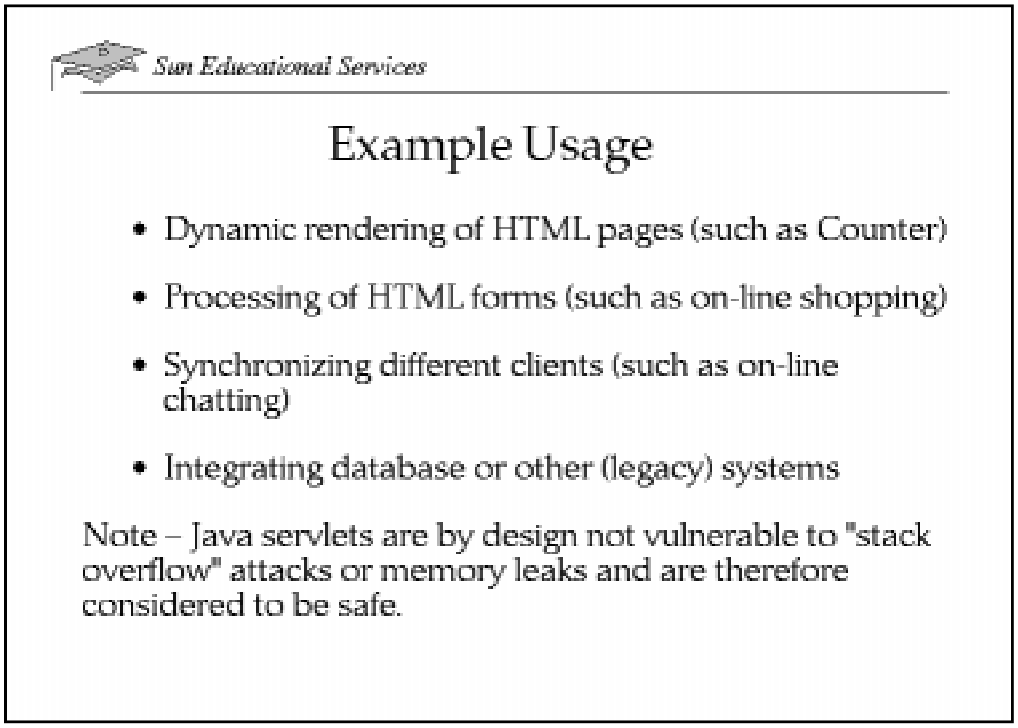
Using an HTTP Servlet

- Access a simple servlet:
  - URL: `http://localhost:8080/servlet/simple`
  - HTML link: `<A HREF="http://localhost:8080/servlet/simple">Click the Servlet</A>`
  - Server side includes: `<servlet name=simple ...>...</servlet>`
- Send a request to an HTTP servlet using an HTML form:
  - HTML-form: `<form action=http://localhost:8080/servlet/survey method=POST>`

Using an HTTP Servlet

HTTP servlets are usually invoked using a Web browser. Servlets only providing data, such as the `DateServlet` shown before, are referenced by a URL, such as `http://localhost:8080/servlet/dateservlet`. The exact path is determined by the administrator of the server and by the server's architecture. The Java Web Server, for example, uses the path `/servlet/`*`servletname`* to reference servlet *`servletname`*. The user either enters the URL or clicks on a hypertext link that leads to the servlet.

HTTP servlets that require user input are embedded in an HTML form. The servlet to be used to process the data entered is referenced by a usual HTML form tag. The class `javax.servlet.http.HttpServlet` supports only the *post* method to receive the data entered in the form.


Sun Educational Services

Example Usage

- Dynamic rendering of HTML pages (such as Counter)
- Processing of HTML forms (such as on-line shopping)
- Synchronizing different clients (such as on-line chatting)
- Integrating database or other (legacy) systems

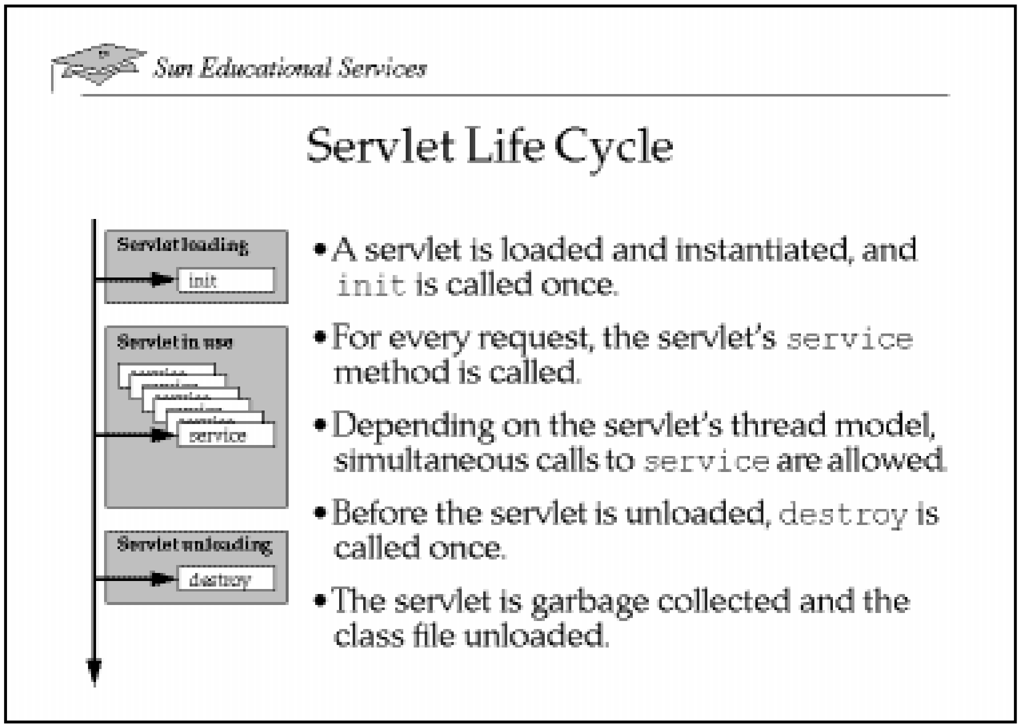
Note – Java servlets are by design not vulnerable to "stack overflow" attacks or memory leaks and are therefore considered to be safe.

Example Usage

When dynamic HTML pages must be created “on the fly” (while the program is running) or HTML forms must be processed, you can use HTTP servlets. For HTML page generation, two approaches are used:

- As in the `DateServlet` example, a complete page is generated by the servlet. The disadvantage is that any change of the static part of the HTML page must be accomplished by editing and recompiling the servlet.
- Some Web servers offer Server Side Includes (SSI), where the Web server parses the requested HTML pages. Special HTML tags can denote servlets, which are executed by the server. The HTML tag is replaced by the output of the servlet.

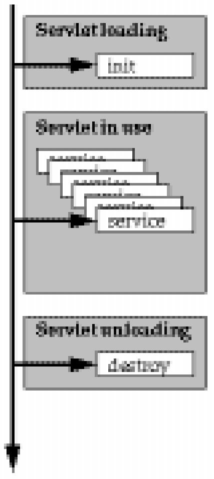
Many Web applications use some sort of HTML form processing, such as feedback or registration pages, online shopping, or access to other systems.



Sun Educational Services

Servlet Life Cycle

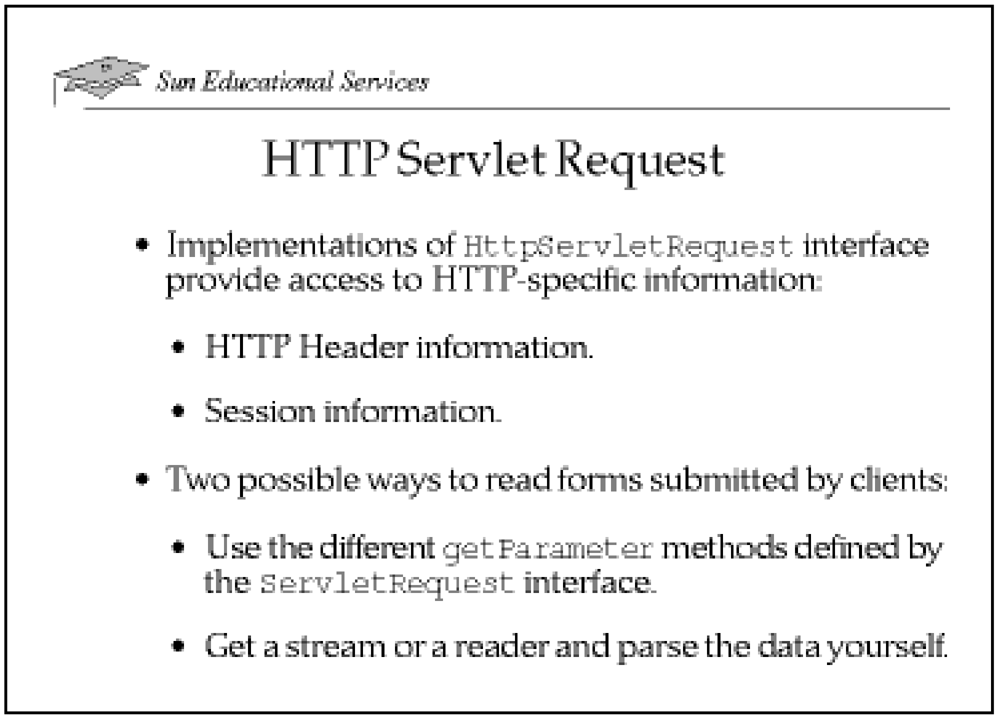
- A servlet is loaded and instantiated, and `init` is called once.
- For every request, the servlet's `service` method is called.
- Depending on the servlet's thread model, simultaneous calls to `service` are allowed.
- Before the servlet is unloaded, `destroy` is called once.
- The servlet is garbage collected and the class file unloaded.



Servlet Life Cycle

Servlets have their own life cycle that is independent of the server's life cycle. You can load and unload servlets at runtime. On the first request for a specific servlet, the server loads the appropriate class files either from a local disk or through the network. Exactly one instance of the servlet is created and the servlet's `init` method is called. A `ServletConfig` object is passed along to the servlet. If the servlet does not throw an `UnavailableException`, it is ready to answer requests through the `service` method. The server ensures that no `service` method call is performed before `init` returns.

Servlets can be unloaded at runtime. Unloading means unreferencing and garbage collecting the servlet and also removing the servlet's class file from the Java virtual machine. This enables servers to reload, for example, a newer version of the same servlet without restarting the server. Before a servlet is unloaded, the server calls the `destroy` method. `Destroy` removes all resources allocated either during runtime or by `init`. `Destroy` can be called before all service requests are completed. The servlet must either wait for any pending *requests* to complete or abort the requests. After `destroy` returns, the servlet is unloaded.


Sun Educational Services

HTTP Servlet Request

- Implementations of `HttpServletRequest` interface provide access to HTTP-specific information:
  - HTTP Header information.
  - Session information.
- Two possible ways to read forms submitted by clients:
  - Use the different `getParameter` methods defined by the `ServletRequest` interface.
  - Get a stream or a reader and parse the data yourself.

HTTP Servlet Request

The `HttpServletRequest` interface extends the underlying `ServletRequests` class by adding methods to access HTTP-specific header and status information, such as current session or authentication type information. This information is provided with every HTTP request.

To read data entered in an HTML form, two different approaches are offered:

- Obtain a stream or a reader object from which the data sent by the Web browser can be read. The input gathered this way must be parsed according to the HTML specification to get the data entered into the form.

- Use the various `getParameter` methods to get to the form data directly. This way the HTTP servlet parses the data sent by the Web browser and builds a dictionary using the field names as key and the form entries as values.

The two approaches cannot be mixed. Using the `getParameter` method is usually the simpler way of processing a request as the parsing is done by the request object.

Exercise: Snoop Servlet

Exercise objective – Access additional HTTP header fields in a request.

Tasks

Access the Snoop Servlet and Study Its Output

Complete the following steps:

1. Change the directory to `labfiles/mod5-servlets/lab.`
2. Compile `SnoopServlet.java`, and copy the resulting .class file to `/jsdk2.1/webpages/WEB-INF/servlets`.
3. Modify `SnoopServlet.html` as needed (modify the URL string), copy it to `/jsdk2.1/webpages,` and open it in Netscape. Click the link to the Snoop servlet and let the servlet snoop your request.

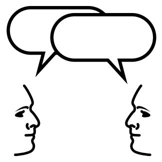
Run `DateLinkServlet`

`DateCounterServlet` gets another feature. Add a link in the HTML output that points back to the servlet itself like in the Snoop servlet.

1. Copy `DateCounterServlet.java` to `DateLinkServlet.java` and rename the class appropriately.
2. Look at `SnoopServlet.java` and add a link from the `DateLinkServlet` to itself in a similar way.

   Why should you not hardcode the link? (Optional)

Exercise Summary



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

✓ ***Manage the discussion here based on the time allowed for this module, which was given in the "About This Course" module. If you find you do not have time to spend on discussion, then just highlight the key concepts students should have learned from the lab exercise.***

- Experiences

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

- Interpretations

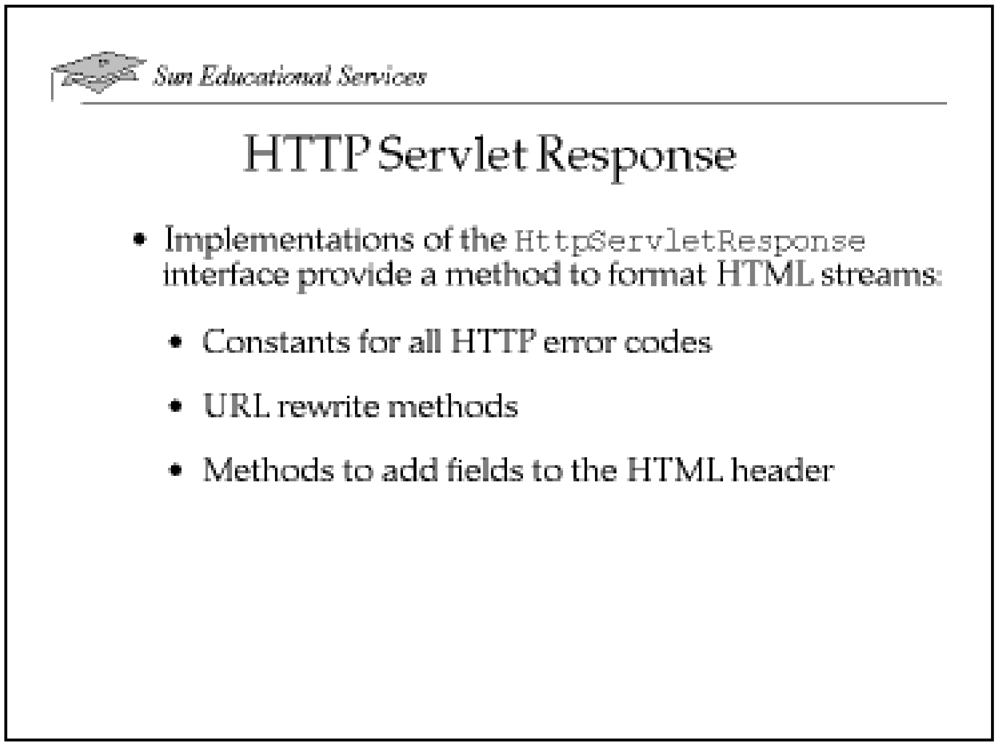
✓ ***Ask students to interpret what they observed during any aspects of this exercise.***

- Conclusions

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

- Applications

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



Sun Educational Services

HTTP Servlet Response

- Implementations of the HttpServletResponse interface provide a method to format HTML streams:
  - Constants for all HTTP error codes
  - URL rewrite methods
  - Methods to add fields to the HTML header

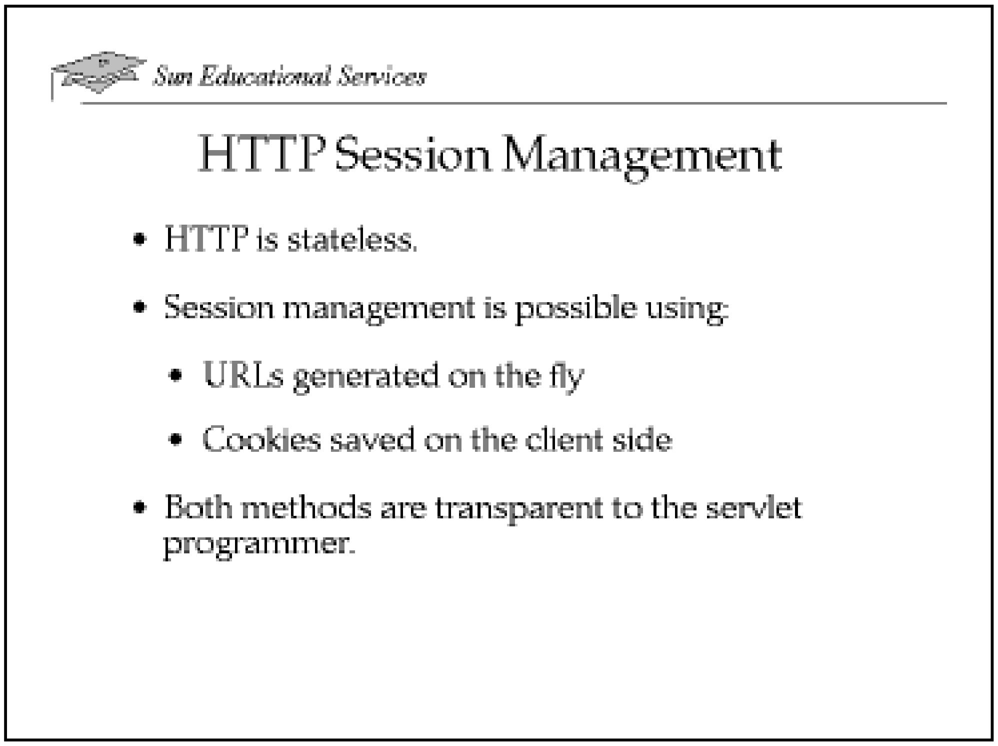
Client Interaction

HTTP Servlet Response

The `HttpServletResponse` interface provides methods to format and send HTTP-specific header and error codes. The HTTP error codes are provided as constants in the interface.

To send data to the Web client, a stream or a writer object is provided by the response object. Before retrieving a stream or a writer from the response object, you must set the content's Multipurpose Internet Mail Extension (MIME) type using the `setContentType` method.

✓ ***MIME means Multipurpose Internet Mail Extension. MIME is used for providing a mapping of what helper application is to be launched when a particular data format (GIF, audio, and so on) is received.***



Sun Educational Services

HTTP Session Management

- HTTP is stateless.
- Session management is possible using:
  - URLs generated on the fly
  - Cookies saved on the client side
- Both methods are transparent to the servlet programmer.

HTTP Session Management

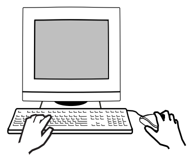
HTTP requests are stateless. However, there are several methods of building a session management layer on top of HTTP. You cannot build applications, such as online shopping without sessions because you must track user state information throughout multiple page requests.

✓ ***For example, when the abstraction of a "shopping cart" is to be provided.***

Sessions are either implemented using cookies or special URLs that are generated on the fly. The session management facilities within the HTTP servlets package do both in a transparent manner. Implementations of the `Session` interface manage sessions. They provide methods to identify sessions and store and retrieve data associated with the session.

Session management in conjunction with user authentication and secured communication (HTTPS) allow you to develop sophisticated Web-based applications.

Exercise: Session Servlet



Exercise objective – Learn about sessions.

Preparation

This exercise requires a Web browser that supports cookies.

Tasks

Use the Session Servlet

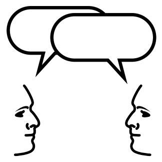
Access the session servlet and compare it to `DateCounterServlet`.

1. Change the directory to `labfiles/mod5-servlets/lab.`
2. Copy `SessionServlet.html` to /jsdk2.1/webpages, open it in Netscape, and follow the link to the servlet.
3. Try to connect to your neighbor's `server` and let your neighbors connect to yours (if the setup in your classroom allows it). If you do the same thing with `DateCounterServlet`, what will be the difference? Check your answer.

✓ ***The*** `DateCounterServlet` ***sums up all connections. The session Servlet counts all accesses per user.***

4. Study the source code and find out how session management is done using `HttpServlets`.

Exercise Summary



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

✓ ***Manage the discussion here based on the time allowed for this module, which was given in the "About This Course" module. If you find you do not have time to spend on discussion, then just highlight the key concepts students should have learned from the lab exercise.***

- Experiences

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

- Interpretations

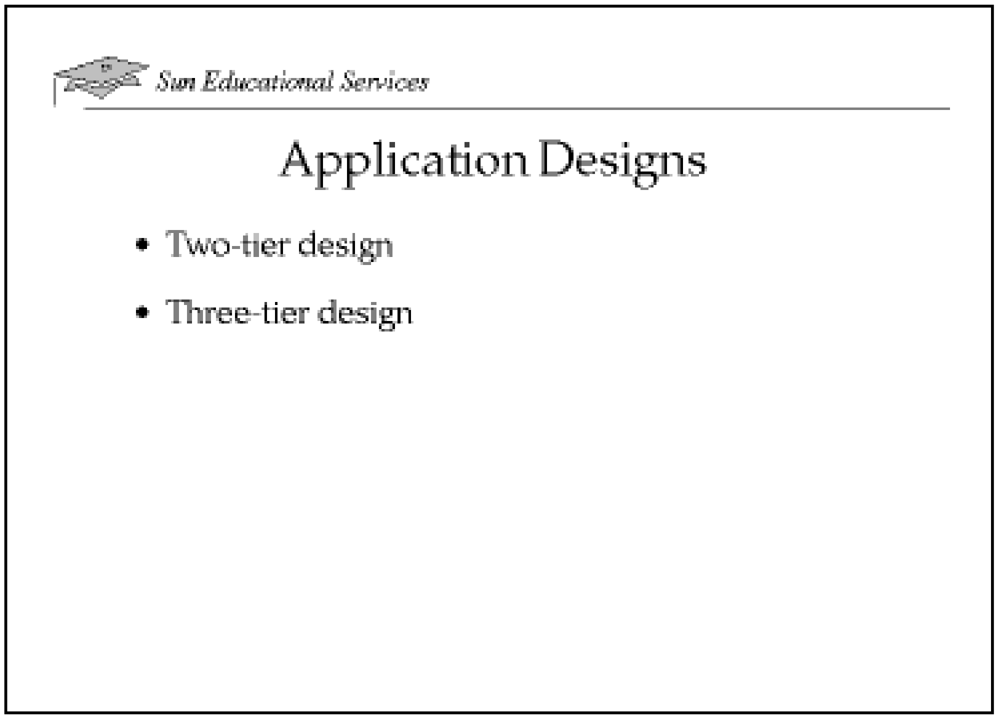
✓ ***Ask students to interpret what they observed during any aspects of this exercise.***

- Conclusions

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

- Applications

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



Sun Educational Services

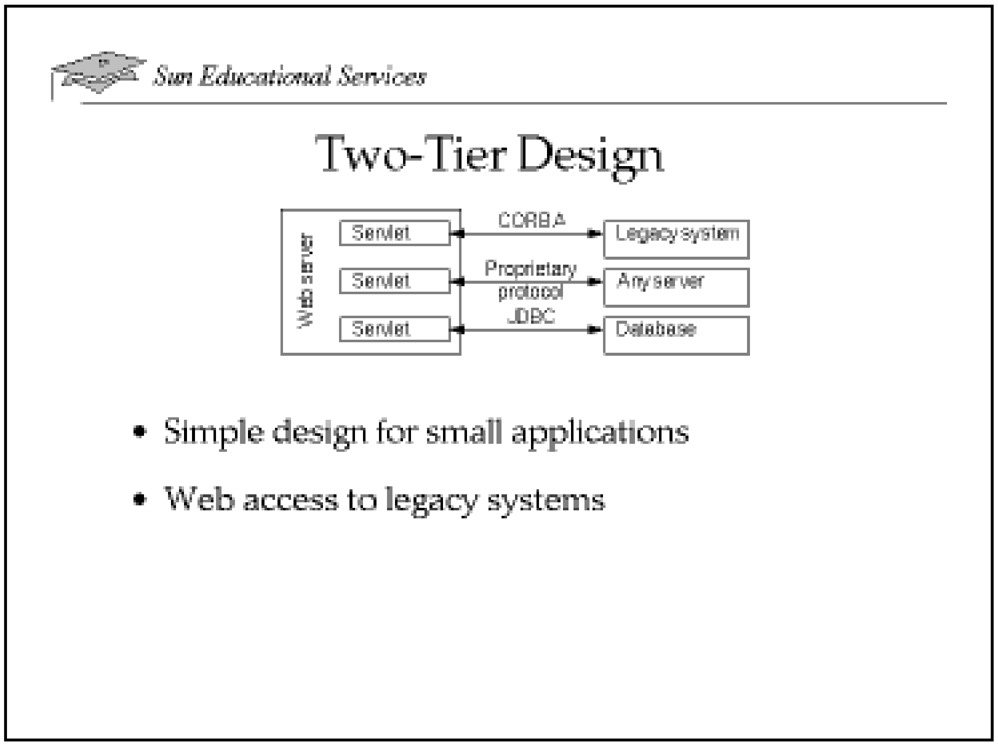
Application Designs

- Two-tier design
- Three-tier design

Application Designs

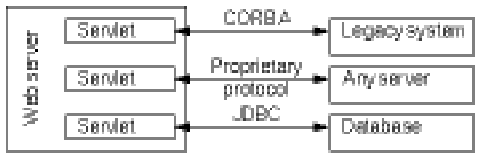
The servlets API is generic; therefore, you can build many different applications with completely different designs. This section looks at application designs using HTTP servlets.

An application consists of a client, which does user interaction; a server handling data; and some business or application logic. The Web browser and the servlet are considered the first *tier*, because the browser renders only HTML and feeds data entered by the user, and is not part of the system to be built.



Sun Educational Services

Two-Tier Design



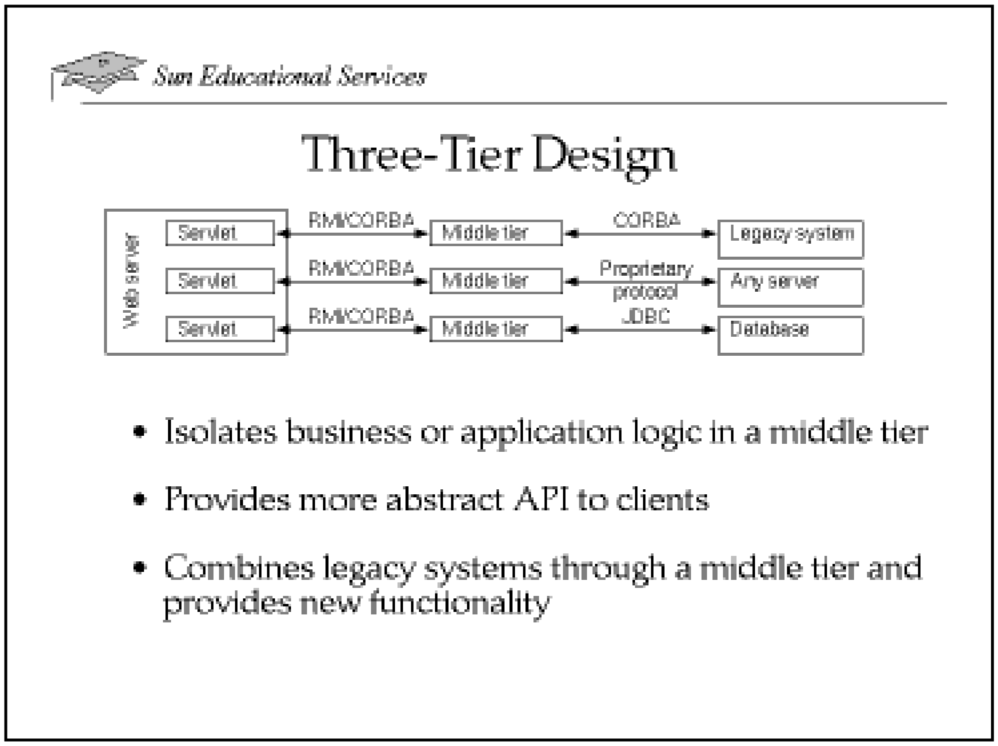
- Simple design for small applications
- Web access to legacy systems

Two-Tier Design

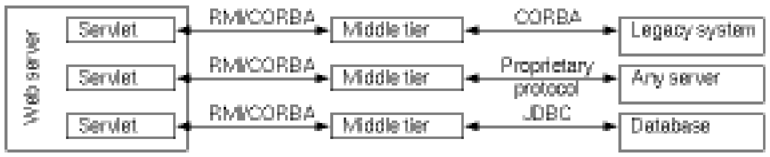
In a two-tier architecture, the servlet interacts with the users and gets and sends data to another system: a database, for example. This design is best suited for providing a Web front end to legacy systems, which already contain all the business and application logic.

Here servlets correspond directly to applets, which would be a good alternative, but might be considered inappropriate if a browser capable of running Java programs is not available on the client side, or when the applet security model is considered too restrictive.

Of course, you can enhance servlets with business logic and directly access databases using JDBC. For small applications, with the servlet as the only client of the database, this design might be a viable solution. For larger applications with many clients, it would be a bad choice, because the business logic needs to be rebuilt in the other clients as well.


Sun Educational Services

Three-Tier Design

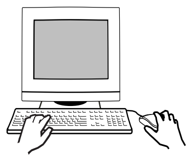


- Isolates business or application logic in a middle tier
- Provides more abstract API to clients
- Combines legacy systems through a middle tier and provides new functionality

Three-Tier Design

Within a three-tier design using HTTP servlets, the business or application logic is isolated in a middle tier. The middle tier offers a more abstract API to the top tier using RMI or CORBA, for example. The top tier provides for user interaction and forwards the user’s requests to the middle tier by API calls. The top tiers are small and lightweight and can be implemented in many different ways, such as full Java programs, applets, programs built using some other language, and HTTP servlets. The bottom tier is used to store and retrieve data. Instead of rebuilding legacy systems, a middle tier can integrate several systems and databases and build new applications on top of them. The new API is then used by the clients to interact with the end users.

Exercise: LeShop Servlet (Optional)



Exercise objective – Show how data entered in a form could be processed by an application.

Preparation

Check that the path specified in the `servlet.properties` for the `LeShopServlet` exists. Depending on the setup in your classroom, your path might be different. Look in the `properties` file or ask your instructor.

Tasks

Use the LeShop Servlet to Store Data in a File

Fill out the HTML form in `LeShopServlet.html` and let the servlet process the data entered.

1. Change the directory to `labfiles/mod5-servletslab.`
2. Open `LeShopServlet.html` in your browser.
3. Fill out the form and send it to the servlet for further processing.
4. Watch the file where the data entered is stored. See how it grows when more data is entered.
5. Study the source code and learn how data entered in a form is extracted and processed.

Exercise Summary

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

✓ ***Manage the discussion here based on the time allowed for this module, which was given in the "About This Course" module. If you find you do not have time to spend on discussion, then just highlight the key concepts students should have learned from the lab exercise.***

- Experiences

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

- Interpretations

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

- Conclusions

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

- Applications

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

Check Your Progress

Before continuing on to the next module, check that you were able to accomplish the following in this module:

- ❑ Understand the basic concepts of servlets
- ❑ Create your own servlets for Web-based applications
- ❑ Compare HTTP servlets to other technologies, such as CGI

Think Beyond

How could you take advantage of servlets in an existing application?

Object Bus Systems 6

Objectives

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

- Understand the fundamentals of the Object Bus model
- Understand its main application areas
- Explain the differences between iBus, RMI, and CORBA

Object Bus systems enable developers to write applications that interact in a one-to-many and many-to-many fashion by transmitting events through ubiquitous communication channels.

Relevance

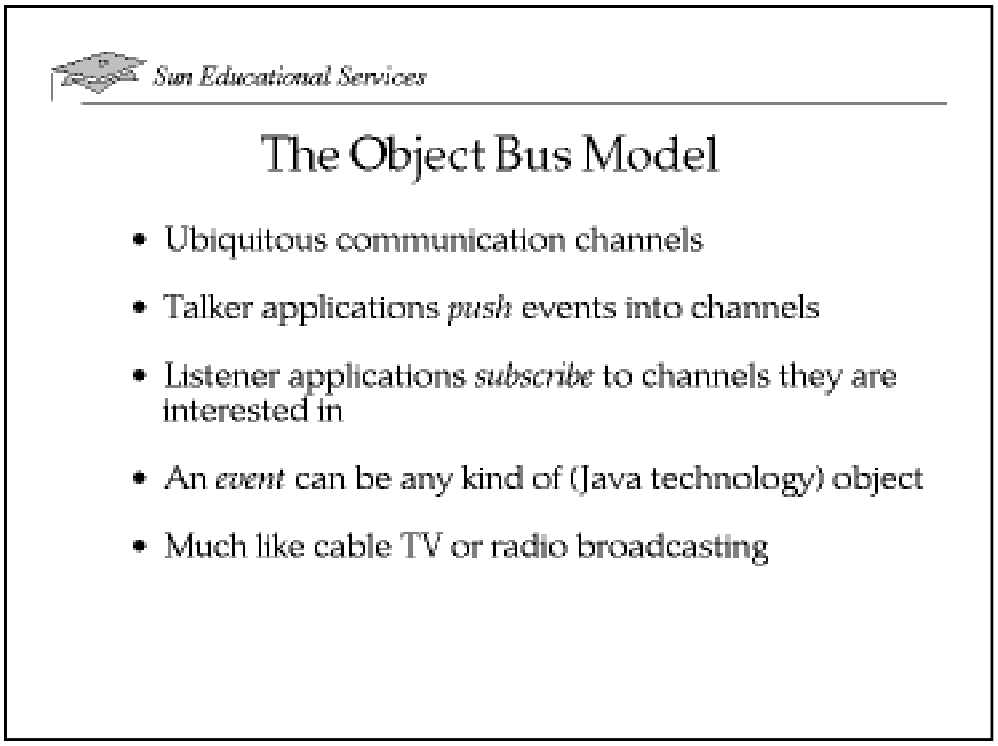
Discussion – Consider the following questions:

- How do you design an RMI application that distributes near real-time events, such as stock quotes to a dynamically changing set of listener applications?

✓ ***You need to provide a network-centric broker daemon to which the listeners of the events connect. A listener typically is a graphical user interface on the workstation of a stock trader. When the talker application has a new stock quote to deliver, it requests that the broker daemon distribute that event to any registered listener application. Another solution consists of the receivers setting up an RMI connection to the talker application, to periodically poll the talker for new quotes.***

- What impact does this have on network load?

✓ ***Each quote results in (at least) one RMI request from the talker to the daemon and in one RMI per registered listener, to "broadcast" the quote. The polling solution also results in considerable network traffic.***


Sun Educational Services

The Object Bus Model

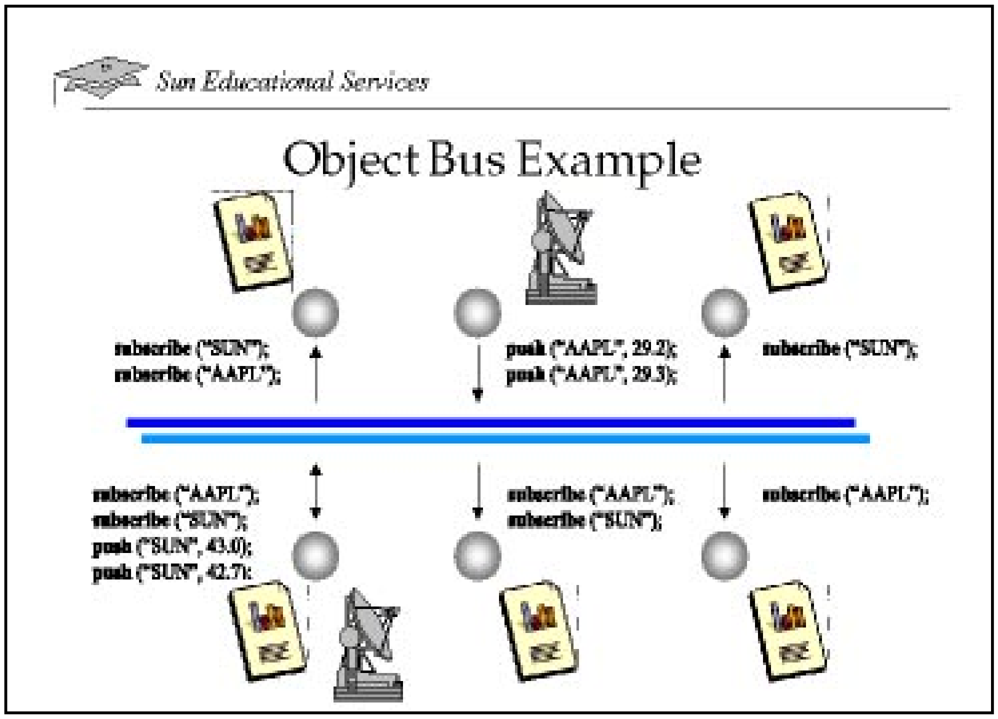
- Ubiquitous communication channels
- Talker applications *push* events into channels
- Listener applications *subscribe* to channels they are interested in
- An *event* can be any kind of (Java technology) object
- Much like cable TV or radio broadcasting

The Object Bus Model

The basic communication paradigm of CORBA and RMI is “request and reply.” An application explicitly requests specific data from a server object by invoking a method on a local communication stub. However, applications, such as financial data delivery require an approach where listener applications *subscribe* for events, and talker applications *push* events to their listeners through ubiquitous communication channels. This is called the Object Bus model.

In this model, an event can cause any kind of (Java technology) object, (such as a stock quote, an audio data packet, an alert indicating that an aircraft engine is getting too hot, or even a user interface component) to be displayed by the listeners.

Object Bus middleware functions much like radio transmission. A radio station transmits on a certain channel and radio listeners tune into that channel. The radio station does not need to know its receivers to transmit. The listeners do not need to know the location of their transmitter.



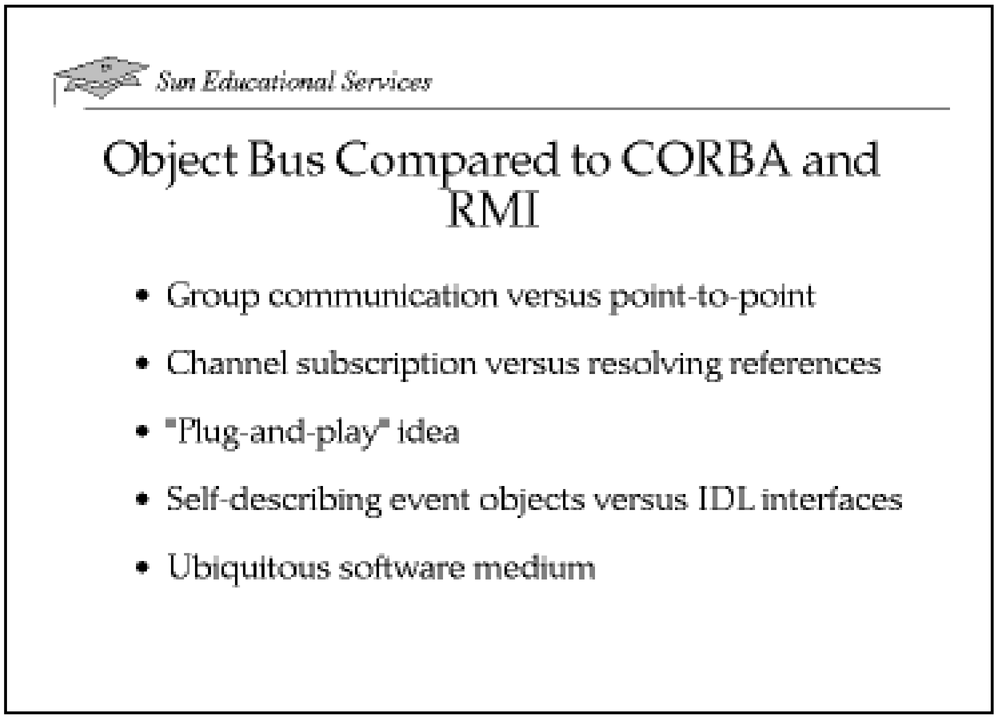
Object Bus Example

Consider the stock quote example. With Object Bus middleware, you can implement the application without providing a network-centric broker daemon, and without polling the talker for new events. The Object Bus provides ubiquitous communication channels to which listeners of stock quotes (user interfaces on the traders' desks) subscribe and to which producers of stock quotes (so-called *data collectors*) transmit.

The satellite dishes denote the equipment necessary to receive financial events from a data provider, such as Reuters. The bar charts depict graphical user interfaces residing on traders' desks.

✓ ***In this example there are two channels, one for carrying Apple Computer stock quotes and one for Sun Microsystems quotes.***

✓ ***An assumption is that events are transmitted with a real multicast protocol such as IP multicast. The network load is then constant, no matter how many listeners are active.***



Sun Educational Services

Object Bus Compared to CORBA and RMI

- Group communication versus point-to-point
- Channel subscription versus resolving references
- "Plug-and-play" idea
- Self-describing event objects versus IDL interfaces
- Ubiquitous software medium

Object Bus Compared to CORBA and RMI

There are a few fundamental differences between Object Bus, and CORBA and RMI. First, Object Bus systems are based on a group communication model where talkers transmit events to a dynamically growing and shrinking group of listeners. The logic necessary to support registration and unregistration of listeners is transparent to the developer of Object Bus applications.

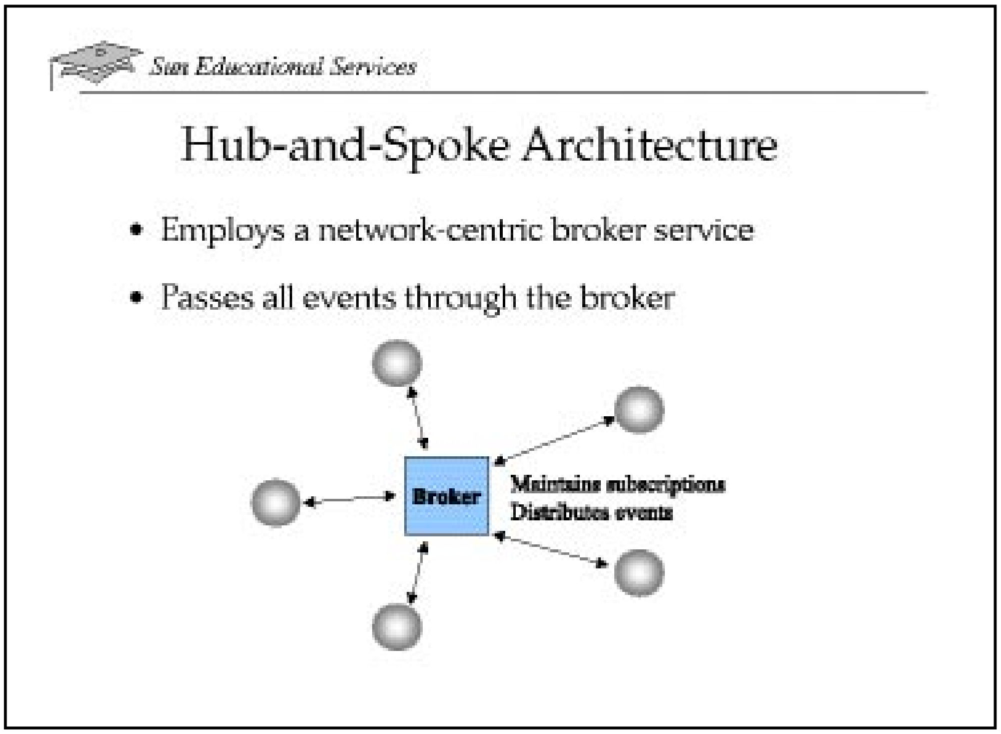
In CORBA and RMI, a server object installs its object reference in a name server. Clients retrieve that reference from the name server to invoke the server. The Object Bus model does not make use of any object references. Channels are typically named using character strings, such as `/dataproviders/financial/stockquotes/SUN`. Applications plug into a channel simply by specifying its name.

Events are self-describing and self-contained. Typically, Object Bus systems do not provide any interface definition language (IDL). A talker can change the format of an event at runtime. However, the listeners must be prepared to cope with format changes and reject any event that does not provide the information needed by the listener.

The Object Bus model provides the abstraction of a ubiquitous software medium extending through a whole company. Applications "tap" into the medium at any location to send and receive events.

✓ ***A disadvantage is that type checking of event objects cannot be done at compile time any more: in complex object bus applications the listeners typically need to cope with event format changes due to software upgrades and the like.***

✓ ***An advantage is that Object Bus systems can be extended at runtime since they do not depend on static invocation stubs, as is the case in CORBA and RMI. Object Bus systems are thus better suited to support a 7 days/24 hours per day of continuous operation since talkers and listeners can be added (and even migrated) at runtime.***



Sun Educational Services

Hub-and-Spoke Architecture

- Employs a network-centric broker service
- Passes all events through the broker

Object Bus Architectures

Object Buses are implemented in two different ways: hub-and-spoke and multicast bus.

Hub-and-Spoke Architecture

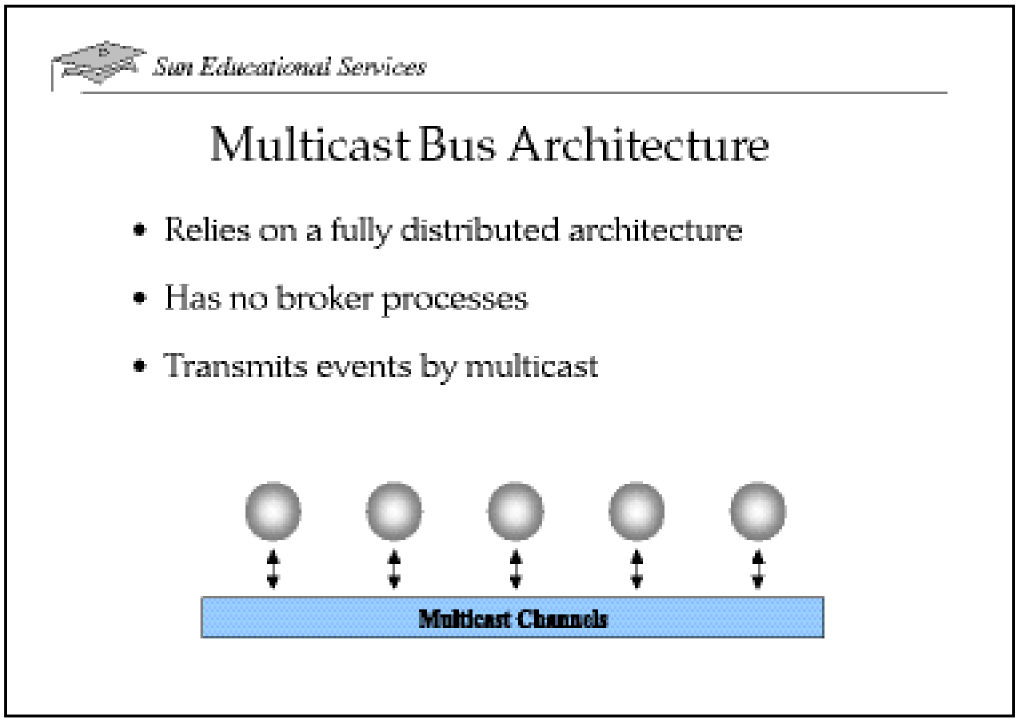
The straightforward approach provides a network-centric "hub" daemon. Listener applications connect to the hub; talker applications use the hub to deliver an event on their behalf. There might be a hub per channel, or a hub serving multiple channels simultaneously.

✓ ***Hub-and-spoke corresponds to the RMI broker architecture outlined in the "Relevance" section.***

The advantage of the hub-and-spoke architecture is that it is straightforward to implement with RMI, CORBA, or plain TCP/IP sockets. Another advantage is that you can use the hub for accounting and access control purposes, because all events and subscriptions pass through the hub.

The disadvantage of hub-and-spoke is that each quote results in (at least) one RMI request from the talker to the daemon, and in one RMI per registered listener, to deliver the quote. RMIs are performed whenever a listener registers or unregisters. This solution is unsuitable in real-world scenarios where hundreds of different channels of information are to be supported, each carrying real-time events to many listeners. Furthermore, the hub embodies a single point of failure.

✓ ***As was pointed out in the "Relevance" section, hub-and-spoke can also be implemented by having the listeners poll the talker at regular time intervals, to check whether there are any new events. This approach does not require any hub but consumes even more network resources.***



Sun Educational Services

Multicast Bus Architecture

- Relies on a fully distributed architecture
- Has no broker processes
- Transmits events by multicast

Multicast Channels

Multicast Bus Architecture

A more scalable solution consists of deploying a network multicast protocol, such as IP multicast. This approach is more easily scalable, because with IP multicast, the network traffic is independent of the number of listeners subscribed to a channel.

✓ ***IP multicast deploys hardware-enabled multicast which is available in Ethernet and Token Ring.***

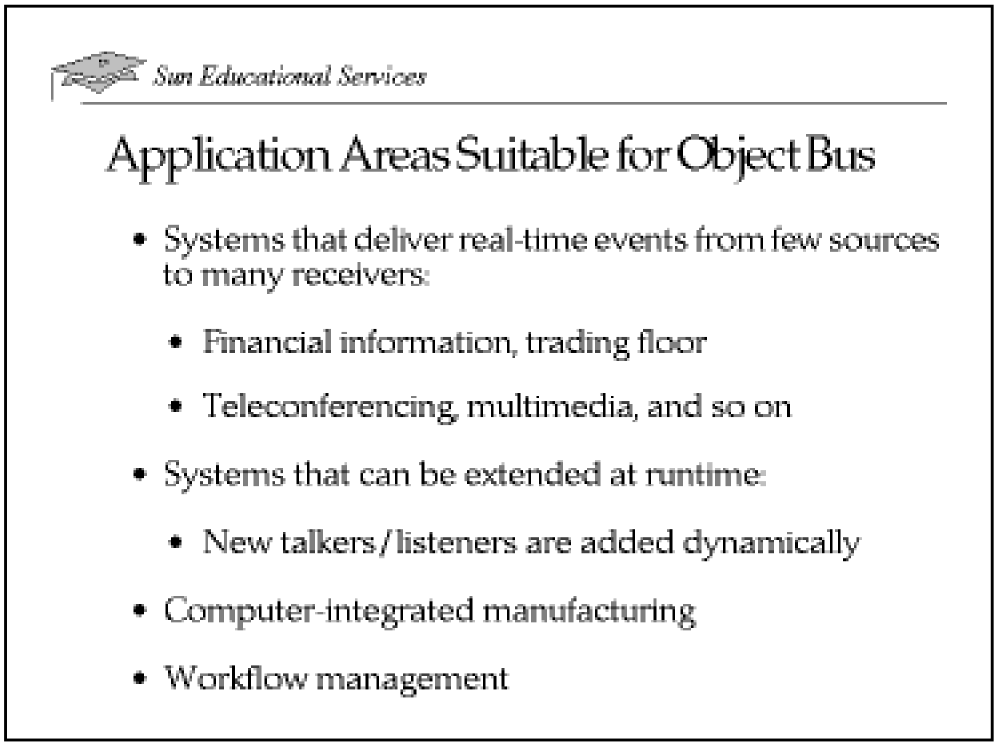
The disadvantage of the multicast bus architecture is that such an Object Bus is more difficult to implement. Today’s operating systems support only unreliable multicast, where messages can get lost on overload or on network problems.

The advantage is that IP multicast, due to its fully distributed and ubiquitous architecture, ideally supports the Object Bus model. In addition, more efficient usage of network resources is ensured.

✓ ***IP multicast is well supported by Windows NT, Windows 95/98, all major UNIX environment types, Linux, MacOS, and so on.***

Notes

- ✓ ***A special range of IP addresses (the class D) has been reserved for IP multicast. The difference from conventional IP addresses is that applications running in an Intranet can all bind to the same class D address and receive IP packets sent to the address. IP multicast addresses thus are location independent.***
- ✓ ***IP multicast is an unreliable protocol. Messages can get lost or duplicated. It is up to the Object Bus middleware to provide the necessary reliability protocols. There is no TCP over IP multicast. One typically creates UDP sockets to communicate through IP multicast.***



Sun Educational Services

Application Areas Suitable for Object Bus

- Systems that deliver real-time events from few sources to many receivers:
  - Financial information, trading floor
  - Teleconferencing, multimedia, and so on
- Systems that can be extended at runtime:
  - New talkers/listeners are added dynamically
- Computer-integrated manufacturing
- Workflow management

Application Areas Suitable for Object Bus

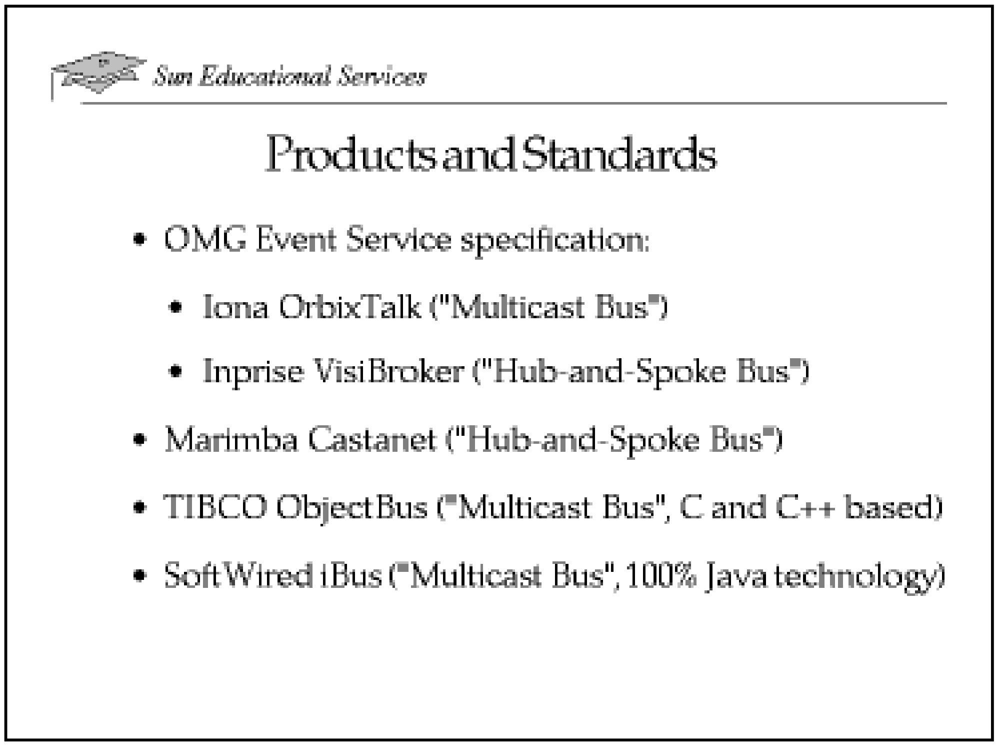
The Object Bus model is well suited for applications that require one-to-many or many-to-many transmission of real-time events to a dynamically changing set of receivers. Examples are systems that push financial events from a data collection unit to many trader workstations. Teleconferencing applications and distributed multimedia are other examples.

Another interesting application area is systems that need to be extended with new functionality at runtime, without shutting the system down. This is important for medical systems, flight reservation systems, and other applications that require high availability.

Control systems for computer-integrated manufacturing (CIM) and Workflow management are other examples.

✓ ***Workflow management systems are programmed to support the main work flows in a company, such as a clerk taking a rental-car reservation by phone, checking a database for a suitable car, assigning the car to the customer, and instructing the staff to prepare the car.***

✓ ***Numerous large banks have been deploying this kind of middleware for many years.***


Sun Educational Services

Products and Standards

- OMG Event Service specification:
  - Iona OrbixTalk ("Multicast Bus")
  - Inprise VisiBroker ("Hub-and-Spoke Bus")
- Marimba Castanet ("Hub-and-Spoke Bus")
- TIBCO ObjectBus ("Multicast Bus", C and C++ based)
- SoftWired iBus ("Multicast Bus", 100% Java technology)

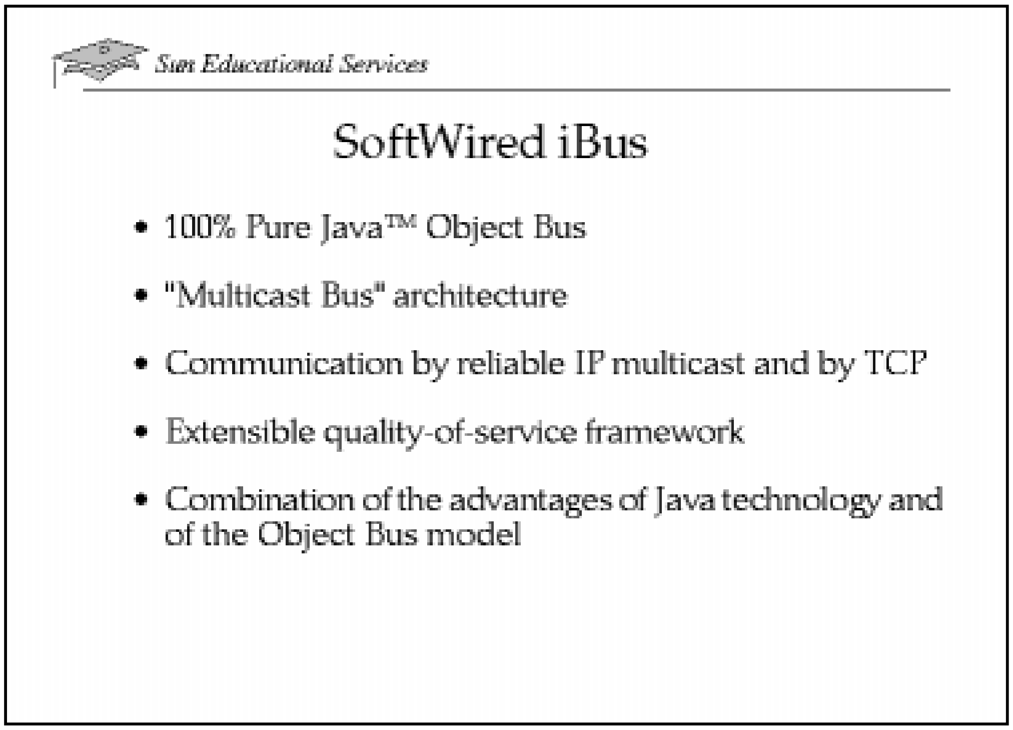
Products and Standards

✓ ***Hints on the architecture of the product appear in parentheses. OrbixTalk is a pure C++ product. VisiBroker Event Service is available both for C++ and Java programming language. Castanet is a pure Java technology middleware. TIBCO ObjectBus is implemented in C and C++. iBus is a lightweight "pure Java" Object Bus.***

The Object Management Group (OMG) Event Service standard defines an Object Bus infrastructure on top of CORBA (`www.omg.org`). The Event Service specification distinguishes event producers, event consumers, and communication channels. A variety of implementations of the standard are available today; for example, OrbixTalk from Iona Ltd. (`www.iona.com`) and VisiBroker Event Service from Inprise (`www.inprise.com`).

Marimba's Castanet (`www.marimba.com`) is a Java technology Object Bus targeted at distributing software upgrades and other data from one source to many listeners.

TIBCO ObjectBus is a CORBA-based middleware in use at many financial companies (`www.tibco.com`). SoftWired iBus is described on the next overhead.



Sun Educational Services

SoftWired iBus

- 100% Pure Java™ Object Bus
- "Multicast Bus" architecture
- Communication by reliable IP multicast and by TCP
- Extensible quality-of-service framework
- Combination of the advantages of Java technology and of the Object Bus model

SoftWired iBus

The *iBus* Object Bus middleware by SoftWired Ltd. (`www.softwired.ch`) is a pure Java technology implementation of the multicast bus architecture. Its main characteristics are compact size, easy installation, and an intuitive API.

iBus is designed so that it can run on top of various communication protocols, notably reliable IP multicast and TCP. You can extend it with new quality-of-service features, such as proprietary encryption engines.

iBus provides an intuitive API allowing developers to subscribe to communication channels and to transmit arbitrary Java objects using those channels.

✓ ***iBus mainly consists of one JAR file. No naming services or broker daemons need to be installed in order to run iBus applications. Since it is pure Java technology it can run on any platform providing a Java runtime.***

Sample iBus Application

The Talker Program

The following example provides a simple but complete iBus application that transmits a stock quote on a channel carrying financial information. Before any data can be transmitted, you must declare a communication stack. In the constructor of class `Stack,` a quality-of-service string is specified. Reliable multicast is used because the example application requires one-to-many communication, and you want to ensure that all quotes are received by all listeners, in spite of packet loss, which might happen at the IP multicast level.

✓ ***iBus employs a negative-acknowledgments reliable multicast protocol but can be configured for other protocols as well.***

You must declare a URL for the channel to which you want to transmit. `registerTalker` informs iBus that transmission will take place using the stack and to the given channel.

Now it is time to create a posting object to fill with a quote for Sun Microsystems shares. Transmit the posting by invoking the `push` operation on the stack object. `push` is an asynchronous operation meaning that it returns immediately without waiting until the listeners have received the posting.

✓ ***iBus also provides a synchronous (that is, blocking) push operation as well as a request/reply style operation, called `pull`.***

The iBus protocol stack takes care of fragmenting large postings into chunks fitting an IP datagram size, recovering from packet loss, bringing packets in a first-in-first-out order, and so on. More sophisticated stacks might perform failure detection, posting encryption, and so on.

```
import iBus.iBusURL;
import iBus.Posting;
import iBus.Stack;

public  class Talker {
    public static void main(String [] argv)
    throws Exception {
        String quote = new String("SUN: 42.7");

        // handle any command line argument:
        if(argv.length > 0) quote = argv[0];

        // create an iBus protocol stack for
        // reliable multicast:
        Stack stack = new Stack("Reliable");

        //create an iBus URL for the destination
        //channel:
        iBusURL url = new iBusURL(
            "ibus://226.1.2.3/financial/Text");

        //open the channel:
        stack.registerTalker(url);

        //create a Posting to hold a quote string:
        Posting posting = new Posting();
        posting.setLength(1);
        posting.setObject(0, quote);

        //push the quote through the iBus channel
        //"url":
        for (;;) {
            stack.push(url, posting);
            Thread.currentThread().sleep(2000);
        }
    }
}
```

The Listener Program

This application provides a listener that subscribes to the channel and displays stock quotes. iBus implements an *upcall* event-handling model in which `iBus.Receiver` objects are subscribed to with a channel to receive postings. iBus invokes the `Receiver.dispatchPull` method when a posting arrives on a channel to which the receiver object is subscribed.

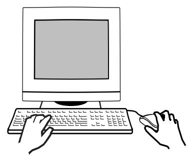
A class `QuoteReceiver` that implements interface `iBus.Receiver` is defined. The `dispatchPush` operation extracts the quote string from the posting and writes it to the output stream.

Now the main body of the listener creates an instance of `QuoteReceiver` and subscribes it with the stack object and with the channel. Finally, `waitTillExit` is called to tell iBus to suspend the main thread and to wait for postings to arrive.

✓ ***Without calling*** `waitTillExit` ***the application would terminate immediately without receiving any postings.***

```
import iBus.iBusURL;
import iBus.Posting;
import iBus.Receiver;
import iBus.Stack;

public class Listener {
    public static void main (String [] argv)
    throws Exception {
        Stack stack = new Stack("Reliable");
        QuoteReceiver receiverObject =
            new QuoteReceiver();
        iBusURL url = new iBusURL(
            "ibus://226.1.2.3/financial/Text");

        stack.subscribe (url, receiverObject);
        stack.waitTillExit();
    }
}


// A listener class to receive quotes:
class QuoteReceiver implements iBus.Receiver {
    public void dispatchPush(iBusURL source, Posting p) {
        // to display the quote string:
        System.out.print("QuoteReceiver: got a quote: ");
        System.out.println(p.getObject(0));
    }

    // not used in exercise
    public Posting dispatchPull(iBusURL c, Posting p) {
        return null;
    }
    public void error(iBusURL channel, String details) {}
};
```

Exercise: iBus



Exercise objective – Experience using an Object Bus.

Tasks

Compile and Run the Sample Application

Complete the following steps:

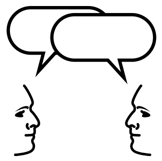
1. Change the directory to `labfiles/mod6-objectbus/lab/simple/`.
2. Compile `Talker.java` and `Listener.java`.
3. Start the Listener program in a console window.
4. Start the Talker program in a console window.
5. Watch the output. How many Talker applications do you see?

✓ ***Each Talker in the network should show up immediately on every Listener. This may surprise students and is a good demonstration for object bus technology.***

Add and/or Remove Talkers and Listeners (Optional)

You can add more Listeners and Talkers now. You can shut a Talker down, relocate it to another console window, for example, and restart it without disturbing the Listeners.

Exercise Summary



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

✓ ***Manage the discussion here based on the time allowed for this module, which was given in the "About This Course" module. If you find you do not have time to spend on discussion, then just highlight the key concepts students should have learned from the lab exercise.***

- Experiences

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

- Interpretations

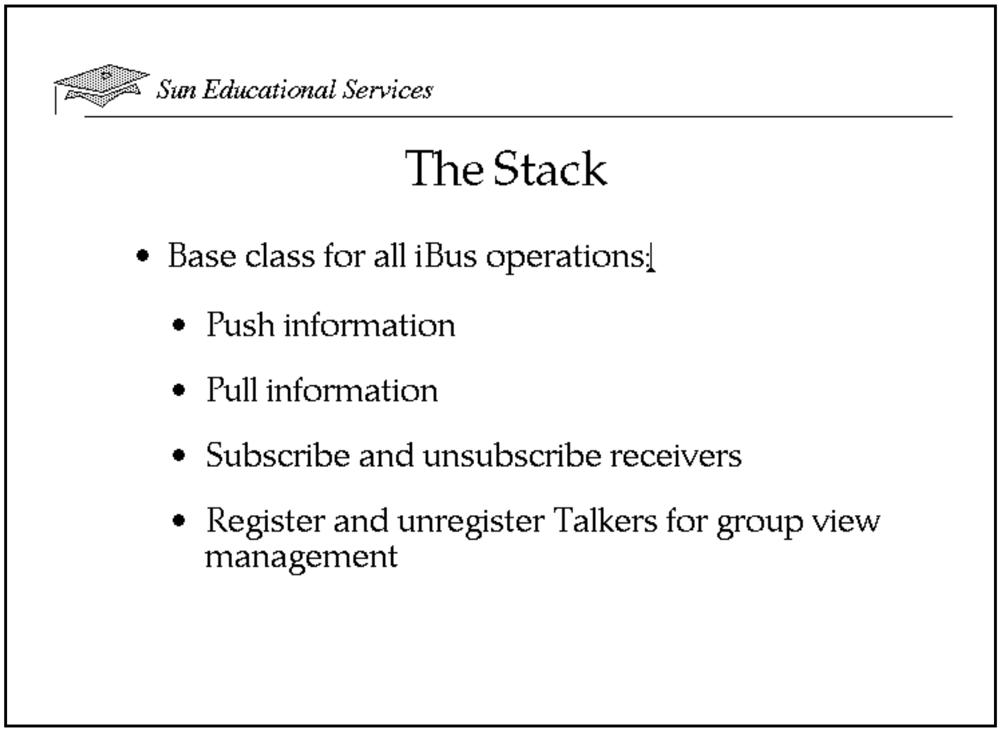
✓ ***Ask students to interpret what they observed during any aspects of this exercise.***

- Conclusions

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

- Applications

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


Sun Educational Services

The Stack

- Base class for all iBus operations:
  - Push information
  - Pull information
  - Subscribe and unsubscribe receivers
  - Register and unregister Talkers for group view management

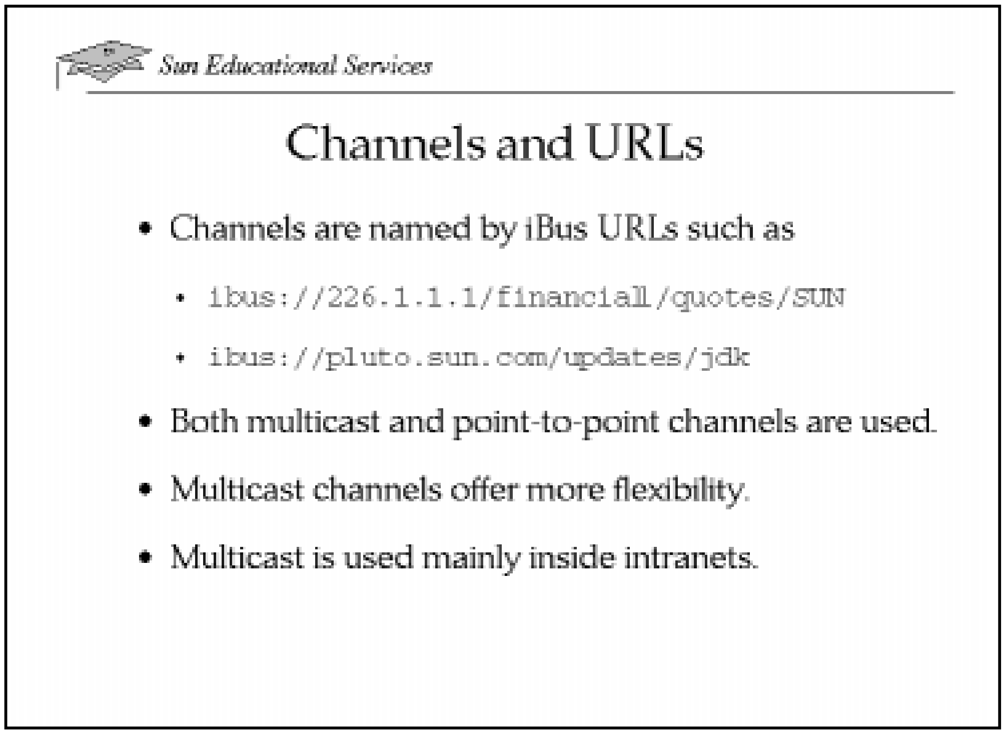
The iBus API

The Stack

iBus applications communicate through so-called communication *stacks*. A stack defines a quality of service, such as reliable multicast, real-time multicast, reliable point-to-point streaming, data encryption, and so on. iBus is extensible in respect to incorporating yet unsupported qualities of services.

The following is an example of the iBus API for sending and receiving events.

```
  package iBus;

public class Stack {
     /** Create a protocol stack the application can use to send
     * and receive Postings. Specify a quality of service such as
     * reliable multicast or reliable streaming.
     */
     public Stack(String qos) { ... }

     /** Register as a talker of a channel. This is to be done
     * before any data can be pushed to the channel.
     */
     public void registerTalker(iBusURL channel) { ... }

     /** Unregister as a talker.
     */
     public void unregisterTalker(iBusURL channel) { ... }

     /** Subscribe a listener object to a channel.
     */
     public void subscribe(iBusURL channel, Receiver rcv) { ... }

     /** Unsubscribe a listener object.
     */
     public void unsubscribe(iBusURL channel, Receiver rcv) { ... }

     /** Push a posting through a channel. (one-way and asynchronous)
     */
     public void push(iBusURL channel, Posting p) { ... }

     /** Pull a posting through a channel. Much like RMI.
     */
     public Posting[] pull(iBusURL channel, Posting request) { ... }
}
```



Sun Educational Services

Channels and URLs

- Channels are named by iBus URLs such as
  - `ibus://226.1.1.1/financial/quotes/SUN`
  - `ibus://pluto.sun.com/updates/jdk`
- Both multicast and point-to-point channels are used.
- Multicast channels offer more flexibility.
- Multicast is used mainly inside intranets.

Channels and URLs

When a Java talker application wants to communicate using iBus, it must specify a transmission channel using a URL. iBus URLs obey the following format:

`ibus:``//`*`address`*`[:`*`port`*`]/`*`subject`*

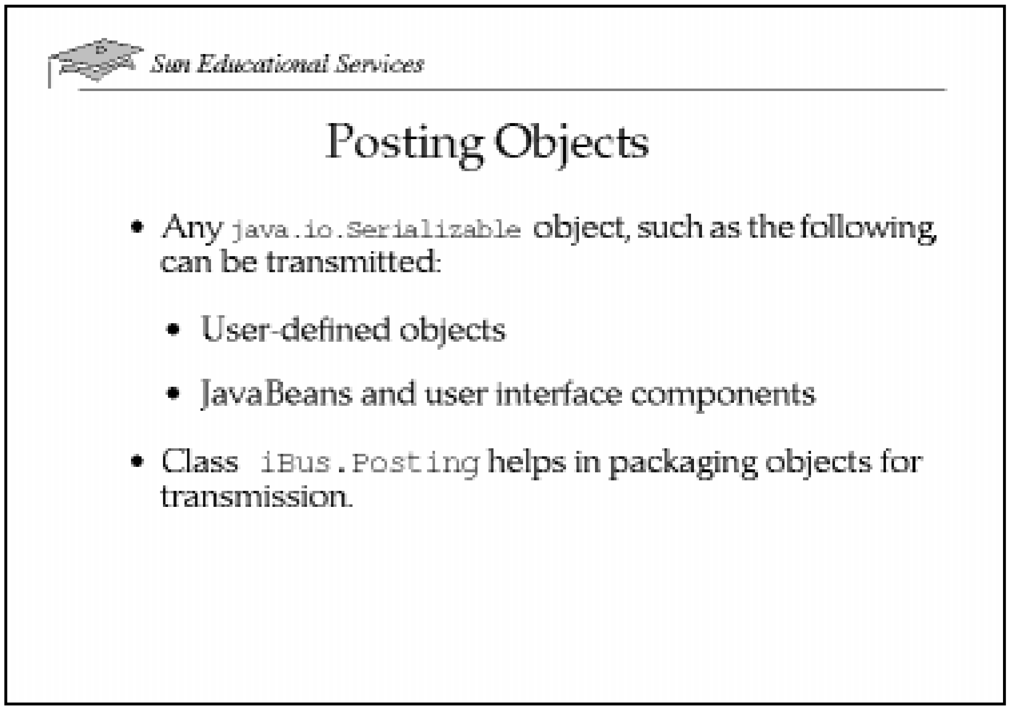
The address part contains a host name or an IP address. IP addresses can be of the class A, B, C, or D.

✓ ***A, B, and C are "conventional" IP addresses used for point-to-point datagrams. Class D addresses are multicast, these range from 224.0.0.0 to 239.255.255.255. This means there is over a quarter of a billion possible IP multicast channel addresses!***

A URL denotes a multicast or a point-to-point channel, depending on its address part. The overhead on this page provides an example of a multicast and a point-to-point URL.

Multicast channels offer more flexibility because one-to-many and many-to-many communication is supported. Another advantage is that such channels are not tied to particular computers but are truly ubiquitous. This means that you can relocate iBus applications from one machine to another in a “plug-and-play” fashion, without needing to contact any central registry or naming service.

iBus multicast channels are used primarily within intranets. Secure TCP bridges can be established to route iBus events from one intranet to another.



Sun Educational Services

Posting Objects

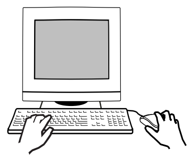
- Any `java.io.Serializable` object, such as the following, can be transmitted:
  - User-defined objects
  - JavaBeans and user interface components
- Class `iBus.Posting` helps in packaging objects for transmission.

Posting Objects

Any Java object that is serializable can be transmitted using iBus. This includes user-defined objects, JFC components, JavaBeans, and so on. iBus provides the class `Posting` to help in packing several Java objects and transmitting them at once.

```
T  package iBus;

public class Posting {
        /**
        * Create a posting object to be transmitted through iBus
        */
        public Posting() { ... }

        /**
        * Pack an object into the posting.
        */
        public void setObject(int index, Serializable object) { ... }
        /**
        * Extract an object from the posting.
        */
        public Serializable getObject(int index) { ... }

        /**
        * Set the length property of the posting.
        */
        public void setLength(int length) { ... }

        /**
        * Get the length property of the posting.
        */
        public int getLength() { ... }

        /**
        * Get the iBus URL of the sender of the posting.
        */
        public iBusURL getSender() { ... }
}
```

Exercise: Creating a Stock Quote Application



Exercise objective – Show how you can use objects more complex than simple Strings in object bus applications.

Tasks

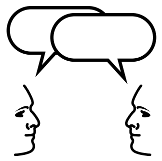
Build a Small Stock Quote Application

Use the `Quote` class to send and receive stock quotes in a extremely simplified stock quote example. Complete the Talker and the Listener and run the application.

1. Change the directory to `labfiles/mod6-objectbus/lab/quote.`
2. Compile `Quote.java`.
3. Write a Talker that generates and sends stock quotes described by instances of the `Quote` class. Use the skeleton `TalkerQuote.java`.
4. Write a Listener that receives stock quotes and processes them (displaying on the console). Use the skeleton `ListenerQuote.java`.
5. Compile the Talker and the Listener and run the application.
6. As more and more people in your class complete the exercise and run their Talkers, your Listener will automatically get all stock quotes sent by all Talkers.

Exercise Summary

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



✓ ***Manage the discussion here based on the time allowed for this module, which was given in the "About This Course" module. If you find you do not have time to spend on discussion, then just highlight the key concepts students should have learned from the lab exercise.***

- Experiences

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

- Interpretations

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

- Conclusions

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

- Applications

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

Check Your Progress

Before continuing on to the next module, check that you were able to accomplish the following:

- ❑ Understand the fundamentals of the Object Bus model
- ❑ Understand its main application areas
- ❑ Explain the differences between iBus, RMI, and CORBA

Think Beyond

What benefits are there in using an Object Bus middleware for building a mission-critical system, such as financial data feed?

- ✓ ***Extensibility. Add further listeners and talkers to a running system and add new channels to a running system.***
- ✓ ***Simple architecture. Components need to interface only to the bus. This makes for a "flat" architecture since components do not reference each other directly.***
- ✓ ***Instant delivery of events.***

Supporting Technologies 7

Objectives

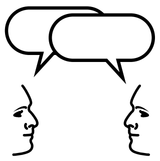
Upon completion of this module, you should be able to:

- Describe the Java Naming and Directory Interface
- Write a simple JNDI client that binds a CORBA object into the service and later looks it up
- Describe the Java Transaction Service
- Describe the Java Message Service

No matter which technology you choose to distribute your application, you will almost always need one or more supporting technologies: a naming service to look up addresses or objects by name; a transaction service to manage distributed consistency; or a messaging service to let your components communicate asynchronously.

Relevance

✓ ***Present the following questions to stimulate the students and get them thinking about the issues and topics presented in this module. They are not expected to know the answers to these questions. The answers to these questions should be of interest to the students, and inspire them to learn the content presented in this module.***



Discussion – Consider the following questions:

- If all you know about a remote object is its name, how do you get its reference?

✓ ***So far, a technology-specific naming service, rmiregistry with RMI and COS naming with JavaIDL, has been used.***

- Would it be preferable to have the same API for all available naming services?

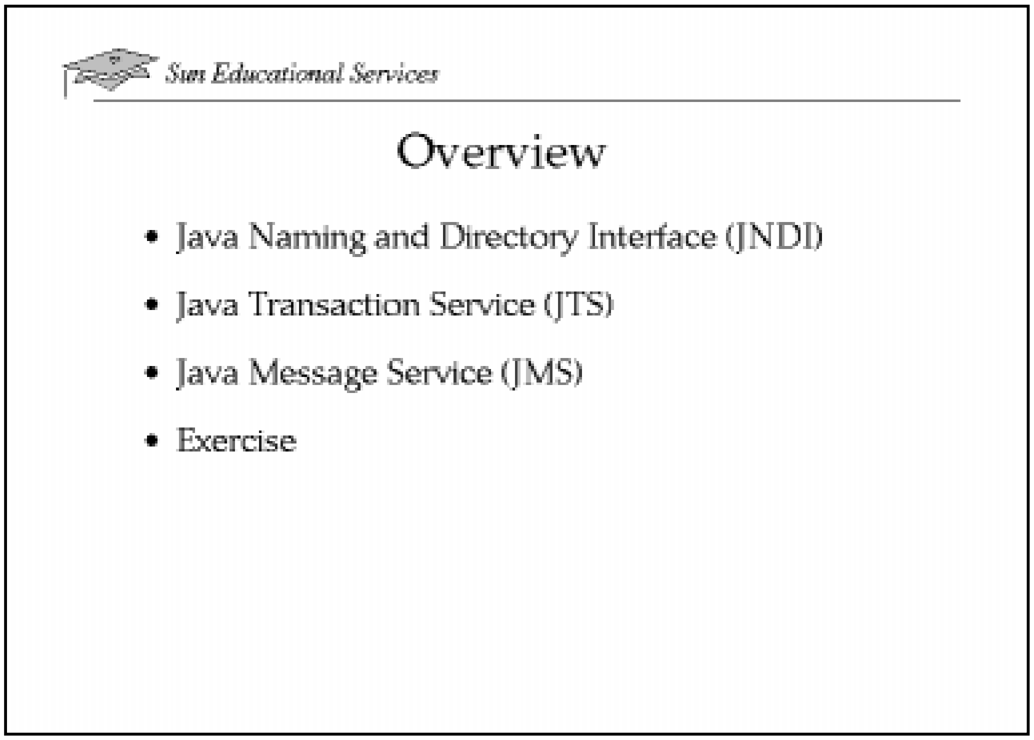
✓ ***There is no standard answer to this question. If you already know the APIs to several naming services, why learn yet another one? On the other hand, if there is just one API, there is less to learn if you don't know any naming service APIs yet.***

- How would you design a vendor-independent transaction API?

✓ ***Two possibilities come to mind: either use an already common and widely used standard API, or design a new one. The Java Transaction Service does both: It supports the CORBA COS Object Transaction Service, and it supports a new, Java technology-specific transaction API.***

- What do you think about when you hear the term *messaging*?

✓ ***Often,*** email ***comes to mind. However, messaging is*** not ***email. While messaging shares some characteristics with email (especially the asynchronous nature of the service), messaging is about asynchronous requests, reports, or events that are consumed by enterprise applications, not humans.***



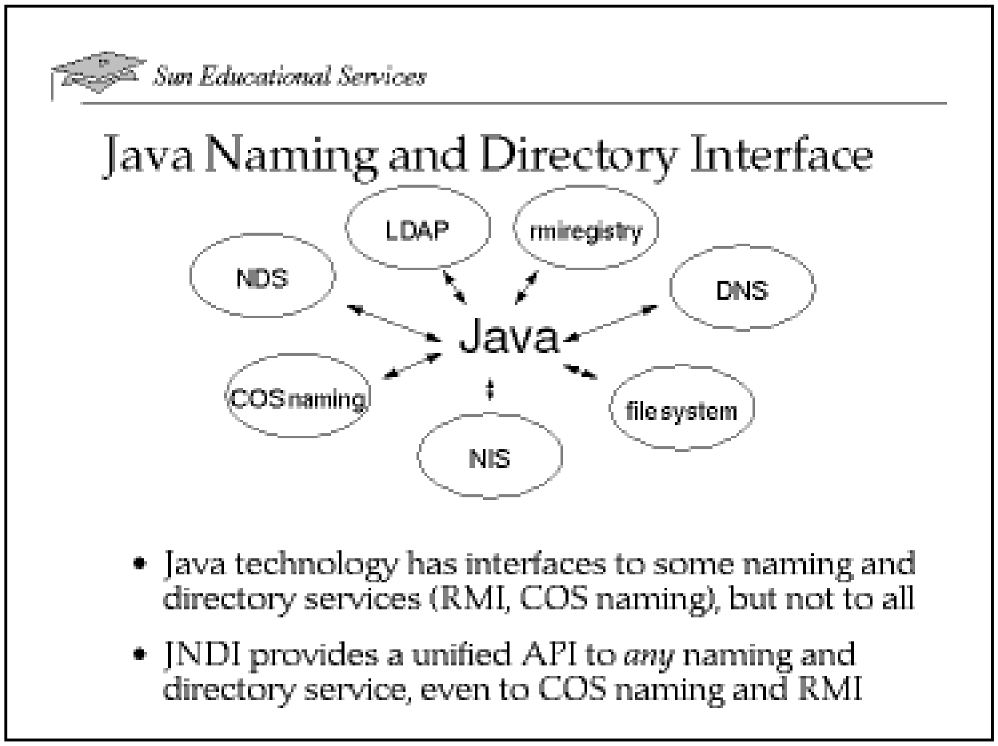
Sun Educational Services

Overview

- Java Naming and Directory Interface (JNDI)
- Java Transaction Service (JTS)
- Java Message Service (JMS)
- Exercise

Overview

This module looks at three supporting technologies from JavaSoft: Java Naming and Directory Interface (JNDI), Java Transaction Service (JTS), and Java Message Service (JMS).


Sun Educational Services

Java Naming and Directory Interface

LDAP
rmiregistry
NDS
DNS
Java
COS naming
file system
NIS

- Java technology has interfaces to some naming and directory services (RMI, COS naming), but not to all
- JNDI provides a unified API to *any* naming and directory service, even to COS naming and RMI

Java Naming and Directory Interface

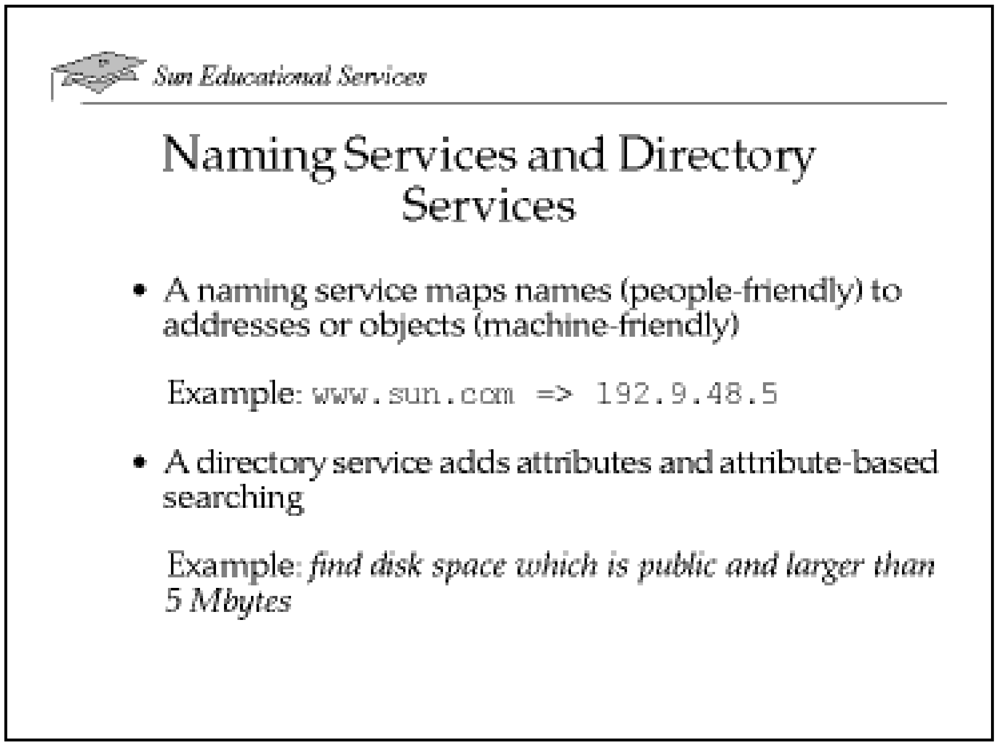
In today's information technology (IT) world, there is a wide range of naming and directory technologies and products available. The following lists some of these products:

- Domain Name System (DNS)
- Network Information Services (NIS and NIS+)
- Lightweight Directory Access Protocol (LDAP)
- CORBA naming service (COS naming)
- RMI naming service (`rmiregistry`)
- The OSI protocol for managing online directories of users and resources (X.500)
- NetWare Directory Service (NDS)

Each of these products has its API (mostly for C or C++). To access these products from a Java application, a Java API is necessary. There are a couple of Java technology APIs; Netscape developed a Java technology API to LDAP, you can reach DNS using the `java.net` package, COS naming is part of the Java 2 SDK, and RMI is a Java technology. However, to reach all the products from Java applications, more APIs are necessary.

JNDI takes another approach. It provides a unified Java API to *any* of the naming and directory services available. The JNDI API can even be used if another Java API already exists; you can reach COS naming, LDAP, DNS or the rmiregistry using JNDI.

A developer now faces this choice: use JNDI, or use the “older” API? How you decide depends on whether you know the “older” API. If you know how to reach COS naming, there is no advantage in using JNDI. On the other hand, once you know JNDI, you can use this knowledge to reach other naming and directory services easily.



Sun Educational Services

Naming Services and Directory Services

- A naming service maps names (people-friendly) to addresses or objects (machine-friendly)

  Example: `www.sun.com => 192.9.48.5`

- A directory service adds attributes and attribute-based searching

  Example: *find disk space which is public and larger than 5 Mbytes*

Naming Services and Directory Services

Naming and directory services are often confused, but the following sections describe the differences.

Naming Services

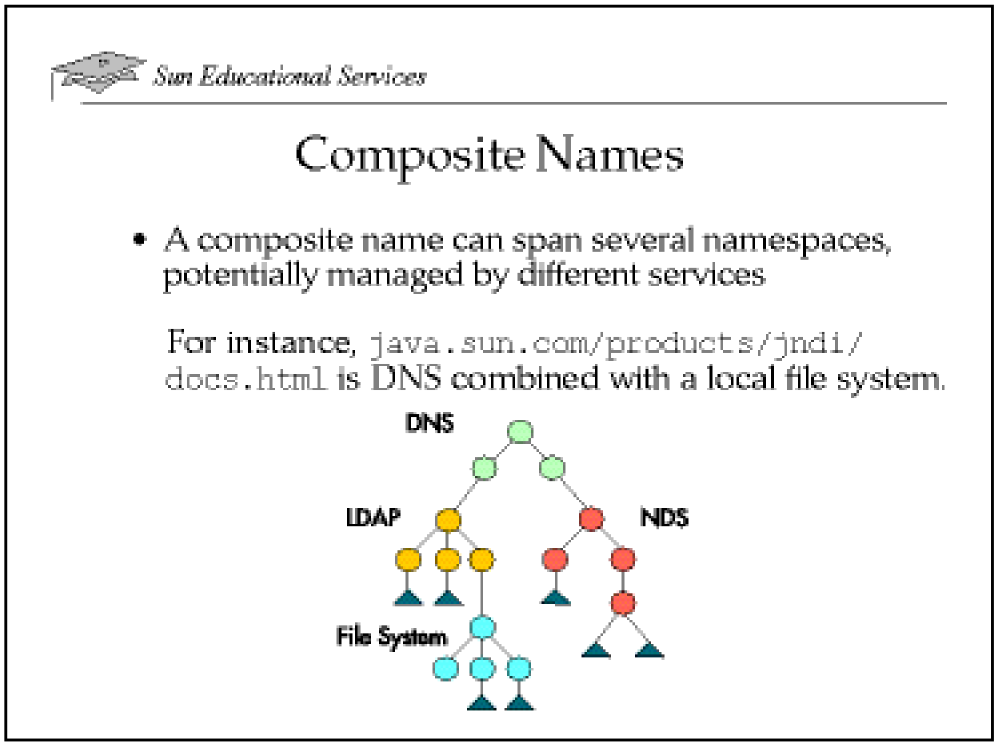
A naming service does only one task: it looks up an object, given the object's name. You have to know the name of the object you are looking for, or you will not find anything. The types of objects that are supported depends on the particular naming service. DNS, for example, supports only IP addresses; COS naming service supports CORBA objects.

Directory Services

A directory service adds the capability for searching. You no longer have to know the name of the object for which you are looking. Instead, you can provide a set of attributes that your desired object should have set.

In a real-world example, you have to know the name to find a person's phone number and address. This is insufficient if you want the address of all people who like skiing, for example. If you can find a company who has a phone book on disk, and has an attribute "hobby" attached to each person, you might get what you want. This company would provide a directory service, extending the possibilities of a name service alone.

For more information about the directory service part of JNDI, consult the Java Naming and Directory Interface specification (`http://java.sun.com/products/jndi/`).

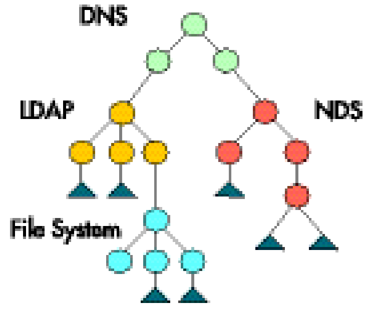


Sun Educational Services

Composite Names

- A composite name can span several namespaces, potentially managed by different services

For instance, `java.sun.com/products/jndi/docs.html` is DNS combined with a local file system.

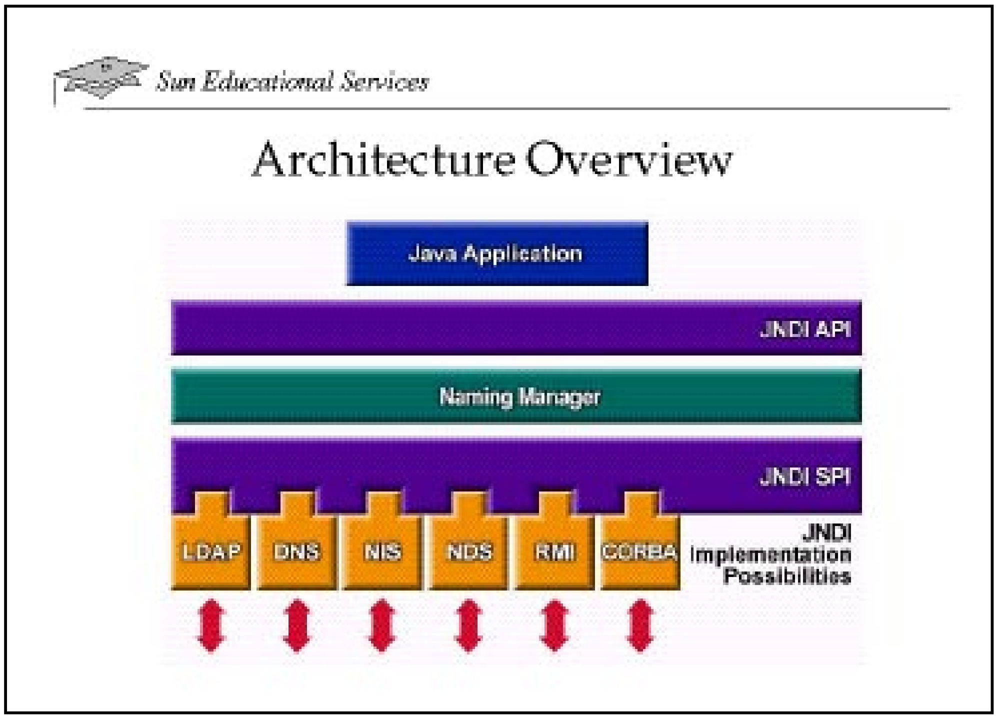


Composite Names

Usually, you do not have one large dictionary that contains all the names. Instead, the dictionary is distributed in a hierarchical manner. The domain name system, for example, is highly distributed; each company manages the machines in its own domain. The domains, in turn, are managed differently for example, by a public organization and grouped to a country-wide context. To reach a domain in another country, still higher hierarchical levels are needed. A single dictionary would be a *namespace*.

Namespaces are usually linked; in the address `java.sun.com` there is the `com` namespace, which points to the `sun` namespace. You can use the `sun` namespace to look up the address of the `java` host.

Namespaces can even be a combination of several name services. The address `java.sun.com/products/jndi/docs.html` is a combination of DNS (`java.sun.com`) and a local file system (`products/jndi/docs.html`). JNDI can manage namespaces with an arbitrary number of different services involved.

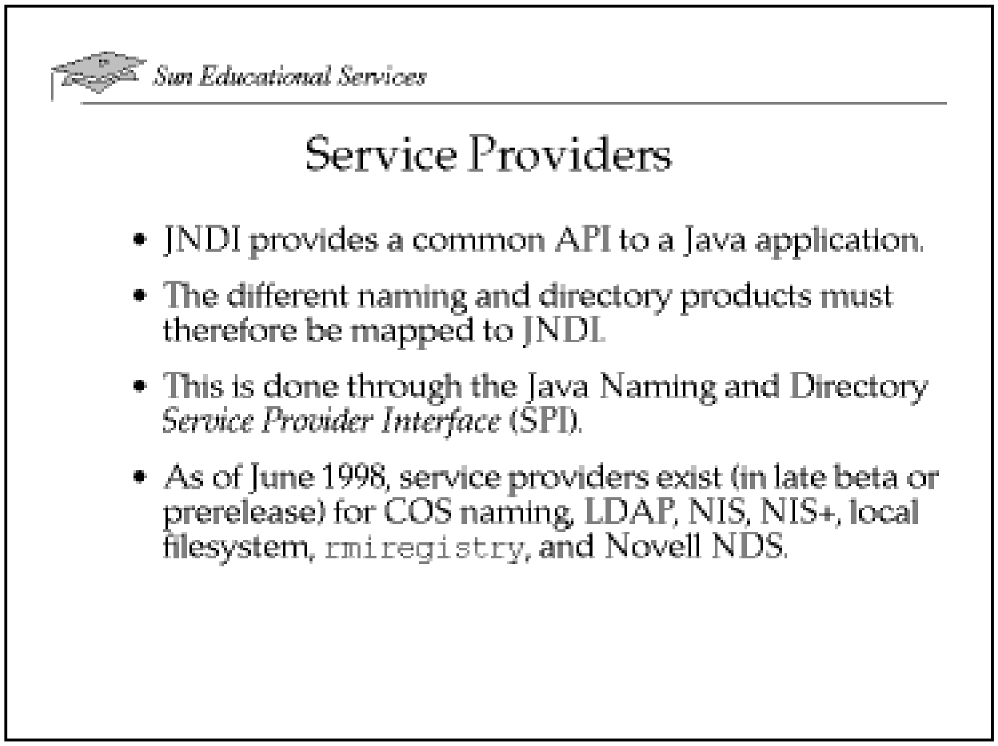


Architecture Overview

The Java Naming and Directory Interface specification declares two APIs: the JNDI API, and the JNDI Service Provider Interface.

The JNDI API is used by developers to get at the services of a naming or directory service. A part of this API will be used later in the exercises.

The JNDI Service Provider API is used to link a specific naming or directory service to the JNDI system. A module that plugs into JNDI using the Service Provider API is called a *service provider.*


Sun Educational Services

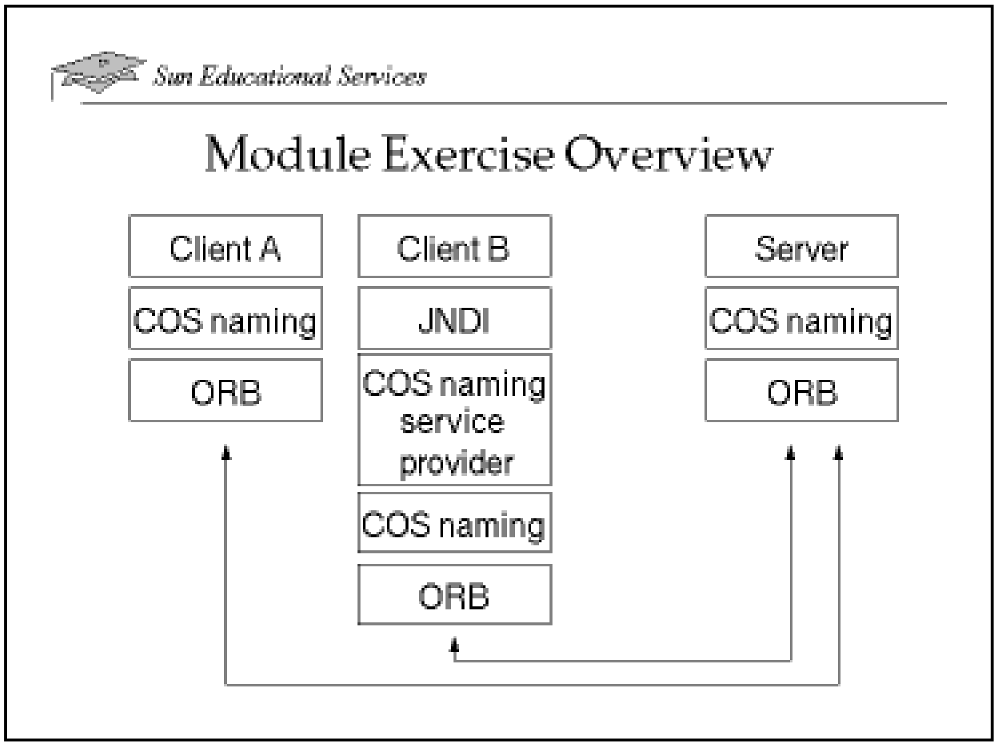
Service Providers

- JNDI provides a common API to a Java application.
- The different naming and directory products must therefore be mapped to JNDI.
- This is done through the Java Naming and Directory *Service Provider Interface* (SPI).
- As of June 1998, service providers exist (in late beta or prerelease) for COS naming, LDAP, NIS, NIS+, local filesystem, `rmiregistry`, and Novell NDS.

Service Providers

The goal of JNDI is to eventually reach all available naming and directory services using JNDI. To reach this goal, service providers, which plug using the Service Provider API to JNDI, must be developed. The following service providers are available already:

- COS naming
- LDAP
- NIS, NIS+
- Novell NDS
- File system
- `rmiregistry`



Module Exercise Overview

Most of the services are not available or not configured correctly in a classroom. However, there are still two to choose from: COS naming, which is part of the Java 2 SDK, and the local file system. Of the two, COS naming is more interesting, because there is already an application that uses it: the IDL version of Echo.

In Module 4, the client and the server both used COS naming directly to bind and look up the Echo implementation. If you convert to use JNDI, the picture looks different; the client uses JNDI to look up the object reference. This is translated using the COS naming service provider to COS naming. Because it is translated on the client, the server can continue to use COS naming directly.

According to the previous description, it should also be possible to convert the server to JNDI, and use either the “old” client (which uses COS naming directly) or the JNDI version of the client, to reach the JNDI-based server.

Using JNDI to Access COS Naming

Using the Java Naming and Directory Interface is straightforward. The lookup of an object consists of three steps:

1. Set the necessary properties.
2. Get a first `Context.`
3. Use the `Context` to look up either of the following:
   - ▼ Another `Context` further down the hierarchy
   - ▼ The desired object.

Of course, step 3 might be repeated several times.

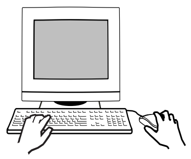
You use properties to configure the JNDI core and the individual service providers. As usual, you can set a property in the form of a system property (using the `-D` command-line option), or by passing a `Properties` object on JNDI initialization. Because the class `java.util.Properties` accepts only `Strings`, but some service providers require arbitrary objects to configure themselves, you can use a `java.util.Hashtable` instead of a `Properties` object. Setting such a property with the `-D` command-line option does not work.

The following is the necessary code to look up a CORBA object:

```
Hashtable env = new Hashtable(5, 0.75f);
env.put("java.naming.factory.initial",
"com.sun.jndi.CosNaming.CNCtxFactory");
env.put("java.naming.corba.orb", orb);
Context ic = new InitialContext(env);
Echo echoRef = EchoHelper.narrow(
(org.omg.CORBA.Object)ic.lookup("Echo"));
```

The first of the two properties that are set, `java.naming.factory.initial`, is used by the JNDI core to generate the initial context object. The second property, `java.naming.corba.orb`, is used by the COS naming service provider to get a reference to the ORB.

Exercise: Java Naming and Directory Interface



Exercise objective – Learn about JNDI and the COS naming service provider by coding an example.

Preparation

Install the JNDI JAR file, and two of the service providers: COS naming and local file system.

Thanks to the new extensions framework of the Java 2 SDK, the installation is easy. Copy the JAR files to the `$JAVA_HOME/lib/ext` directory. There is no need to update the `CLASSPATH`.

Tasks

Modify the JavaIDL Echo Server to Use JNDI Instead of COS Naming

So far, in all the JavaIDL examples, the CORBA native naming service, COS naming, was used. Thanks to the COS naming service provider, you can use the Java Naming and Directory Interface instead. A modified client is already provided as part of this exercise. Your job is to modify the server as well, and then to test whether "old" and "new" clients and servers can interoperate.

1. Change the directory to `labfiles/mod7-support/lab/jndi.`
2. Analyze the code of the modified client, which uses JNDI instead of COS naming to get the remote Echo object reference.
3. Compile and run the application. Can the modified client interoperate with the current server?

Modify the JavaIDL Echo Server to Use JNDI Instead of COS Naming (Continued)

4. Modify the server to use JNDI as well.

5. Try the different combinations of COS naming or JNDI on the server and on the client. Do COS naming and JNDI with COS naming service provider really work together transparently?

Can You Use the File System Service Provider Instead of COS Naming?

The Java Naming and Directory Interface unifies the API to different products. Does it also mask the implementation differences? You can find out by trying to use the file system service provider instead of the COS naming service provider.

1. Start with the server code, and replace the COS naming service provider with the file system service provider. The initial factory is `com.sun.jndi.fscontext.FSContextFactory`. Instead of a handle to the ORB, the file system service provider needs a URL-like string that indicates the root of the file system. The property called `java.naming.provider.url` can be set, for example, to `file:///tmp`, or `file:///D:/temp`

   If it does not work, seek an explanation and discuss it with the other students.

✓ ***The file system service provider does not accept CORBA objects in a*** `bind` ***or*** `rebind` ***call. If you think about this for a second, it becomes clear why. Even with "pure" COS naming, the object to*** `bind` ***on the server and the object to look up on the client is not the same. You bind an "***`EchoImpl`***" to the COS naming service, but get a stub out of COS naming on the client. This functionality is specific to the COS naming service and is, of course, not implemented in the file system service provider. Even if you could bind arbitrary objects to the file system, you would get the wrong object out of it on the client. (You would get an*** `EchoImpl` ***instead of a stub.)***

Exercise Summary

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

✓ ***Manage the discussion here based on the time allowed for this module, which was given in the "About This Course" module. If you find you do not have time to spend on discussion, then just highlight the key concepts students should have learned from the lab exercise.***

- Experiences

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

- Interpretations

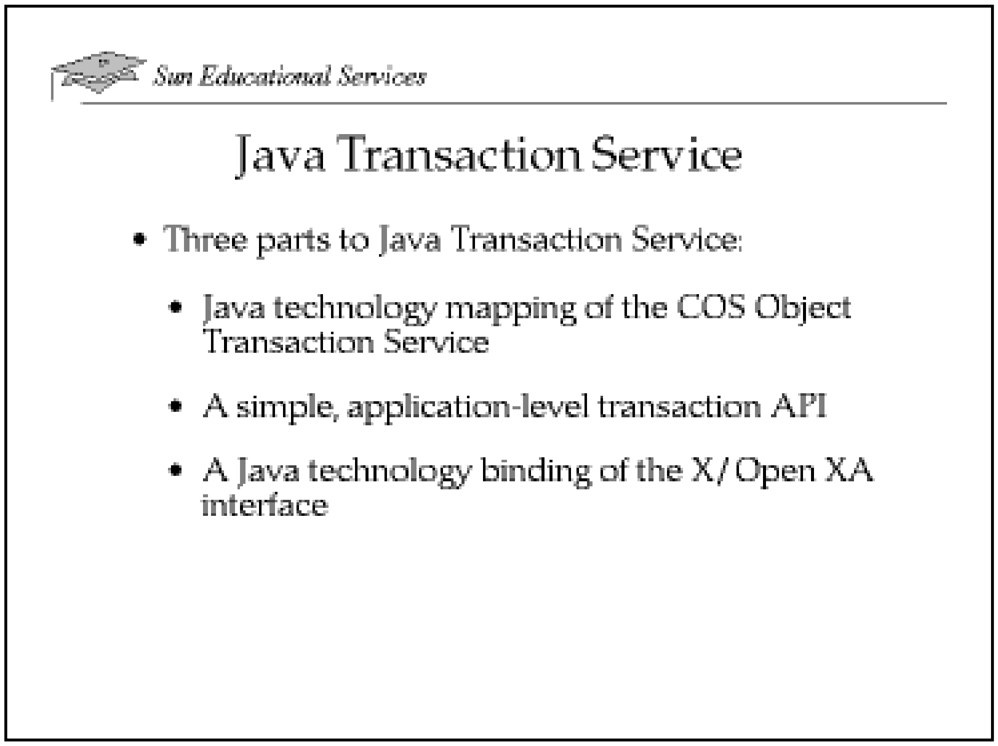
✓ ***Ask students to interpret what they observed during any aspects of this exercise.***

- Conclusions

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

- Applications

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



Sun Educational Services

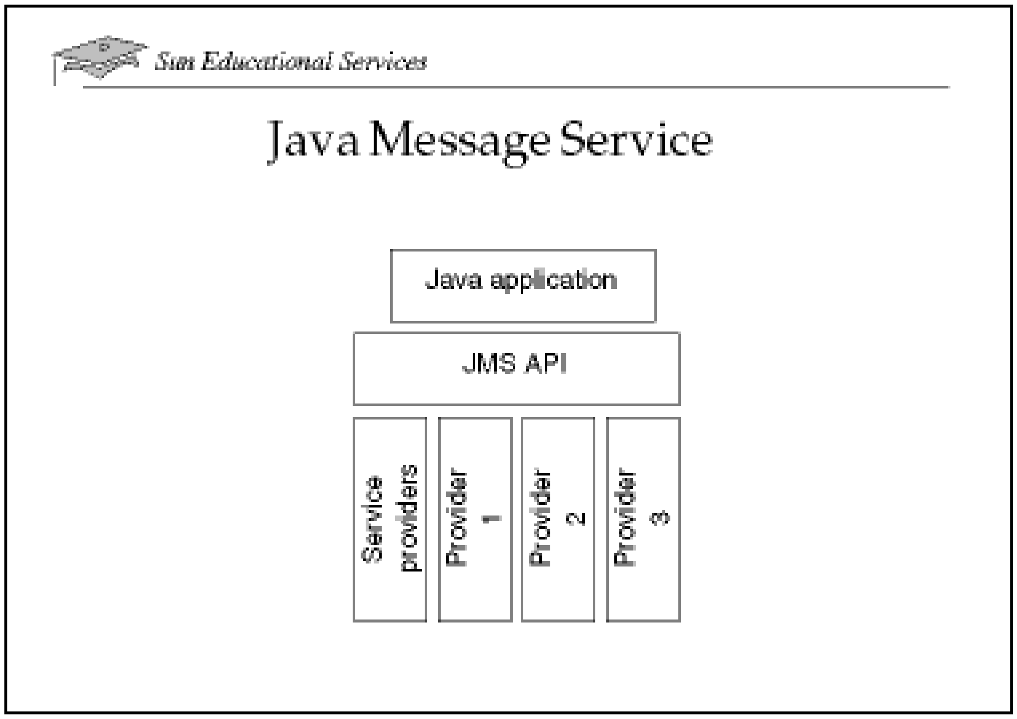
Java Transaction Service

- Three parts to Java Transaction Service:
  - Java technology mapping of the COS Object Transaction Service
  - A simple, application-level transaction API
  - A Java technology binding of the X/Open XA interface

Java Transaction Service

The Java Transaction Service (JTS) defines a standard transaction management API for the Java platform. JTS consists of the following elements:

- A standard Java technology mapping ("Java mapping") of the OMG Object Transaction Service (OTS). The Java mapping of the OTS interfaces is defined in the packages `org.omg.CosTransactions` and `org.omg.CosTSPortability`.
- An application-level transaction demarcation API. This API is defined in the package javax.jts.
- A Java mapping of the industry-standard XA interface. This API is defined in the `javax.jts.xa package`. This package is a Java binding of the X/Open XA interface. The XA interface defines the API through which a transactional resource manager attaches to an external transaction manager.



Java Message Service

Just as Java Naming and Directory Service provides a unified interface to existing naming and directory products, the Java Messaging Service (JMS) provides a generalized API to various enterprise messaging service products. JMS provides a common way for Java programs to create, send, receive, and read an enterprise messaging system's messages.

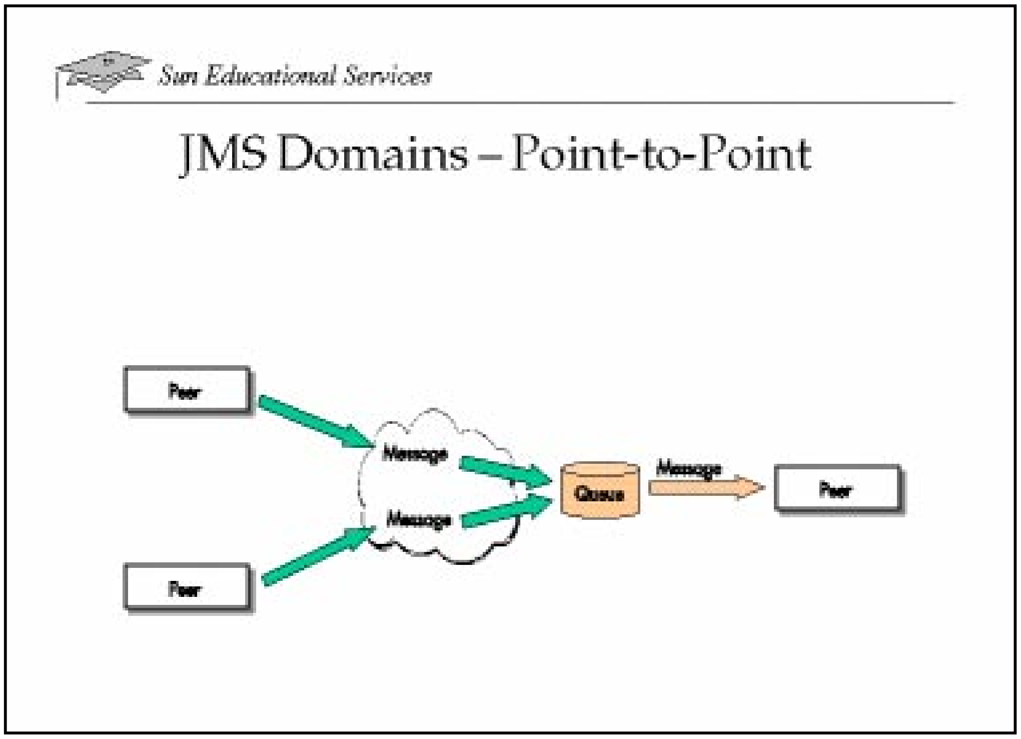
JMS is a set of interfaces and associated semantics that defines how a JMS client accesses the facilities of an enterprise messaging product. While you could write and compile a client application, it would not run with JMS alone, you need at least one service provider. Whether the final version of JMS will include a simple service provider for testing purposes is not known at this time.

JMS Service Provider

As with JNDI, a JMS provider maps the JMS API to a specific messaging product. Ideally, JMS providers will be written in 100% Pure Java™ so they can run in applets, simplify installation, and work across architectures and operating systems. However, early implementations will probably provide only a JMS API on top of a legacy product.

JMS Domains

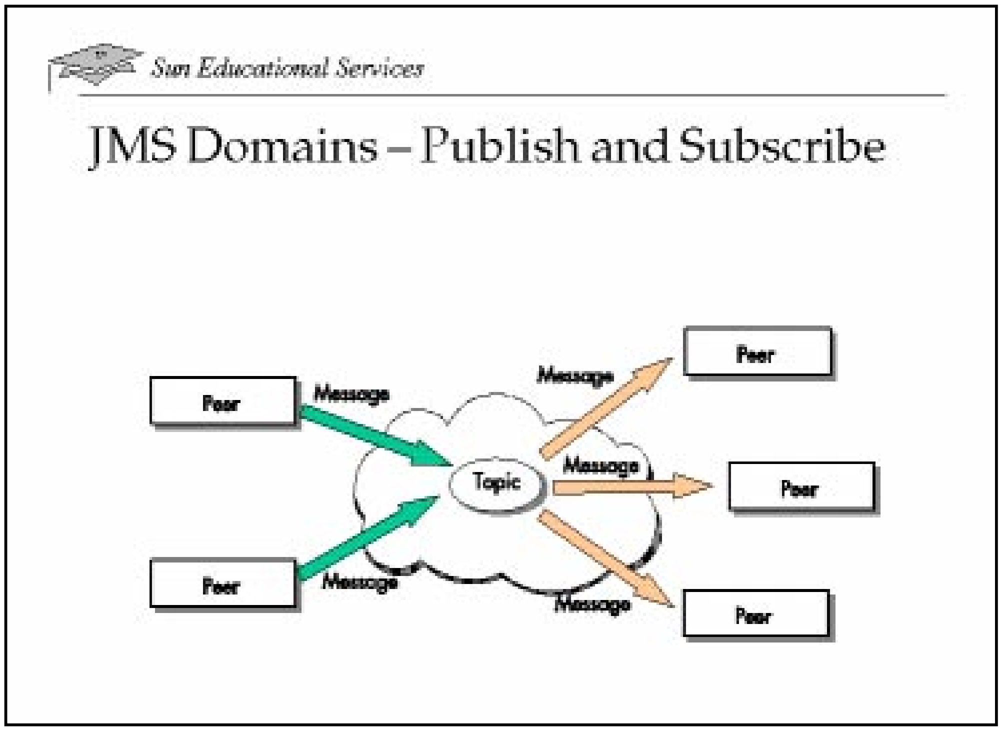
There are two different types of messaging products on the market today. One group of products uses point-to-point semantics; that is, a message is sent from one sender to one receiver. The other group uses publish and subscribe semantics, which is typically a many-to-many approach. The JMS API provides support for both types of messaging products.


Sun Educational Services

JMS Domains – Point-to-Point

JMS Domains – Point-to-Point

Point-to-point (PTP) products are built around the concept of message queues. Each message is addressed to a specific queue and clients extract messages from the queue(s) established to hold their messages.


Sun Educational Services

JMS Domains – Publish and Subscribe

JMS Domains – Publish and Subscribe

Publish and subscribe (or "publish-subscribe") clients address messages to some node in a content hierarchy. Publishers and subscribers are generally anonymous and can dynamically publish or subscribe to the content hierarchy. The system takes care of distributing the messages arriving from a node's multiple publishers to its multiple subscribers.

Check Your Progress

Before continuing on to the next module, check that you were able to accomplish the following:

- ❑ Describe the Java Naming and Directory Interface
- ❑ Write a simple JNDI client that binds a CORBA object into the service and later looks it up
- ❑ Describe the Java Transaction Service
- ❑ Describe the Java Message Service

Think Beyond

How can the supporting technologies help in your development of a distributed system?

Technology Summary and Comparison 8

Objectives

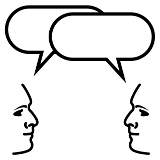
Upon completion of this module, you should be able to:

- Explain the role of JDBC and servlets in a distributed application
- Compare and contrast RMI and JavaIDL
- Compare and contrast request-reply and publish-subscribe architectures

This module summarizes and compares the technologies presented in this course. It will help you decide which technology is appropriate to use in different circumstances.

Relevance

✓ ***Present the following questions to stimulate the students and get them thinking about the issues and topics presented in this module. They are not expected to know the answers to these questions. The answers to these questions should be of interest to the students, and inspire them to learn the content presented in this module.***



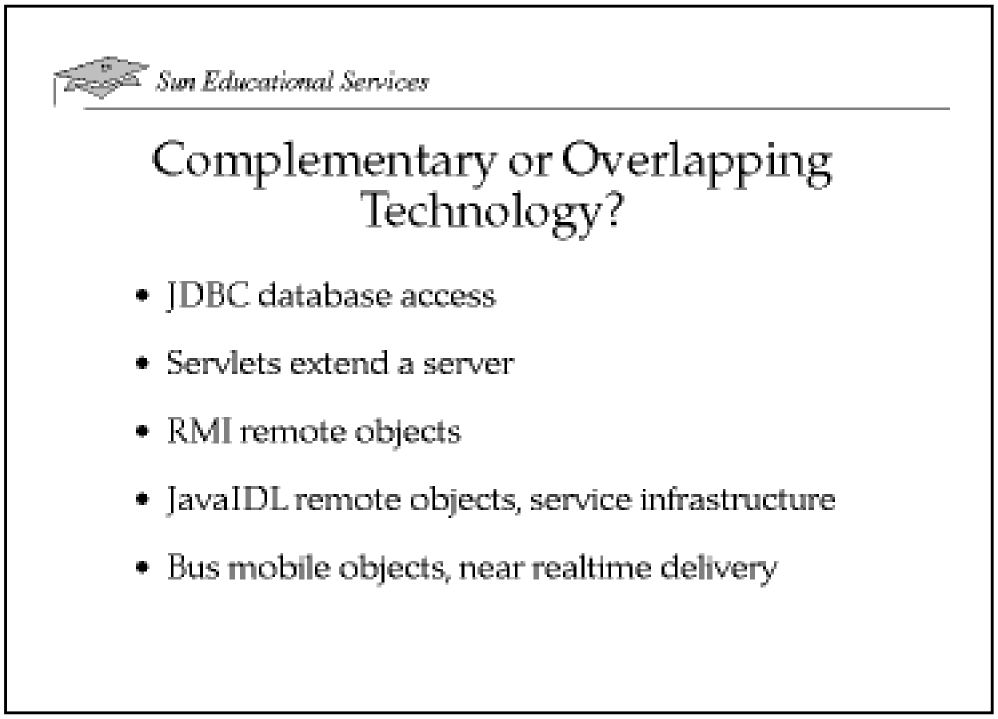
Discussion – Consider the following questions:

- Will you be using all the technologies you learned about in one and the same application?

✓ ***Usually not. RMI and JavaIDL tend to exclude each other. This is not a technical issue, but one of personal preference, company regulations, or existing legacy code.***

- Is JDBC always a part of a distributed application?

✓ ***Often it is, because in real-world applications there is usually a persistent data store around; that is, a database. However, if there is no database, there will be no JDBC.***


Sun Educational Services

Complementary or Overlapping Technology?

- JDBC database access
- Servlets extend a server
- RMI remote objects
- JavaIDL remote objects, service infrastructure
- Bus mobile objects, near realtime delivery

Complementary or Overlapping Technology?

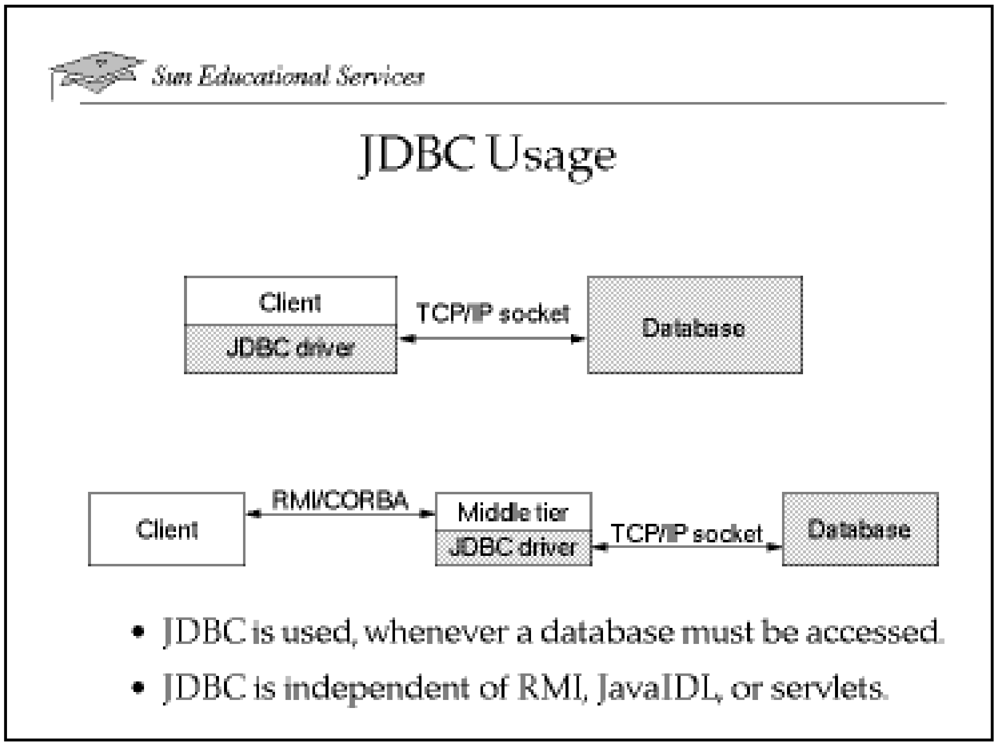
This course describes five distributed computing technologies:

- JDBC for database connectivity
- Servlets for extending a Java technology-based server
- RMI to create and use remote objects
- JavaIDL to create and use remote objects
- Object Bus to support mobile objects and near realtime object delivery

From these five technologies, only two overlap to a large degree: RMI and JavaIDL. Both technologies serve basically the same purpose: they allow for the development of remote objects. To a lesser degree RMI/JavaIDL and object buses overlap, because you can solve several distributed computing problems using either technology.

In this module, the comparison is therefore between RMI and JavaIDL on one hand, and between request/reply style communication (for example, RMI or JavaIDL) and publish/subscribe style communication (for example, object buses) on the other hand.

Additionally, this module describes the roles of JDBC and servlets in a distributed computing environment.


Sun Educational Services

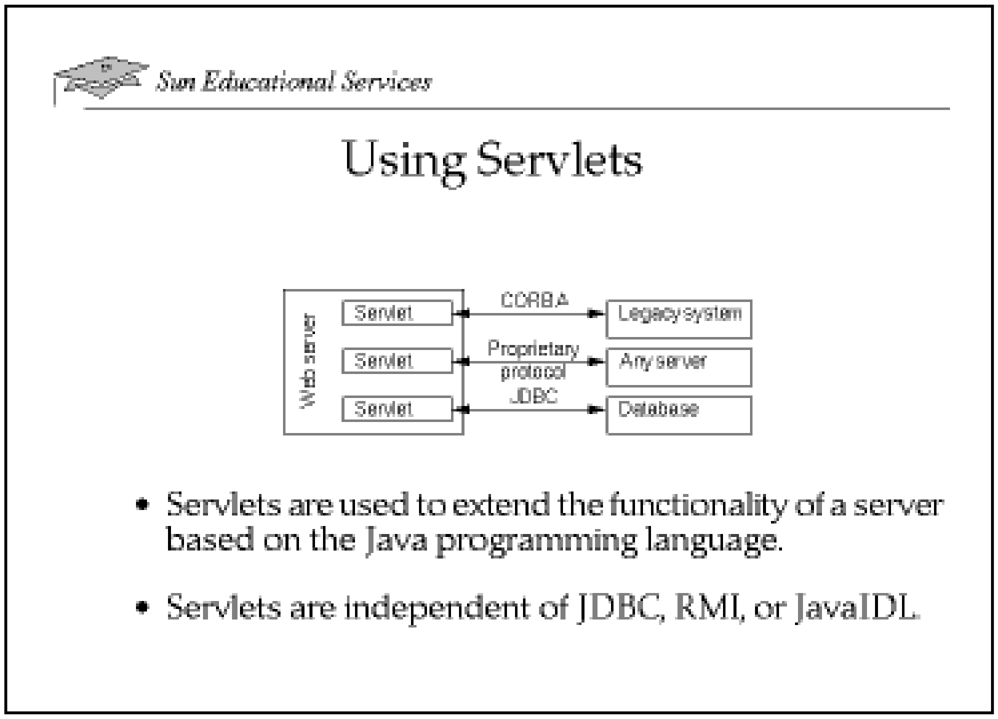
JDBC Usage

- JDBC is used, whenever a database must be accessed.
- JDBC is independent of RMI, JavaIDL, or servlets.

JDBC Usage

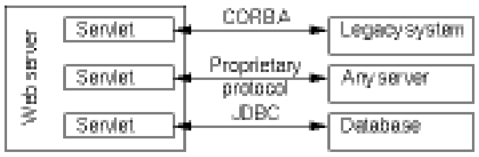
JDBC is used for one simple purpose: to work with databases. In the Java technology environment, there is no alternative to JDBC. There are commercial tools to assist the developer, such as Java Blend™ from Sun Microsystems. But even this tool produces JDBC code as a result.

You can use JDBC directly on a client system, although it is not the preferred usage. Mostly JDBC is used on a business logic server, which translates the raw JDBC calls into a higher-level business object. This object is then called remotely, using RMI or JavaIDL.



Sun Educational Services

Using Servlets

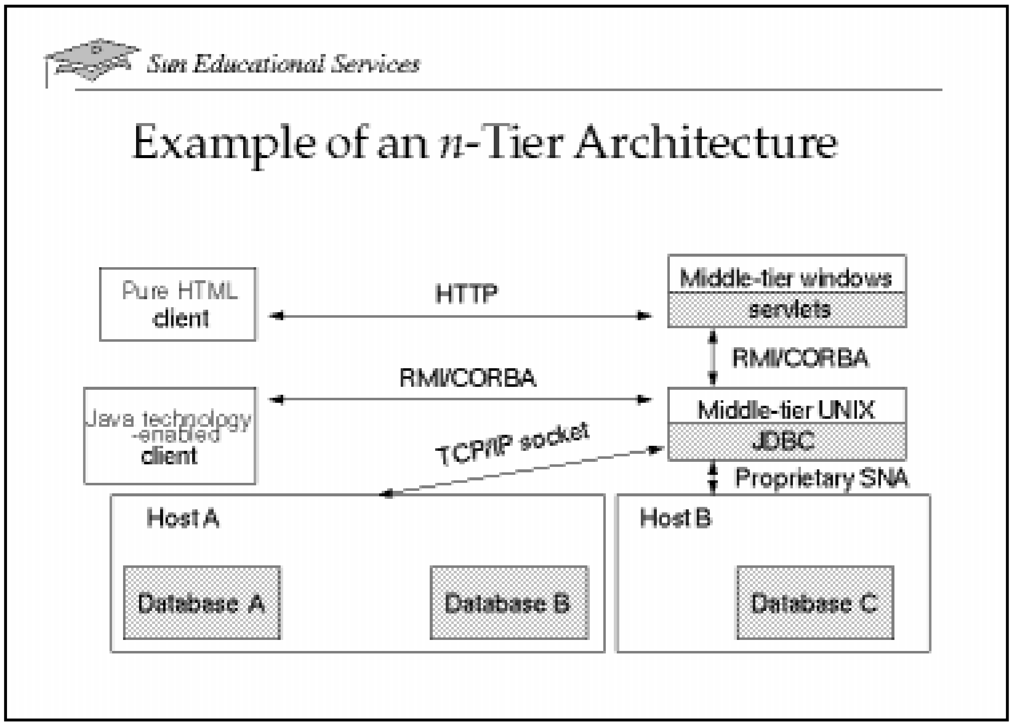


- Servlets are used to extend the functionality of a server based on the Java programming language.
- Servlets are independent of JDBC, RMI, or JavaIDL.

Using Servlets

The application area of servlets is broader than that of JDBC. In a generic sense, servlets are used to extend the functionality of any server based on the Java programming language. However, the primary usage of servlets today is extending a Web server.

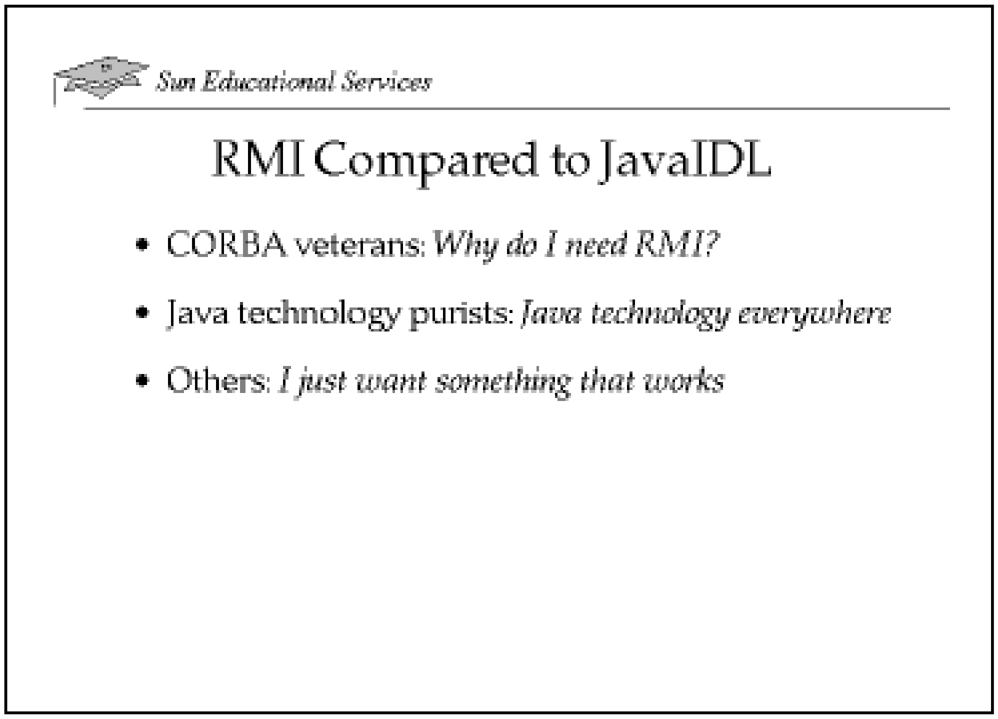
Because servlets are written in the Java programming language, they can use JDBC, RMI, or JavaIDL.



Example of an n-*Tier Architecture*

The above overhead shows the picture of a simple *n*-tier architecture. The core is a business logic server, running the UNIX operating system. It uses JDBC to access data residing on two different legacy hosts.

A client capable of running Java programs can access the business logic server directly, using RMI or JavaIDL. A pure HTML client needs a helper to present the data, for example, in an HTML form. This helper is written as a servlet. This servlet communicates with the business logic server using the same remote objects as the client.



Sun Educational Services

RMI Compared to JavaIDL

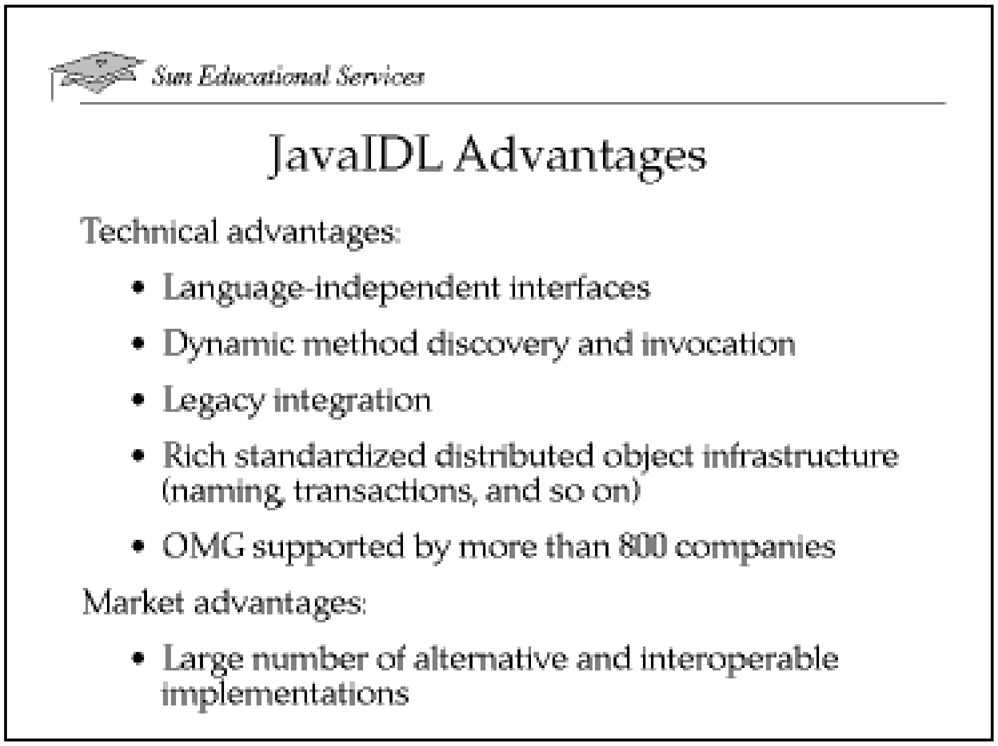
- CORBA veterans: *Why do I need RMI?*
- Java technology purists: *Java technology everywhere*
- Others: *I just want something that works*

RMI Compared to JavaIDL

The Java 2 platform allows for the building of distributed object applications using two different methods: RMI and JavaIDL. Both technologies have their advantages and disadvantages, which should be compared and contrasted to solve the confusion of having two apparently overlapping solutions.

There are three types of Java technology developers:

- CORBA veterans have developed CORBA applications using C++ or other languages. The Java technology environment, for them, promises to take CORBA to the place it belongs. RMI, in contrast, is completely unnecessary for them and should be ignored.
- For Java technology purists, all applications must be written or rewritten in the Java programming language. If all components are developed with the Java programming language, why use a technique that is made for the support of different languages?
- The third group consists of all the developers that are new to distributed objects in the Java technology environment, and are not prejudiced towards a certain solution.



Sun Educational Services

JavaIDL Advantages

Technical advantages:

- Language-independent interfaces
- Dynamic method discovery and invocation
- Legacy integration
- Rich standardized distributed object infrastructure (naming, transactions, and so on)
- OMG supported by more than 800 companies

Market advantages:

- Large number of alternative and interoperable implementations

JavaIDL Advantages

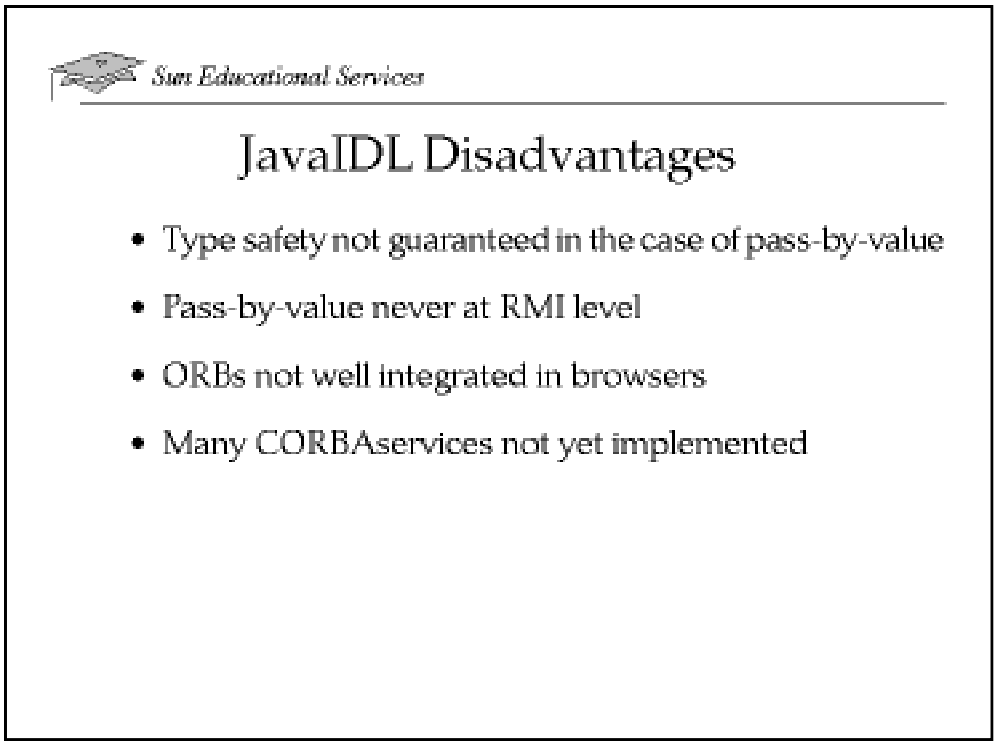
There are two kinds of advantages of the CORBA architecture:

- Technical advantages
- Market advantages

Technical advantages include the fact that CORBA was designed without a special language in mind. This gives advantages as well as disadvantages; however, if there is legacy code or code written in a language other than the Java programming language, CORBA is probably the superior solution for a given project. Interfaces between clients and servers are defined in IDL, providing the advantages of multi-language and multi-platform environments and enforcing a clear separation between interfaces and implementations. Another fact is the rich set of standardized services that are useful for distributed object applications, such as the Lifecycle, Naming, or Event Service.

Another technical advantage is the support for dynamic method discovery and invocation.

It is significant that the CORBA architecture is supported by a consortium of over 800 companies, representing the whole variety of the computer industry. They are willing to make this architecture the next generation of middleware. The results of this support are the many product offerings and various commercial alternatives from which customers can choose.



Sun Educational Services

JavaIDL Disadvantages

- Type safety not guaranteed in the case of pass-by-value
- Pass-by-value never at RMI level
- ORBs not well integrated in browsers
- Many CORBAservices not yet implemented

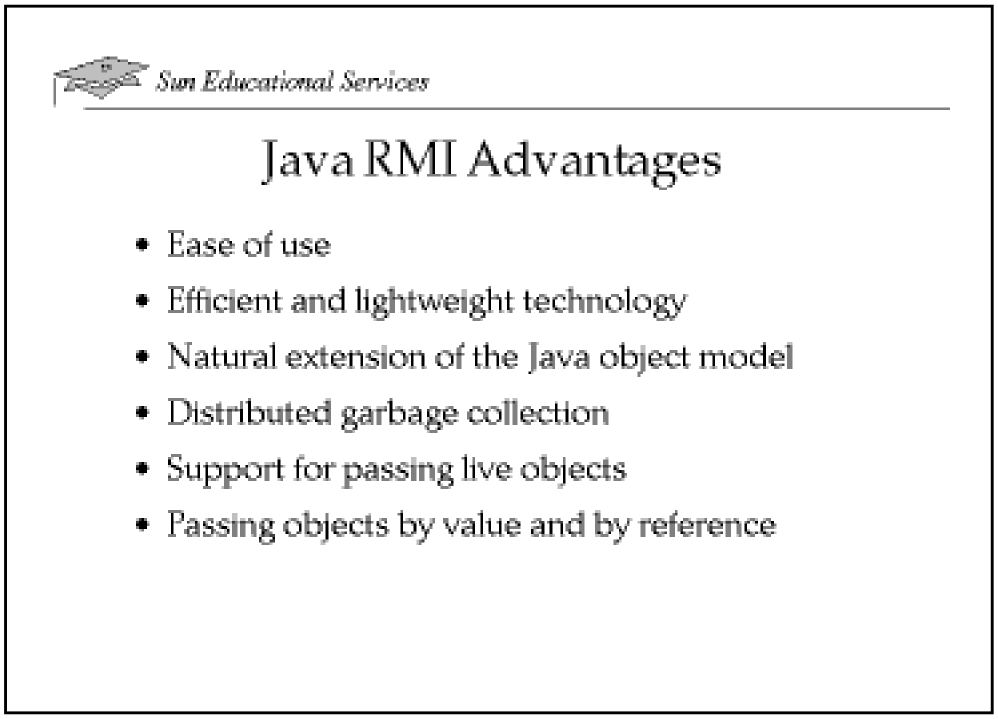
JavaIDL Disadvantages

Current CORBA implementations do not allow the passing over an object and its current state to the remote system (objects by value). However, there is already a draft standard to support this, which soon will be followed by implementations.

The CORBA architecture is made for heterogeneous systems. Its goal is to make all languages cooperate, so that arguments can easily be passed over. To achieve this goal, a lowest common denominator approach is required. You cannot pass a Java object to a COBOL program, because the COBOL program cannot “know” what to do with it. Instead, objects are mapped to IDL, and implemented for every target language and platform. This approach provides a certain level of interoperability, while losing the power of a specific language environment.

RMI, on the other hand, can take a different stance; it can make the assumption that it deals with only one language environment on both sides of the wire, leveraging all the inherent advantages of the Java object environment. CORBA cannot allow a type-safe object passing, because it cannot make any assumptions about which language is used on the other side. With RMI, it is even possible to pass a subclass of some object to a remote VM, and the remote VM will then download the code, and invoke methods of the subclass polymorphically from the base classes interface. With CORBA it is mandatory that both sides know in advance exactly what object will be passed.

A Java applet that wants to talk to a remote CORBA object needs an ORB to communicate with. Therefore, a so-called orblet must be downloaded to the client together with the applet code. This happens each time the client tries to start the applet. While the common bootstrapping protocol might help on this issue, not every ORB has implemented locally all of the potentially required CORBAservices, like the full Security or Trading Service.



Sun Educational Services

Java RMI Advantages

- Ease of use
- Efficient and lightweight technology
- Natural extension of the Java object model
- Distributed garbage collection
- Support for passing live objects
- Passing objects by value and by reference

Java RMI Advantages

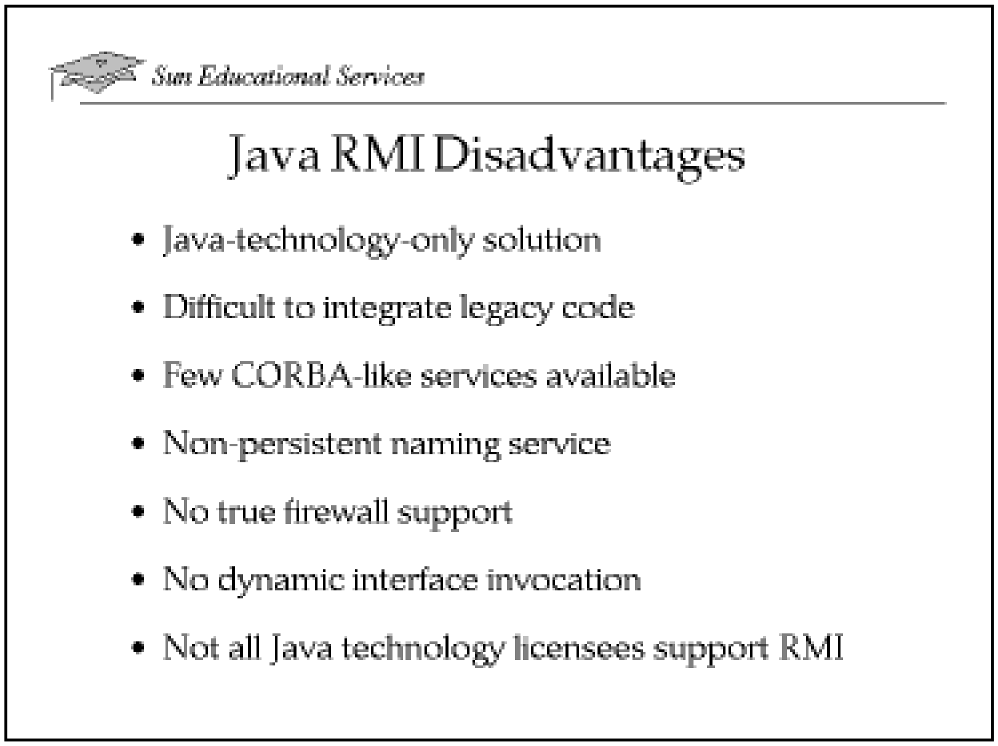
Java RMI is tightly integrated into the Java technology environment and once the remote object is located, you can invoke methods as if it were local.

RMI also uses an optimized protocol for the communication between remote Java objects. Data marshalling and object serialization are handled transparently for the developer.

With RMI, Java objects in different VMs can transparently invoke each other's methods. Since VMs can exist on different systems, RMI can provide full polymorphism.

The RMI system enables the garbage collection of active objects. The distributed garbage collection mechanism frees the developer of all the tedious memory management.

RMI allows behavior to be moved between VMs. Java classes can be built into clients and then sent to the server or vice versa. Passing Java objects in this way preserves the encapsulation. Unlike CORBA, RMI allows you to pass objects by reference as well as by value.



Sun Educational Services

Java RMI Disadvantages

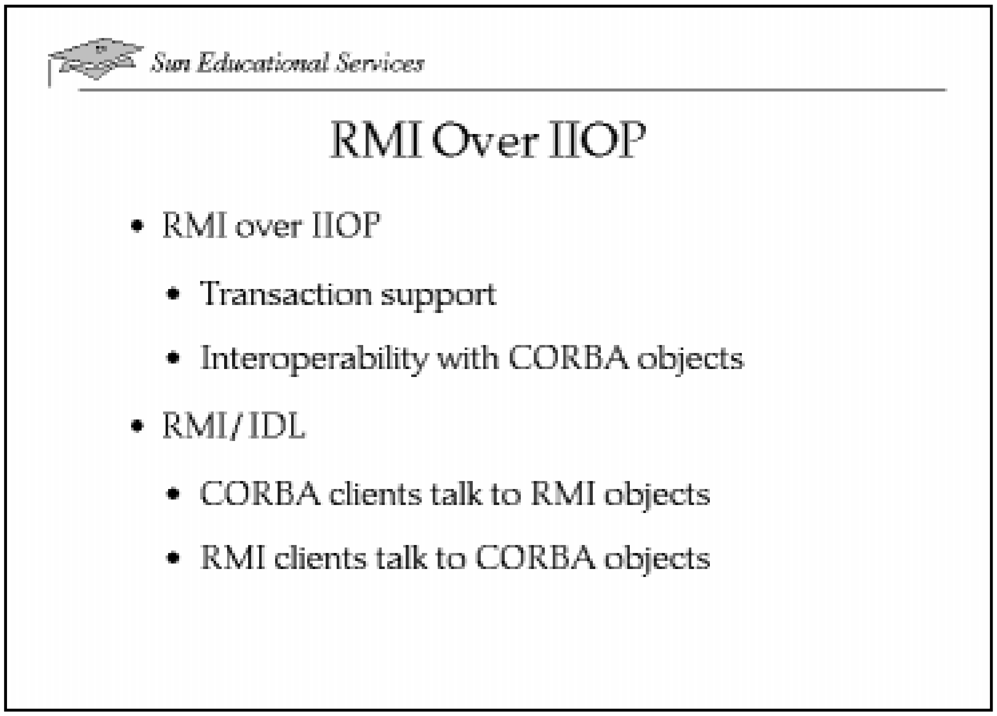
- Java-technology-only solution
- Difficult to integrate legacy code
- Few CORBA-like services available
- Non-persistent naming service
- No true firewall support
- No dynamic interface invocation
- Not all Java technology licensees support RMI

Java RMI Disadvantages

The 100% Pure Java RMI solution has an obvious drawback: the interaction with objects written in different languages is not possible. RMI does not provide methods to use services offered by C++ or Smalltalk objects. The integration of legacy code is a major problem for pure RMI applications.

Today CORBA offers services that are not available in RMI. There is, for example, no event support in RMI; a Java applet waiting for a specific event notification has to poll its server frequently to get the updates. The RMI naming service today is non-persistent and quite primitive. If an object is changed, the `rmiregistry` daemon has to be stopped, and the whole RMI subsystem must be restarted.

RMI is implemented on TCP/IP instead of IIOP. Most firewalls refuse TCP/IP communication, while others like Iona's wonderwall at least allow IIOP through firewalls. RMI tries to achieve the same effect by tunnelling its communication in HTTP, in this way disguising communication as HTTP POSTs and GETs.



Sun Educational Services

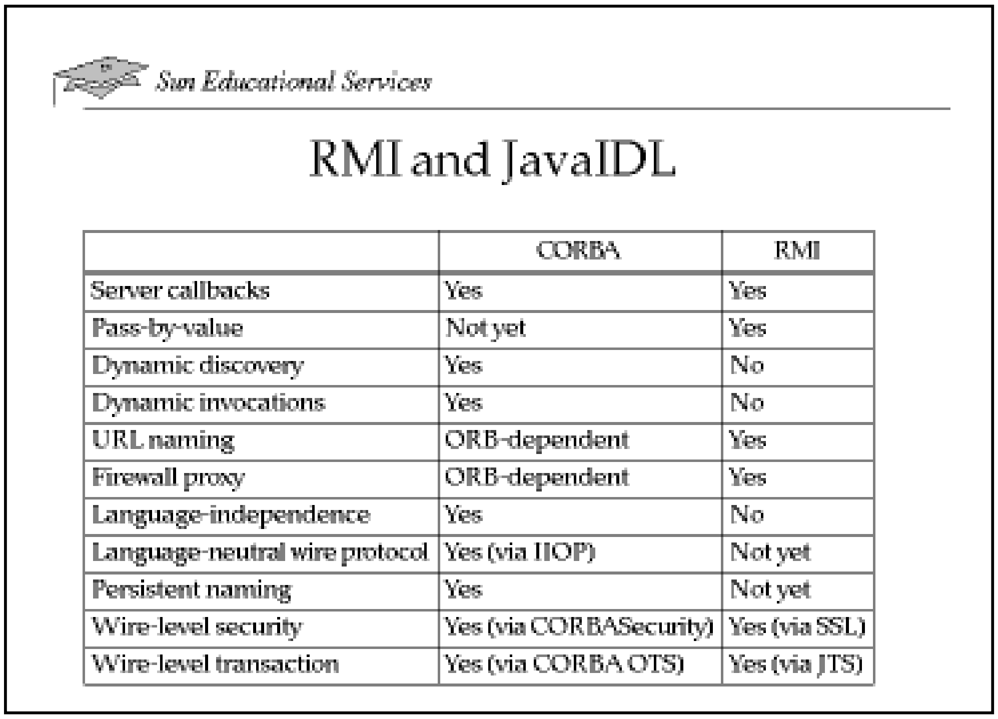
RMI Over IIOP

- RMI over IIOP
  - Transaction support
  - Interoperability with CORBA objects
- RMI/IDL
  - CORBA clients talk to RMI objects
  - RMI clients talk to CORBA objects

RMI Over IIOP

JavaSoft and OMG are currently working on a convergence of the CORBA and RMI object models. This concerns two areas:

- RMI over IIOP. With the release of the Java 2 SDK comes the support for RMI that to work on top of IIOP, thus providing the following benefits to RMI:
  - Built-in transaction support, using OMG OTS
  - Interoperability with objects written in different languages via the RMI/IDL subset
- RMI/IDL. With the CORBA Java-to-IDL standard, it is possible to define CORBA interfaces with Java RMI semantics. The RMI/IDL allows CORBA clients to call RMI objects and vice versa. RMI clients can therefore invoke methods on CORBA objects written in different languages.

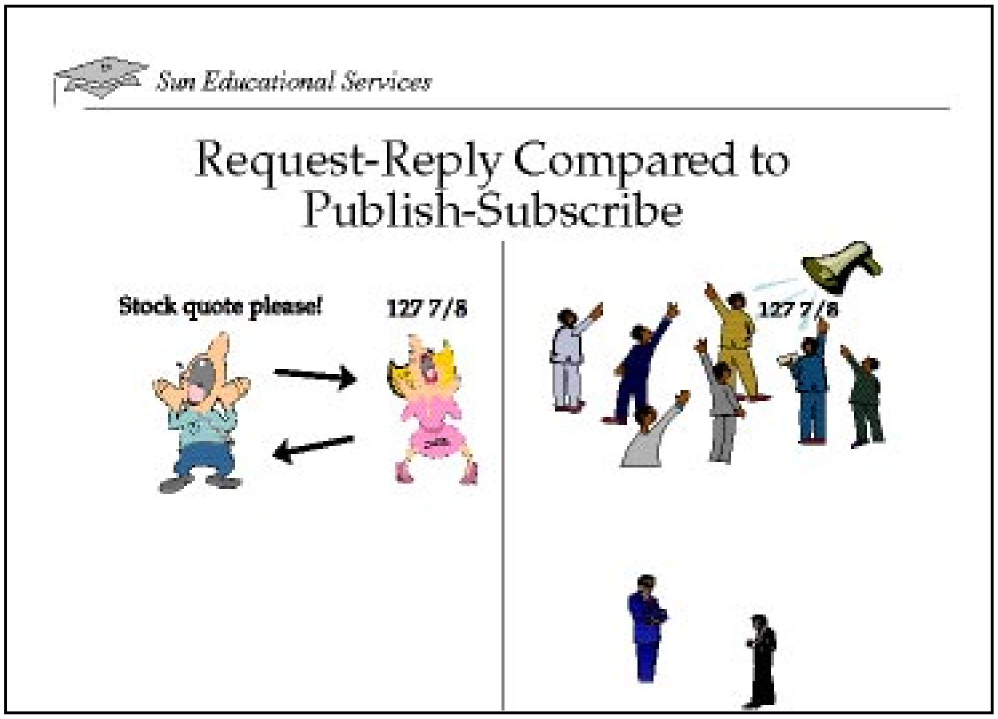

Sun Educational Services

RMI and JavaIDL

	CORBA	RMI
Server callbacks	Yes	Yes
Pass-by-value	Not yet	Yes
Dynamic discovery	Yes	No
Dynamic invocations	Yes	No
URL naming	ORB-dependent	Yes
Firewall proxy	ORB-dependent	Yes
Language-independence	Yes	No
Language-neutral wire protocol	Yes (via IIOP)	Not yet
Persistent naming	Yes	Not yet
Wire-level security	Yes (via CORBASecurity)	Yes (via SSL)
Wire-level transaction	Yes (via CORBA OTS)	Yes (via JTS)

RMI and JavaIDL

The table in the above overhead summarizes the features of RMI and JavaIDL. This course shows that the processes of building applications using these technologies are fairly similar. Each has inherent advantages the other does not match. Java RMI's advantage of being built right into the Java programming language, allowing it to exploit the full capabilities of the Java programming language will never exist in JavaIDL. The JavaIDL, though, has the advantage of working with existing legacy code.



Sun Educational Services

Request-Reply Compared to Publish-Subscribe

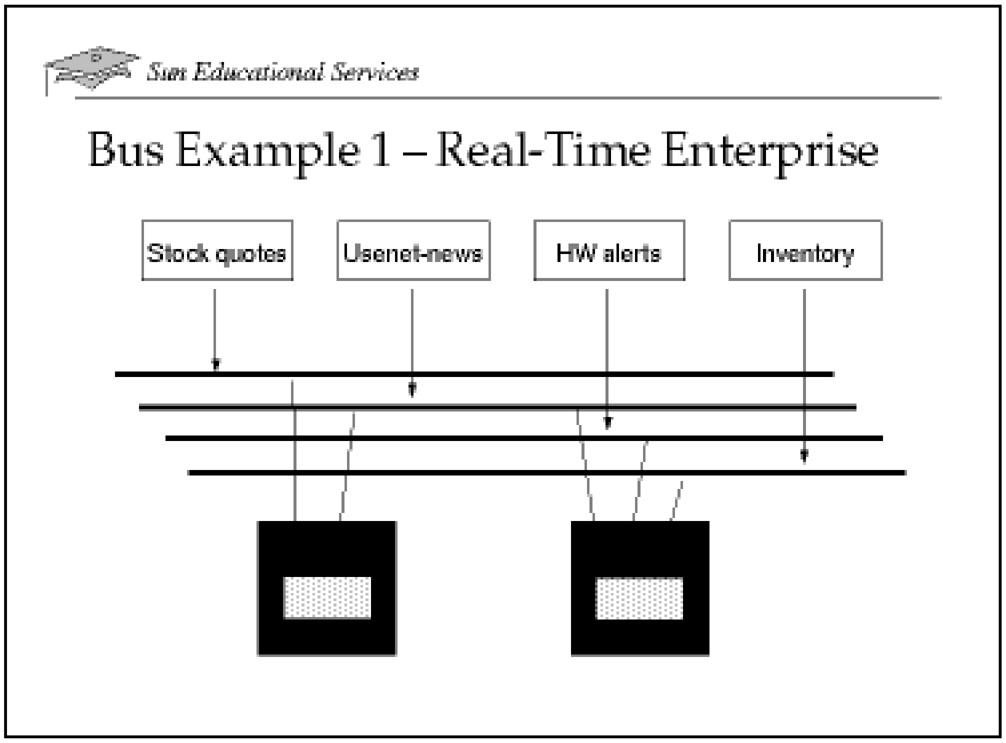
Request-Reply Compared to Publish-Subscribe

Request-reply is what most developers grew up with. A request (that is, a function call with arguments) is sent to a well-known service provider (that is, a remote object), and a reply (a return value) is sent back. Such a conversation is inherently point to point. An example of request-reply is shown on the above overhead. A person requests a stock quote, and receives the answer later. If several people want the stock quote, they all have to ask individually.

An example of *publish-subscribe* is also shown on the above overhead. Several people gather around a specific loudspeaker (they “subscribe” to whatever the speaker has to say). Somebody then talks into the microphone (something is “published”) and announces a stock quote. Everybody around the speaker can hear the quote at the same time: at the precise moment the quote is announced. Those not near the speaker will not hear the announcement.

Another difference from request-reply is the anonymity. The subscribers need to know only the position of the loudspeaker (the “channel name” in software bus terminology), not the person talking into the microphone (the remote reference of the object that is publishing).

Since most people are familiar with request-reply, but not with publish-subscribe, two publish-subscribe scenarios are shown on the next two overheads. The first scenario is an application that is clearly intended for a bus. The second scenario is an application that could be solved (and typically is) with request-reply. This would be a case where request-reply and publish-subscribe overlap.


Sun Educational Services

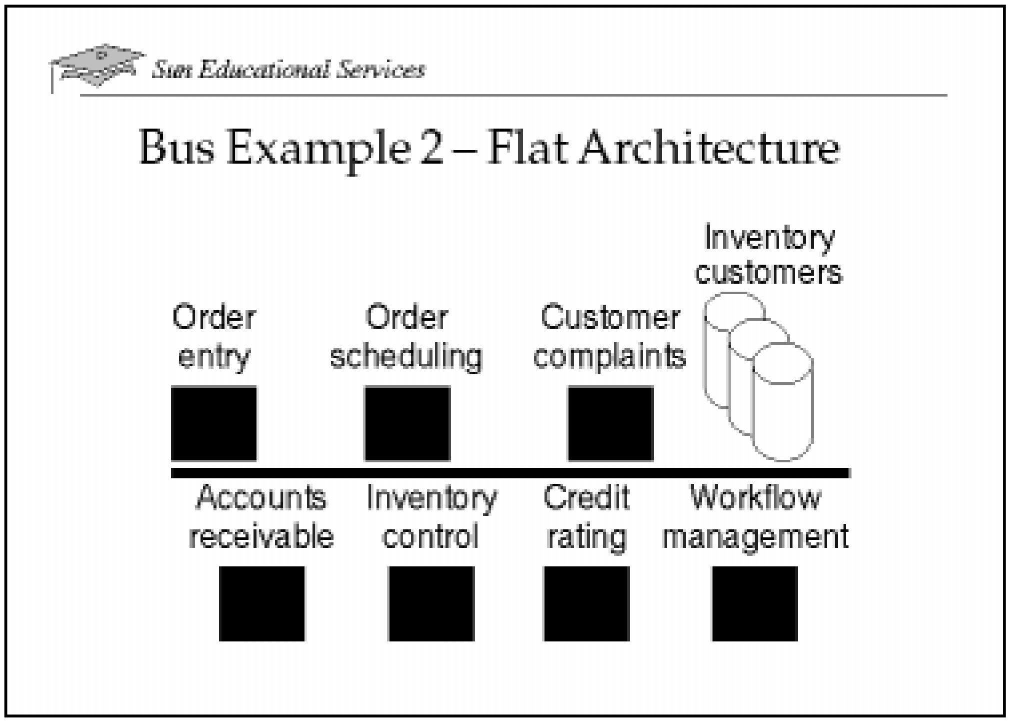
Bus Example 1 – Real-Time Enterprise

Bus Example 1 – Real-Time Enterprise

Today's enterprises need specific information at near-real time. Not everybody, however, needs the same information. Sending mass email is inefficient and not easy to administer. A publish-subscribe medium can help in this scenario.

On the above overhead, four different channels are shown: one carrying stock quotes, one carrying selected usenet-news, one carrying alerts generated by intranet hardware, and one carrying inventory changes.

The client application can be seen as a versatile tuner, which allows the user to select relevant channels for display. More function-rich clients are possible as well; for example, the hardware alerts could be processed in a network management application.



Bus Example 2 – Flat Architecture

The above overhead shows the components of a simplified but typical enterprise environment. Usually, the components are linked together individually: order entry is linked with the customer database, with order scheduling, with accounts receivable, and so on. These connections are point to point (via CORBA or RMI) and hardcoded. If the organization is changed, the system must be rebuilt.

On the overhead, the components are linked only to the bus. The order entry components publish all the information about a newly entered order onto a specific channel on the bus. Every component that needs to know about orders is registered to this channel, and acts accordingly. New functionality can be added by creating new components and subscribing them to relevant channels. Organizational changes can usually be dealt with by updating a few components only. Other non-affected components never know about the change, since the channel continues to deliver the expected information.

Check Your Progress

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

- Explain the role of JDBC and servlets in a distributed application
- Compare and contrast RMI and JavaIDL
- Compare and contrast request-reply and publish-subscribe architectures

Think Beyond

How do all the distributed technologies work together?

Flags for the `idltojava` *Utility* *A*

This appendix describes the options and environmental files for the `idltojava` utility.

The IDL-to-Java Compiler: `idltojava`

The `idltojava` utility compiles IDL files to Java source code.

Syntax

```
idltojava [ options | flags ] filename.idl ...
```

Description

The `idltojava` utility compiles IDL source code into Java source code. You then use the `javac` compiler to compile that source to Java bytecodes. The IDL declarations from the named IDL files are translated to Java programming language declarations according to the mapping from IDL to the Java programming language.

Options

Options are used to pass some environment-specific information to the `idltojava` utility. The following three options deal with preprocessor directives:

- -I *`directoryName`* – Deals with #`include`
- -D *`symbol`* – Deals with #`define`
- -U *`symbol`* – Deals with #`undefine`

All three tell the `idltojava` utility to pass on information to the preprocessor.

The following are the options you can use with the `idltojava` utility:

- `-j` *`javaDirectory`* – Specifies that generated Java files should be written to the given *`javaDirectory`*. There must be a space between `-j` and the directory name. Also, you must create the directory before specifying it in this option.
- `-I` *`directoryName`* – Tells `idltojava` to pass on to the preprocessor the information that the directory *`directoryName`* is the place to search for the files in #`include` directives contained in the IDL file.
- `-D` *`symbol`* – Specifies that *`symbol`* must be defined during preprocessing of the IDL file(s).
- `-U` *`symbol`* – Specifies that *`symbol`* must be undefined during preprocessing of the IDL files.

Flags

Flags give instructions to the `idltojava` utility and can be turned on or off. To turn on a flag, type the following:

`idltojava -f`*flag-name.*

For example, the following directs `idltojava` to print a list giving the current state of each flag:

```
idltojava -flist-flags hello.idl
```

To turn off a flag, type the following:

`idltojava -fno-`*`flag-name`*

The following line directs `idltojava` not to use the C/C++ preprocessor before compiling the IDL file `hello.idl`:

```
idltojava -fno-cpp hello.idl
```

Each flag is set to a default value. Turning a flag off when it is already off does nothing; similarly, turning a flag on when it is already turned on does nothing.

If you run `idltojava` on an IDL file and supply no flags, `idltojava` runs the C/C++ preprocessor on the IDL file, produces a portable client stub file, produces a portable server skeleton file, considers case when comparing identifiers, and writes out the `.java` files it produces.

The following are the flags you can use with `idltojava`:

- `-fcaseles` – When ON, requests that capitalization *not* be considered in the comparison of identifiers. The default is OFF. In other words, the default behavior of the `idltojava` compiler is to consider case when comparing identifiers. Use `-fno-caseless`.ogm to tell `idltojava` not to consider case.
- `-fclien` – When ON, requests the generation of the client side of the IDL files supplied. Default is ON.
- `-fcp` – When ON, runs the C/C++ preprocessor on the IDL file supplied. Default is ON.

Note – `idltojava` is hard-coded to use a default preprocessor. It uses the path `/usr/ccs/lib/cpp` when looking for the preprocessor. You can change the preprocessor that `idltojava` uses by setting two environment variables:

`CPP` – Set this environment variable to the full path name of the preprocessor executable you want to use.

`CPPARGS` – Set this environment variable to the complete list of arguments to be passed to the preprocessor. The preprocessor needs to write to standard output, so if it does not do so by default, include the argument appropriate to your preprocessor to accomplish that.

`-f list-flag` – When ON, requests that the current state of all the `-f` flags be printed. Default is OFF.

- `-f list-option` – When ON, requests a list of command-line options. Default is OFF.
- `-f map-included-file` – When ON, requests that Java code be generated for the IDL file and all the files listed in `#include` directives in the IDL file. Default is OFF.
- `-f portabl` – When ON, requests the generation of portable stubs and skeletons. Default is ON.

- -f serve – When ON, requests the generation of the server side of the IDL files supplied. Default is ON.
- -f t – When ON, requests the generation of the Operations interface and Tie class, which are required for the implementation of a delegation-based skeleton. For the file hello.idl, idltojava creates the files _helloTie.java and _helloOperations.java. This flag does nothing if the -fserver flag is turned off. Default is OFF.
- -f verbos – When ON, requests that idltojava print comments on the progress of the compilation. Default is OFF.
- -f versio – When ON, requests that idltojava print its version and timestamp. Default is OFF.
- -f write-file – When ON, requests that the Java files that are generated be written out. idltojava creates a directory named after the module for the .idl file and puts the .java files in it. This new directory is in the same directory as the .idl file. Turning the flag off allows the programmer to validate the IDL code before writing out the generated Java files. Default is ON.

Using the #pragma *Compiler Directive*

Java IDL supports the #pragma compiler directive. A #pragma directive must appear at the beginning of an IDL file so that it has global scope.

To request a repository prefix, type the following:

```
#pragma prefix "requested_prefix_name"
```

To wrap the default package in one called package, type the following:

```
#pragma javaPackage "package"
```

For example, compiling an IDL module M normally creates a Java package M. If the module declaration is preceded by the directive

```
#pragma javaPackage "browser"
```

the compiler creates the package M inside package browser. This #pragma is useful when the definitions in one IDL module are used in multiple products.

SQL Syntax *B*

This appendix provides an overview of the commonly used SQL statements.

SQL 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 set:

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

The syntax of these clauses is meant to be read like English commands and can be spoken aloud. For example, an SQL command could be "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 that held a name, employee ID, date of hire, social security number, and current salary, you would expect to receive a single employee record with these values.

In mSQL, this command would be written using the following statement syntax:

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

Note – The `employee_id` field is presented here as a string, but if it were an integer or a real number, you would not require the single quotes.

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

SELECT *Statement*

Syntax

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 mSQL's SELECT is:

```
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 that of standard SQL:

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

Examples

```
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`, use the following query:

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

Note – Here `DESC` applies to both `last_name` and `first_name.`

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

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

Note – `DISTINCT` is useful when you do not have a primary key.

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

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

Joins

Note – The SQL JOIN keyword is not supported.

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; 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 are forced to qualify every column name as soon as you access more than one table in a single SELECT.

INSERT *Statement*

Note – Unlike ANSI SQL, you cannot nest a SELECT within an INSERT (in other words, you cannot insert the data returned by a SELECT).

The INSERT keyword is used to add new SQL records to a table. 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.

```
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')
```

Note – Single quotes are possible within a field item by escaping the single quote: "\".

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.

DELETE *Statement*

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

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

Where

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

For example:

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

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 is

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

Where

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

For example:

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

Glossary

address

A location on a computer network, on a peripheral device, or in computer memory.

address space

The range of memory locations to which a CPU can refer; effectively, the amount of memory a CPU could use if all of the memory were available.

applet

A Java program that can be included in an HTML page using the `applet` tag.

application

Any specific use of the computer. The term is often used synonymously with *program*.

application programmer's interface (API)

The interface to a library or package of language-specific functions or methods.

browser

A program used to view World Wide Web materials that is capable of interpreting URLs and understanding different Internet protocols.

class loader

A class loader is the foundation of the Java virtual machine (JVM). A class loader is an executable class object that converts a named class into the bits that make up an implementation of that class. Class loaders enable the JVM to load a class without having to know anything about the underlying file system semantics. Class loaders extend the abstract class `java.lang.ClassLoader` and implement the `loadClass` method at minimum.

client

A software program that requests information or services from another software application (server). For example, a browser is a client that accesses data from HTTP servers.

Common Object Request Broker Architecture (CORBA)

The architecture and specifications aimed at software developers and designers who want to produce applications that comply with OMG standards for the ORB.

content handler

A specialized Java program that enhances Java technology functionality by providing a means to understand a new content type; for example, email, video, or audio files of nonsupported types. It is responsible for reading data from a stream (provided by a protocol handler) and returning an object representation of the stream's content.

distributed application

A program that makes calls to other address spaces, possibly on another physical machine.

distributed computing

The technique of allowing applications running on one machine to access applications that are running on another machine. That is, client programs make calls to programs in other address spaces, either local or remote.

distributed object computing

An extension of distributed computing, where objects are implemented in an address space separate from the client program.

externalizable

Java objects are externalizable when they implement the `java.io.Externalizable` interface and implement the methods `writeExternal(ObjectOutput out)` and `readExternal(ObjectInput in)`. Externalizable objects are serializable, but only the identity of the class is saved by default. It is the responsibility of the class to save and restore the contents of the object.

firewall

A machine or machines that run filtering and logging software, which restrict and/or monitor traffic passing from one network to another. A firewall is the single point through which all traffic between two networks must pass, so that network and application security policies can be implemented.

FTP

File Transfer Protocol. It is an Internet client-server protocol for transferring files between computers.

GUI

Graphical user interface.

HTTP

Hypertext Transfer Protocol. The most common protocol used on the World Wide Web to transfer hypertext documents.

`idlgen`

A command used to compile JavaIDL.

Interface Definition Language (IDL)

The Object Management Group (OMG) defined a set of language constructs that can be used to define the interface between an implementation of the interface and an application that executes operations on the interface. The Interface Definition Language (IDL) is the result of their work. The IDL defines a “contract” of services. Each service encapsulates operations and attributes and can specify exceptions that occur. IDL is not a programmatic language; the IDL file must be compiled to the specific language implementation desired.

Internet

The worldwide network of computers communicating using the TCP/IP protocols.

Java

An object-oriented programming language developed by Sun Microsystems to solve a number of problems in modern programming practice.

JDBC API

A set of interfaces designed to insulate a database application developer from a specific database vendor.

JavaIDL API

A set of classes and interfaces that enables developers to define a set of remote interfaces using the CORBA IDL standard, and maps IDL constructs onto a set of stubs and skeletons that can be used to create a CORBA implementation.

multicast

A special form of broadcast where copies of the packet are delivered to only a subset of all possible destinations.

network

A group of connected computers.

NFS

A distributed application that enables remote file systems to be accessed by the end-user in the same way that a user would access a local file system.

Object Management Group (OMG)

A nonprofit international consortium dedicated to promoting the theory and practice of object technology for the development of distributed computing systems.

Object Request Broker (ORB)

A program that provides the communications infrastructure that enables objects to transparently make and receive requests and responses in a distributed environment.

Object Serialization API

A set of classes and interfaces that enables developers to write Java code that creates persistent storage for Java objects.

one-tier database design

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

protocol

An agreed convention for inter-computer communication.

protocol handler

A specialized Java program that enhances Java technology functionality by providing a means to understand a new protocol type; for example, IPX/SPX or Asynchronous Transfer Mode (ATM). It is responsible for establishing a connection and defining data stream syntax (how data will pass from one endpoint to another). It returns a connection object that can then be used for opening data streams.

RMI API

A set of classes and interfaces designed to enable developers to make calls to remote objects that exist in the runtime of a different virtual machine invocation.

Remote Procedure Call (RPC)

A paradigm for implementing the client-server model of distributed computing. A request is sent to a remote system to execute a designated procedure, using supplied arguments, and return the result to the caller.

replicated

An object that has one or more exact copies of itself available from multiple address spaces. For example, under NIS+, replica servers provide the same information to clients as the master server, and it is irrelevant to the client which server the information comes from.

`rmic`

The RMI compiler that generates the RMI sub and skeleton classes.

`rmiregistry`

An application that provides a simple naming lookup service of remote objects in the RMI API.

Security Manager API

A class that enables developers to set and control the security policy that must be followed by Java programs running on a system.

serializable

Java objects are serializable if they implement the `java.io.Serializable` interface and do not contain references to nonserializable objects (for example, `java.lang.Thread`, `java.io.FileOutputStream`) unless these references are marked with the `transient` keyword. An object that is serializable may be serialized.

serializing or serialized

Java objects that are serializable can be sent as a stream of bytes over any type of `java.io.Output` stream and retrieved using any type of `java.io.InputStream`. This includes files, pipes, and sockets. An object is said to be serialized when it is converted from its internal in-memory format to the stream of bytes that can be sent over the wire.

server

In the client-server model for file systems, the server is a machine with compute resources (and is sometimes called the compute server) and large memory capacity.

skeleton

A server-side entity that contains a method that dispatches calls to the actual remote object implementation.

socket

A software endpoint for network communication. Two programs on different machines each open a socket in order to communicate over the network. This is the low-level mechanism that supports most networking programs.

stub

A proxy for a remote object that is responsible for forwarding method invocations on remote objects to the server where the actual remote object implementation resides.

TCP

Transport Control Protocol. A virtual circuit protocol of the Internet protocol family. It provides reliable, flow-controlled, in-order, two-way transmission of data in a byte stream.

thread

In programming, a process that is part of a larger process or program.

three-tier database design

A database design that introduces an intermediary tier between the database front end and the database engine. This intermediary tier can support functionality such as mirroring, secure transactions, and caching.

two-tier database design

A database design that separates the database front end from the database engine, enabling the data to reside locally or remotely.

UDP

User Data Protocol. A transport protocol in the Internet suite of protocols using datagrams.