

Distributed Programming

Gabriel Oteniya and Milton Chau Keng Fong

UNU-IIST

The Course

- 1) **objectives** - what do we intend to achieve?
- 2) **outline** - what content will be taught?
- 3) **resources** - what teaching resources will be available?
- 4) **organization** - duration, major activities, daily schedule

Course Objectives

- 1) learn the fundamental concepts of distributed programming for enterprise application development
- 2) learn the various distributed programming architectures and how to apply them
- 3) learn the importance of distributed computing and outline the factors to consider when designing a distributed system
- 4) presents different Distributed Architecture

Course Outline

- 1) introduction
- 2) streams
- 3) networking
- 4) database connectivity
- 5) architectures
 - a) message-orientation
 - 1) javamail
 - 2) jms
 - b) distributed objects
 - 1) rmi
 - 2) corba
 - 3) JavaIDL
- 6) summary

Outline: Introduction

Presents an overview of the distributed programming.

Main points:

- 1) what are distributed systems?.
- 2) why distributed programming?.
- 3) nature and design considerations.
- 4) types of networks.
- 5) distributed architectures.

Outline: Stream

Presents the `java.io` package

Main points:

- 1) what is a stream?
- 2) types of Streams
- 3) characteristics of Streams
- 4) working with streams
- 5) bridging Streams
- 6) stream chaining

Outline: Networking

Presents the network programming in Java language.

Main points:

- 1) review the basic network concepts and Java Implementation
- 2) discuss the usage of java.net package.
- 3) introduce the Secure Socket.
- 4) introduce the New I/O API.
- 5) introduce the Java implementation for UDP protocol.

Outline: Database Connectivity

Presents JDBC API from the basics of SQL to the more esoteric features of advanced JDBC.

Main points:

- 1) introduction to Database and Structured Query Language
- 2) JDBC architecture
- 3) JDBC core interfaces
- 4) query processing
- 5) transaction Management

Outline: Message Orientation

Presents how to build a loosely coupled application using messaging mechanism.

Main points:

- 1) JavaMail – to provide asynchronous communication between application components and human users.
- 2) Java Message Service – to provide asynchronous/synchronous communication software components

Outline: Distributed Objects

This section basically addresses:

- 1) Remote Method Invocation (RMI)
- 2) Common Object Request Broker Architecture (CORBA)
- 3) Interface Definition Language (IDL)
- 4) IDL to Java mapping (JavaIDL)

Outline: Summary

Revision of the material introduced during the course.

How this course provides a foundation for the remaining courses:

- 1) Java XML processing
- 2) Java Web Services
- 3) J2EE web components
- 4) J2EE business components

Course Resources

1) books

- a) Distributed Programming with Java, Qusay H. Mahmoud, Manning Publisher 2000
- b) Java in Distributed Systems: Concurrency, Distribution and Persistence, Marko Boger, 2001
- c) Developing Distributed and E-commerce Applications, 2nd edition, Darrel Ince, 2nd edition, Pearson Addison Westly, 2004.
- d) Java Message Service (O'Reilly Java Series), Richard Monson-Haefel, David Chappell

2) tools

- a) mySQL Database engine
- b) JBoss 4.0.1
- c) JBossMQ
- d) Hermes 1.8 (JBossMQ Browser)

Course Logistics

- 1) **duration** - 36 hours
- 2) **activities** - lecture (hands-on), development
- 3) **sessions/day** - morning 09:00–13:00 and afternoon 14:30–16:30
- 4) **number of sessions** - 6 morning and 6 afternoon
- 5) **style** – interactive, Lab work and tutorial

Course Prerequisite

- 1) some experience in object-oriented programming:
 - a) C++
 - b) Delphi
 - c) Java Programming Language
 - d) any other object-oriented language
- 2) basic understanding of TCP/IP networking concepts

Introduction

Course Outline

- 1) introduction
- 2) streams
- 3) networking
- 4) database connectivity
- 5) architectures
 - a) message-orientation
 - 1) javamail
 - 2) jms
 - b) distributed objects
 - 1) rmi
 - 2) corba
 - 3) JavaIDL
- 6) summary

Distributed System

Distributed system can be defined as a combination of several computers with separate memory, linked over a network, and on which it is possible to run a distributed applications.

Characteristics:

- 1) capable of communicating over a network
- 2) the network is usually stable
- 3) fail-safe
- 4) each device has a permanent identification within the network

Hence, it is a collection of independent computers, interconnected via a network, capable of collaborating on a task.

Distributed Application

A distributed application consist of several parts of a program communicating with each other, which cooperate to carry out a common task.

For example, client server application.

Typically, but not necessarily, the parts of the application are distributed across several computers.

The distribution can also be simulated on one computer.

In this case, however, information is not transmitted via a common memory or address space, but with the aid of techniques of remote communication.

Distributed Programming

Distributed programming is a model in which processing occurs in many different places (or nodes) around a network.

Characteristics:

- 1) processing can occur whenever it makes the most sense
- 2) carried out on a distributed system
- 3) making calls to other address spaces possibly on different machines
- 4) tasks are handled in parallel

Why Distributed Programming?

- 1) balance resource loading
- 2) lower cost of development since clients can access remote codes for services
- 3) separation of concerns
- 4) Platform independence

Design Considerations

In general, three aspects need to be put into consideration:

- 1) **Concurrency** – actual or apparent parallelism of control flows
issues: how to manage both heavy and light weight processes

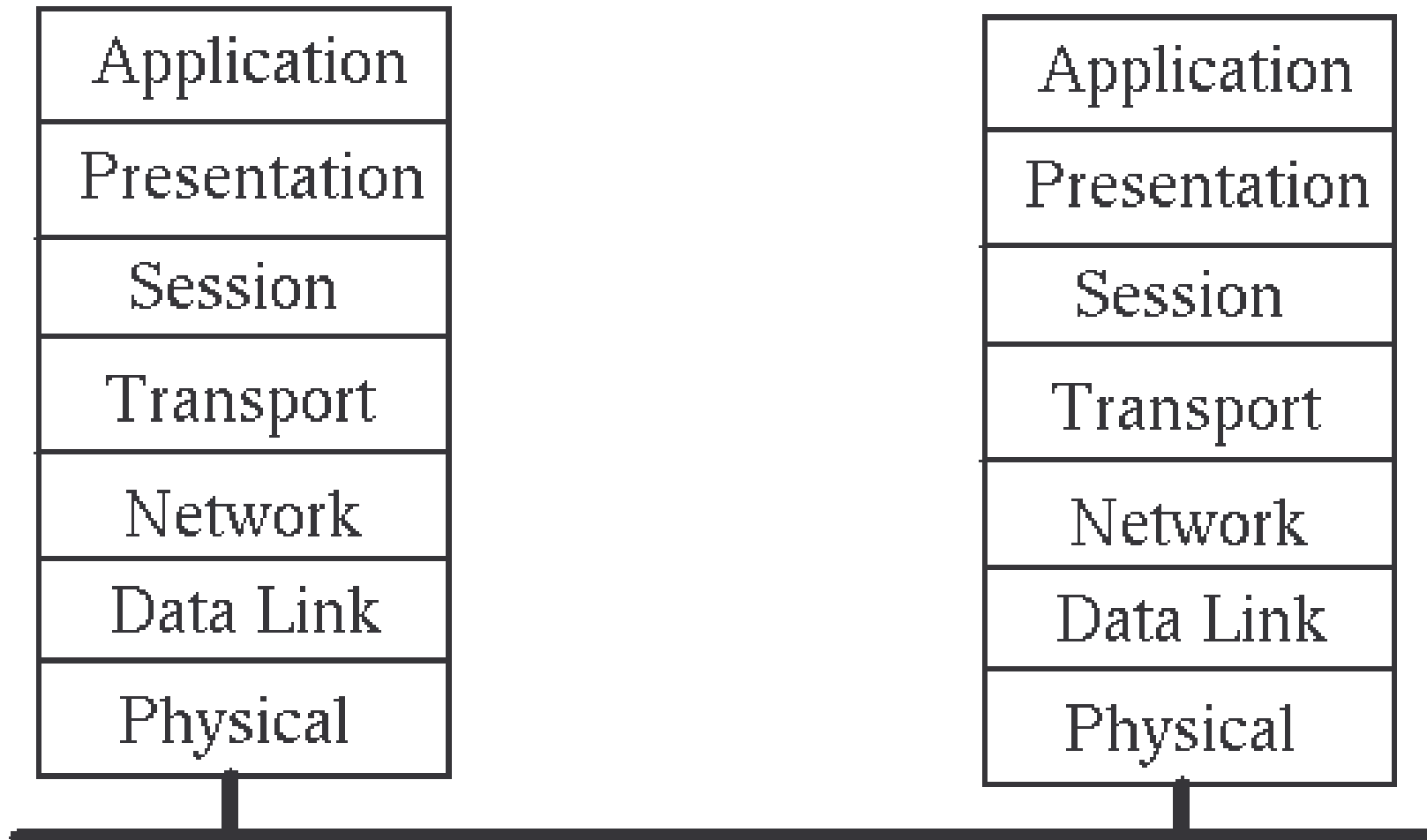
- 2) **Distribution** – is the logical and spatial distance of objects from each other
Issue: how these object can locate, access and communicate with each other

- 3) **Persistence** – is the long-term storage of data or objects on non-volatile media
issues: how to persist data and objects. Persistence achives the distribution of data or objects in time.

Protocol Layers

- 1) Communications between processes takes place using agreed conventions - protocols
- 2) Network communications requires protocols to cover high-level application communication all the way down to wire communication
- 3) Complexity handled by encapsulation in protocol layers

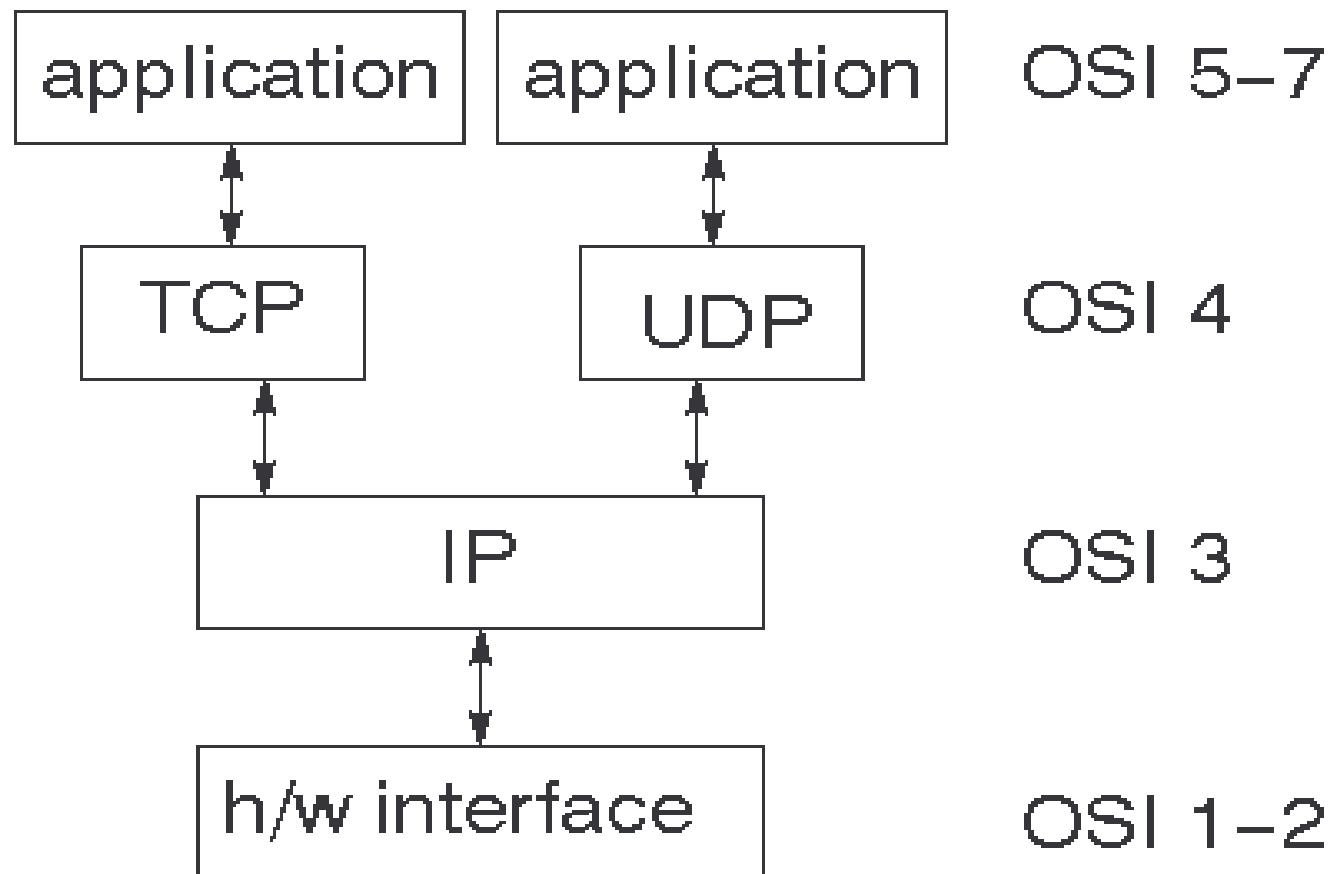
ISO OSI Protocol



OSI layers

- 1) Network layer provides switching and routing technologies
- 2) Transport layer provides transparent transfer of data between end systems and is responsible for end-to-end error recovery and flow control
- 3) Session layer establishes, manages and terminates connections between applications.
- 4) Presentation layer provides independence from differences in data representation (e.g. encryption)
- 5) Application layer supports application and end-user processes

TCP/IP Protocol



Port and Socket

1) port

- a) conduit into a computer through which information flows and assigned a unique number
- b) usually port numbers 0 to 1023 are reserved for special purposes (e.g. HTTP – 80, FTP – 21, SMTP – 25)
- c) TCP/IP-based computer is identified by a pair of IP address and Port number

2) socket

- a) a socket is one end of a process that an application is using to communicate
- b) defined by two addresses: the IP address of the host computer; and the port address of the application or process running on the host

Connection Models

There are two types of connection models:

- 1) Connection oriented
- 2) Connectionless

Connection oriented transports may be established on top of connectionless ones –TCP over IP

Connectionless transports may be established on top of connection oriented ones – HTTP over TCP

Connection oriented

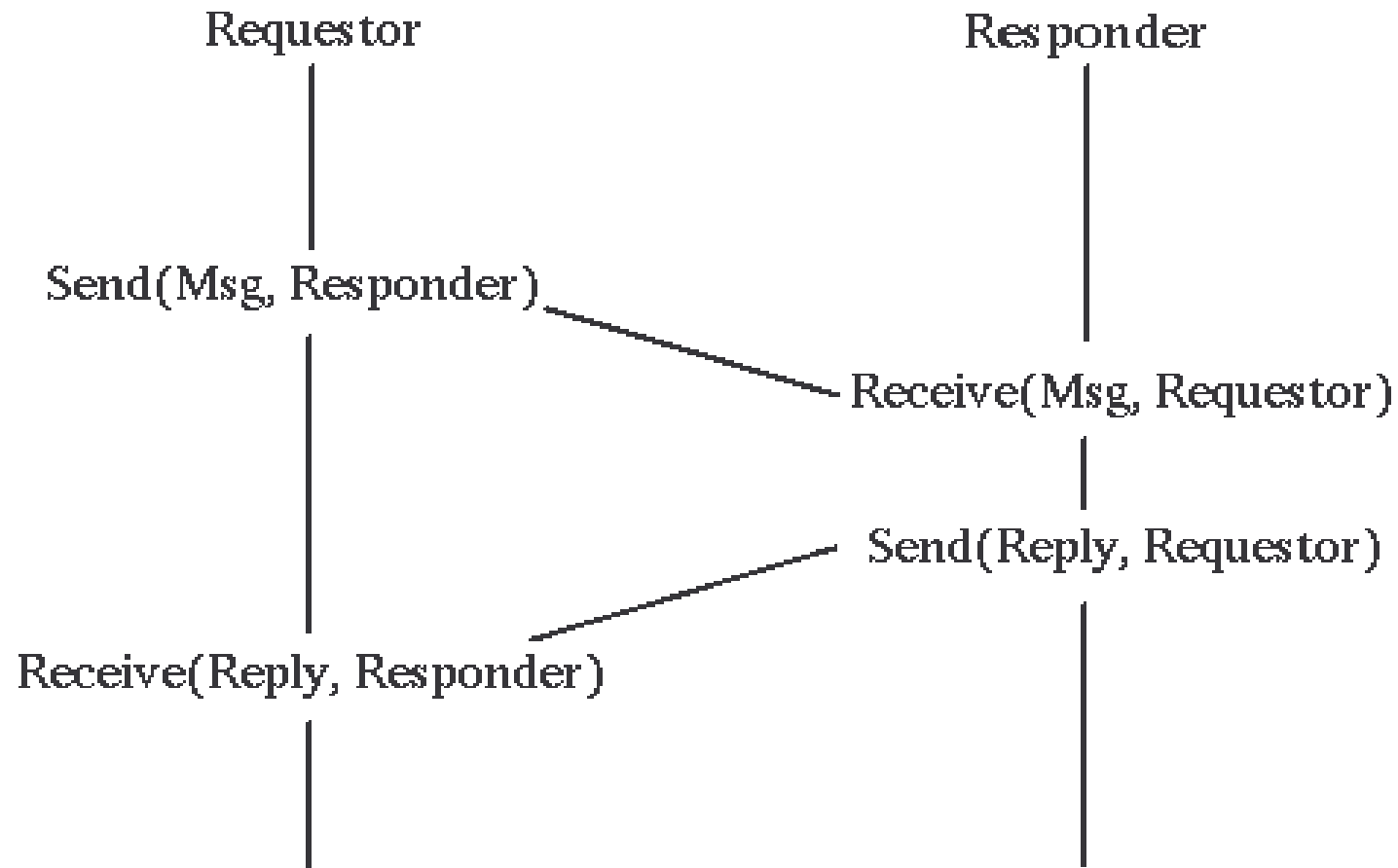
- 1) A single connection is established for the session
- 2) Two-way communications flow along the connection
- 3) When the session is over, the connection is broken
- 4) The analogy is to a phone conversation
- 5) An example is TCP

Connectionless

- 1) In a connectionless system, messages are sent independent of each other
- 2) Ordinary mail is the analogy
- 3) Connectionless messages may arrive out of order
- 4) An example is the IP protocol

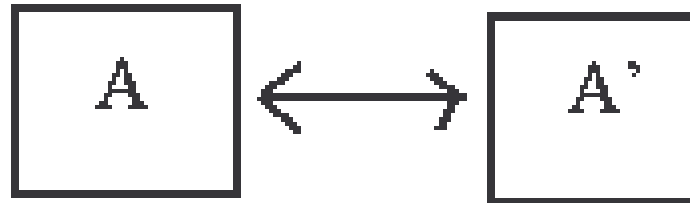
Communications Model

Message passing



Distributed Computing Models

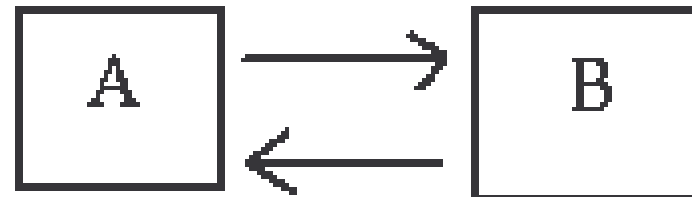
peer-to-peer



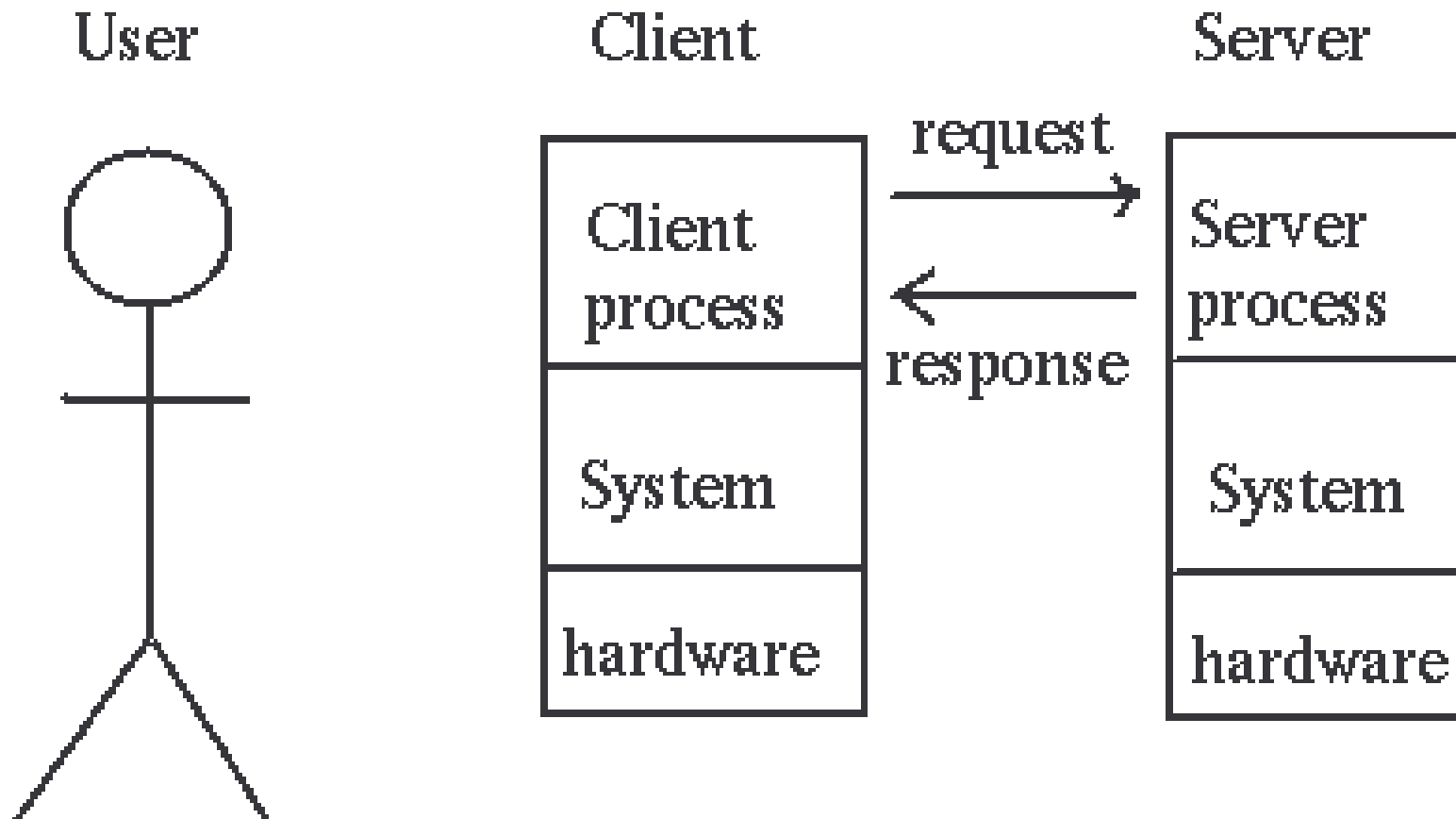
filter



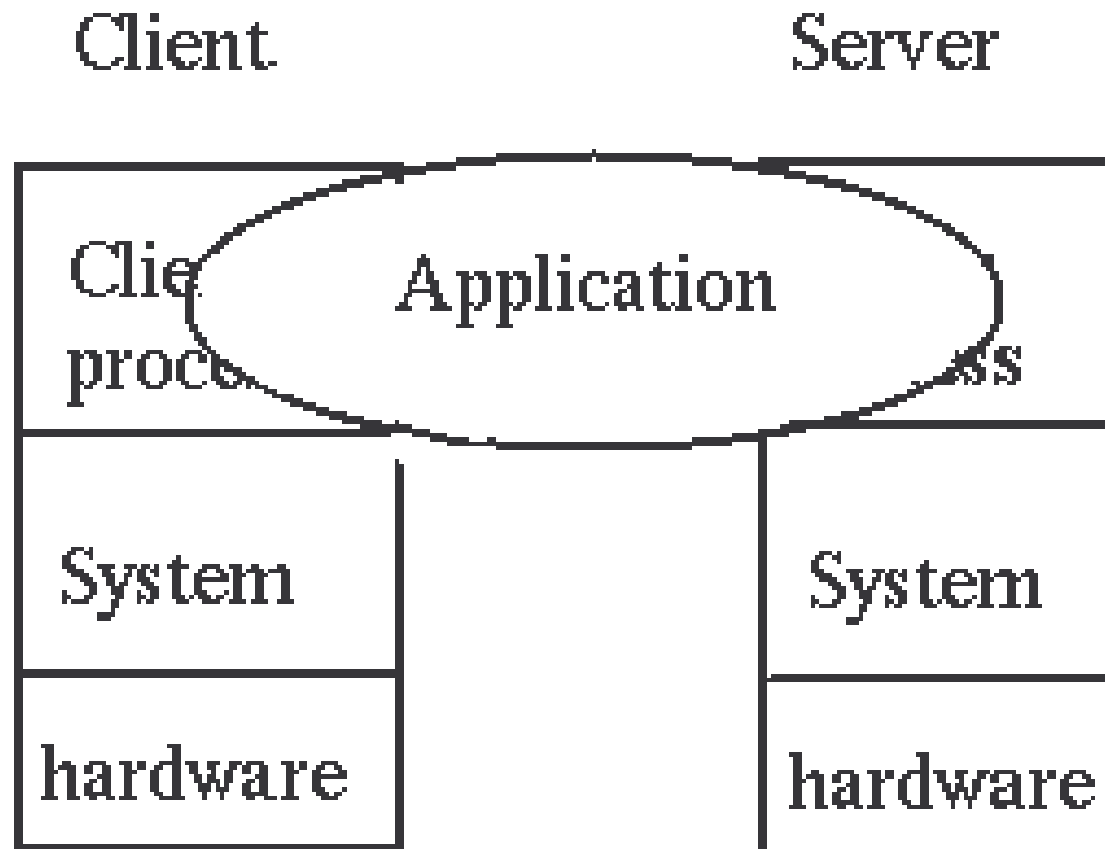
client-server



Client/Server System



Client/Server Application

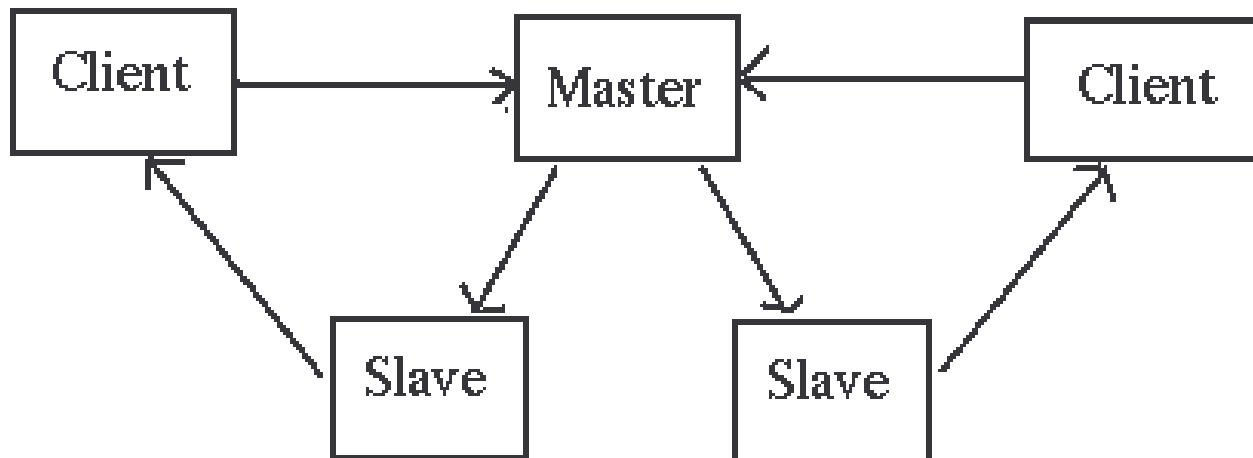


Server Distribution 1

Single client, single server



multiple clients, single server



Server Distribution 2

single client, multiple servers



multiple clients, multiple servers

Component Distribution

Every distribution is made up of three components:

- 1) Presentation component
- 2) Application logic
- 3) Data access

Middleware 1

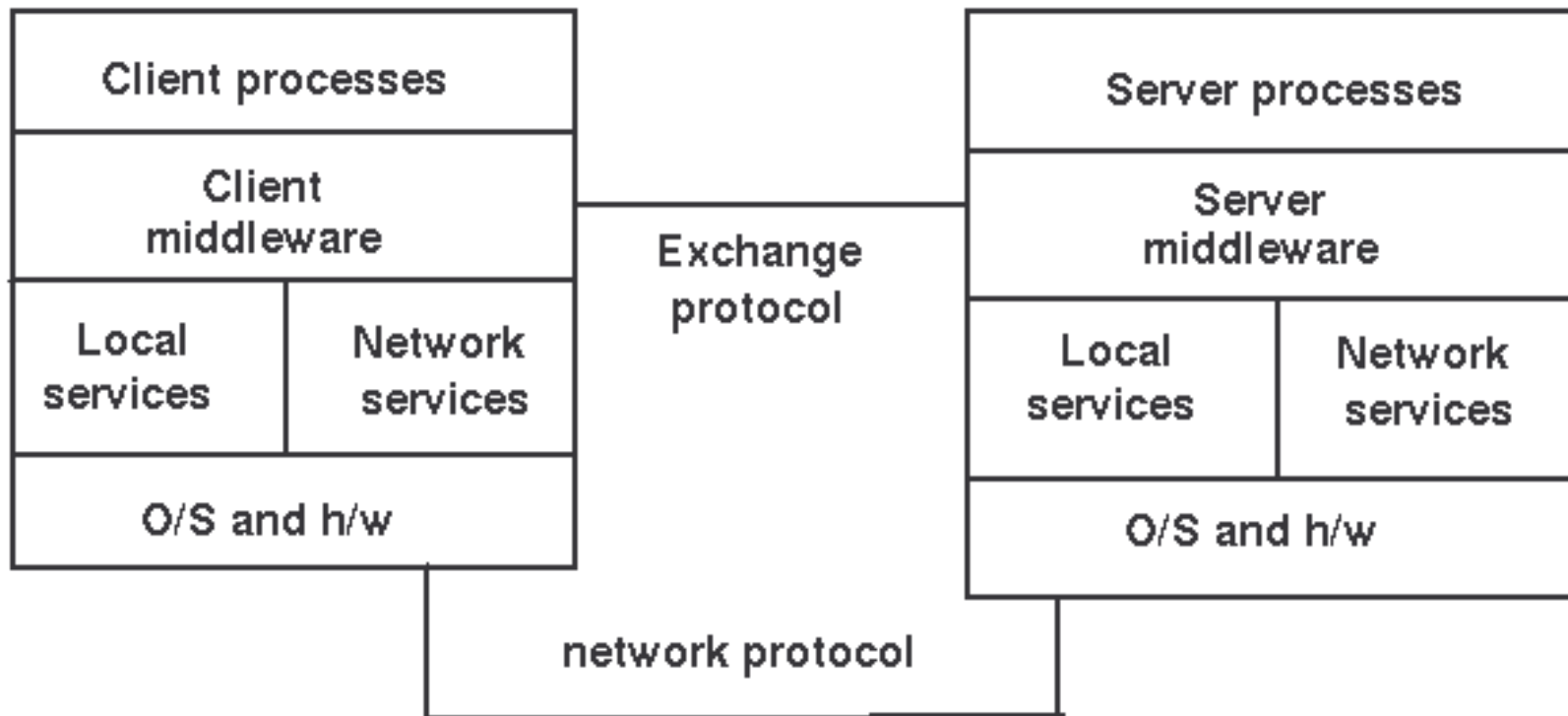
- 1) intermediate layers between client and server
- 2) what exactly is it?
 - a) a vague term that covers all the distributed software needed to support interactions between client and server
- 3) where does the middleware start and where does it end?
 - a) It starts with the API set on the client side that is used to invoke a service, and it covers the transmission of the request over the network and the resulting response”

Middleware 2

- 1) The network services include things like TCP/IP
- 2) The middleware layer is application-independent s/w using the network services
- 3) Examples of middleware are: DCE, RPC, Corba
- 4) Middleware may only perform one function (such as RPC) or many (such as DCE)

Middleware Model

The middleware model is



Example: Middleware

1) Primitive services such as terminal emulators, file transfer, email

- Basic services such as RPC

1) Integrated services such as DCE, Network O/S

- Distributed object services such as CORBA, OLE/ActiveX
- Mobile object services such as RMI, Jini
- World Wide Web

Middleware Functions

- 1) Initiation of processes at different computers
 - Session management
- 1) Directory services to allow clients to locate servers
 - remote data access
 - Concurrency control to allow servers to handle multiple clients
 - Security and integrity
 - Monitoring
 - Termination of processes both local and remote

Project Exercise 1

- 1) Describe a typical distributed system in use in your agency
- 2) Which of the following distributed architecture models best represents the distributed system described in question 1?
- 3) List the different components of the systems listed in 1
- 4) Provide a model of the system described in question 1 using a UML deployment diagram showing the various components listed in question two as well as the nodes hosting these components.
- 5) Identify the possible points of failures in the distributed system using the model presented in question 4.

Streams

Course Outline

- 1) introduction
- 2) **streams**
- 3) networking
- 4) database connectivity
- 5) architectures
 - a) message-orientation
 - 1) javamail
 - 2) jms
 - b) distributed objects
 - 1) rmi
 - 2) corba
 - 3) JavaIDL
- 6) summary

Overview

- 1) what is a stream?
- 2) types of Streams
- 3) characteristics of Streams
- 4) working with streams
- 5) bridging Streams
- 6) stream chaining

Introduction

Most programs use data in one form or another, whether as input, output, or both.

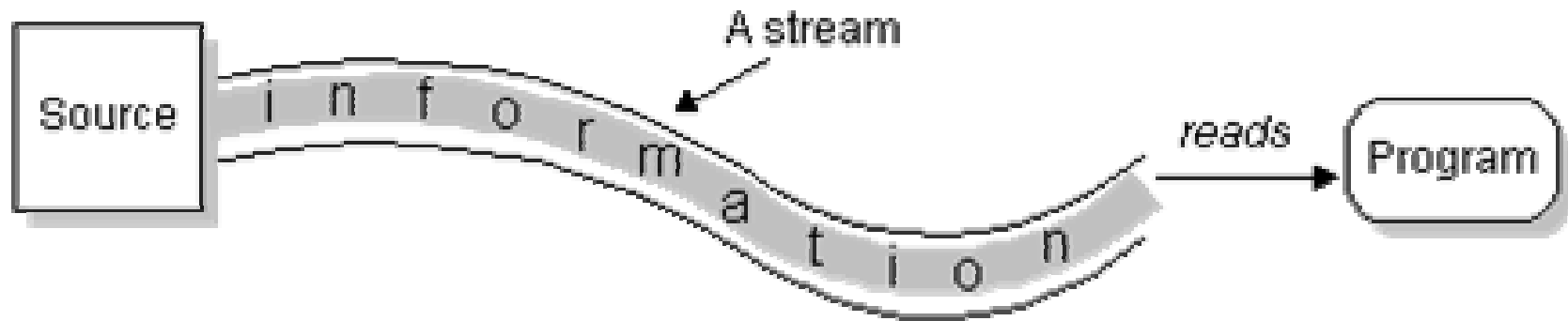
The sources of input and output can vary between a **local file**, a **socket on the network**, a **database**, **variables in memory**, or **another program**.

Even the type of data can vary between **objects**, **characters**, **multimedia**, and others.

Reading Data

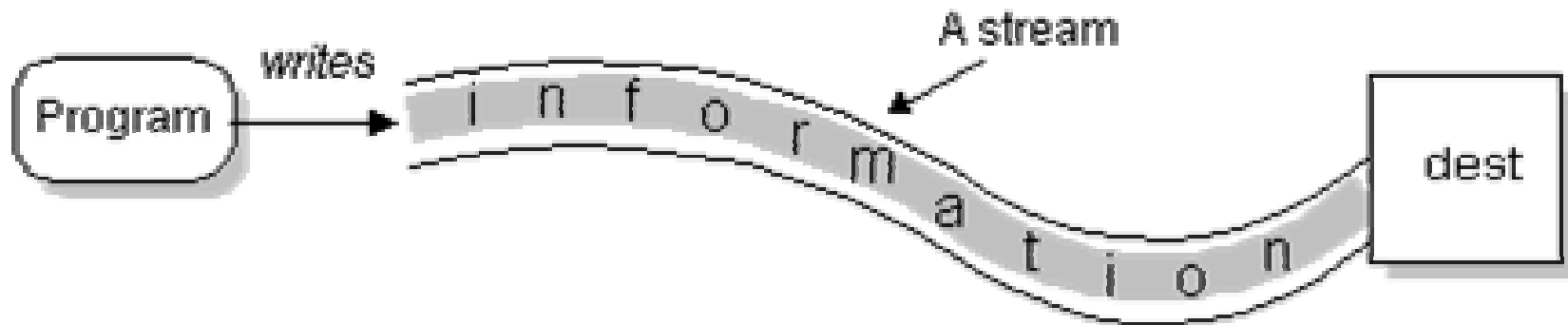
To bring data into a program, a Java program:

- 1) opens a stream to a data source, such as a file or remote socket
- 2) and reads the information serially



Writing Data

On the flip side, a program can open a stream to a data source and write to it in a serial fashion.



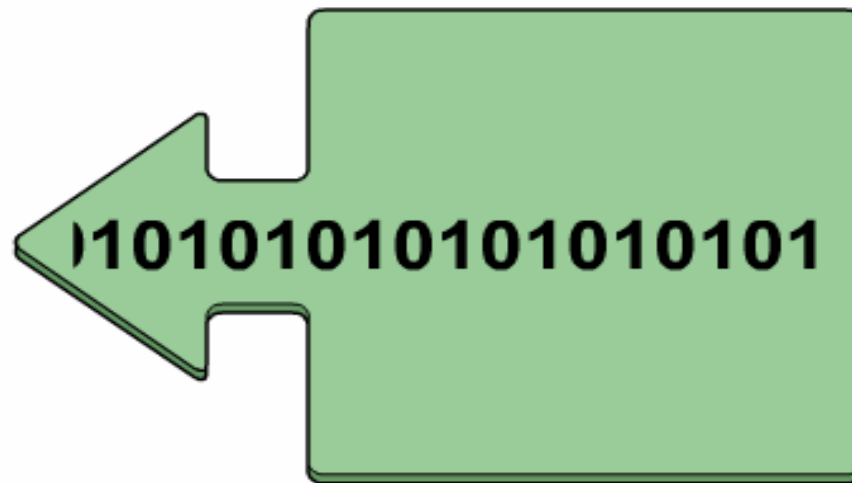
Reading and Writing Data

The concept of serially reading from, and writing to different data sources is the same.

For that very reason, once you understand the top level classes the remaining classes are straightforward to work with.

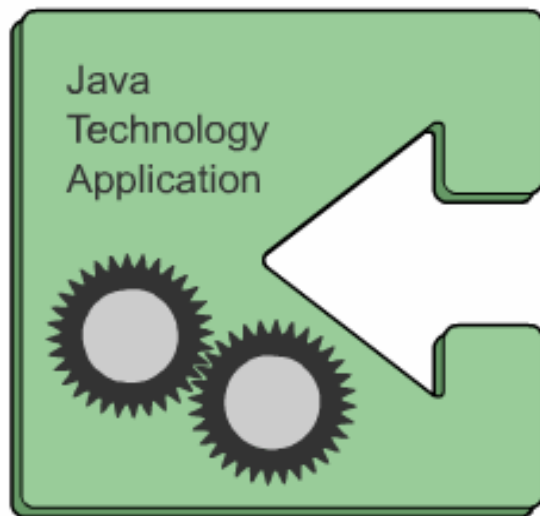
These classes are stored in the `java.io` package.

Streams and Data Sources 1



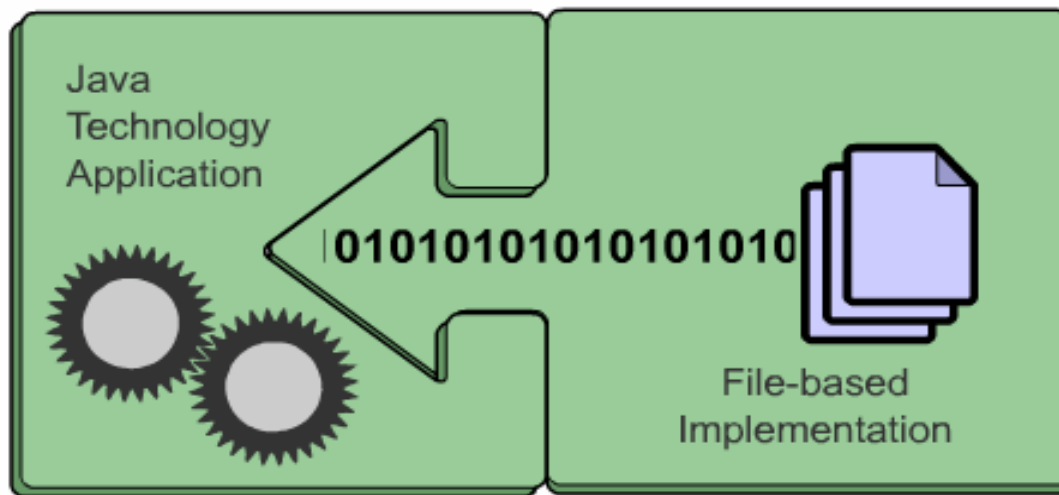
Applications often need to read data to and write data from other sources. Streams provide a means for reading and writing sequences of bytes.

Streams and Data Sources 2



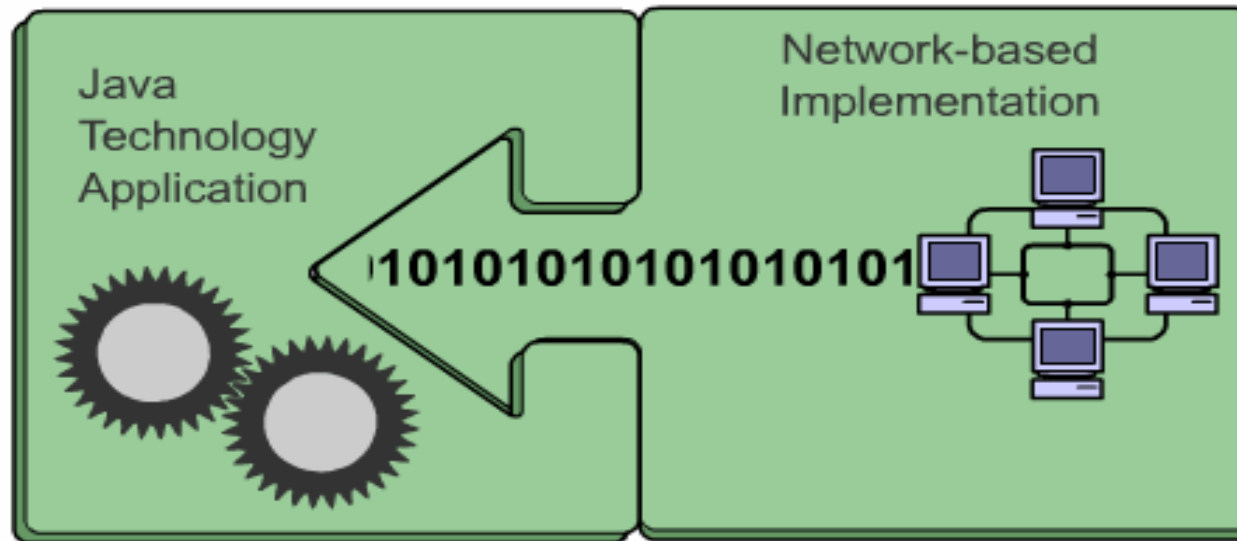
Streams are often used for character values, although this is not the only possibility. The stream concept gives consistent access to any source of data. The source of the data might be a file.

Streams and Data Sources 3



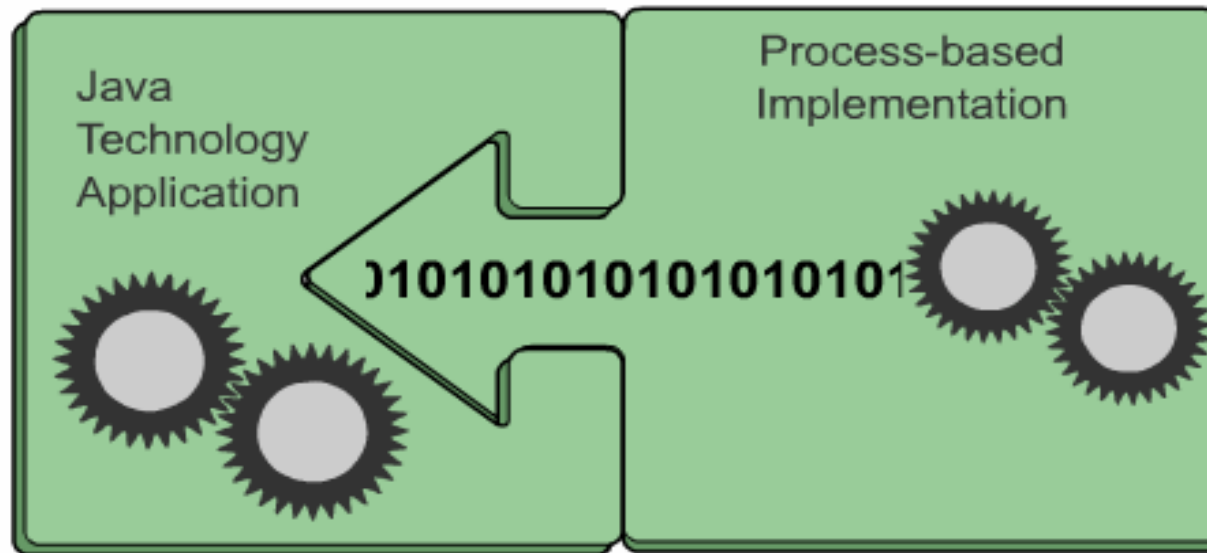
Streams are often used for character values, although this is not the only possibility. The stream concept gives consistent access to any source of data. The source of the data might be a file.

Streams and Data Sources 4



The source of this data might be a network.

Streams and Data Sources 5



The source of this data might even be another process.

Reading and Writing Algorithms

No matter where the data is coming from or going to and no matter what its type, the algorithms for **sequentially reading** and **writing data** are basically the same:

Reading:

```
open a stream
while more information
    read information
close the stream
```

Writing:

```
open a stream
while more information
    write information
close the stream
```

Example: Reading Text from File

```
try {
    BufferedReader in = new BufferedReader(new
                                           FileReader("file"));

    String str;
    while ((str = in.readLine()) != null) {
        process(str);
    }
    in.close();
} catch (IOException e) {
    e.printStackTrace();
}
```


Lab Work: Reading a File

- 1) Based on the code snippet write a program to read a file and displays the content to the console.

Stream Types

There are two categories of streams:

- 1) 8-bit `byte` streams
- 2) 16-bit Unicode `character` streams

Prior to JDK 1.1, the input and output classes (mostly found in the `java.io` package) only supported 8-bit `byte` streams.

The concept of 16-bit Unicode `character` streams was introduced in JDK 1.1.

Stream Support

Support for `byte` streams are provided by:

- 1) `java.io.InputStream` **abstract** class
- 2) `java.io.OutputStream` **abstract** class
- 3) and their subclasses.

While the support for `character` streams are provided by:

- 1) `java.io.Reader` **abstract** class
- 2) `java.io.Writer` **abstract** class
- 3) and their subclasses.

Character versus Byte 1

Most of the functionality available for `byte` streams is also provided for `character` streams.

Methods for character streams generally accept parameters of data type `char` while methods for byte streams accept `byte`.

The names of the methods in both sets of classes are almost identical except for the **suffix**

- 1) character-stream classes end with the suffix **Reader** or **Writer**
- 2) byte-stream classes end with the suffix **InputStream** and **OutputStream**

Character versus Byte 2

For example:

1) to read files using `character` streams you would use

`java.io.FileReader` class.

2) to read files using `byte` streams you would use

`java.io.FileInputStream`.

Unless you are working with binary data, such as **image** and **sound** files, you should use **readers** and **writers** (`character` streams) to read and write the data.

Why?

Character versus Byte 3

`Character` streams are always preferred to `byte` streams when reading and writing information because:

- 1) They can handle any character in the Unicode character set (while the byte streams are limited to ISO-Latin-1 8-bit bytes).
- 2) They are easier to internationalize because they are not dependent upon a specific character encoding.
- 3) They use buffering techniques internally and are therefore potentially much more efficient than byte streams.

I/O Streams Organization

`java.io` is a large collection of classes, consisting of over 50 classes.

For the purpose of understanding the relationships that exist among these classes, they are categorized using the following criteria:

- 1) Data flow
- 2) Function and
- 3) Type of Data they process

Data Flow

Stream classes that channel data into a program are called **input streams**.

Example:

- 1) `FileInputStream`
- 2) `FileReader`
- 3) `ObjectInputStream`
- 4) `PipedInputStream`
- 5) etc.

Stream classes that channel data out of a program are called **output streams**.

Example:

- 1) `FileOutputStream`
- 2) `FileWriter`
- 3) `ObjectOutputStream`
- 4) `PipedOutputStream`
- 5) etc.

Function

Streams can also be grouped by the function they perform.

There are two categories:

- 1) **Node** or **Data sink Streams** - nature of the resource at the other end of the stream, For example,
 - a) `FileInputStream` reads byte data from a **file**,
 - b) `PipedWriter` writes `character` data to a **pipe** (Thread)

- 2) **Process** or **Filter Streams** - type of processing performed on the contents of the stream,
For example,
 - a) `BufferedReader` to buffer reading to reduce disk/network access
 - b) `ObjectOutputStream` for object serialization

Data Sink Streams

<code>CharArrayReader,</code> <code>CharArrayWriter</code>	For reading from or writing to character buffers in memory
<code>FileReader, FileWriter</code>	For reading from or writing to files
<code>PipedReader,</code> <code>PipedWriter</code>	Used to forward the output of one thread as the input to another thread
<code>StringReader</code> <code>StringWriter</code>	For reading from or writing to strings in memory
<code>ByteArrayInputStream</code> <code>ByteArrayOutputStream</code>	For reading from or writing to byte buffers in memory
<code>FileInputStream,</code> <code>FileOutputStream</code>	For reading from or writing bytes to files
<code>PipedInputStream,</code> <code>PipedOutputStream</code>	Used to forward the output of one thread as the input to another thread

Example: Data Sink Streams

Displays contents of a file.

```
import java.io.*;

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

        FileReader fr = new FileReader(args[0]);
        PrintWriter pw = new PrintWriter(System.out,
                                           true);

        char c[] = new char[4096];
        int read = 0;
        while ((read = fr.read(c)) != -1)
            pw.write(c, 0, read);
        fr.close(); pw.close();
    }
}
```

Filter Streams

<code>BufferedReader</code> , <code>BufferedWriter</code>	For buffered reading/writing to reduce disk/network access for more efficiency
<code>InputStreamReader</code> , <code>OutputStreamWriter</code>	Provide a bridge between byte and character streams
<code>SequenceInputStream</code>	Concatenates multiple input streams.
<code>ObjectInputStream</code> , <code>ObjectOutputStream</code>	Use for object serialization.
<code>DataInputStream</code> , <code>DataOutputStream</code>	For reading/writing raw bytes to Java native data types.
<code>PushbackReader</code>	Allows to "peek" ahead in a stream by one character.
<code>LineNumberReader</code>	For reading while keep tracking of the line number.

Example: Filter Streams

Displays contents of many files

```
import java.io.*;
class cat {
    public static void main (String args[]) {
        String thisLine;
        for (int i=0; i < args.length; i++) {
            try {
                BufferedReader br = new BufferedReader(new
                    FileReader(args[i]));
                while ((thisLine = br.readLine()) != null) {
                    System.out.println(thisLine);
                }
            } catch (IOException e) {
                System.err.println("Error: " + e);
            }
        }
    }
}
```

Data Type

At the Simplest level, the `java.io` package can be decomposed into classes that process either of two types of data:

- 1) Byte Stream
- 2) Character Stream

Fundamental Stream Classes

	Byte Stream	Character Stream
Read Data	<code>InputStream</code>	<code>Reader</code>
Write data	<code>OutputStream</code>	<code>Writer</code>

Byte Stream Classes

They process raw bytes.

They come in two basic forms:

1) InputStreams – channel `byte` data into the program

Example:

```
java.io.FileInputStream
```

2) OutputStreams – channel `byte` data from the program

Example:

```
java.io.FileOutputStream
```

Byte-stream classes end with the suffix `InputStream` and `OutputStream`

Byte-Stream Parent Classes

`InputStream` and `OutputStream` are the **abstract** parent classes for byte-stream based classes in the `java.io` package.

Usage:

- 1) `InputStream` classes are used to read 8-bit byte streams and
- 2) `OutputStream` classes are used to write to 8-bit byte streams.

Methods for reading and writing to streams:

```
int read()
int read(byte[] b)
int read(byte[] b, int offset, int length)
int write(int b)
int write(byte[] b)
int write(byte[] b, int offset, int length)
```

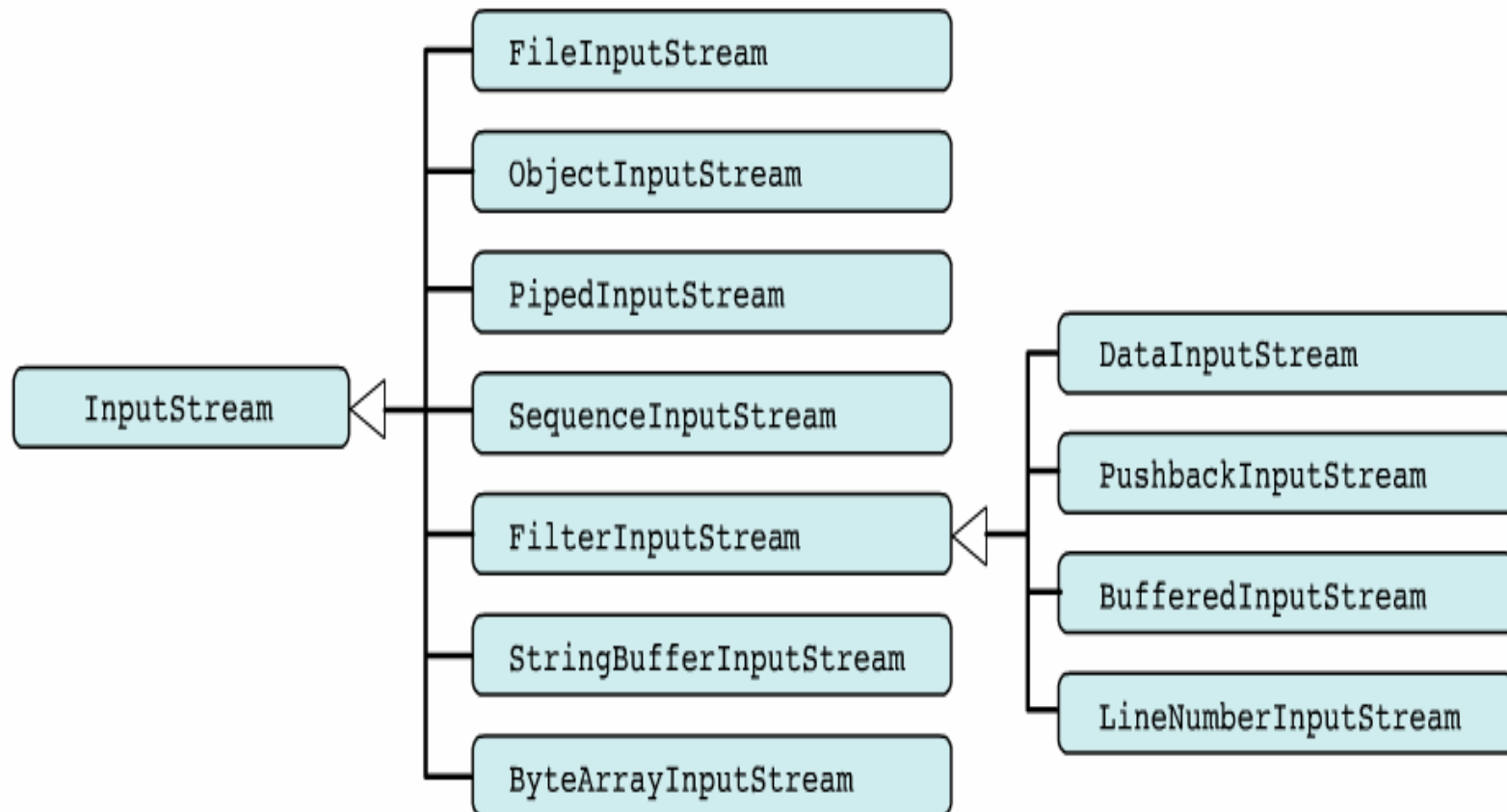

InputStream

This abstract class provides the core methods used to read bytes from an input node.

The methods are:

```
int read()
int read(byte[] b)
int read(byte[] b, int offset, int length)
void close()
int available()
long skip( long l)
boolean markSupported()
void mark( int i)
void reset()
```

InputStream Hierarchy



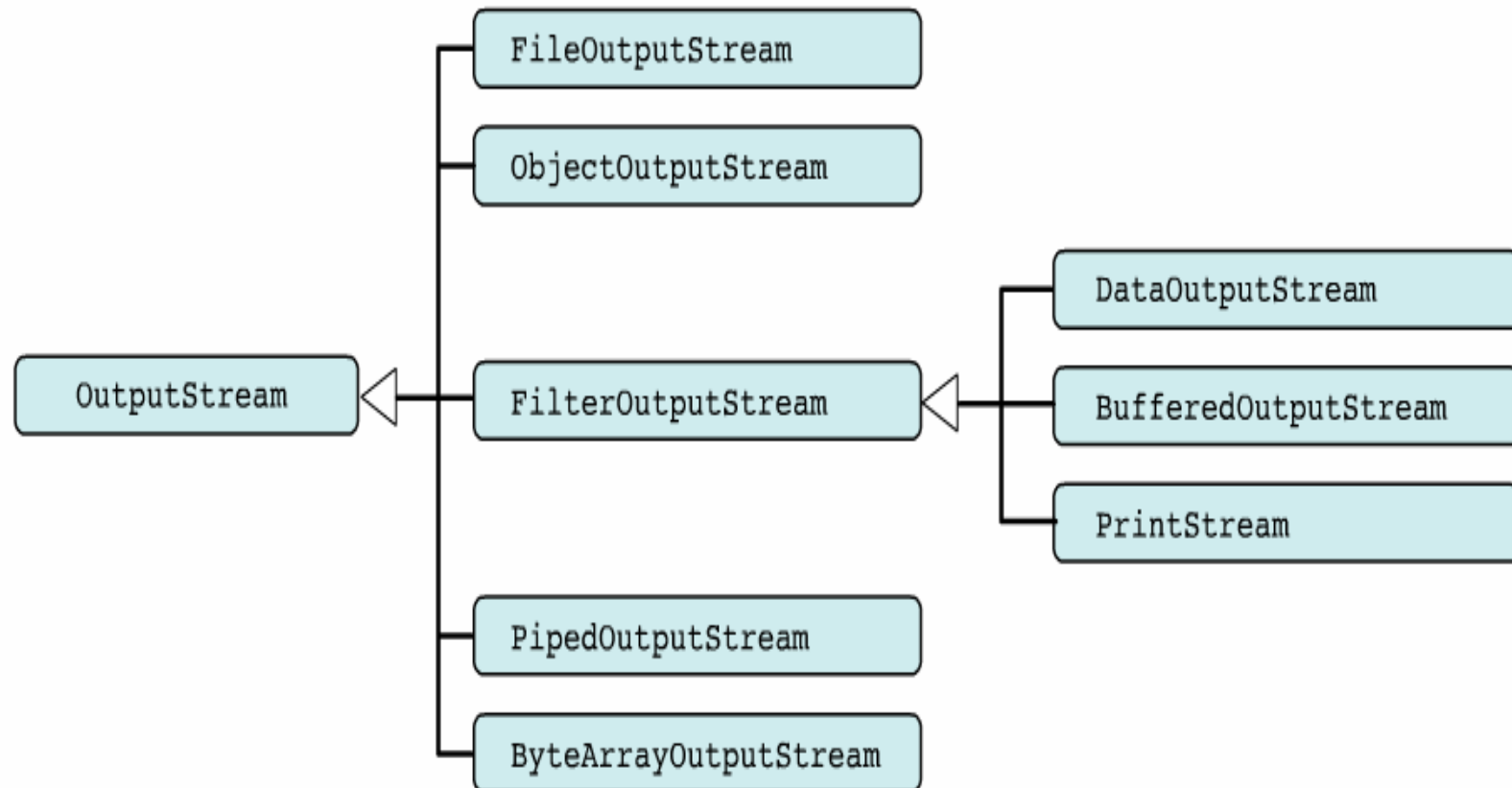
OutputStream

This abstract class provides the core methods used to write bytes to an output node.

The methods are:

```
int write(int b)
int write(byte[] b)
int write(byte[] b, int offset, int length)
void close()
void flush()
```

OutputStream Hierarchy



Data Sink Byte-Stream

Classes that take **byte input** from different types of nodes (file, pipe, byte array, etc)

Example:

```
FileInputStream  
PipedInputStream
```

Classes that send **byte output** to different types of node (file, pipe, byte array, etc)

Example:

```
FileOutputStream  
PipedOutputStream
```

Example: Data Sink Byte-Stream

1) FileInputStream/FileOutputStream

Usage: is meant for reading/writing streams of raw bytes such as image data to/from files

```
File f = new File("mydata.txt");
FileInputStream fis = new FileInputStream(f);
BufferedInputStream bis = new
BufferedInputStream(fis);
DataInputStream dis = new DataInputStream(bis);
```

2) ByteArrayInputStream/ByteArrayOutputStream

Usage: are useful for holding data when the underlying data type is irrelevant to the purpose of the application

```
byte[] c
...
ByteArrayInputStream r = new ByteArrayInputStream(c);
```

Example: Data Sink Byte-Stream

3) `PipedInputStream/PipedOutputStream`

Usage: read from or write to pipes. Often used to exchange data between threads.

```
PipedInputStream pi = new PipedInputStream();  
PipedOutputStream po = new PipedOutputStream(pi);
```

Or

```
PipedInputStream pi = new PipedInputStream();  
PipedOutputStream po = new PipedOutputStream(pi);
```

Or

```
PipedInputStream pi = new PipedInputStream();  
PipedOutputStream po = new PipedOutputStream();  
pi.connect(po);
```

Example: FileInputStream 1

```
import java.io.*;

class FileInputStreamDemo {
    public static void main(String args[]) throws
        Exception {
        int size;
        InputStream f =
            new FileInputStream("FileInputStreamDemo.java");

        System.out.print("Total Available Bytes: " )
        System.out.println((size = f.available()));
        int n = size/40;
        System.out.println("First " + n +
            " bytes of the file one read() at a time");
        for (int i=0; i < n; i++) {
            System.out.print((char) f.read());
        }
    }
}
```


Example: FileInputStream 2

```
System.out.println("\nStill Available: " +
                    f.available());
System.out.println("Reading the next " + n +
                    " with one read(b[])");
byte b[] = new byte[n];
if (f.read(b) != n) {
    System.err.println("couldn't read " + n + "
                        bytes.");
}
System.out.println(new String(b, 0, n));
System.out.println("\nStill Available: " + (size =
                    f.available()));
System.out.println("Skipping half of remaining
                    bytes with skip()");
f.skip(size/2);
System.out.println("Still Available: " +
                    f.available());
```

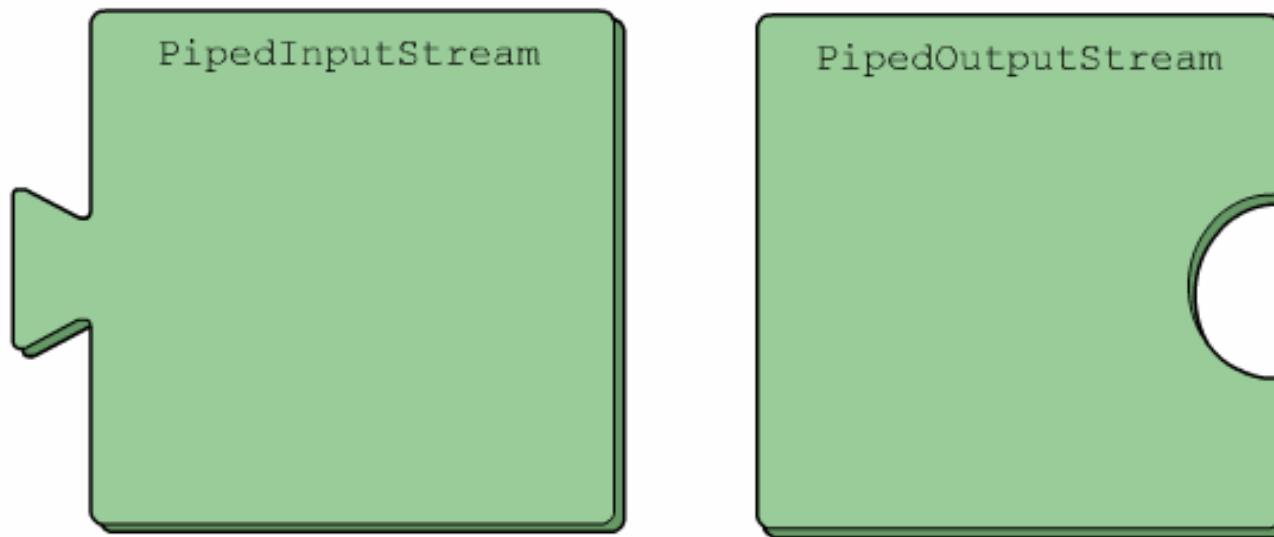
Example: FileInputStream 3

```
System.out.println("Reading " + n/2 + " into the  
                        end of array");  
if (f.read(b, n/2, n/2) != n/2) {  
    System.err.println("couldn't read " + n/2 + "  
                        bytes.");  
}  
System.out.println(new String(b, 0, b.length));  
System.out.println("\nStill Available: " +  
                    f.available());  
f.close();  
}  
}
```

Lab Work: Reading a File

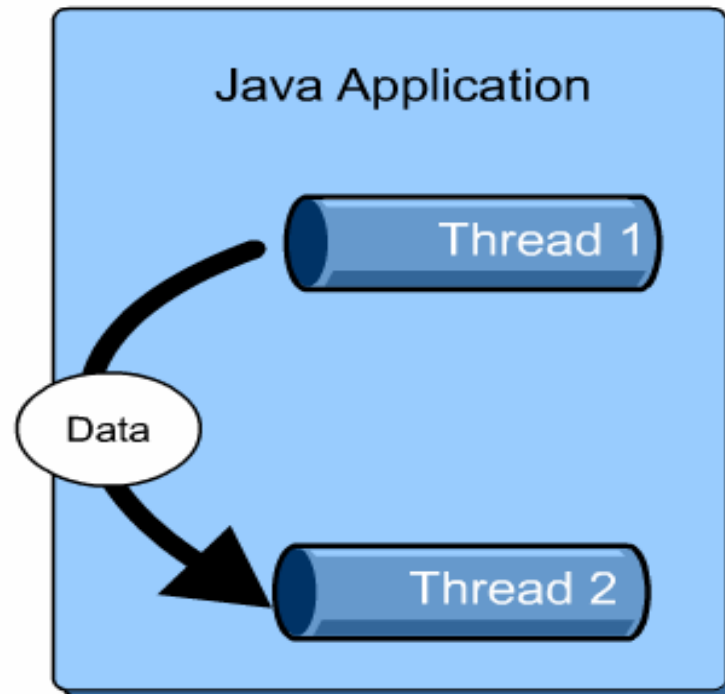
1) Write a program that reads an MP3 file.

PipedInput/Output Stream 1



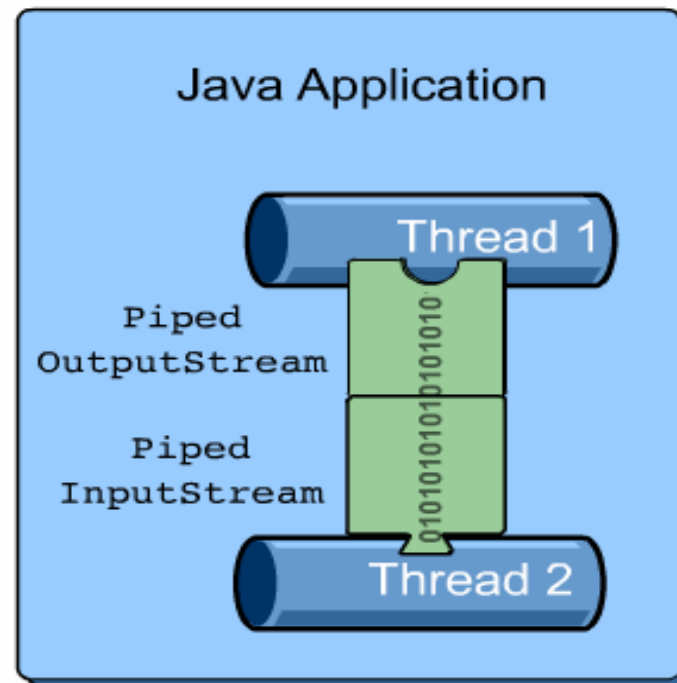
The `PipedInputStream` and `PipedOutputStream` classes are designed to be used as a pair. They allow a mechanism for communication among threads, although they have other uses, too.

PipedInput/Output Stream 2



To illustrate how the piped streams can be used, suppose you have an application which must transfer data between two separate threads.

PipedInput/Output Stream 3



The `PipedInputStream`/`PipedOutputStream` are created as a pair, so that what is written into the output stream can be read from the input stream. The effect is much like a pipeline that carries data from one point in an application to another.

Example: Piped Stream 1

Shows how to exchange data between two threads

```
import java.io.*;

class ReadThread extends Thread implements Runnable {
    InputStream pi = null;
    OutputStream po = null;
    String process = null;
    ReadThread( String process, InputStream pi,
                OutputStream po) {

        this.pi = pi;
        this.po = po;
        this.process = process;
    }
}
```

Example: Piped Stream 2

```
public void run() {  
    int ch;  
    byte[] buffer = new byte[512];  
    int bytes_read;  
    try {  
        for(;;) {  
            bytes_read = pi.read(buffer);  
            if (bytes_read == -1) { return; }  
            po.write(buffer, 0, bytes_read);  
        }  
    } catch (Exception e) {  
        e.printStackTrace();  
    } finally { }  
}  
  
}
```


Example: Piped Stream 3

```
class SystemStream {
    public static void main( String [] args) {
        try {
            int ch;
            while (true) {
                PipedInputStream in = new PipedInputStream();
                PipedOutputStream out = new PipedOutputStream(
                                                            in );

                FileOutputStream writeOut = new
                    FileOutputStream("out");

                ReadThread rt = new ReadThread("reader",
                                                System.in, out );
                ReadThread wt = new ReadThread("writer", in,
                                                System.out );

                rt.start();
                wt.start();
            }
        }
    }
}
```

Example: Piped Stream 4

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

Filter Byte-Stream

They convert bytes to primitive data

They write primitive data

Example:

`BufferedInputStream/BufferedOutputStream`

`DataInputStream/DataOutputStream`

Example: Filter Byte-Stream 1

1) `BufferedInputStream/BufferedOutputStream`

Usage:

These classes buffer data emanating from an `InputStream` object or in route to an `OutputStream` object.

Benefits:

- a) **Improved performance** - Buffered streams cache data to reduce the need to access slower transmission media.
- b) **Simplicity** - Buffered streams manage the data cache themselves, so you do not have to.

```
File f = new File("mydata.txt");  
FileInputStream fis = new FileInputStream(f);  
BufferedInputStream bis = new BufferedInputStream(fis);  
DataInputStream dis = new DataInputStream(bis);
```

Example: Filter Byte-Stream 2

2) `DataInputStream/DataOutputStream`

Usage:

These classes transform bytes emanating from an `InputStream` type into primitives (such as `int`, `long`, or `double`) or primitives in route to an `OutputStream` type into bytes.

Attach a `DataInputStream` filter to an `InputStream` object when you need to read primitives from a stream.

Attach a `DataOutputStream` filter to an `OutputStream` object when you need to write primitives to a stream.

```
File f = new File("mydata.txt");  
FileInputStream fis = new FileInputStream(f);  
BufferedInputStream bis = new BufferedInputStream(fis);  
DataInputStream dis = new DataInputStream(bis);
```

Example: BufferedInputStream 1

```
import java.io.*;

class BufferedInputStreamDemo {
    public static void main(String args[]) throws
        IOException {
        String s = "This is a &copy; copyright symbol " +

            "but this is &copy; not.\n";
        byte buf[] = s.getBytes();
        ByteArrayInputStream in = new

            ByteArrayInputStream(buf);
        BufferedInputStream f = new BufferedInputStream(in);
        int c;
        boolean marked = false;
```

Example: BufferedInputStream 2

```
while ((c = f.read()) != -1) {  
    switch(c) {  
        case '&':  
            if (!marked) {  
                f.mark(32);  
                marked = true;  
            } else {  
                marked = false;  
            }  
            break;  
        case ';':  
            if (marked) {  
                marked = false;  
                System.out.print("(c)");  
            } else
```

Example: BufferedInputStream 3

```
        System.out.print ((char) c);  
        break;  
case ' ':  
    if (marked) {  
        marked = false;  
        f.reset();  
        System.out.print("&");  
    } else  
        System.out.print ((char) c);  
    break;  
default:  
    if (!marked)  
        System.out.print ((char) c);  
    break;  
}}}}
```


Serialization

Serialization is a process of writing an object to a byte stream.

Writing an Object

```
FileOutputStream out = new FileOutputStream("tmp");  
ObjectOutput objOut = new ObjectOutputStream(out);  
objOut.writeObject(Color.red);
```

Reading an Object

```
FileInputStream in = new FileInputStream("tmp");  
ObjectInputStream objIn = new ObjectInputStream(in);  
Color c = (Color)objIn.readObject();
```

Object Serialization 1

Provides a way for objects to be written as a stream of bytes and then later recreated from that stream of bytes.

The job of an `ObjectInputStream` class is to convert collections of bytes into objects.

Sending an object over a stream was a cumbersome process. How?

Essentially, you had to decompose the object into its constituent parts, sending each to the stream individually, and then reconstruct the object manually at the other end of the stream.

Process is cumbersome. Solution?

Object Serialization 2


The introduction of `new` interface to the `java.io` package, the `Serializable` interface

The `Serializable` interface eliminates the drawbacks of sending objects across streams.

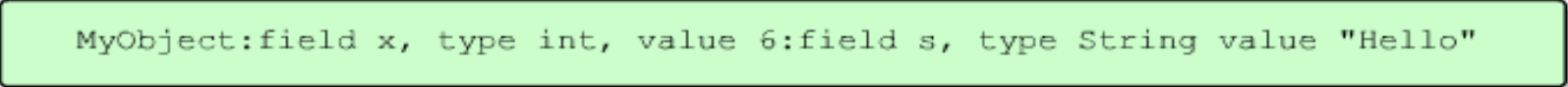
Each object to be sent has to implement this interface

```
import java.io.* ;
class Date implements Serializable {
    int m, d, y ;
    public Date( int m, int d, int y ) {
        this.m = m ; this.d = d ; this.y = y ;
    }
}
```

Object Serialization 3



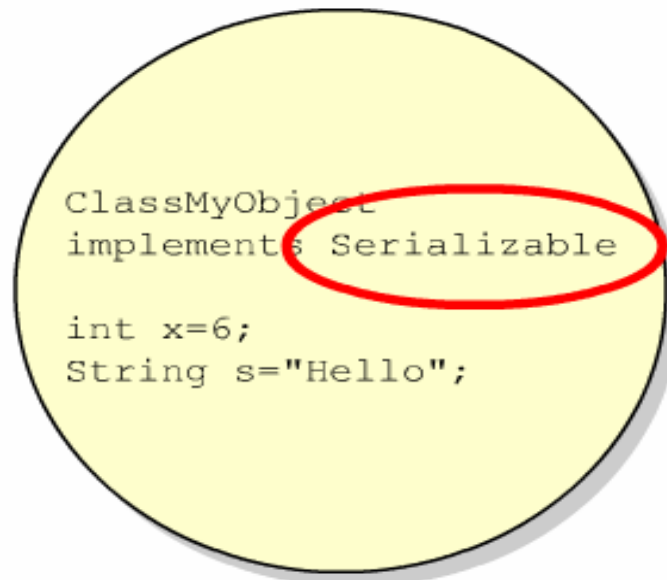
```
ClassMyObject  
implements Serializable  
  
int x=6;  
String s="Hello";
```



```
MyObject:field x, type int, value 6:field s, type String value "Hello"
```

Serialization takes the state (that is the instance variables) of an object and represents them as a sequence of bytes

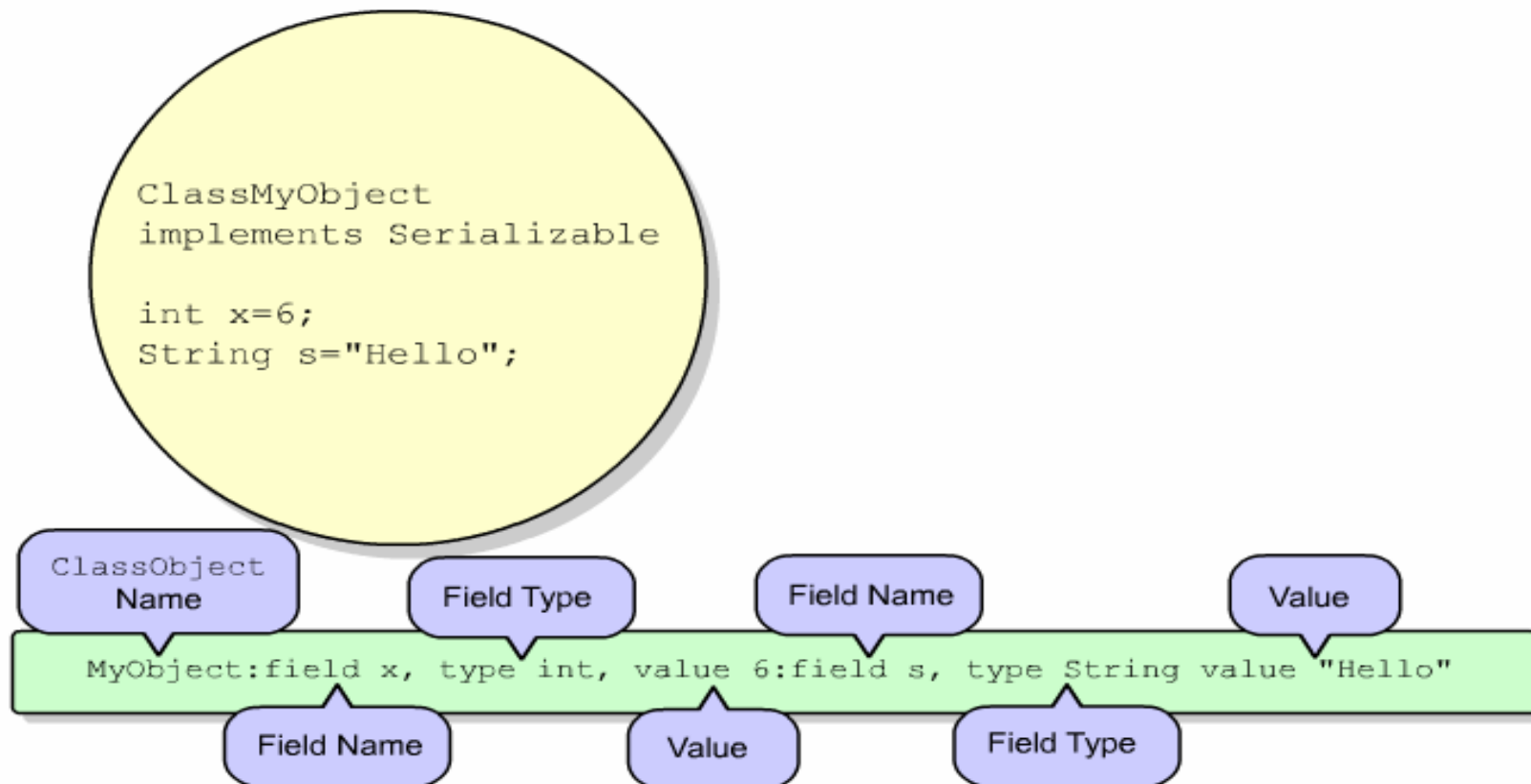
Object Serialization 4



```
MyObject:field x, type int, value 6:field s, type String value "Hello"
```

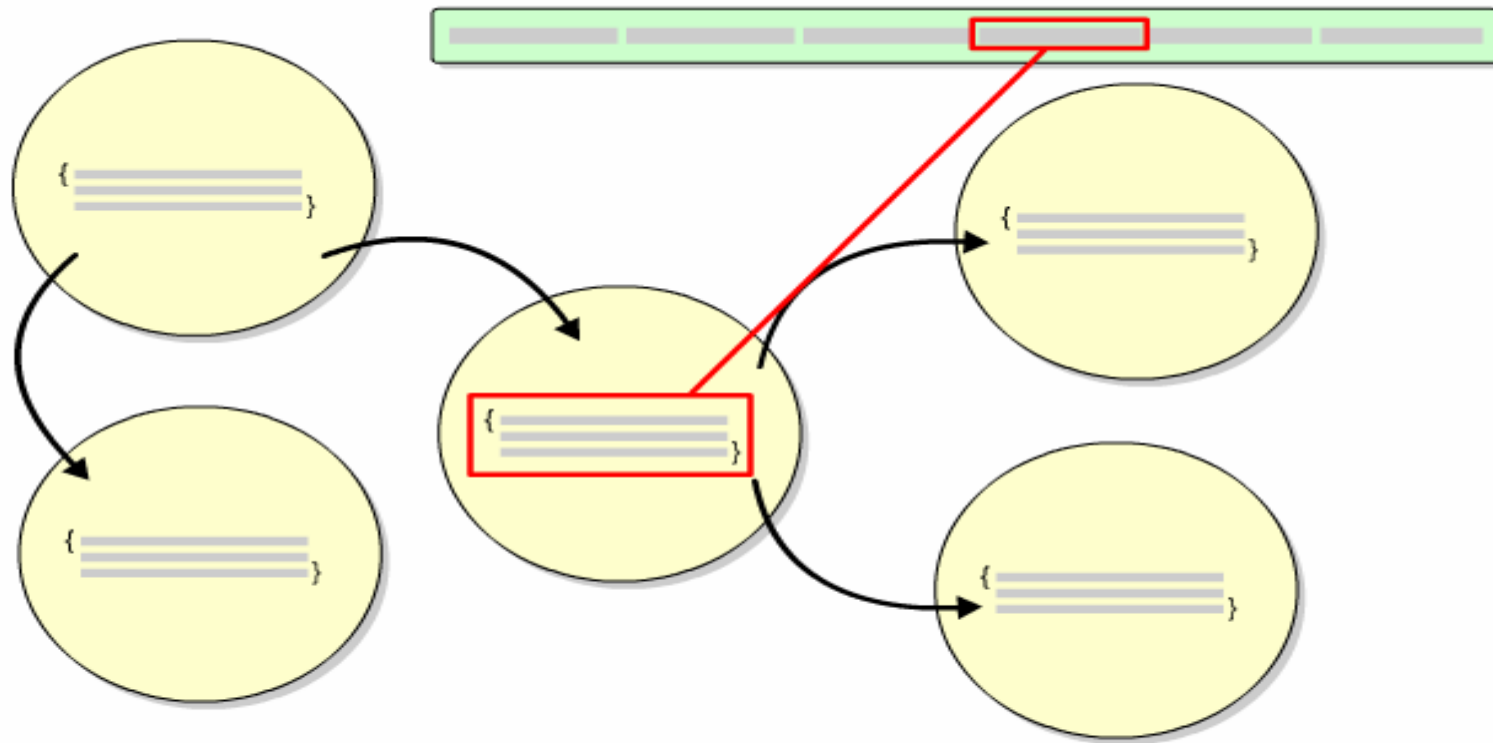
It is important that the class that defines the object declares that the object is serializable. Otherwise, this process does not work.

Object Serialization 5



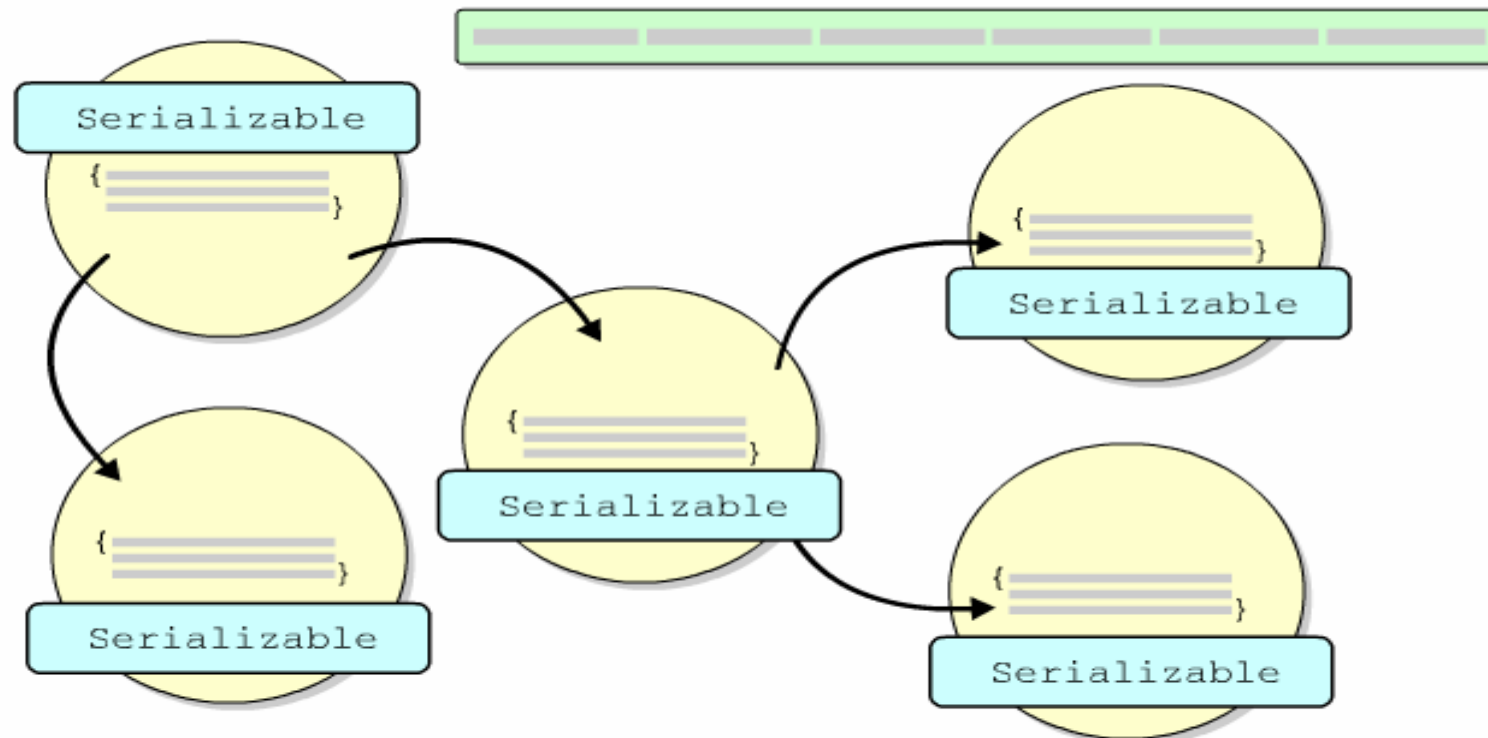
Notice that the serialized form contains information about the name of the class of the object, the names and types of the fields, as well as the values themselves.

Object Serialization 6



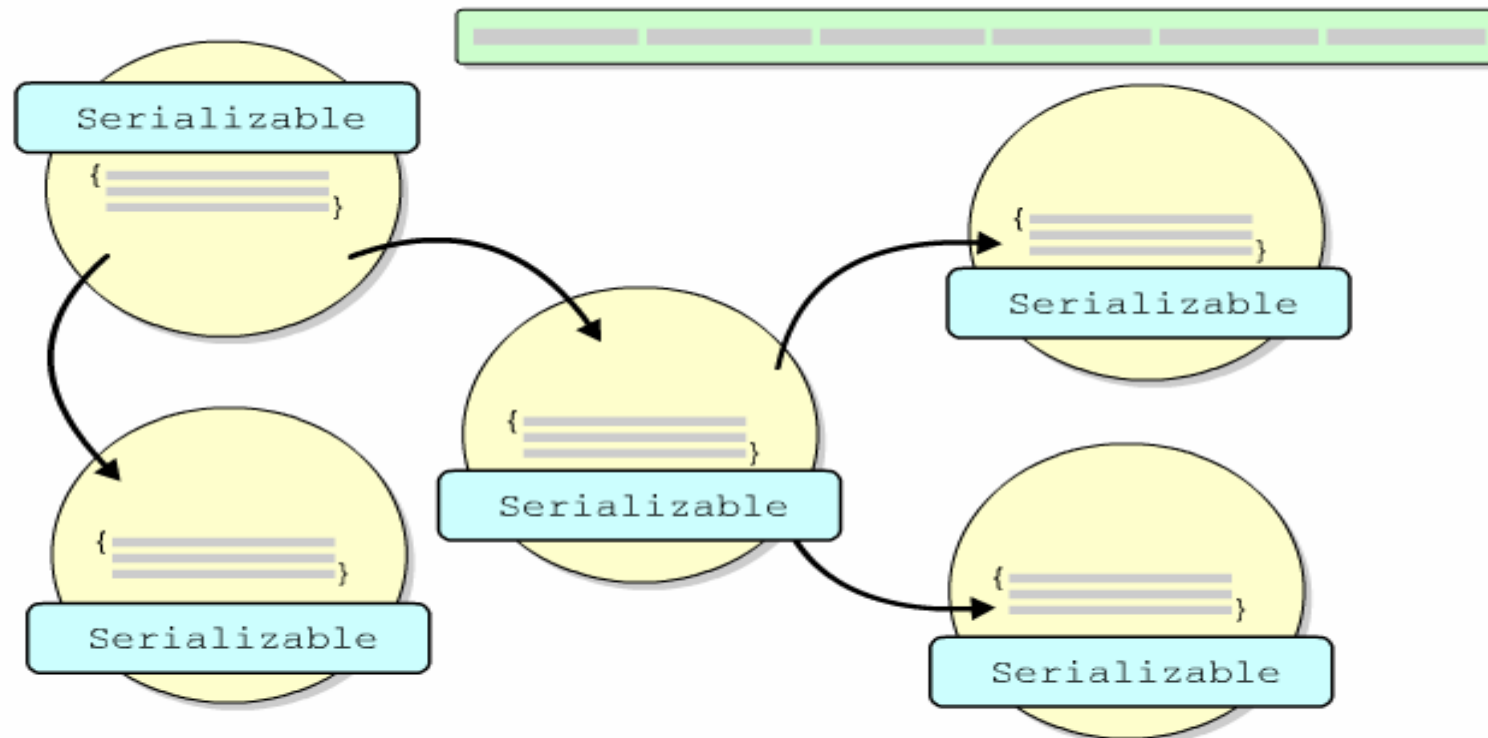
Most objects have references to other objects within them, and sometimes those reference objects also have other references to other objects. All the objects that are reachable through these references form the object graph.

Object Serialization 7



Normally, when an object is serialized, all the objects in the object graph must be serialized. This means that all the objects in the object graph must implement `Serializable` as well. Otherwise, the serialization will fail.

Object Serialization 8



Normally, when an object is serialized, all the objects in the object graph must be serialized. This means that all the objects in the object graph must implement `Serializable` as well. Otherwise, the serialization will fail.

Character Stream Classes

They process 16 bits Unicode characters.

They come in two basic forms:

1) Reader – channel `character` data into the program

Example:

```
java.io.FileReader
```

2) Writer – channel `character` data from the program

Example:

```
java.io.FileWriter
```

Character-stream classes end with the suffix **Reader** and **Writer**

Character Parent Classes

`Reader` and `Writer` are the **abstract** parent classes for byte-stream based classes in the `java.io` package.

Usage:

- 1) `Reader` classes are used to read 16-bit character streams and
- 2) `Writer` classes are used to write to 16-bit character streams.

Methods for reading and writing to streams:

```
int read()
int read(char[] c)
int read(char[] c, int offset, int length)
int write(int c)
int write(char[] c)
int write(char[] c, int offset, int length)
```

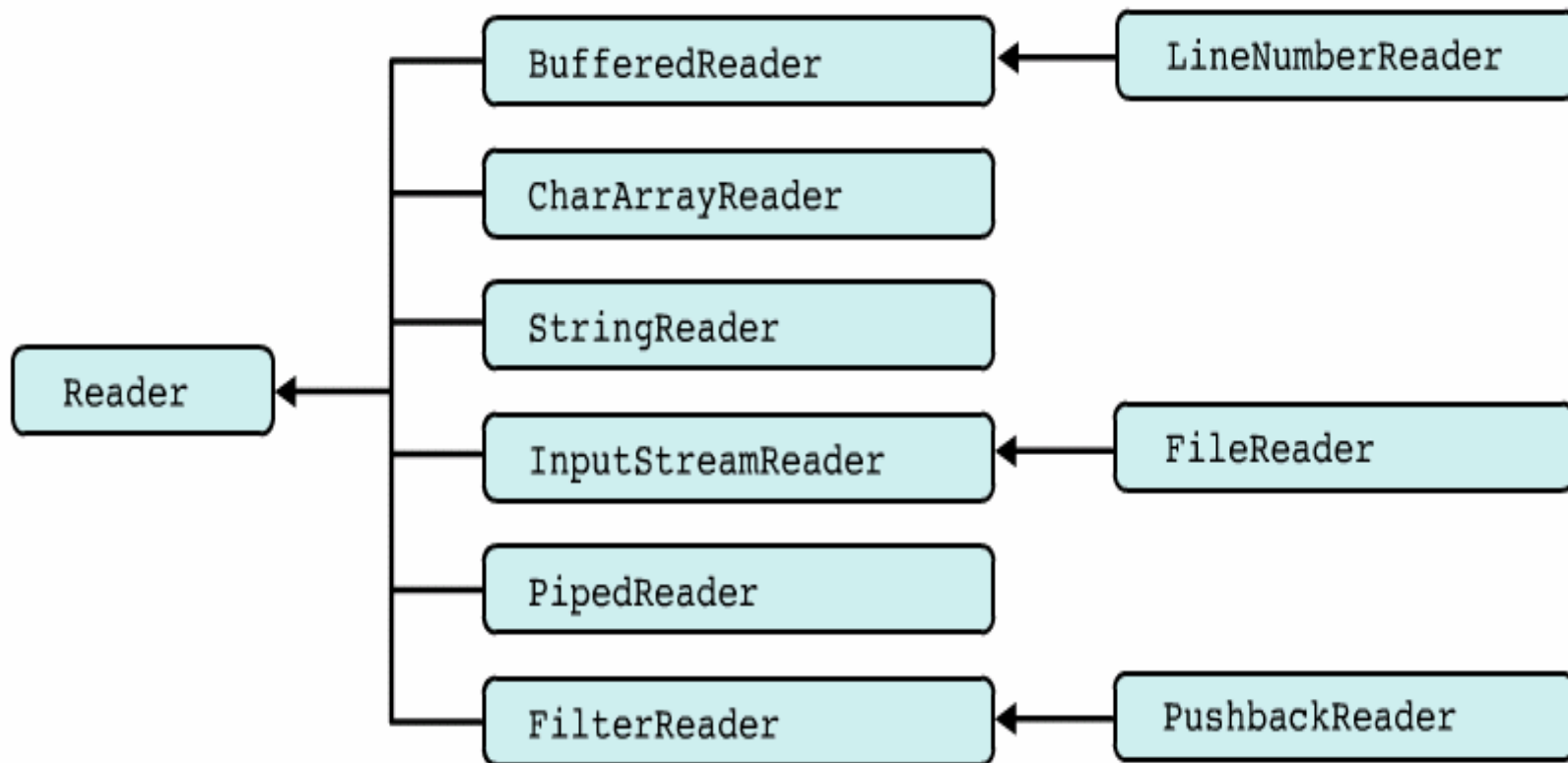
Reader

This abstract class provides the core methods used to read characters from an input node.

The methods are:

```
int read()  
int read(char[] c)  
int read(char[] c, int offset, int length)  
void close()  
long skip( long l)  
boolean markSupported()  
void mark( int i)  
void reset()
```

Reader Hierarchy



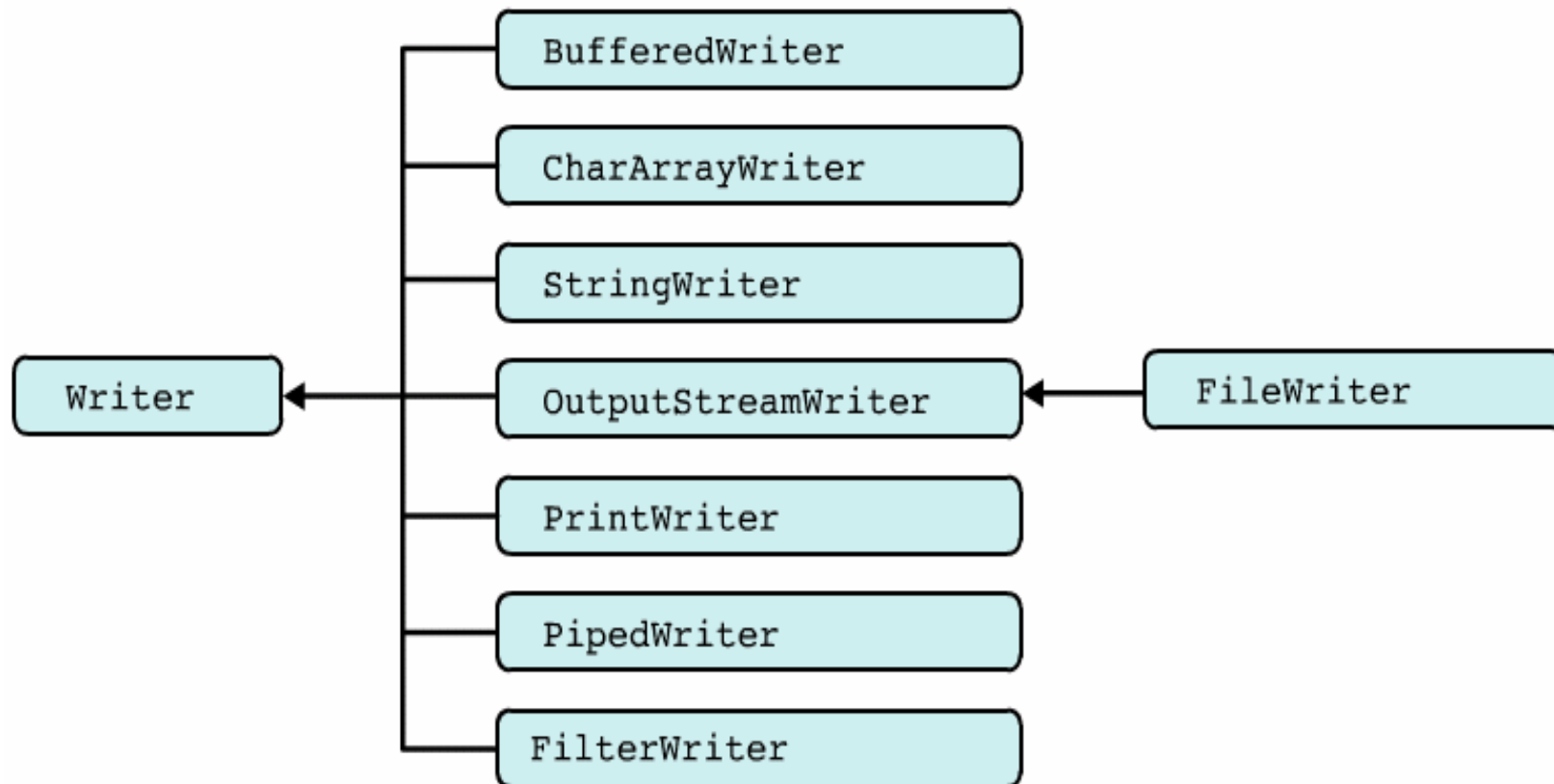
Writer

This abstract class provides the core methods used to write characters to an output node.

The methods are:

```
int write(int b)
int write(char[] c)
int write(char[] c, int offset, int length)
void close()
void flush()
```

Writer Hierarchy



Data Sink Character-Stream

Classes that take **input** from different types of nodes (file, pipe, char array, etc)

Example:

`FileReader`

`PipedReader`

Classes that send **output** to different types of node (file, pipe, char array, etc)

Example:

`FileWriter`

`PipedWriter`

Example: Node Character-Stream

1) `FileReader/FileWriter`

Usage: is meant for reading/writing streams of character data to/from files

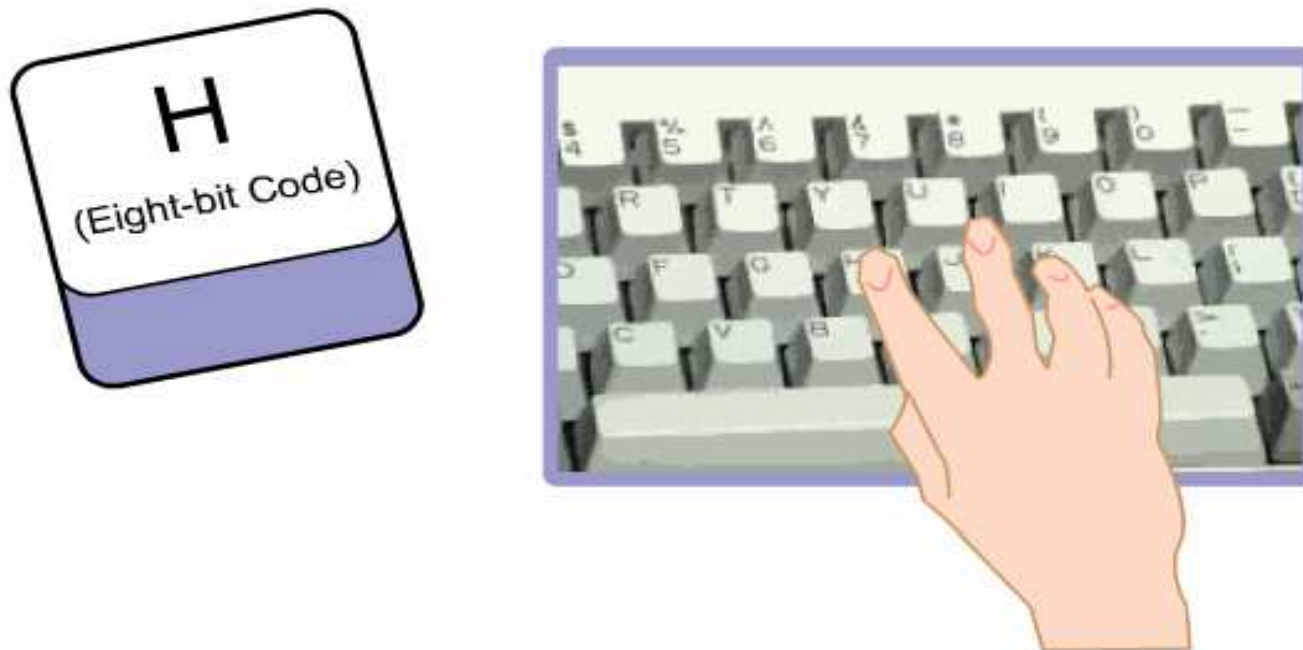
```
File f = new File("mydata.txt");  
FileReader fis = new FileReader(f);
```

2) `CharArrayReader/CharArrayWriter`

Usage: are useful for holding data when the underlying data type is irrelevant to the purpose of the application

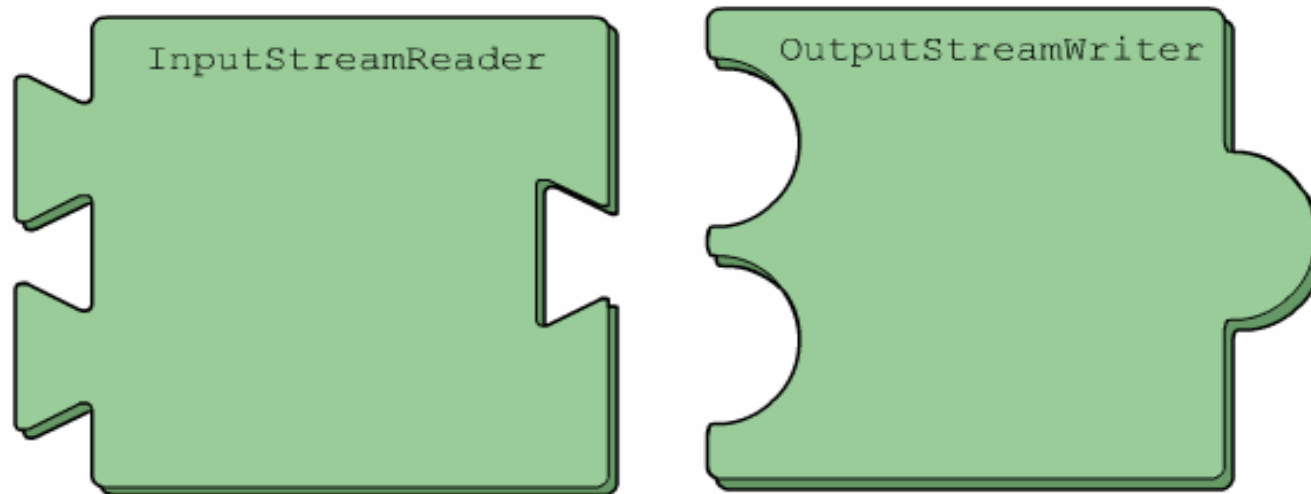
```
char[] c  
...  
CharArrayReader r = new CharArrayReader(c);
```


Bridging Streams 2



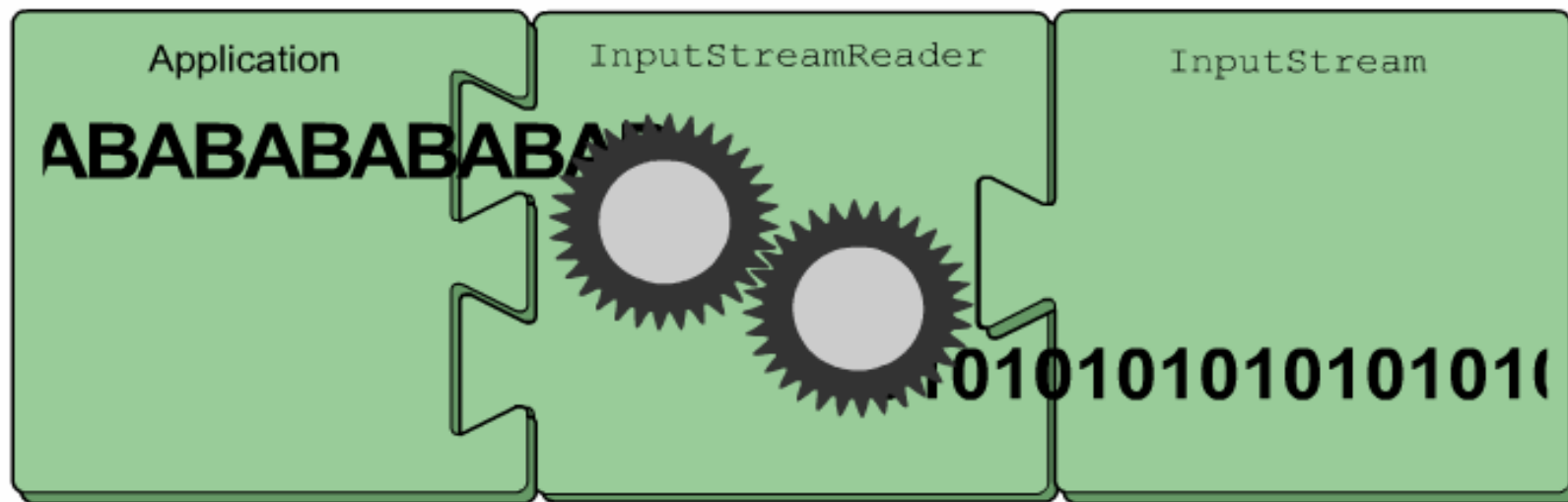
When a key is pressed, most platforms produce an 8bit code to represent a character. There are many different character sets that a platform may use to encode characters. Internally, the JVM uses 16bit Unicode characters.

Bridging Streams 3



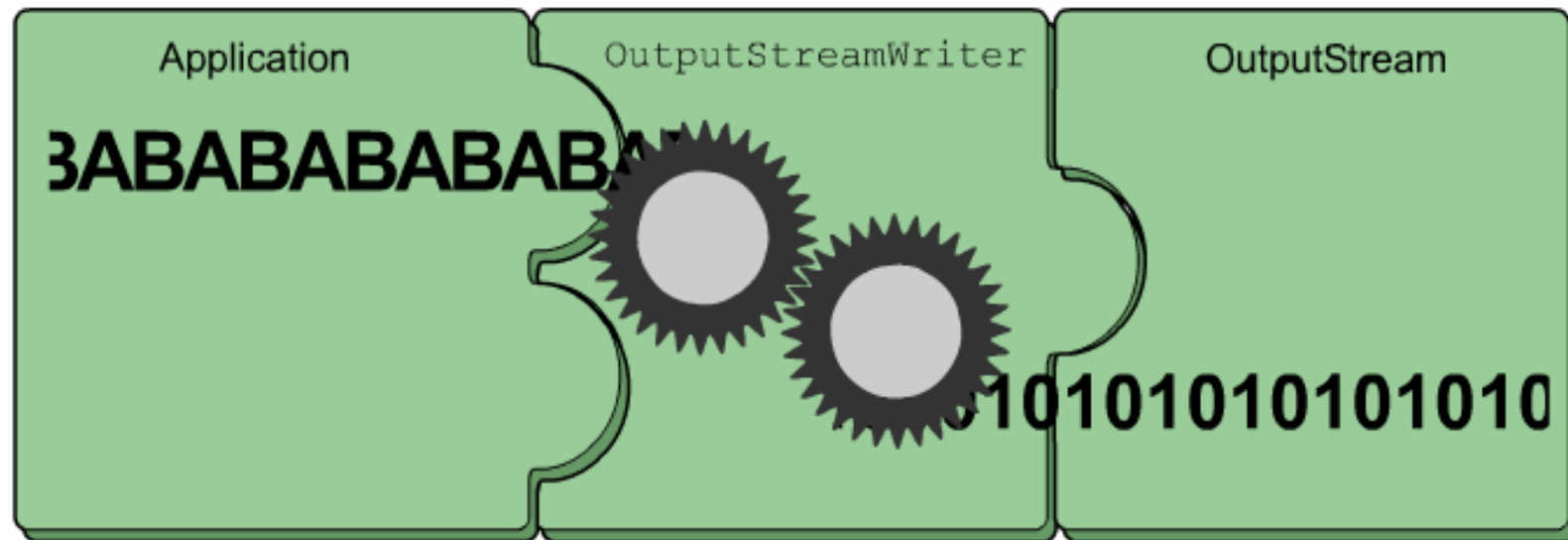
When characters enter or leave the program, they must be converted between these distinct encoding formats. This is the job of the `InputStreamReader` and `OutputStreamWriter` classes.

Bridging Streams 4



This is the `InputStreamReader` object. This class uses a `Reader` type object on the left hand side, but can plug into an `InputStream` type object on the right hand side. This allows it to read data that are encoded in a platform-specific way from an `InputStream` object and convert those data into 16 bit unicode characters that can be read from the `Reader` interface by the client program.

Bridging Streams 5



When an application wants to write data out to the local platform, the `OutputStreamWriter` performs the inverse conversion, taking Unicode characters and mapping them to bytes in the local encoding format.

Summary: Filter Streams

Type	Character Streams	Byte Streams
Buffering	BufferedReader BufferedWriter	BufferedInputStream BufferedOutputStream
Filtering	FilterReader FilterWriter	FilterInputStream FilterOutputStream
Converting between bytes and character	InputStreamReader OutputStreamWriter	
Object serialization		ObjectInputStream ObjectOutputStream
Data conversion		DataInputStream DataOutputStream
Counting	LineNumberReader	LineNumberInputStream
Peeking ahead	PushbackReader	PushbackInputStream
Printing	PrintWriter	PrintStream

Reading Text from Standard Input

```
try {  
    BufferedReader in = new BufferedReader(new  
        InputStreamReader(System.in));  
    String str = "";  
    while (str != null) {  
        System.out.print("> prompt ");  
        str = in.readLine();  
        process(str);  
    }  
} catch (IOException e) {  
}
```


Reading Text from a File

```
try {  
    BufferedReader in = new BufferedReader(new  
        FileReader("infilename"));  
  
    String str;  
    while ((str = in.readLine()) != null) {  
        process(str);  
    }  
    in.close();  
} catch (IOException e) {  
}
```

Writing to a File

```
try {  
    BufferedWriter out = new BufferedWriter(new  
        FileWriter("outfilename"));  
    out.write("aString");  
    out.close();  
} catch (IOException e) {  
  
}
```

Appending to a File

```
try {  
    BufferedWriter out = new BufferedWriter(new  
        FileWriter("filename", true));  
    out.write("aString");  
    out.close();  
} catch (IOException e) {  
}
```

Serializing an Object

```
Object object = new javax.swing.JButton("push");
try {
    // Serialize to a file
    ObjectOutputStream out = new
        ObjectOutputStream(new
            FileOutputStream("filename.ser"));
    out.writeObject(object);
    out.close();
    // Serialize to a byte array
    ByteArrayOutputStream bos = new
        ByteArrayOutputStream();
    out = new ObjectOutputStream(bos);
    out.writeObject(object);
    out.close();
    // Get the bytes of the serialized object
    byte[] buf = bos.toByteArray();
} catch (IOException e) {
```

Deserializing an Object

```
try {
    // Deserialize from a file
    File file = new File("filename.ser");
    ObjectInputStream in = new
        ObjectInputStream(new
            FileInputStream(file));
    // Deserialize the object
    javax.swing.JButton button =
        (javax.swing.JButton) in.readObject();
    in.close();

} catch (ClassNotFoundException e) {

} catch (IOException e) {

}
```

Lab Work: Reading and Writing

Based on the code snippets, write a program that

- 1) reads text from standard input
- 2) copies the content of one file and writes to another file
- 3) appends `"This is emacao training"` to the end a text file
- 4) a program that joins series of files together

Stream Chaining

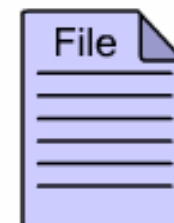
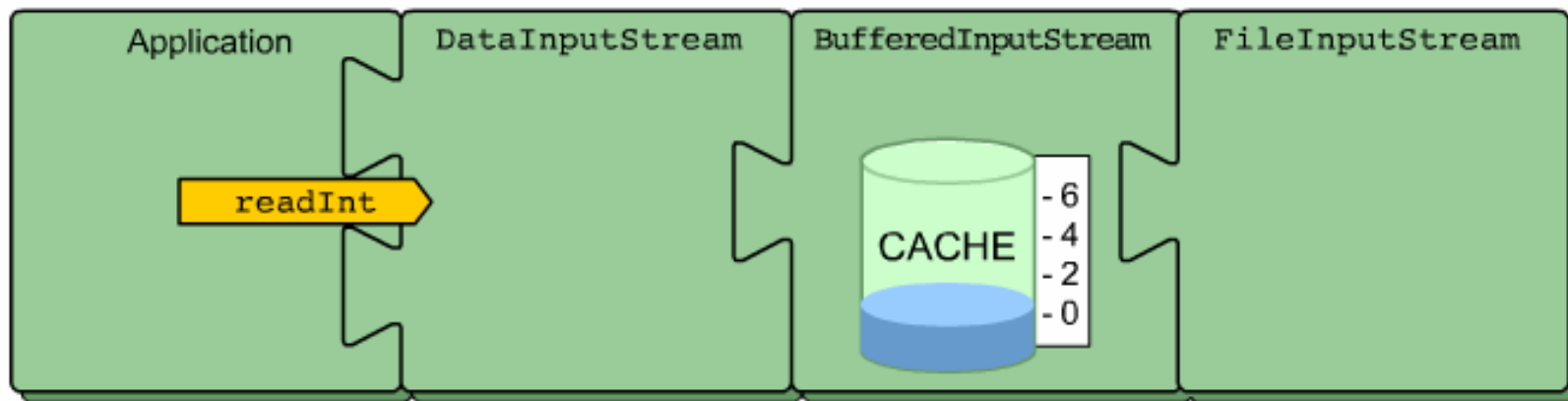
Stream chaining is a way of connecting several stream classes together to get the data in the form required.

Each class performs a specific task on the data and forwards it to the next class in the chain.

The output produced by one component becomes the input to the next component in the chain.

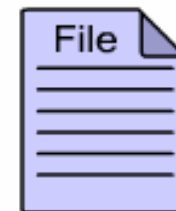
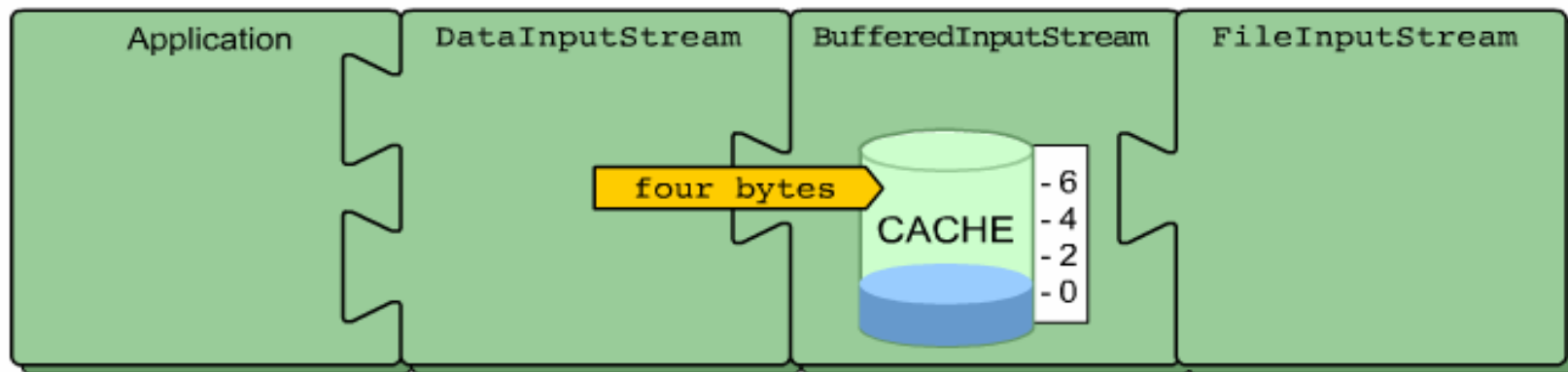
Consider this.

Example: Stream Chaining 1



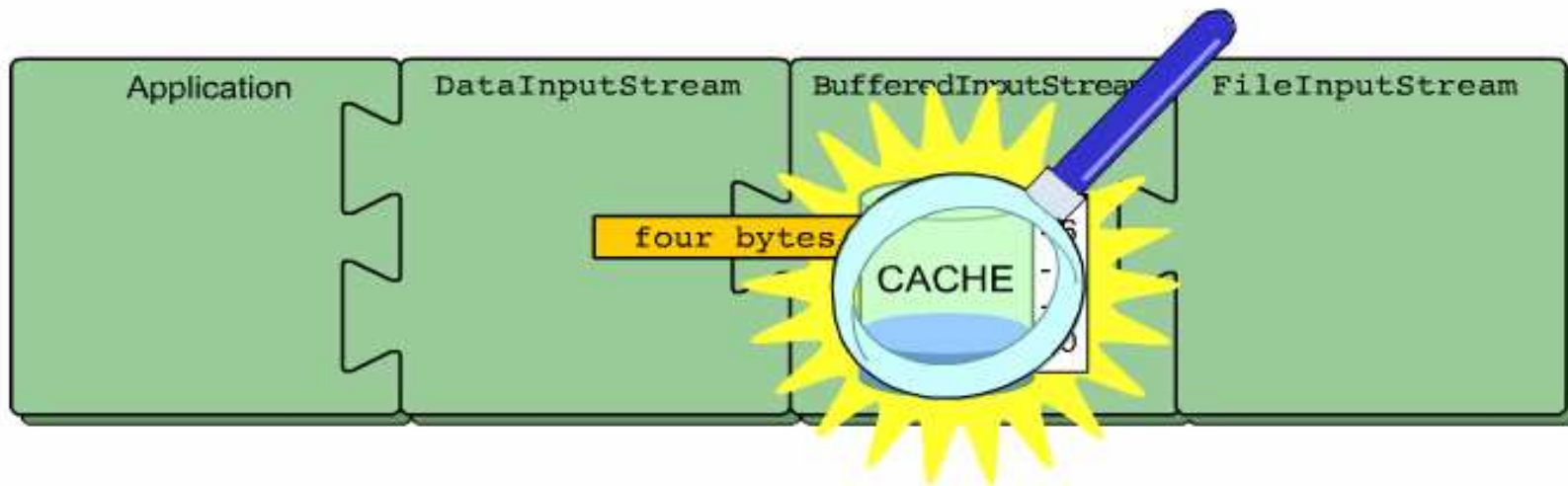
When an application invokes a method like `readInt` on a `DataInputStream` object, what happens beneath the surface?

Example: Stream Chaining 2



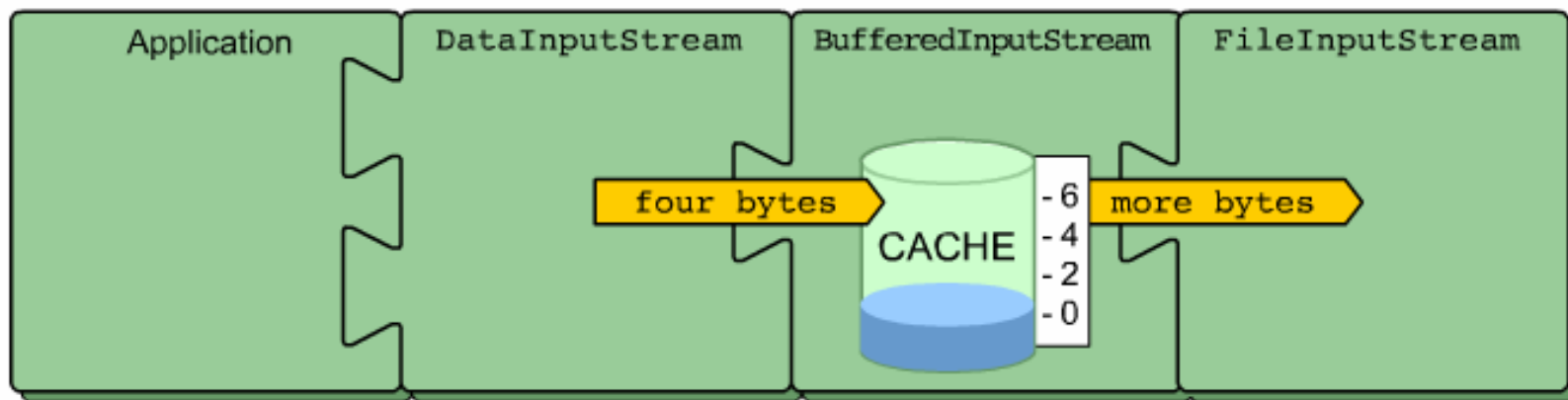
The `DataInputStream` object requests four bytes (the representation for an `int` type) from the `BufferedInputStream` object (a reference to which is contained within the `DataInputStream` object).

Example: Stream Chaining 3



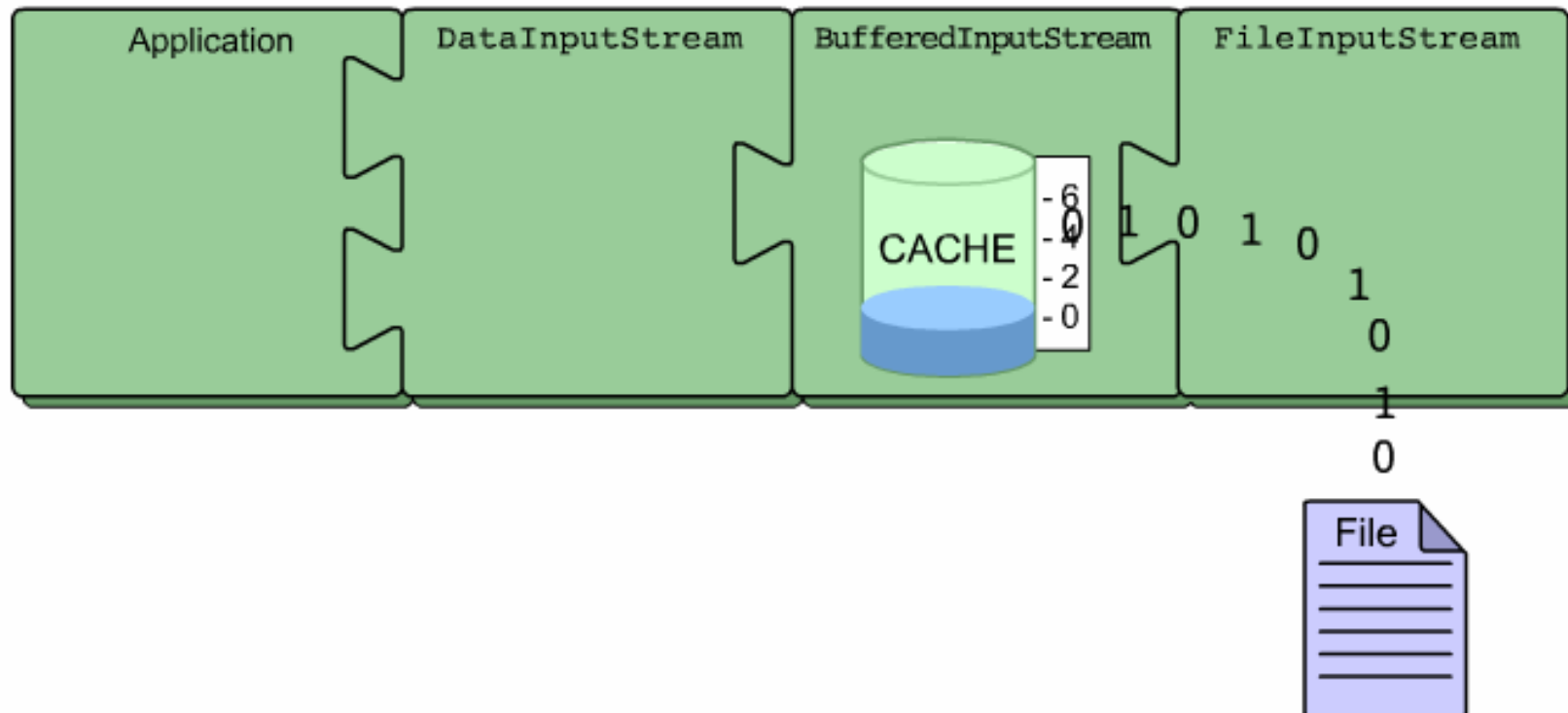
The `BufferedInputStream` object inspects its internal cache. If the cache does not contain four bytes of leftover data from a previous read, the `BufferedInputStream` object will request additional bytes from the `FileInputStream` object (a reference to which is contained within the `BufferedInputStream` object).

Example: Stream Chaining 4



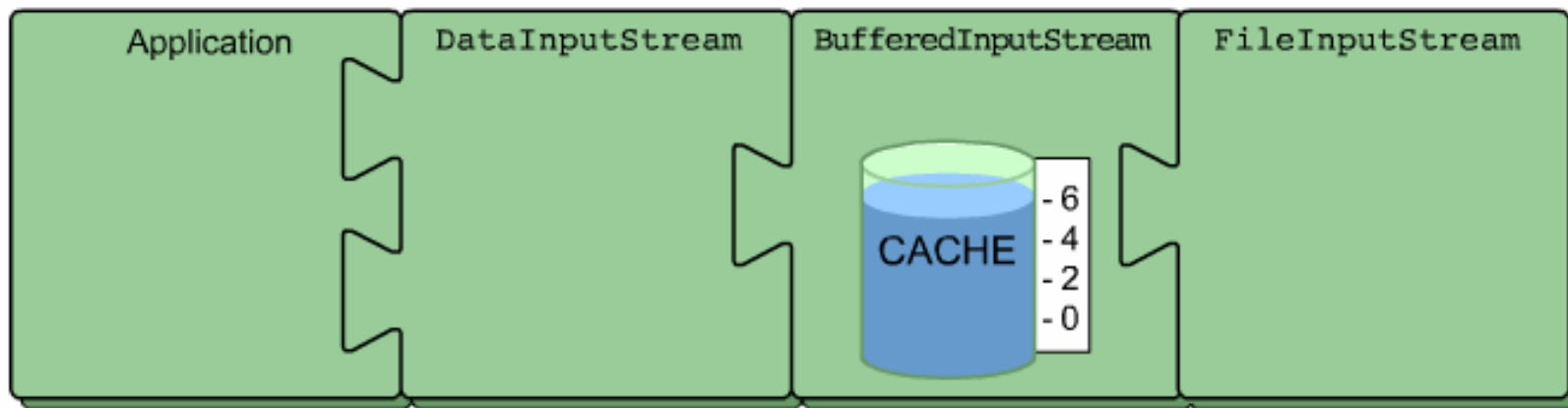
The `BufferedInputStream` object inspects its internal cache. If the cache does not contain four bytes of leftover data from a previous read, the `BufferedInputStream` object will request additional bytes from the `FileInputStream` object (a reference to which is contained within the `BufferedInputStream` object).

Example: Stream Chaining 5



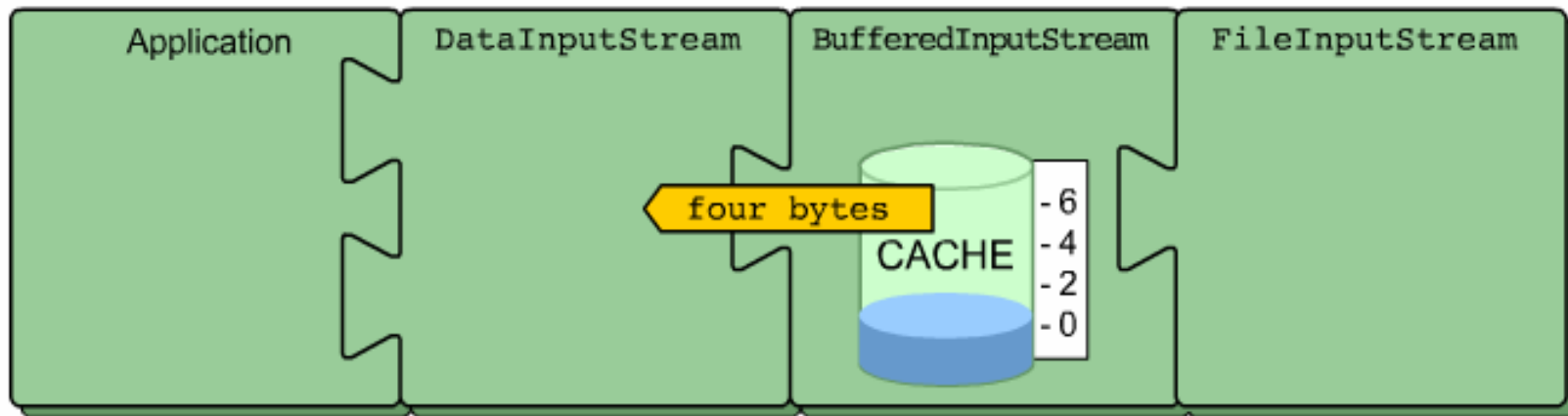
The `FileInputStream` object reads the requested numbers of bytes from the file and returns them to the `BufferedInputStream` object, which then caches the data in its internal storage buffer.

Example: Stream Chaining 6



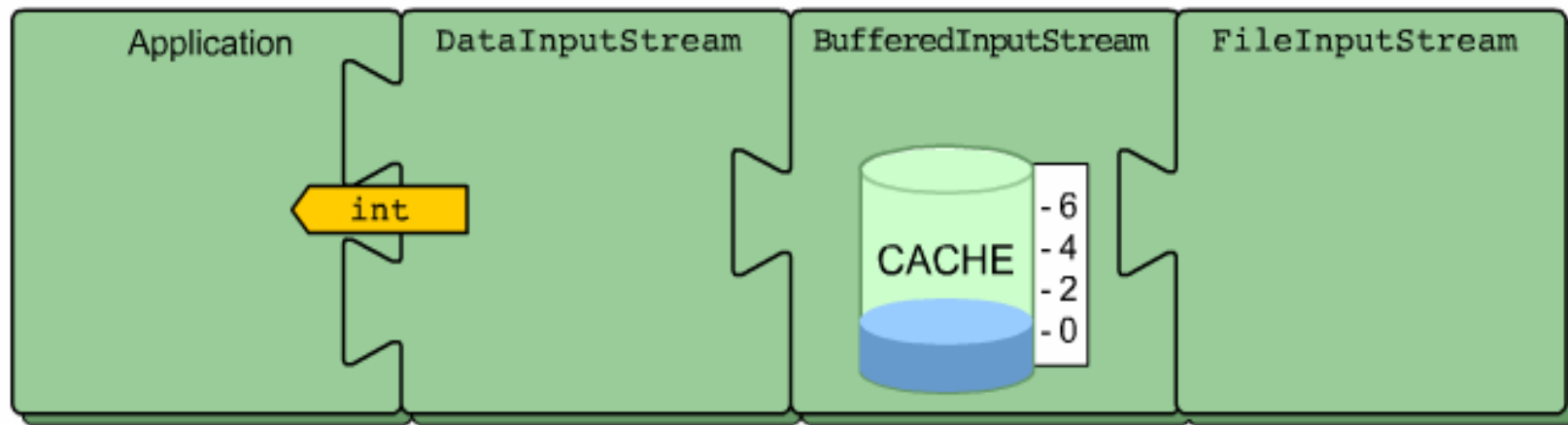
The `FileInputStream` object reads the requested numbers of bytes from the file and returns them to the `BufferedInputStream` object, which then caches the data in its internal storage buffer.

Example: Stream Chaining 7



The `BufferedInputStream` object extracts the four bytes requested by the `DataInputStream` object from the cache and returns them to the calling method.

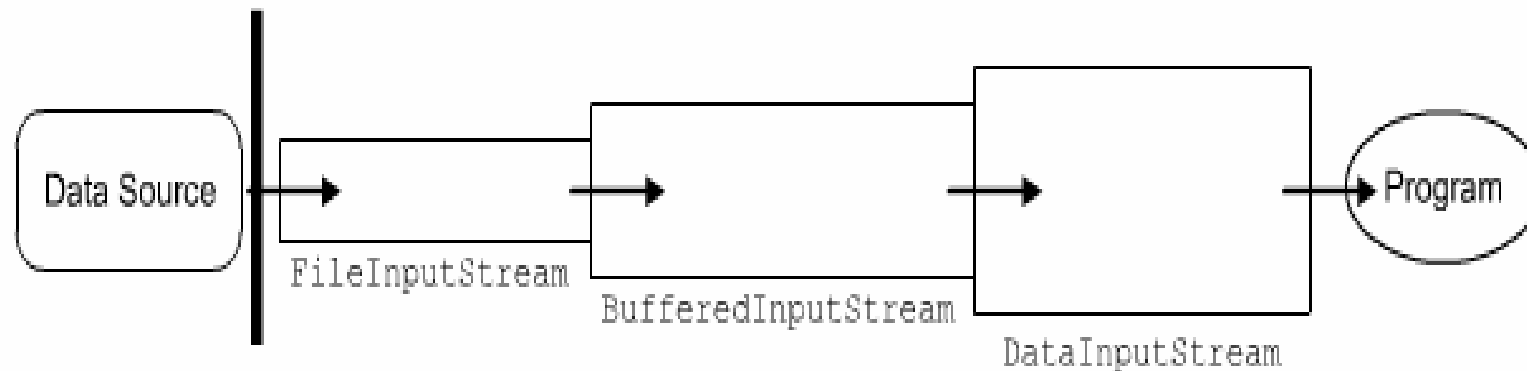
Example: Stream Chaining 8



The `DataInputStream` object returns the four bytes to the application in the form of an `int` type.

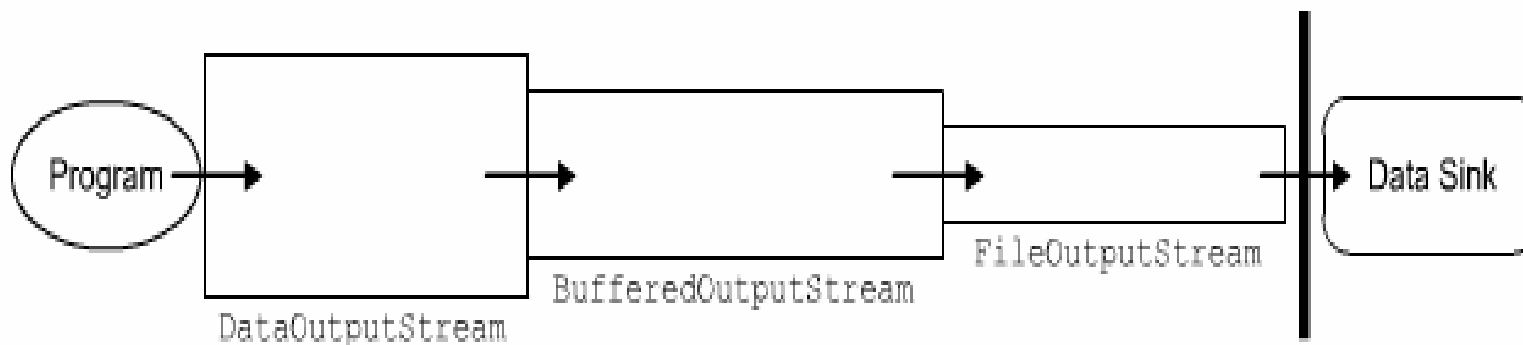
InputStream Chain

```
FileInputStream theFile = new FileInputStream( "input.dat" ) ;  
BufferedInputStream theBuffer = new BufferedInputStream ( theFile ) ;  
DataInputStream theData = new DataInputStream( theBuffer ) ;
```



OutputStream Chain

```
FileOutputStream theFile = new FileOutputStream( "output.dat" ) ;  
BufferedOutputStream theBuffer = new BufferedOutputStream ( theFile ) ;  
DataOutputStream theData = new DataOutputStream( theBuffer ) ;
```



File Operations

There are three non stream classes in `java.io` package.

- 1) `File` Class - represents a file on the local system
- 2) `FilenameFilter` class - is an interface used to filter a list of filenames

Each will be considered in details.

File Class

Represents a file on the local filesystem.

Usage:

- 1) to identify a file
- 2) obtain information about the file
- 3) and even change information about the file

Constructors:

- 1) `File(File parent, String child)`
- 2) `File(String pathname)`
- 3) `File(String parent, String child)`
- 4) `File(URI uri)`

Example: File 1

```
import java.io.File;

class FileDemo {
    static void p(String s) {
        System.out.println(s);
    }

    public static void main(String args[]) {
        File f1 = new File("filename here");
        p("File Name: " + f1.getName());
        p("Path: " + f1.getPath());
        p("Abs Path: " + f1.getAbsolutePath());
        p("Parent: " + f1.getParent());
        p(f1.exists() ? "exists" : "does not exist");
        p(f1.canWrite() ? "writeable" : "not writeable");
        p(f1.canRead() ? "is readable" : "is not readable");
    }
}
```

Example: File 2

```
p("is " + (f1.isDirectory()?" " : "not a directory"));  
p(f1.isFile() ? "normal file" : " a named pipe");  
p(f1.isAbsolute() ? "absolute" : "not absolute");  
p("File last modified: " + f1.lastModified());  
p("File size: " + f1.length() + " Bytes");  
}  
}
```

Directories

A directory is a `File` that contains a list of other files and directories.

When you create a `File` object and it is a directory, the `isDirectory()` method will return true.

In this case you can call `list()` on that object to extract the list of other files and directories inside.

The form of `list()` is

```
String[] list();
```

The list of file is returned in an array of `String` objects.

Example: Directories 1

```
import java.io.File;

class DirList {
    public static void main(String args[]) {
        String dirname = "/java";
        File f1 = new File(dirname);

        if (f1.isDirectory()) {
            System.out.println("Directory of " + dirname);
            String s[] = f1.list();

            for (int i=0; i < s.length; i++) {
                File f = new File(dirname + "/" + s[i]);
                if (f.isDirectory()) {
                    System.out.println(s[i] + " is a directory");
                } else {
                    System.out.println(s[i] + " is a file");
                }
            }
        }
    }
}
```

Example: Directories 2

```
    }  
  } else {  
    System.out.print(dirname + " is not a");  
  
    System.out.println("directory");  
  }  
}  
}
```


Creating Directories

Another two useful `File` utility methods are:

- 1) `mkdir()` – creates a directory , returning `true` on success and `false` on failure.

Failure indicates that the path specified in the `File` object already exists, or that the directory cannot be created because the entire path does not exist yet.

- 2) `mkdirs()` – to create directories for which no path exists, it creates both a directory and all the parents of the directory

FilenameFilter

To limit the number of files returned by the `list()` method to include only those files that match a certain type of filename pattern, or filter, use the second form of `list()`.

The second form of `list()` is:

```
String[] list(FilenameFilter fobj);
```

`FilenameFilter` defines only a single method, `accept()`, which is called once for each file in a list.

The `accept()` method returns `true` for files in the directory specified by directory that should be included in the list and returns `false` if otherwise.

Example: FilenameFilter 1

```
import java.io.*;

public class OnlyExt implements FilenameFilter {
    String ext;

    public OnlyExt(String ext) {
        this.ext = "." + ext;
    }

    public boolean accept(File dir, String name) {
        return name.endsWith(ext);
    }
}
```

Example: FilenameFilter 2

Here is a modified version of directory listing program. Now it display only classes that use `.html` extension.

```
import java.io.*;

class DirListOnly {
    public static void main(String args[]) {
        String dirname = "/java";
        File f1 = new File(dirname);
        FilenameFilter only = new OnlyExt("html");
        String s[] = f1.list(only);

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

Stream Benefits

The Streaming interface to I/O in Java provides:

- 1) A clean abstraction for complex and often cumbersome task.
- 2) The composition of filtered stream classes allows you to dynamically build the custom streaming interface to suit your data transfer requirements.
- 3) Java programs written to adhere to the abstract, high level-level `InputStream`, `OutputStream`, `Reader` and `Writer` classes will function properly in the future even when new and improved concrete stream classes are invented.
- 4) Serialization of object is expected to play an increasingly important role in Java Programming in the future.

Lab Work: Input and output

- 1) Write a program that reads the content of `"C:\Documents and Settings\All Users"` on your local system.
- 2) Write a program that counts the total number of directories and files you have in the path.
- 3) Archive at least one of the subdirectories of `All Users` folder and save it in zip format in your folder on the network.
- 4) Study and use `java.util.zip` package by referring to the API documentation for the appropriate classes to use in this exercise.

Project Exercise 2

- 1) Implement the client component of your software architecture which satisfies your use case model. Provide a web interface for users and desktop interface for back office processing. Implementation should be carried out using Java swing library.
- 2) Check for the consistency between your implementation and your design class diagrams (in your design model). For instance, are all your design classes implemented?
- 3) Check for the consistency between the dynamic aspect of your architecture (instance level collaboration diagrams) and your implementation
- 4) Update your implementation model indicating the implementing artifacts for your client component.

Note: Provide appropriate version control for all artifacts (models and codes)

Networking

Course Outline

- 1) introduction
- 2) streams
- 3) **networking**
- 4) database connectivity
- 5) architectures
 - a) message-orientation
 - 1) javamail
 - 2) jms
 - b) distributed objects
 - 1) rmi
 - 2) corba
 - 3) JavaIDL
- 6) summary

Outline: Networking

Presents the network programming in Java language.

Main points:

- 1) Review the basic network concepts and Java Implementation.
- 2) Discuss the usage of java.net package.
- 3) Introduce the Secure Socket.
- 4) Introduce the New I/O API.
- 5) Introduce the Java implementation for UDP protocol.

Overview

- 1) introduction
- 2) basic network concepts and Java Implementation
- 2) Implementation
- 3) JMS programming model and implementation
- 4) advance configuration
- 5) summary

Why Java

Why use Java for Networking?

- a) Java was the first programming language designed from the ground up with networking in mind.
- b) Java provides easy solutions to two crucial problem for Internet networking — platform independence and security.
- c) It is far easier to write network programs in Java than in almost any other language.
 - In the fully functional applications, very little code is devoted to networking.

Network programs with Java 1

Examples that a Network Program can do:

a) Server-Client

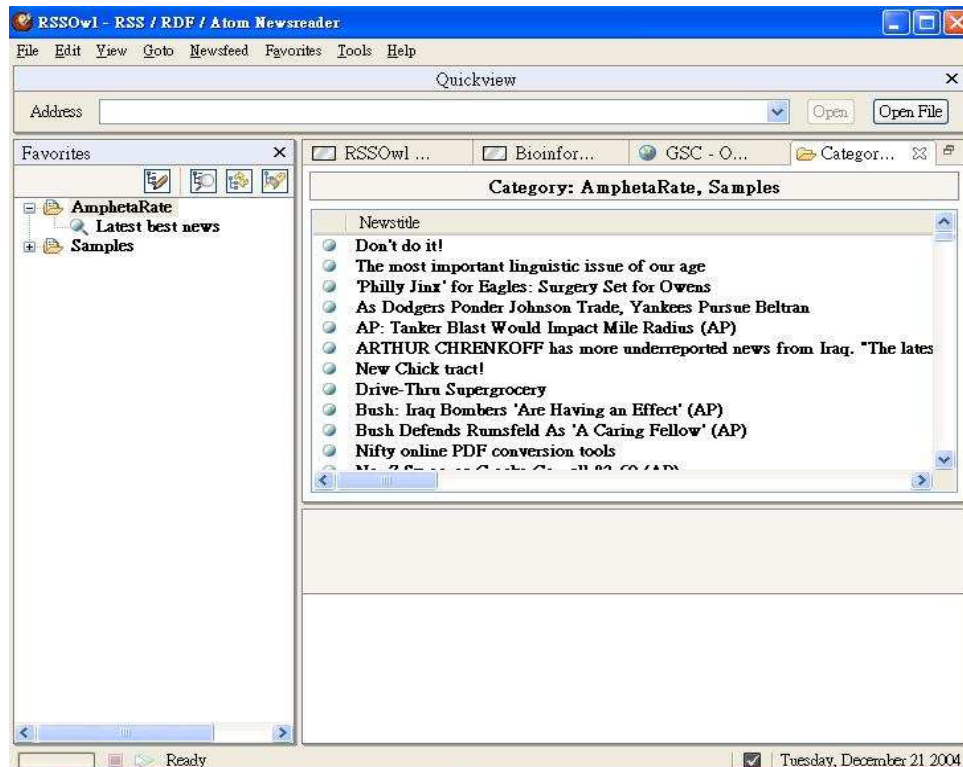
Examples: RssOwl (<http://rssowl.sourceforge.net/>)

b) Peer-to-Peer

Examples: LimeWire (<http://limewire.org/>)

Azureus (<http://azureus.sourceforge.net/>)

Network programs with Java 2



<http://rssowl.sourceforge.net/>

- RssOwl combines news from different sources and allows the user to browse using a modern graphical user interface.
- Unlike a web browser, this program can continuously update the data in real time.

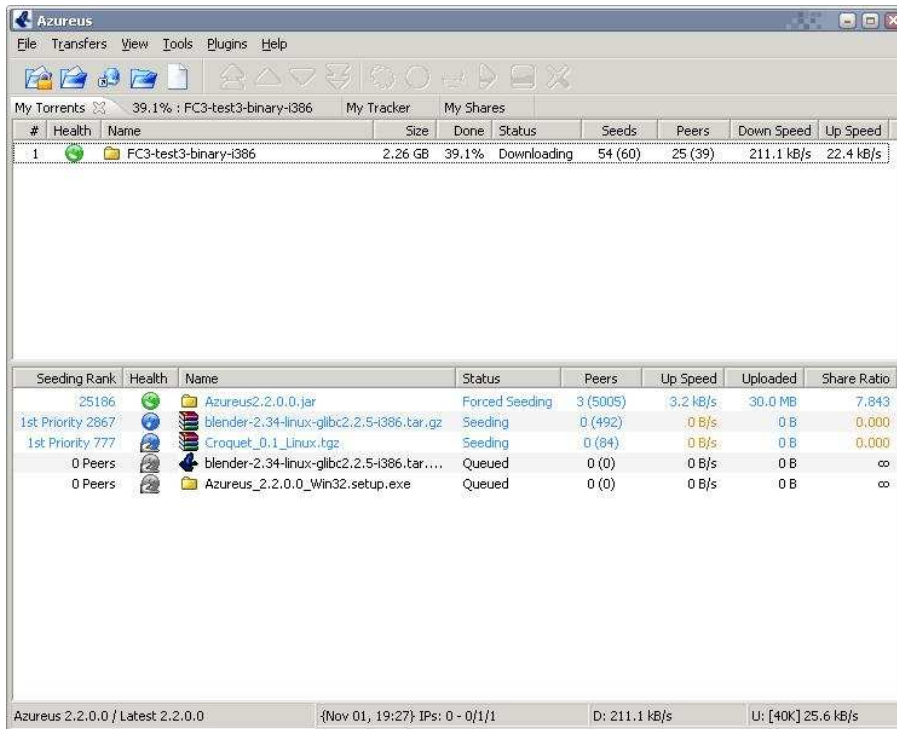
Network programs with Java 3



- LimeWire enables the clients to query each other and transfer files among themselves.
- LimeWire is an open source pure Java application that uses a Swing GUI and standard Java networking classes.

<http://www.limewire.org/>

Network programs with Java 4



<http://azureus.sourceforge.net>

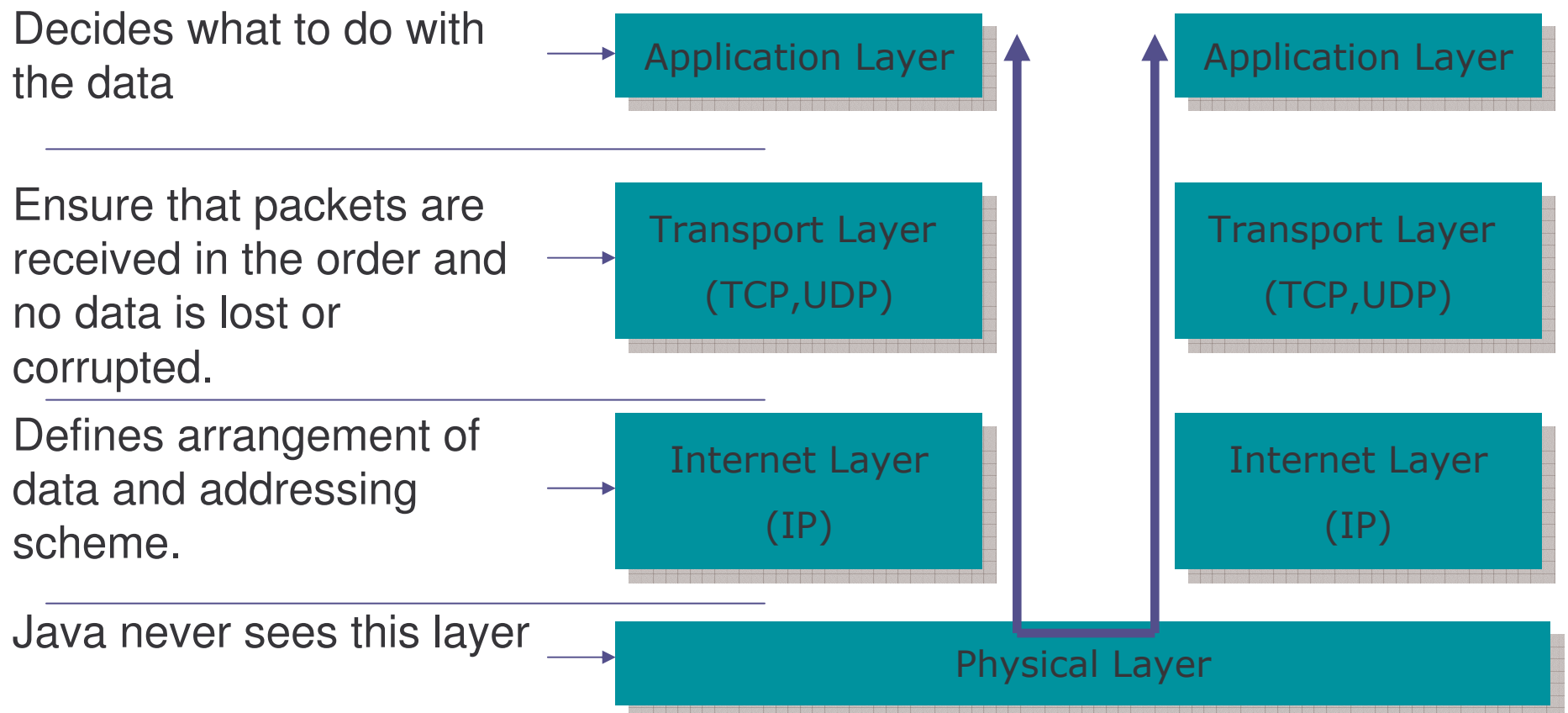
- Azureus is one of the BitTorrent clients written in pure Java.
- BitTorrent is designed to serve files that can be referenced from known keys
- Downloaders can sharing a file while they're still downloading it.

Concepts for network program

Important concepts needed for writing network program in Java

- 1) Communication protocols: TCP and UDP
- 2) Ports and Internet Addresses
- 3) Sockets
- 4) Uniform Resource Locator (URL)
- 5) Uniform Resource Identifier (URI)
- 6) Streams and Threads (Covered in previous sessions)
- 7) Classes in java.net and java.io packages

Network Layers



TCP and UDP

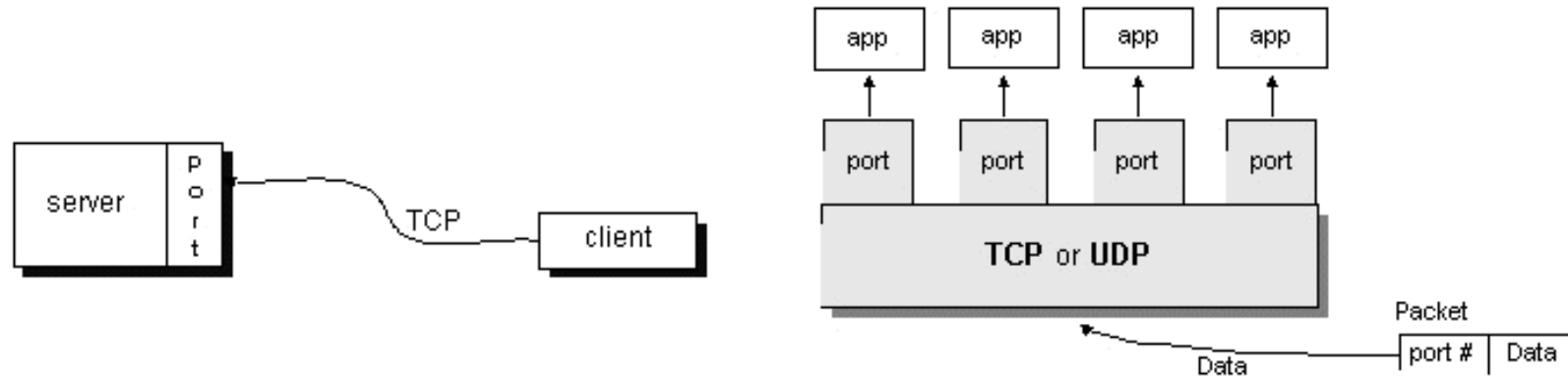
Java only supports TCP (Transmission Control Protocol), UDP (User Datagram Protocol) and application layer protocols built on top of these.

Characteristics of TCP and UDP :

TCP	UDP
<ul style="list-style-type: none">• Provides the ability to acknowledge receipt of IP packets and request retransmission of lost or corrupted packets.• Allows the received packets to be put back together in the order they were sent.• Requires a lot of overhead.• Supported classes in java.net : URL, URLConnection, Socket, and ServerSocket	<ul style="list-style-type: none">• Is an unreliable protocol that does not guarantee that packets will arrive at their destination.• Allow the receiver to detect corrupted packets but does not guarantee that packets are delivered in the correct order.• Requires less overhead and faster.• Supported classes in java.net : DatagramPacket, DatagramSocket, and MulticastSocket

Ports

Each port from the server can be treated by the clients as a separate machine offering different services.



Port numbers are represented by 16-bit numbers. (0 to 65,535)

The port numbers ranging from 0 - 1023 reserved for use by well-known services such as HTTP and FTP and other system services.

Sockets

You can reach required service via its network and port IDs. what then?

- a) If you are a client
 - you need an API that will allow you to send messages to that service and read replies from it
- b) If you are a server
 - you need to be able to create a port and listen at it.
 - you need to be able to read the message comes in and reply to it.

The **Socket** and **ServerSocket** are the Java client and server classes to do this.

Example : Sending Email 1

E-mail is sent by socket communication with port 25 on a computer system.

open a socket connected to port 25 on some system, and speak “mail protocol” to the daemon at the other end.

Example : Sending Email 2

```
import java.io.*;
import java.net.*;
public class SendEmail {
    public static void main(String args[]) throws
        IOException {
        Socket sock;
        DataInputStream dis;
        BufferedReader br;
        PrintStream ps;
        System.out.println(">>> Connect
                           mailhost.iist.unu.edu");
        sock = new Socket("mailhost.iist.unu.edu", 25);
        dis = new DataInputStream(sock.getInputStream());
```

Example : Sending Email 3

```
br = new BufferedReader (new
    InputStreamReader(dis));
ps = new PrintStream( sock.getOutputStream());
System.out.println( br.readLine() );
System.out.println(">>> Hello UNU/IISTT");
ps.println("Hello UNU/IISTT");
System.out.println( br.readLine() );
System.out.println(">>> Mail From:
    oluotes@yahoo.com");
ps.println("MAIL FROM:milton_hm@hotmail.com");
System.out.println( br.readLine() );
String Addressee= "milton@iist.unu.edu";
```


Example : Sending Email 4

```
System.out.println(">>> Rcpt to: " + Addressee);  
ps.println("RCPT TO: " + Addressee );  
System.out.println( br.readLine() );  
System.out.println(">>> Send \"data\\\"");  
ps.println("DATA");  
System.out.println( br.readLine() );  
System.out.println(">>>>>>>>>");  
System.out.println(">>> This is the message\\n that  
Java sent");  
System.out.println(">>> We are testing Socket  
Programming");  
System.out.println(">>> in eMacao Training  
program.");  
System.out.println(">>>>>>>>>");
```

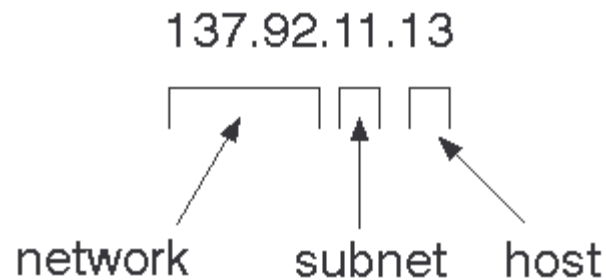
Example : Sending Email 5

```
ps.println("This is the message\n that Java  
sent");  
ps.println("We are testing Socket Programming");  
ps.println("in eMacao Training program.");  
System.out.println(">>> .");  
ps.println(".");  
System.out.println( br.readLine() );  
System.out.println(">>> QUIT");  
ps.println("QUIT");  
System.out.println( br.readLine() );  
ps.flush();  
sock.close();  
}  
}
```

Internet Addressing

Internet address (IP address) is a unique number for identifying a device connected to the Internet.

The current standard is IPv4 which are four bytes long.



The hostname and IP address is, in Java, represented by `java.net.InetAddress`.

`InetAddress` is used by many other networking classes, including `Socket`, `ServerSocket`, `URL`, `DatagramSocket`, `DatagramPacket`, and more.

Example

The following program will print out the IP address of the www.iist.unu.edu

```
import java.net.*;

public class IISTByName {

    public static void main (String[] args) {

        try { InetAddress address =

            InetAddress.getByName      ("www.iist.unu.edu") ;

            System.out.println(address);

        } catch (UnknownHostException ex) {

            System.out.println("Could not find

                www.iist.unu.edu ");

        }

    }

}
```

InetAddress methods

Useful methods:

- a) `static InetAddress getByName(String host)`
- b) `static InetAddress getLocalHost()`
- c) `String getHostAddress();` // in dotted form
- d) `String getHostName();`

The URL Class 1

The `java.net.URL` is an abstraction of a Uniform Resource Locator (URL).

URLs are composed of five pieces:

1. The scheme, also known as the protocol
2. The authority
3. The path
4. The query string
5. The fragment identifier, also known as the section or ref

`<scheme>://<authority><path>?<query>#<fragment>`

The URL Class 2

For example, given the URL :

`http://www.ibiblio.org/javafaq/javabooks/index.html?isbn=123456789#toc`

1. scheme : http
2. authority : www.ibiblio.org
3. path : /javafaq/books/javabooks/index.html
4. query string : isbn=123456789
5. fragment identifier : toc

The URL Class 3

The authority may further be divided into the user info, the host, and the port.

For example, in the URL `http://admin@www.blackstar.com:8080/`

1. user info : admin
2. host : www.blackstar.com
3. port : 8080

The URL Class 4

The `java.net.URL` class provides static methods for getting the above mentioned information:

- a) `getFile()`
- b) `getHost()`
- c) `getPort()`
- d) `getProtocol()`
- e) `getRef()`
- f) `getQuery()`
- g) `getPath()`
- h) `getUserInfo()`
- i) `getAuthority()`

The URL Class 5

Unlike the `InetAddress` objects, you can construct instances of `java.net.URL` using one of its six constructors.

- 1) `public URL(String url) throws MalformedURLException`
- 2) `public URL(String protocol, String hostname, String file) throws MalformedURLException`
- 3) `public URL(String protocol, String host, int port, String file) throws MalformedURLException`
- 4) `public URL(URL base, String relative) throws MalformedURLException`

The URL Class 6

5) `public URL(URL base, String relative,
URLStreamHandler handler) // 1.2 throws
MalformedURLException`

6) `public URL(String protocol, String host, int port,
String file, // 1.2 URLStreamHandler handler)
throws MalformedURLException`

Example 1

The following program will test the protocol supported by the browser:

```
import java.net.*;
public class ProtocolTester {
    public static void testProtocol(String url) {
        try {
            URL u = new URL(url);
            System.out.println(u.getProtocol( ) + " is
                supported");
        }
        catch (MalformedURLException ex) {
            String protocol =
                url.substring(0, url.indexOf(':'));
            System.out.println(protocol + " is not
                supported");
        }
    }
}
```

Example 2

You can test it with the following Tester:

```
public class Tester {  
    public static void main(String[] args) {  
        ProtocolTester.testProtocol("http://www.adc.org");  
        ProtocolTester.testProtocol("https://www.amazon.com/exec/obidos/order2/");  
        ProtocolTester.testProtocol("ftp://metalab.unc.edu/pub/languages/java/javafaq/");  
    }  
}
```

Lab Work: URL 1

1) Write a program that will split the input URL into corresponding parts.

Given: `java URLSplitter`

```
http://www.unu.iist.edu/demoweb/html-  
primer.html#A1.3.3.3
```

Output:

```
The output will be:
```

```
The URL is http://www.unu.iist.edu/demoweb/html-  
primer.html#A1.3.3.3
```

```
The scheme is http
```

```
The user info is null
```

```
The host is www.unu.iist.edu
```

```
The port is -1
```

```
The path is /demoweb/html-primer.html
```

```
The ref is A1.3.3.3
```

```
The query string is null
```

Lab Work: URL 2

Try to test your program using the following URL:

```
ftp://mp3:mp3@138.247.121.61:21000/c%3a/  
http://www.oreilly.com  
http://www.ibiblio.org/nywc/compositions.phtml?category=Piano  
http://admin@www.blackstar.com:8080/
```

2) What is the difference between file and path?

Retrieving Data from a URL

The URL class provides methods for retrieving data from a URL:

```
public InputStream openStream( ) throws IOException
```

```
public URLConnection openConnection( ) throws  
IOException
```

```
public URLConnection openConnection(Proxy proxy)  
throws IOException // 1.5
```

```
public Object getContent( ) throws IOException
```

```
public Object getContent(Class[] classes) throws  
IOException // 1.3
```


Retrieving Data from a URL 1

Procedure to use the methods:

1) Create an URL object

e.g. `URL u = new URL("http://www.iist.unu.edu");`

2) Open an InputStream object directly from the URL object

e.g. `InputStream in = u.openStream();`

3) Or open an URLConnection object from the URL object and then get an InputStream object from the URLConnection object .

e.g. `URLConnection uc = u.openConnection();
InputStream in = uc.getInputStream();`

4) In either case, you will have an InputStream. What's followed is the normal I/O procedure for getting data.

5) Don't forget to put the try catch block for catching the `MalformedURLException` and `IOException`.

Retrieving Data from a URL 2

What is the difference between using the `openStream` and `openConnection` method?

- 1) `openStream` method only give you the access to the raw data and cannot detect the encoding information.
- 2) `openConnection` method opens a socket to the specified URL and returns a `URLConnection` object.
- 3) The `URLConnection` object gives you access to everything sent by the server. You can access all the metadata specified by the protocol such as the scheme. The `URLConnection` class also lets you write data to as well as read from a URL.

Retrieving Data from a URL 3

The following methods are used to access the header fields and the contents after the connection is made to the remote object:

- 1)getContent
- 2)getHeaderField
- 3)getInputStream
- 4)getOutputStream

Retrieving Data from a URL 4

Certain header fields are accessed frequently. The methods:

- 1) `getContentEncoding`
- 2) `getContentLength`
- 3) `getContentType`
- 4) `getDate`
- 5) `getExpiration`
- 6) `getLastModified`

Example : Reading from URL 1

```
import java.net.*;
import java.io.*;
public class URLConnectionReader {
    public static void main(String[] args) throws
        Exception {
        URL yahoo = new URL("http://www.yahoo.com/");
        URLConnection yc = yahoo.openConnection();
        BufferedReader in = new BufferedReader(
            new InputStreamReader (yc.getInputStream()));
        String inputLine;
```

Example : Reading from URL 2

```
while ((inputLine = in.readLine()) != null)
    System.out.println(inputLine);
    in.close();
}
```

Uniform Resource Identifier (URI)

A Uniform Resource Identifier (URI) is an abstraction of a URL.

Most URIs used in practice are URLs, but most specifications and standards such as XML are defined in terms of URIs

In Java 1.4 and later, URIs are represented by the `java.net.URI` class.

you should use the URL class when you want to download the content of a URL and the URI class when you want to use the URI for identification rather than retrieval.

When you need to do both, you may convert from a URI to a URL with the `toURL()` method, and in Java 1.5 you can also convert from a URL to a URI using the `toURI()` method of the URL class.

Uniform Resource Identifier (URI)

A URI reference has up to three parts: a scheme, a scheme-specific part, and a fragment identifier. The general format is:
scheme:scheme-specific-part:fragment .

Getter methods:

- 1) public String getScheme()
- 2) public String getSchemeSpecificPart()
- 3) public String getRawSchemeSpecificPart()
- 4) public String getFragment()
- 5) public String getRawFragment()

Lab Work: URI

- 1) Write a program that will split the input URI into corresponding parts.
- 2) List the methods you can get from the URI class using the javadoc.

Networking Examples

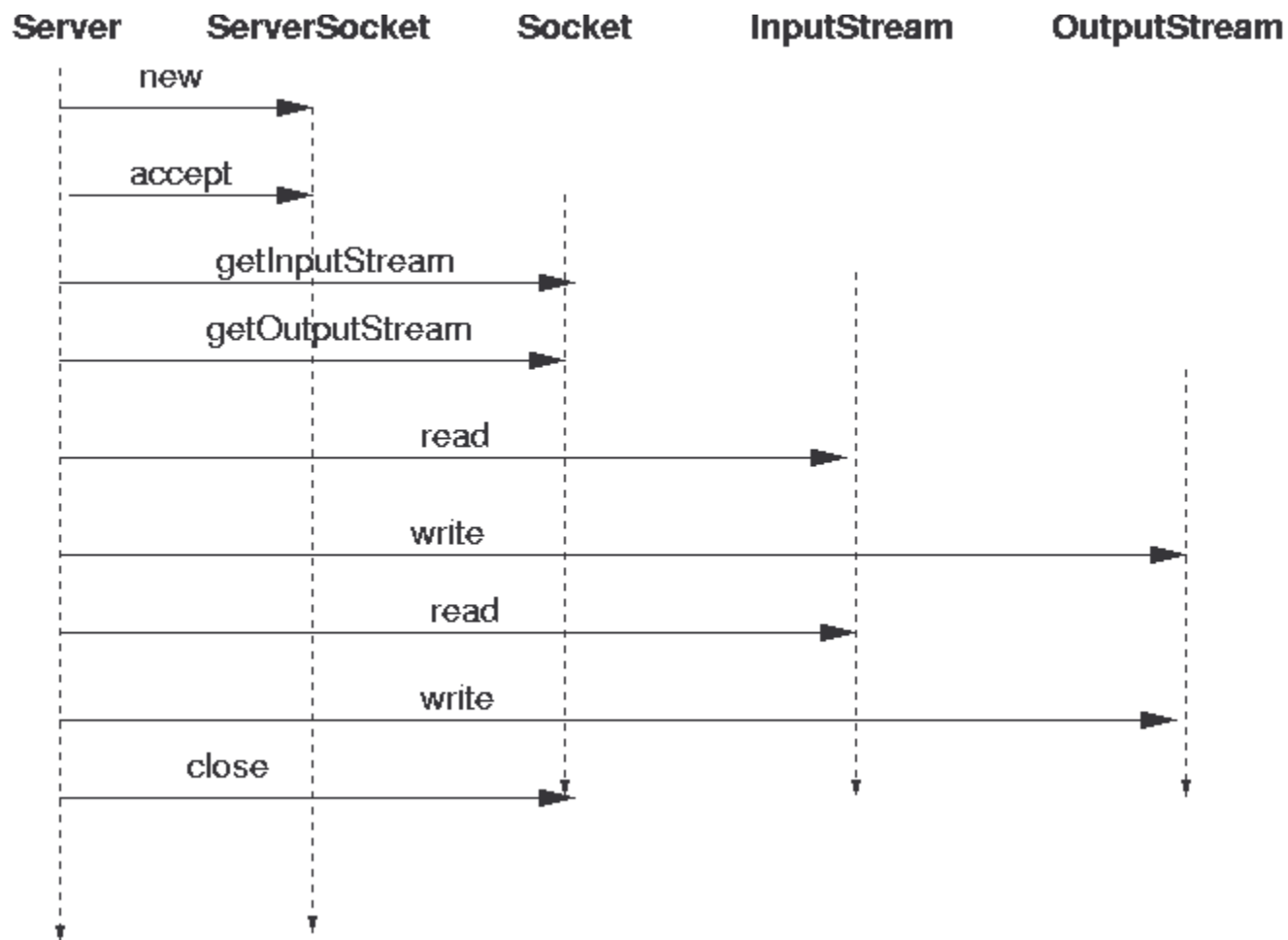
Now you have the basic concepts for different components for Java networking. Let's try to start some simple experiments.

A Simple Example



Create a Server

How to create a server?



Echo Server 1

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

public class EchoServer {
    public static int MYECHOPORT = 8189;

    public static void main(String argv[]) {
        ServerSocket s = null;
        try {
            s = new ServerSocket(MYECHOPORT);
        } catch(IOException e) {
            System.out.println(e);
            System.exit(1);
        }
    }
}
```

Echo Server 2

```
while (true) {  
    Socket incoming = null;  
    try {  
        incoming = s.accept();  
    } catch (IOException e) {  
        System.out.println(e);  
        continue;  
    }  
    try {  
        incoming.setSoTimeout(10000); //10 seconds  
    } catch (SocketException e) {  
        e.printStackTrace();  
    }  
}
```

Echo Server 3

```
try {
    handleSocket(incoming);
} catch (InterruptedException e) {
    System.out.println("Time expired " + e);
} catch (IOException e) {
    System.out.println(e);
}

try {
    incoming.close();
} catch (IOException e) {
    // ignore
}

}

}
```

Echo Server 4

```
public static void handleSocket(Socket incoming)
    throws IOException {
    BufferedReader reader =
        new BufferedReader(new InputStreamReader(
            incoming.getInputStream()));
    PrintStream out =
        new PrintStream(incoming.getOutputStream());
    out.println("Hello. Enter BYE to exit");

    boolean done = false;
    while ( ! done) {
        String str = reader.readLine();
```

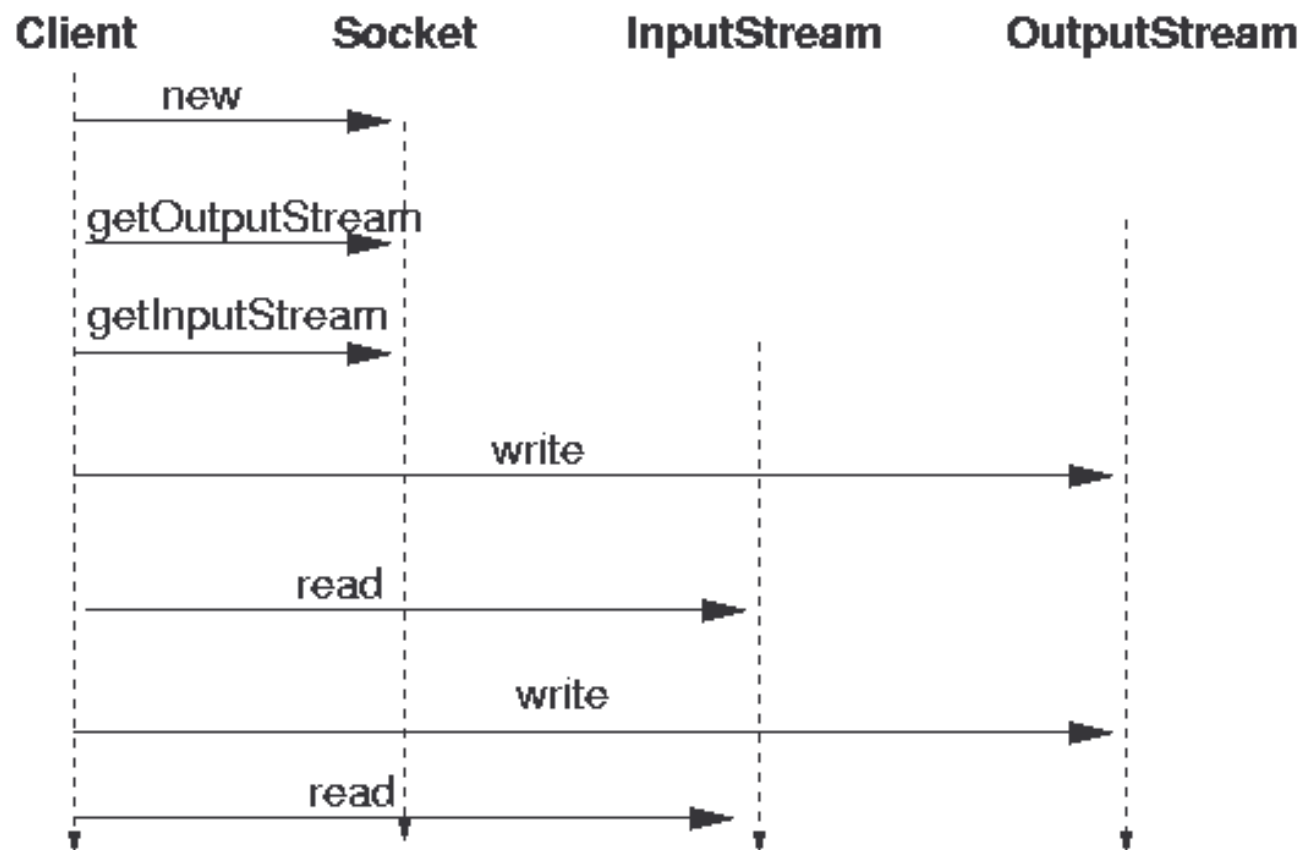

Echo Server 5

```
if (str == null) {
    done = true;
    System.out.println("Null received");
}
else {
    out.println("Echo: " + str);
    if (str.trim().equals("BYE"))
        done = true;
}

incoming.close();
}
```

Create a Client

How to create a client connected to a sever?



Echo Client 1

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

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

        Socket echoSocket = null;
        PrintWriter out = null;
        BufferedReader in = null;
        BufferedReader stdIn = null;
```

Echo Client 2

```
try {  
    echoSocket = new Socket("localhost", 8189);  
    out = new  
    PrintWriter(echoSocket.getOutputStream(), true);  
    in = new BufferedReader(new  
    InputStreamReader(echoSocket.getInputStream()));  
    System.out.println (in.readLine());  
} catch (UnknownHostException e) {  
    System.err.println("Don't know about host.");  
    System.exit(1);  
} catch (IOException e) {  
    System.err.println("Couldn't get I/O for "  
    + "the connection to server.");
```

Echo Client 3

```
        System.exit(1);
    }
    try {
        stdin = new BufferedReader(new InputStreamReader
            (System.in));
        String userInput;
        while ((userInput = stdin.readLine()) != null) {
            out.println(userInput);
            System.out.println("echo: " + in.readLine());
        }
    } catch (SocketException e) {
        System.err.println("Socket closed");
    }
```

Echo Client 4

```
finally {  
    if (out != null)  
        out.close();  
    if (in != null)  
        in.close();  
    if (stdIn != null)  
        stdIn.close();  
    if (echoSocket != null)  
        echoSocket.close();  
}  
  
}
```

Lab Work: Chatting Program

- 1) Modify the previous Echo Sever and Echo Client examples to create a server-client chatting program.

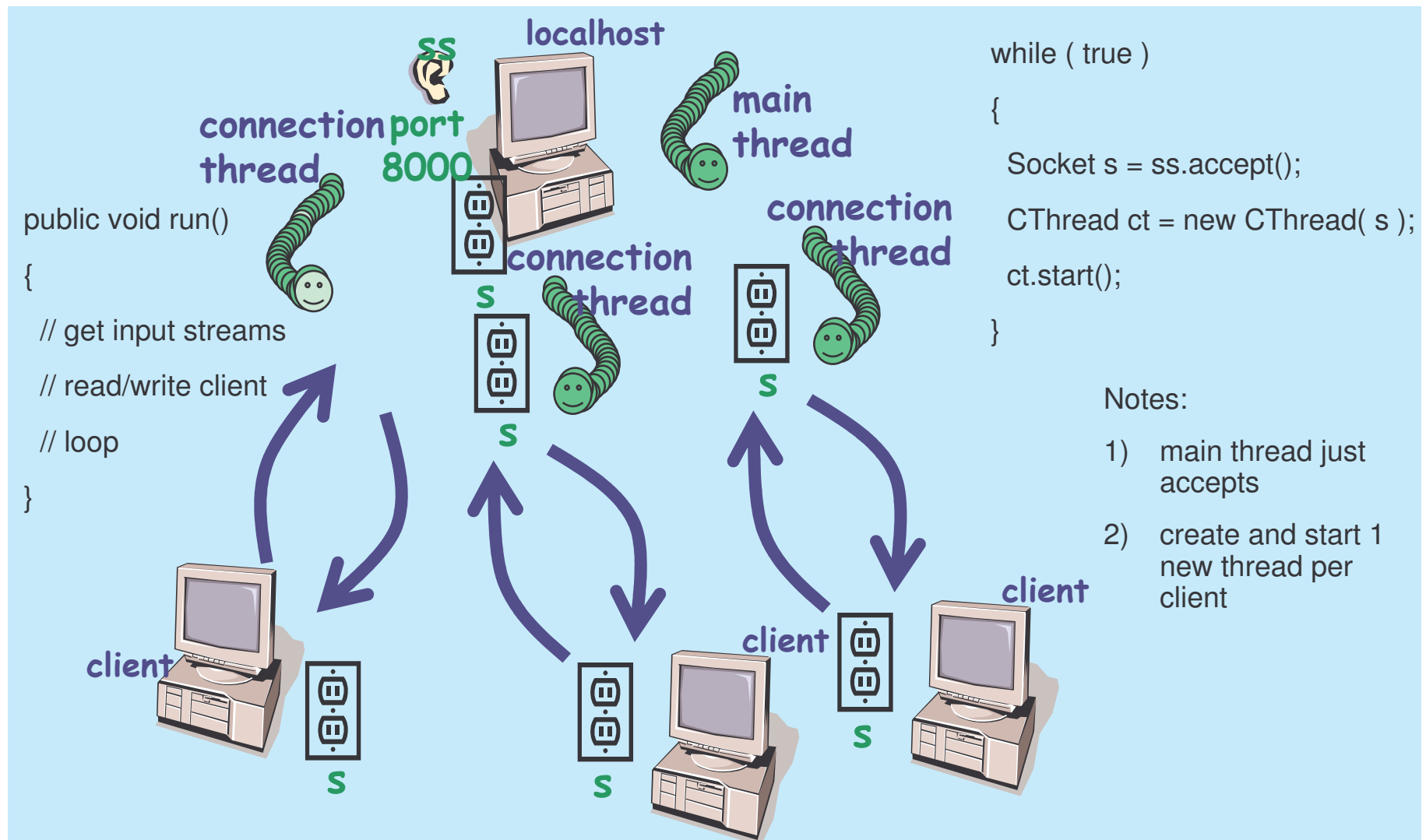
Hints:

- a) You need to create a server which will wait for the client to establish the connection. Then it will print a statement to the client's console notifying that a connection is made.
- b) Once the connection is made, both the server and client will be able read message from the counterpart and write message to it.

Exercise : Multiple Clients 1

- 1) Please modify the previous lab work. you need to make your server to be able to talk to multiple clients connected the server.

Exercise : Multiple Clients 2



Secure Sockets

Starting from JDK 1.4, Java Secure Sockets Extension (JSSE) is part of the standard distribution.

JSSE uses the Secure Sockets Layer (SSL) Version 3 and Transport Layer Security (TLS) protocols and their associated algorithms to secure network communications.

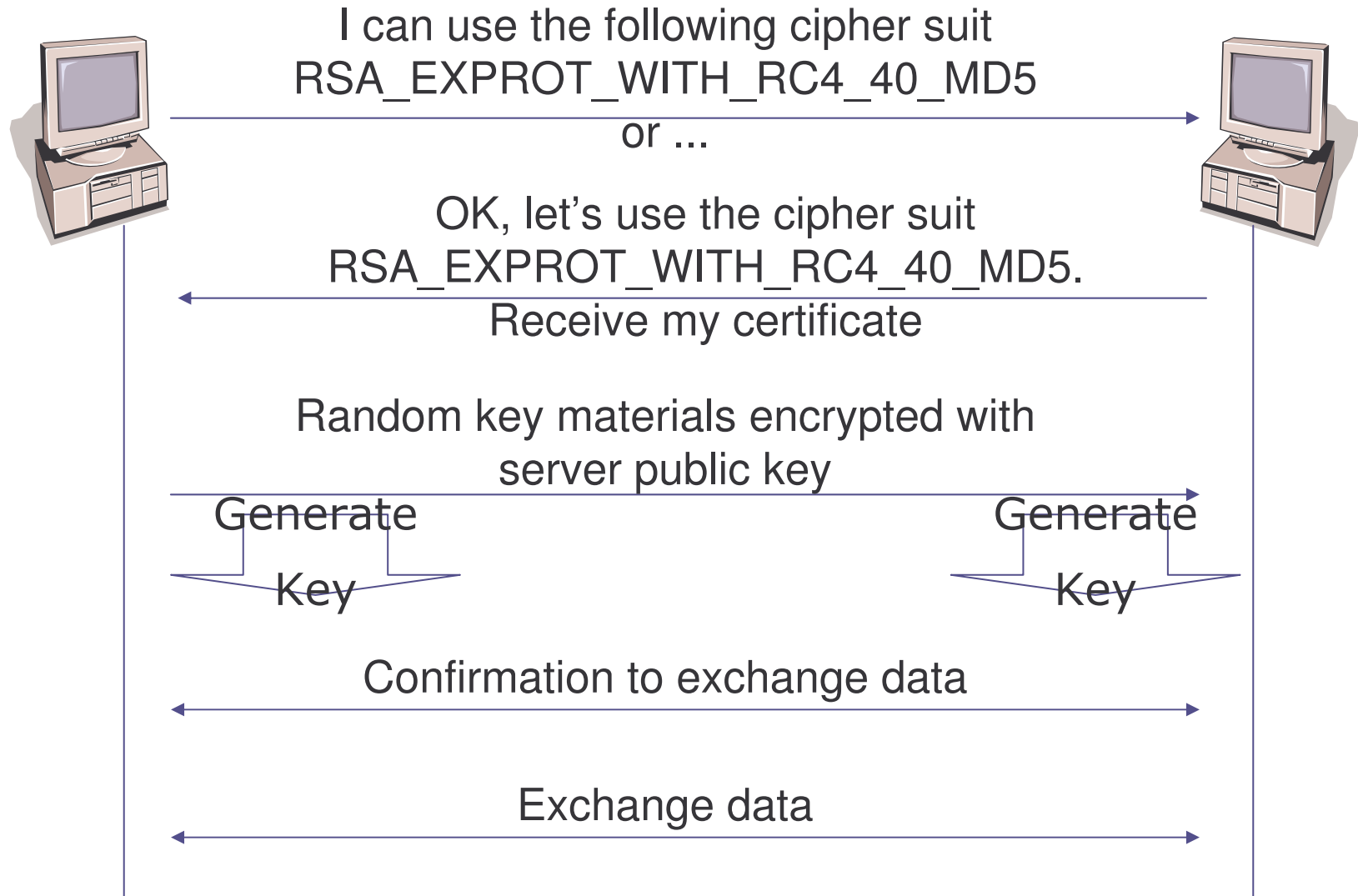
JSSE abstracts all the low-level details such as keys exchange, authentication, and data encryption. All you have to do is to send your data over the streams from the secured sockets obtained.

Java Secure Socket Extension

The Java Secure Socket Extension is divided into four packages:

- 1) `javax.net.ssl` : The abstract classes that define Java's API for secure network communication.
- 2) `javax.net` : The abstract socket factory classes used instead of constructors to create secure sockets.
- 3) `javax.security.cert` : A minimal set of classes for handling public key certificates that's needed for SSL in Java 1.1. (In Java 1.2 and later, the `java.security.cert` package should be used instead.)
- 4) `com.sun.net.ssl` : The concrete classes that implement the encryption algorithms and protocols in Sun's reference implementation of the JSSE.

SSL Handshake



Secure Client Sockets 1

Procedures to create a secure client socket:

- 1) get an instance of `SocketFactory` by invoking the static `SSLSocketFactory.getDefault()` method. e.g.

```
SocketFactory sf = SSLSocketFactory.getDefault();
```

- 2) use one of these five overloaded `createSocket()` methods to build an `SSLSocket`:

```
1. public abstract Socket createSocket(String host,  
    int port) throws IOException,  
    UnknownHostException
```

```
2. public abstract Socket createSocket(InetAddress  
    host, int port) throws IOException
```

Secure Client Sockets 2

3. `public abstract Socket createSocket(String host, int port, InetAddress interface, int localPort) throws IOException, UnknownHostException`
4. `public abstract Socket createSocket(InetAddress host, int port, InetAddress interface, int localPort) throws IOException, UnknownHostException`
5. `public abstract Socket createSocket(Socket proxy, String host, int port, boolean autoClose) throws IOException`

Secure Client Sockets 3

- 3) Once the socket has been created, you use it just like any other socket, through its `getInputStream()`, `getOutputStream()`, and other methods.

For example, if the following purchasing information is required to be sent over the network:

- a) Name: John Smith
- b) Product-ID: 67X-89
- c) Address: 1280 Deniston Blvd, NY NY 10003
- d) Card number: 4000-1234-5678-9017
- e) Expires: 08/05

Using JSSE, the following code will do the work for you:

Secure Client Sockets 4

```
try {  
    SSLSocketFactory factory  
    = (SSLSocketFactory) SSLSocketFactory.getDefault( );  
    Socket socket = factory.createSocket("localhost",  
        7000);  
    Writer out = new  
        OutputStreamWriter(socket.getOutputStream( ),  
            "ASCII");  
    out.write("Name: John Smith\r\n");  
}
```


Secure Client Sockets 5

```
out.write("Product-ID: 67X-89\r\n");
out.write("Address: 1280 Deniston Blvd, NY NY
    10003\r\n");
out.write("Card number: 4000-1234-5678-9017\r\n");
out.write("Expires: 08/05\r\n");
out.flush( );
out.close( );
socket.close( );
} catch (IOException ex) {
    ex.printStackTrace( );
}
```

Configuring Secure Sockets

Methods are available for configuring how much and what kind of authentication and encryption is performed.

- a) `getSupportedCipherSuites()` method tells you which combination of algorithms is available on a given socket
- b) `getEnabledCipherSuites()` method tells you which suites this socket is willing to use
- c) You can change the suites the client attempts to use via the `setEnabledCipherSuites(String[] suites)` method
 - Sun's JDK 1.4 supports 23 cipher suites. For the list of the supported cipher suites, please check with JavaDoc.
- d) There are still methods for handling handshaking and sessions, and I will open these for your further study.

Secure Server Sockets 1

Procedures to create a secure server socket:

- 1) get an instance of `ServerSocketFactory` by invoking the static `SSLConnectionFactory.getDefault()` method. e.g.

```
ServerSocketFactory sf =  
    SSLServerSocketFactory.getDefault();
```

- 2) use one of these three overloaded `createServerSocket()` methods to build an `SSLServerSocket`:

```
1. public abstract ServerSocket  
   createServerSocket(int port) throws IOException  
  
2. public abstract ServerSocket  
   createServerSocket(int port, int queueLength)  
   throws IOException  
  
3. public abstract ServerSocket  
   createServerSocket(int port, int queueLength,  
   InetAddress interface) throws IOException
```

Secure Server Sockets 2

3) Unlike creating the client socket, you need to do more to set up the encryption for the server socket.

This setup varies between different JSSE implementations. In Sun's implementation, you may need to do the followings:

1. Generate public keys and certificates using *keytool*.
2. Pay money to have your certificates authenticated by a trusted third party such as Verisign.
3. Create an SSLContext for the algorithm you'll use.
4. Create a TrustManagerFactory for the source of certificate material you'll be using.

Secure Server Sockets 3

5. Create a KeyManagerFactory for the type of key material you'll be using.
6. Create a KeyStore object for the key and certificate database. (Sun's default is JKS.)
7. Fill the KeyStore object with keys and certificates; for instance, by loading them from the filesystem using the pass phrase they're encrypted with.
8. Initialize the KeyManagerFactory with the KeyStore and its pass phrase.
9. Initialize the context with the necessary key managers from the KeyManagerFactory, trust managers from the TrustManagerFactory, and a source of randomness. (The last two can be null if you're willing to accept the defaults.)

Lab Work: Secure Sockets 1

- 1) Please try to run through the process for setting up a secure socket server as following and implement it in java code.
 - a) Generate public keys and certificates using *keytool*
 - ***D:\JAVA>keytool -genkey -alias ourstore -keystore jnp3e.keys***
 - Answer some questions and please remember the password you entered. You will need it later.
 - A file jnp3e.keys will be generated and protected by the password you entered.
 - b) If you don't want to pay for the digital ID for experiment, you can use the verified keystore file called testkeys, protected with the password "passphrase", included in SUN's JSSE implementation package.
 - c) Create a class named SecureServer.

Lab Work: Secure Sockets 2

d) Import the necessary packages.

e) Define variables:

- int PORT – default port number
- String ALGORITHM – algorithm for setting the SSLContext (“SSL”)
- String KEYFILE – in our case, “keyfiles”
- String PASSWORD – in our case, “passphrase”

f) Create the context using SSLContext.getInstance(arg) method. You need to pass the variable ALGORITHM as argument.

g) As accepted the default, we don't need to create theTrustManagerFactory

Lab Work: Secure Sockets 3

- h) Create the KeyManagerFactory using the static method `KeyManagerFactory.getInstance (arg)`. The Sun implementation will need "SunX509" as argument.
- i) Create the KeyStore using the static method `KeyStore.getInstance(arg)` Use "JKS" as argument.
- j) For security, every key store is encrypted with a pass phrase that must be provided before we can load it from disk. The pass phrase is stored as a `char[]` array so it can be wiped from memory quickly rather than waiting for a garbage collector. Of course using a string literal here completely defeats that purpose.
- k) Use the load method from the KeyStore to load the key file. Check the javadoc for method usage.

Lab Work: Secure Sockets 4

- l) Use the init method from the KeyManagerFactory to initialize the KeyManagerFactory. Check the javadoc for method usage.
- m) Use the init method from the SSLContext to initialize the context. Check the javadoc for method usage.
- 2) You have completed the setup process at this point. Create a secure server socket for this server at port 7000.
- 3) Test your server with the secure client.

New I/O (NIO) API

Java introduce the new I/O (NIO) API in v1.4.

New features:

- a) Buffers for data of primitive types
- b) Character-set encoders and decoders
- c) A pattern-matching facility based on Perl-style regular expressions
- d) Channels, a new primitive I/O abstraction
- e) A file interface that supports locks and memory mapping
- f) A multiplexed, non-blocking I/O facility for writing scalable servers

Why NIO?

Allow Java programmers to implement high-speed I/O.

NIO deals with data in blocks which can be much faster than processing data by the (streamed) byte.

NIO Components

- 1) Buffers
- 2) Channels
- 3) Selectors
- 4) Regular Expressions
- 5) Character Set Coding

Buffers

In the NIO library, all data is handled with buffers.

A buffer is essentially an array. Generally, it is an array of bytes, but other kinds of arrays can be used.

A buffer also provides structured access to data and also keeps track of the system's read/write processes.

Types:

- a) ByteBuffer
- b) CharBuffer
- c) ShortBuffer
- d) IntBuffer
- e) LongBuffer
- f) FloatBuffer
- g) DoubleBuffer

Channels

Channel is like a stream in original I/O.

You can read a buffer from and write a buffer to a channel.

Unlike streams, channels are bi-directional.

Read from a file

Codes for reading from a file:

```
FileInputStream fin = new  
FileInputStream( "readandshow.txt" );  
FileChannel fc = fin.getChannel();  
ByteBuffer buffer = ByteBuffer.allocate( 1024 );  
fc.read( buffer );
```

Write to a file

Codes for writing to a file:

```
FileOutputStream fout = new
FileOutputStream( "writesomebytes.txt" );
FileChannel fc = fout.getChannel();
ByteBuffer buffer = ByteBuffer.allocate( 1024 );
for (int i=0; i<message.length; ++i) {
buffer.put( message[i] );
}
buffer.flip();//prepares the buffer to have the newly-
              //read data written to another channel
fc.write( buffer );
```


Lab Work: NIO

- 1) Refer to the example, please use the NIO to create a program to write a String to a text file and store in your computer.
- 2) Read the text file back and print the content on the screen.
- 3) You may need to use the WritableByteChannel as following:

```
WritableByteChannel wbc =  
    Channels.newChannel(System.out);
```

Server with NIO

Channels and buffers are really intended for server systems that need to process many simultaneous connections efficiently.

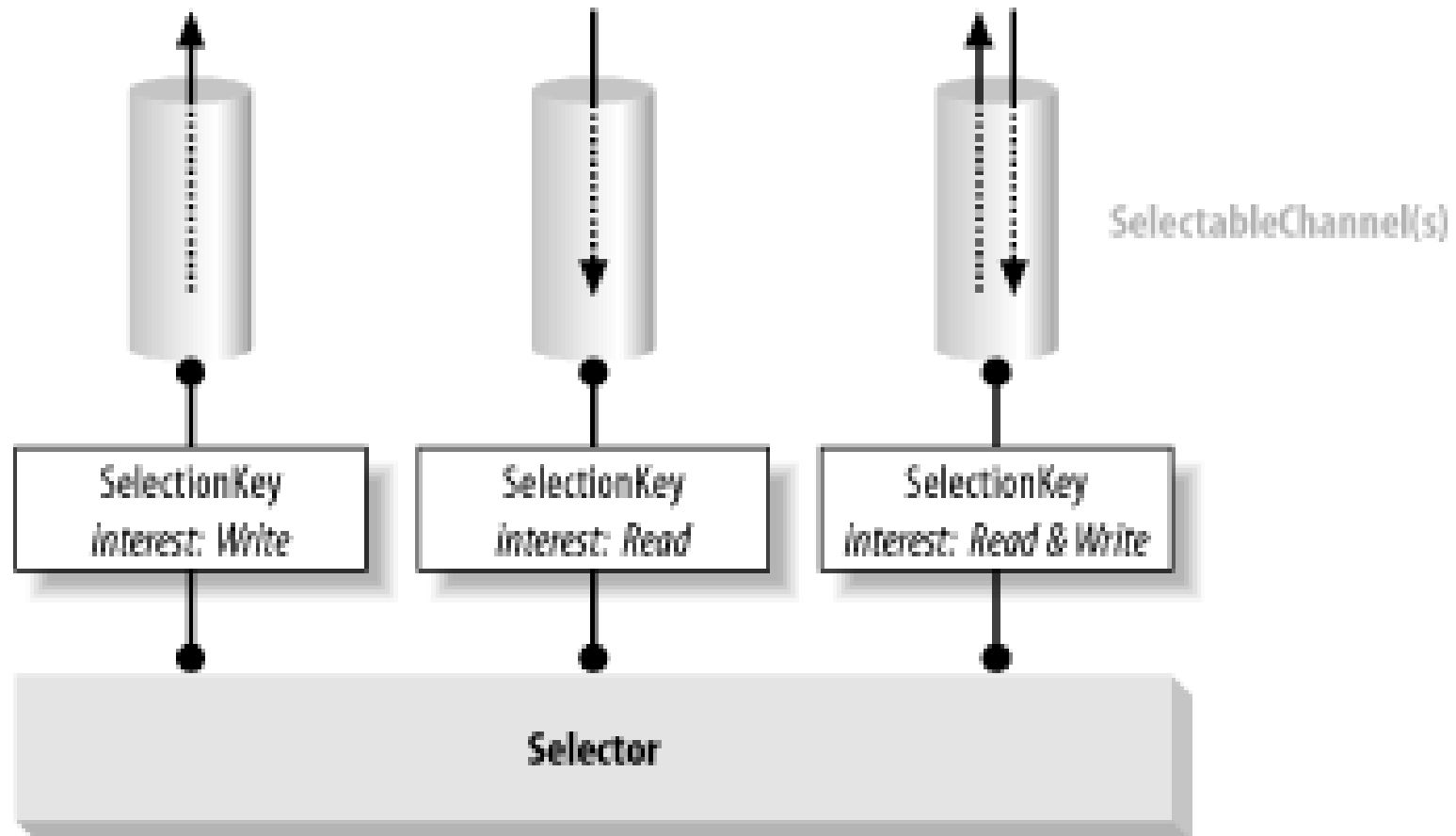
Handling servers requires the new selectors that allow the server to find all the connections that are ready to receive output or send input.

Asynchronous I/O

Asynchronous I/O is made possible in NIO with scalable sockets, which consist of the following components:

- a) Selectable Channel - A channel that can be multiplexed
- b) Selector - A multiplexor of selectable channel
- c) Selection key - A token representing the registration of a selectable channel with a selector

Selectors, Keys and Channels



The Selection Process 1

- 1) Create a Selector and register channels with it

Use `Open()` method to create a Selector.

The `register()` method is on `SelectableChannel`, not `Selector`

- 2) Invoke `select()` method on the `Selector` object

- 3) Retrieve the Selected Set of keys from the Selector

Selected set: Registered keys with non-empty Ready Sets

```
keys = selector.selectedKeys()
```

The Selection Process 2

4) Iterate over the Selected Set

- 1) Check each key's Ready Set

- 2) Remove the key from the Selected Set (`iterator.remove()`)

- 1) Bits in the Ready Sets are never reset while the key is in the Selected Set

- 2) The Selector never removes keys from the Selected Set – you must do so

- 3) Service the channel (`key.channel()`) as appropriate (read, write, etc)

Example : NIO Server 1

Skeleton codes for a simple server with NIO:

```
//Open a ServerSocketChannel

ServerSocketChannel serverChannel =
ServerSocketChannel.open( );

//make the ServerSocketChannel non-blocking.
//necessary for asynchronous i/o

serverChannel.configureBlocking(false);

ServerSocket ss = serverChannel.socket( );

// bind the socket to a specific port
ss.bind(new InetSocketAddress(PORT_NO));
```

Example : Create NIO Server 2

```
//Open the selector
```

```
Selector selector = Selector.open( );
```

```
//use the channel's register() method to register the  
//ServerSocketchannel with the selector.
```

```
serverChannel.register(selector,  
SelectionKey.OP_ACCEPT);
```


Example : Create NIO Server 3

```
//check whether anything is ready to be acted on, call  
//the selector's select( ) method. For a long-running  
//server, this normally goes in an infinite loop:
```

```
while (true) {  
    try {  
        selector.select( );  
    }  
    catch (IOException ex) {  
        ex.printStackTrace( );  
        break;  
    }  
}
```

Example : Create NIO Server 4

```
// process selected keys...

//selectedKeys( ) method returns a java.util.Set
//containing one SelectionKey object for each ready
//channel

Set readyKeys = selector.selectedKeys( );
Iterator iterator = readyKeys.iterator( );
while (iterator.hasNext( )) {

    SelectionKey key = (SelectionKey)
        (iterator.next( ));

    // Remove key from set
    iterator.remove( );
}
```

Example : Create NIO Server 5

```
// You can obtain the channel using the channel()  
//methods of the SelectionKey. Catch the IOException.  
try{  
    if (key.isAcceptable( ))  
    { ServerSocketChannel  
        server = (ServerSocketChannel ) key.channel( );  
        SocketChannel client = server.accept( );  
        // Data manipulation  
        System.err.println("Got connection from  
"+client.socket().getInetAddress().getHostName());
```

Example : Create NIO Server 6

```
//Send some message to client

ByteBuffer bb =
ByteBuffer.allocateDirect(1024);

byte[] message = "Hello... You are
reaching NIO Server".getBytes();

bb.put( message);

bb.flip();

client.write( bb);

}

} catch (IOException e) {

}

}
```

User Datagram Protocol

The User Datagram Protocol (UDP) is an alternative protocol for sending data over IP

UDP is very quick, but not reliable.

Java's implementation of UDP is split into two classes: `DatagramPacket` and `DatagramSocket`.

The `DatagramPacket` class stuffs bytes of data into UDP packets called datagrams and lets you unstuff datagrams that you receive.

A `DatagramSocket` sends as well as receives UDP datagrams.

Constructors for DatagramPacket

For receiving datagrams:

- a) `public DatagramPacket(byte[] buffer, int length)`
- b) `public DatagramPacket(byte[] buffer, int offset, int length)`

For sending datagrams:

- a) `public DatagramPacket(byte[] data, int length, InetAddress destination, int port)`
- b) `public DatagramPacket(byte[] data, int offset, int length, InetAddress destination, int port)`
`// Java 1.2`
- c) `public DatagramPacket(byte[] data, int length, SocketAddress destination, int port)` `// Java 1.4`
- d) `public DatagramPacket(byte[] data, int offset, int length, SocketAddress destination, int port)`
`//Java 1.4`

Constructors for DatagramSocket

For socket bound to an anonymous port:

a) `public DatagramSocket() throws SocketException`

For socket listen for incoming datagrams on a particular port :

a) `public DatagramSocket(int port) throws
SocketException`

Others constructors:

a) `public DatagramSocket(int port, InetAddress
interface) throws SocketException`

b) `public DatagramSocket(SocketAddress interface)
throws SocketException // Java 1.4`

c) `protected DatagramSocket(DatagramSocketImpl impl)
throws SocketException // Java 1.4`

Sending and Receiving

After constructed the DatagramPacket, you can send and receive datagram from it :

a) `public void send(DatagramPacket dp) throws
IOException`

b) `public void receive(DatagramPacket dp) throws
IOException`

Lab Work: UDP

- 1) Write a EchoUDP server which will send back any message received from client.
- 2) Write a UDPclient which sends some message to the EchoUDP server for testing it.

Summary

In this session, we cover the followings:

- 1) Review the basic concepts for networking.
- 2) Discuss how to create simple server-client program.
- 3) Discuss the secure socket implementation in Java.
- 4) Introduce the NIO API.
- 5) Introduce the Java implementation for User Datagram Protocol.

Database Connectivity

Course Outline

- 1) introduction
- 2) streams
- 3) networking
- 4) database connectivity
- 5) architectures
 - a) message-orientation
 - 1) javamail
 - 2) jms
 - b) distributed objects
 - 1) rmi
 - 2) corba
 - 3) JavaIDL
- 6) summary

Overview

We are going to consider the following under this section:

- 1) Introduction
- 2) JDBC Overview
- 3) Information Processing

Introduction

Buried within the term "enterprise" is the idea of a business taken wholistically.

An enterprise solution identifies common problem domains within a business and provides a shared infrastructure for solving those problems.

Example:

If your business is running a bank, your individual branches may have different business cultures, but those cultures do not alter the fact that they all deal with **customers** and **accounts**. Looking at this business from an enterprise perspective means abstracting away from irrelevant differences in the way the individual branches do things, and instead approaching the business from their common ground. It does not mean dismissing meaningful distinctions, such as the need for bilingual support in Macao SAR.

Enterprise Systems: What?

Enterprise Systems are Information systems that support many or all of the various parts of a firm.

They can also refer to many mission-critical applications which are mainframe-based (also referred to as legacy systems).

Also known as enterprise-wide information systems.

Information systems that allow companies to integrate information across operations on a company-wide basis.

Enterprise Systems: Types

Enterprise systems are broadly categorized into two:

1) Relational - Relational Database Management Systems (RDBMS)

2) Non-Relational

- a) Non Relational Databases (Legacy Database Systems)
- b) Legacy Systems (Older systems like old Cobol Applications)
- c) Enterprise Resource Planning
- d) Customer Relationship Management (CRM)
- e) Supply Chain Management

Enterprise Systems Integration

Enterprise Systems Integration is normally defined as the bringing together of:

- 1) People,
- 2) Processes and
- 3) Information

to work together in an harmonized way, and supported by appropriate information systems.

It is also the bringing together of both old and new applications to achieve the overall goal of an organization.

Reasons for Integration

There are many reasons why Enterprise information systems need to be integrated. Some are stated below:

- 1) need to persist and retrieve information from data repositories.
- 2) need to leverage existing systems and resources while adopting and developing new technologies and architectures
- 3) concern over the past years, that the mainframe was going away and that all legacy applications would be scrapped and completely rewritten.

Java Integration Mechanisms

Java provides some technologies to integrate Enterprise systems:

- 1) Java Database connectivity (JDBC) for connecting applications to relational Database systems
- 2) JavaIDL and Java Connector Architecture (JCA) for connecting to Non-Relational systems

The focus of this section is JDBC.

JavaIDL will be addressed later in the course while JCA will be addressed sometimes in the training.

Relational Database 1

Programming is all about data processing; data is central to everything you do with a computer.

Databases, like filesystems are nothing more than specialized tools for data storage.

Filesystems are good for storing and retrieving a single volume of information associated with a single virtual location.

In other words, when you want to save a WordPerfect document, a filesystem allows you to associate it with a location in a directory tree for easy retrieval later.

Relational Database 2

Databases provide applications with a more powerful data storage and retrieval system based on mathematical theories about data devised by Dr. E. F. Codd.

Conceptually, a relational database can be pictured as a set of spreadsheets in which rows from one spreadsheet can be related to rows from another.

Each spreadsheet in a database is called a table. As with a spreadsheet, a table is made up of rows and columns.

A database engine is a process instance of the software accessing your database. For example Oracle, MySQL, Sybase etc

Database engines use a standard query language to retrieve information from databases and is called Structured Query Language (SQL).

SQL

SQL is not much like any programming language you might be familiar with.

Instead, it is more of a structured English for talking to a database.

Characteristics:

- 1) SQL keywords are case-insensitive
- 2) table and column names may or may not be case-insensitive depending on your database engine
- 3) the space between words in a SQL statement is unimportant
- 4) have a newline after each word, several spaces, or just a single space

SQL Usage

With SQL you can ask the following question:

- 1) How do you get the data into the database?
- 2) And how do you get it out once it is in there?

Much of the simplest database access comes in the form of equally simple SQL statements.

Some of these commands are:

- 1) Create
- 2) Insert
- 3) Select
- 4) Update
- 5) Delete

Create Statement

SQL `CREATE` statement handles the creation of database entities.

The major database engines provide GUI utilities that allow you to create tables without issuing any SQL.

Uses:

1) To create database

Syntax:

```
CREATE DATABASE database_name
```

2) To create tables

Syntax:

```
CREATE TABLE table_name (  
    column_name column_type column_modifiers,  
    ...,  
    column_name column_type column_modifiers)
```


Lab Work: Create Statement

2) Open another console and type

```
C:\mysql\bin>mysql -u root
```

2) Test your connection by typing

```
mysql>show databases;
```

5) Create emacao database

```
mysql>create database emacao;
```

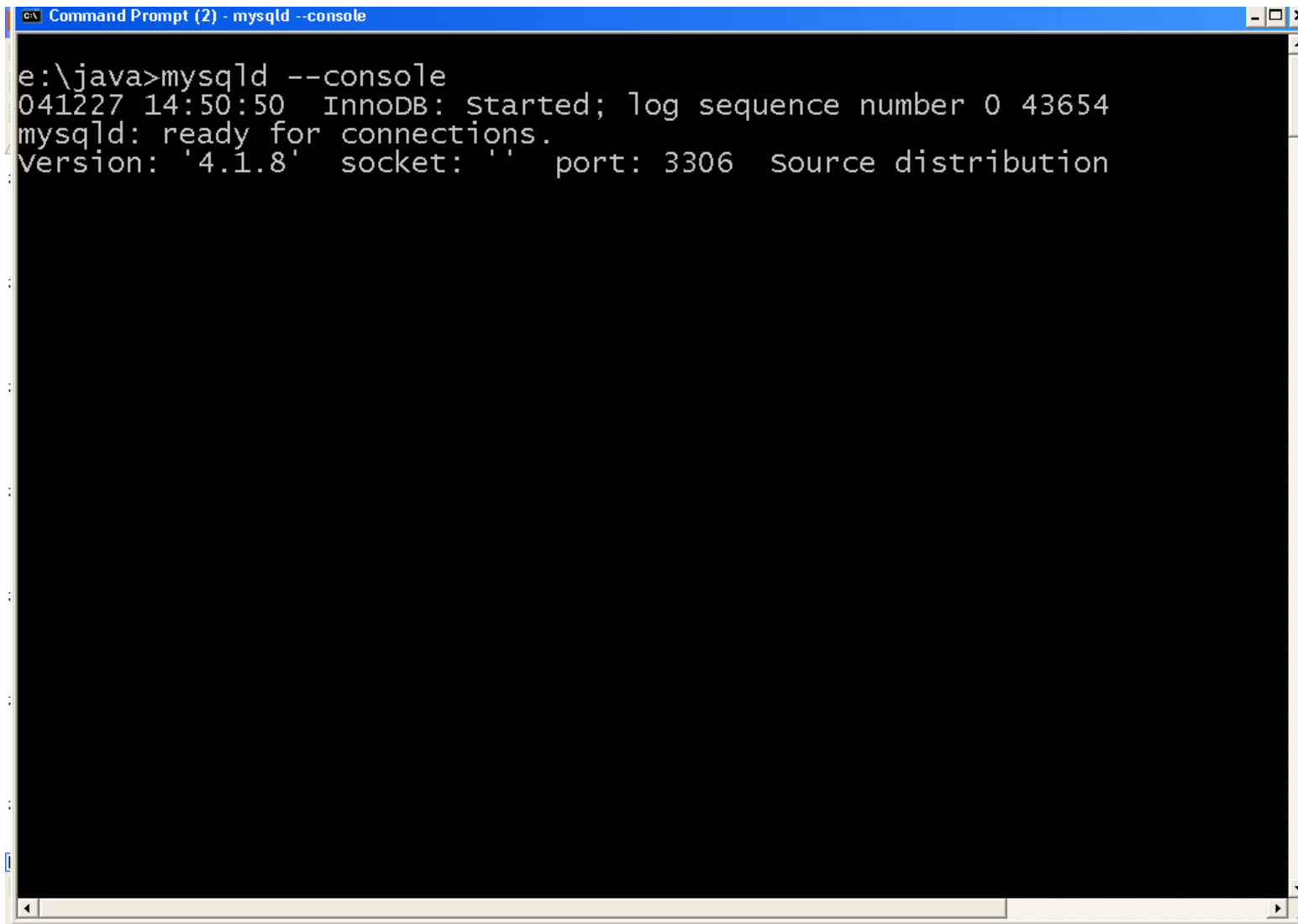
6) Change to the database

```
mysql>Use emacao;
```

6) Create license table

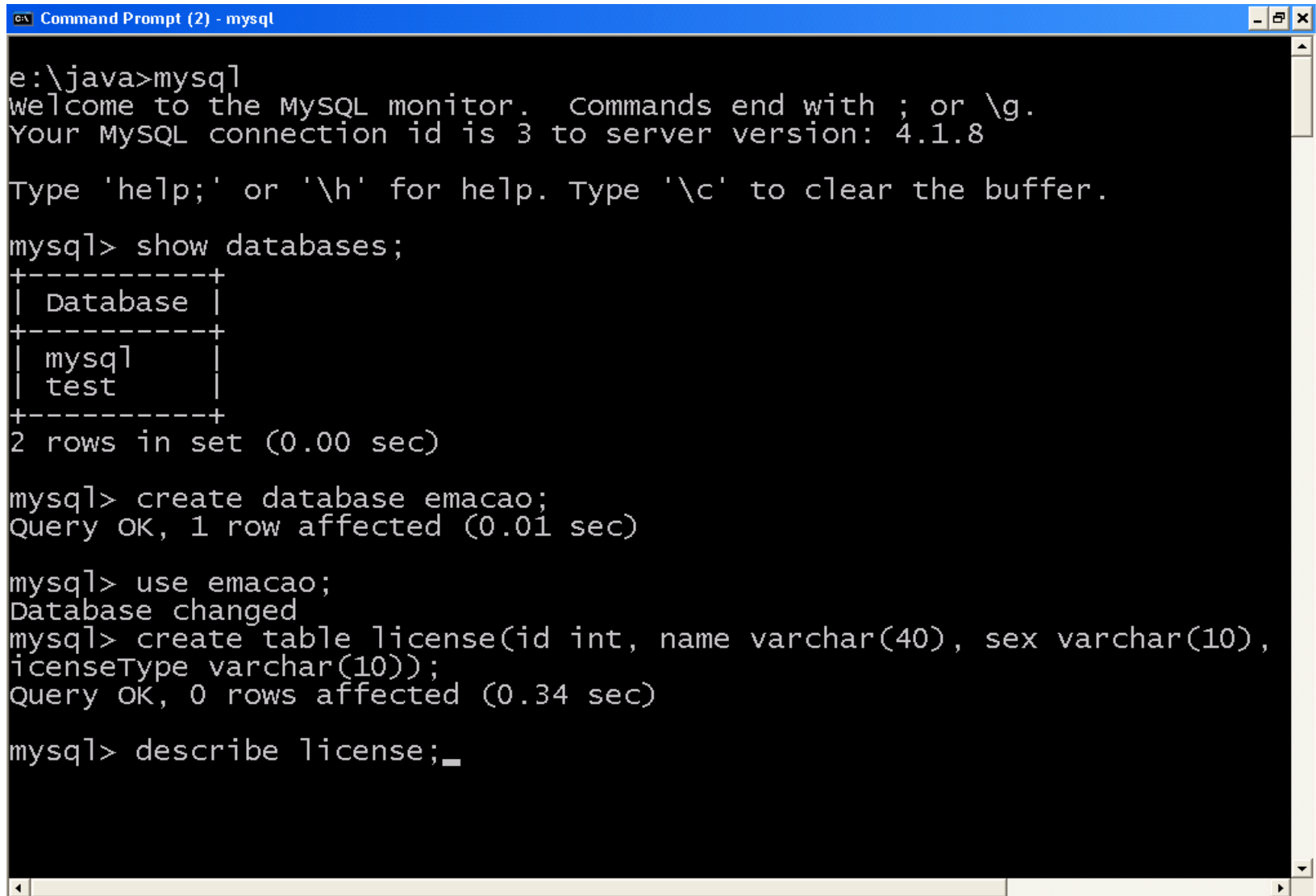
```
mysql>create table license (id int, name  
varchar(40), sex varchar(10), date varchar(12),  
licenseType varchar(10));
```

Lab Work: Console 1



```
Command Prompt (2) - mysqld --console
e:\java>mysqld --console
041227 14:50:50 InnoDB: Started; log sequence number 0 43654
mysqld: ready for connections.
Version: '4.1.8' socket: '' port: 3306 source distribution
```

Lab Work: Console 2



```
GA Command Prompt (2) - mysql

e:\java>mysql
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 3 to server version: 4.1.8

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> show databases;
+-----+
| Database |
+-----+
| mysql   |
| test    |
+-----+
2 rows in set (0.00 sec)

mysql> create database emacao;
Query OK, 1 row affected (0.01 sec)

mysql> use emacao;
Database changed
mysql> create table license(id int, name varchar(40), sex varchar(10),
licenseType varchar(10));
Query OK, 0 rows affected (0.34 sec)

mysql> describe license;_
```

Lab Work: Console 3

```
Command Prompt (2) - mysql
+-----+
| Database |
+-----+
| mysql   |
| test    |
+-----+
2 rows in set (0.00 sec)

mysql> create database emacao;
Query OK, 1 row affected (0.01 sec)

mysql> use emacao;
Database changed
mysql> create table license(id int, name varchar(40), sex varchar(10),
licenseType varchar(10));
Query OK, 0 rows affected (0.34 sec)

mysql> describe license;
+-----+-----+-----+-----+-----+-----+
| Field      | Type          | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| id         | int(11)       | YES  |     | NULL    |       |
| name       | varchar(40)   | YES  |     | NULL    |       |
| sex        | varchar(10)   | YES  |     | NULL    |       |
| date       | varchar(12)   | YES  |     | NULL    |       |
| licenseType | varchar(10)   | YES  |     | NULL    |       |
+-----+-----+-----+-----+-----+-----+
5 rows in set (0.00 sec)

mysql>
```

Insert Statement

With the tables in place, you use the INSERT statement to add data to them.

Its form is:

```
INSERT INTO table_name(column_name, ..., column_name)
VALUES (value, ..., value)
```

The first column name matches to the first value you specify, the second column name to the second value you specify, and so on for as many columns as you are inserting.

If you fail to specify a value for a column that is marked as `NOT NULL`, you will get an error on insert.

Lab Work: Insert Statement

1) Insert the following records into the license table.

a) Id = 123

Name = Chong Gabriel

Sex = male

Date = 27/12/2004

LicenceType = Export

b) Id = 124

Name = martins Gabriel

Sex = Female

Date = 23/12/2004

LicenceType = Import

Update Statement

The `UPDATE` statement enables you to modify data that you previously inserted into the database.

Its form is:

```
UPDATE table_name
SET column_name = value,
...,
column_name = value
WHERE column_name = value
```

This statement introduces the `WHERE` clause. It is used to help identify one or more rows in the database.

Lab Work: Update Statement

- 1) Change the ID and Name of the record with ID 124 to 126 and Martins Leo Gabriel respectively.

Select Statement

The most common SQL command you will use is the `SELECT` statement.

It allows you to select specific rows from the database based on search criteria.

It takes the following form:

```
SELECT column_name, ..., column_name  
FROM table_name  
WHERE column_name = value
```

Lab Work: Select Statement

- 1) Retrieve all records from the table.
- 2) Retrieve all records from the table where `ID` is `126`.

Delete Statement

The `DELETE` command looks a lot like the `UPDATE` statement.

Its syntax is:

```
DELETE FROM table_name WHERE column_name = value
```

Instead of changing particular values in the row, `DELETE` removes the entire row from the table.

Lab Work: Delete Statement

- 1) Remove all records from the table where `ID` is 125.
- 2) Retrieve all records from the table
- 3) Remove all records from the table.

Database Programming

Database programming has traditionally been a technological Tower of Babel.

You are faced with dozens of available database products, and each one talks to your applications in its own private language.

If your application needs to talk to a new database engine, you have to teach it (and yourself) a new language.

As Java programmers, however, you should not worry about such translation issues.

Java is supposed to bring you the ability to "write once, compile once, and run anywhere," so it should bring it to you with database programming, as well.

JDBC Overview

JDBC API is a set of interfaces designed to insulate a database application developer from a specific database vendor.

It 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.

Sun developed a single API for database access—JDBC.

Three main design goals:

- 1) JDBC should be a SQL-level API.
- 2) JDBC should capitalize on the experience of existing database APIs.
- 3) JDBC should be simple.

JDBC and Developer

What does JDBC provide the developer?

- 1) 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
- 2) the developer writes an application using the interfaces described in the API as though they are actual class implementations
- 3) 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 and Driver Vendors

What do the driver vendors provide?

Driver vendors provide implementations of JDBC interfaces.

The JDBC API also enables developers to pass any string directly to the driver.

This makes it possible for developers to make use of custom features of their database without requiring that the application use ANSI SQL

With JDBC you can :

- 1) establish a connection with a database or access any tabular data source
- 2) send SQL statement
- 3) process the results

JDBC Structure

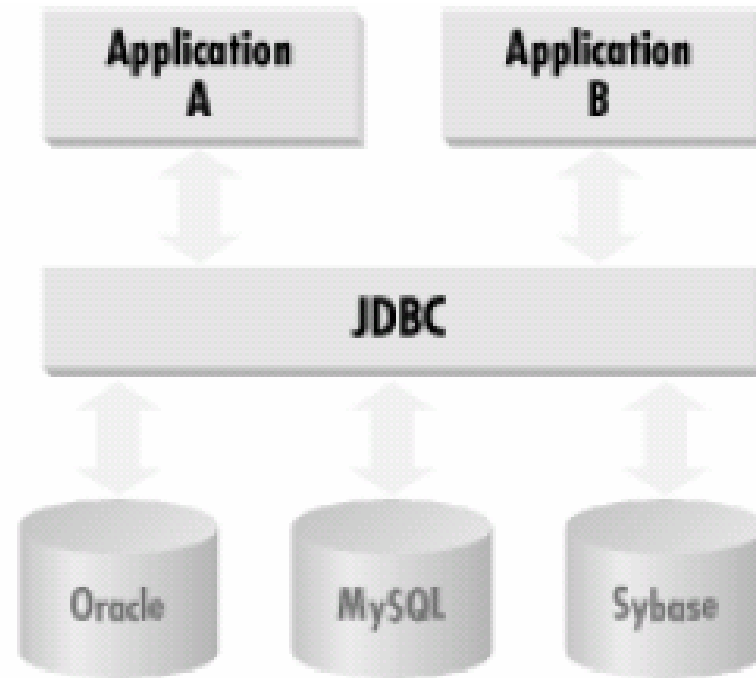
JDBC accomplishes its goals through a set of Java interfaces, each implemented differently by individual vendors.

The set of classes that implement the JDBC interfaces for a particular database engine is called a JDBC driver.

In building a database application, you do not have to think about the implementation of these underlying classes at all.

The whole point of JDBC is to hide the specifics of each database and let you worry about just your application.

JDBC Architecture



JDBC Driver Type 1

JDBC Driver fit into one of the following:

- 1) Type 1:JDBC-ODBC Bridge plus ODBC Driver
- 2) Type 2:A native API partly Java technology-enabled
- 3) Type 3:Pure Java Driver for Database Middleware
- 4) Type 4:Direct-to-Database Pure Java Driver

Type 1: JDBC-ODBC Bridge 1

Type 1: JDBC-ODBC Bridge provides JDBC access via one or more Open Database Connectivity (ODBC) drivers.

Advantage:

- 1) a good approach for learning JDBC
- 2) may be useful for companies that already have ODBC drivers installed on each client machine
- 3) may be the only way to gain access to some low-end desktop databases

Type 1: JDBC-ODBC Bridge 2

Disadvantage:

- 1) Not for large-scale applications. Performance suffers because there's some overhead associated with the translation work to go from JDBC to ODBC.
- 2) doesn't support all the features of Java
- 3) user is limited by the functionality of the underlying ODBC driver

Type 2: Partial Java driver 1

Converts calls to the JDBC API into calls that connect to the client machine's application programming interface for a specific database, such as IBM, Informix, Oracle or Sybase.

Advantage:

- 1) Performance is better than that of Type 1, in part because the Type 2 driver contains compiled code that is optimized for the back-end database server's operating system.

Type 2: Partial Java driver 2

Disadvantage:

- 1) user needs to make sure the JDBC driver of the database vendor is loaded onto each client machine
- 2) must have compiled code for every operating system that the application will run on
- 3) best use is for controlled environments, such as an intranet

Type 3: Pure Java Middleware 1

Type 3: Pure Java driver for database middleware translates JDBC calls into the middleware vendor's protocol, which is then converted to a database-specific protocol by the middleware server software.

Advantage:

- 1) used when a company has multiple databases and wants to use a single JDBC driver to connect to all of them
- 2) Server-based, so no need for JDBC driver code on client machine
- 3) the back-end server component is optimized for the operating system that the database is running on

Type 3: Pure Java Middleware 2

Type 3: Pure Java driver for database middleware translates JDBC calls into the middleware vendor's protocol, which is then converted to a database-specific protocol by the middleware server software.

Advantage:

- 1) used when a company has multiple databases and wants to use a single JDBC driver to connect to all of them
- 2) Server-based, so no need for JDBC driver code on client machine
- 3) the back-end server component is optimized for the operating system that the database is running on

Disadvantage:

- 1) Needs some database-specific code on the middleware server.

Type 4: Direct-to-database Pure

Type 4: Direct-to-database pure Java driver converts JDBC calls into packets that are sent over the network in the proprietary format used by the specific database. Allows a direct call from the client machine to the database.

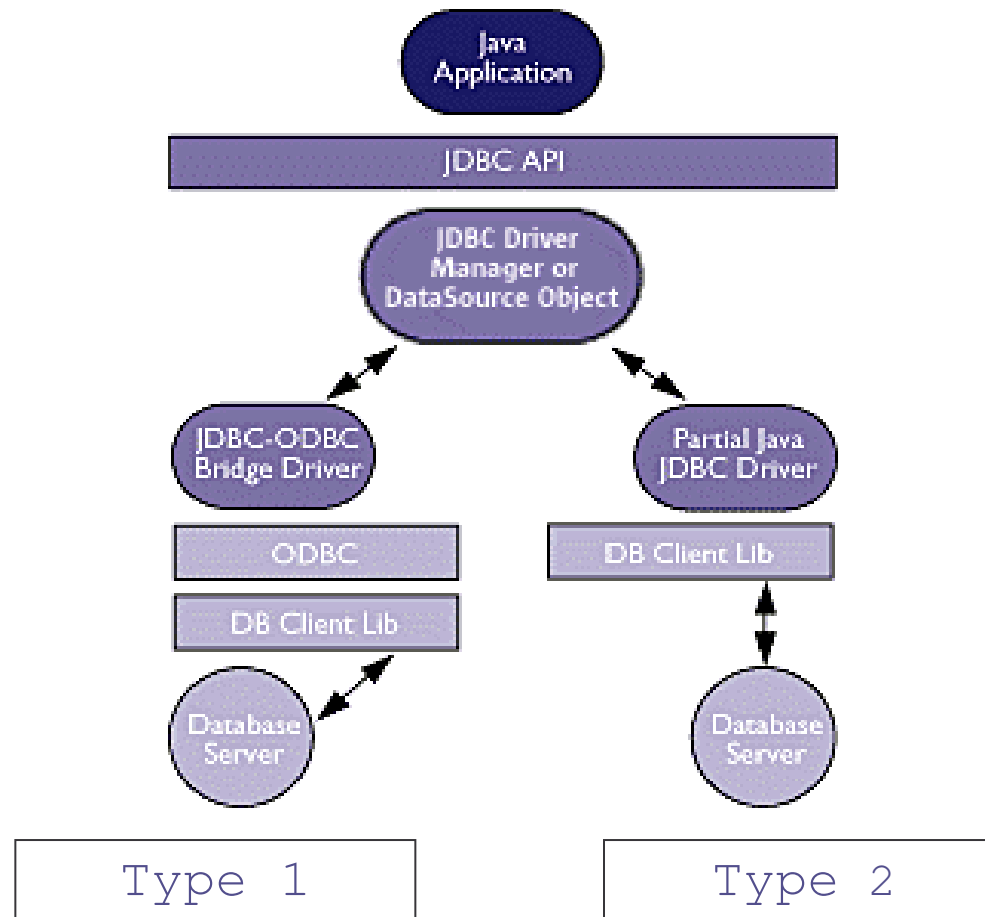
Advantage:

1) No need to install special software on client or server. Can be downloaded dynamically.

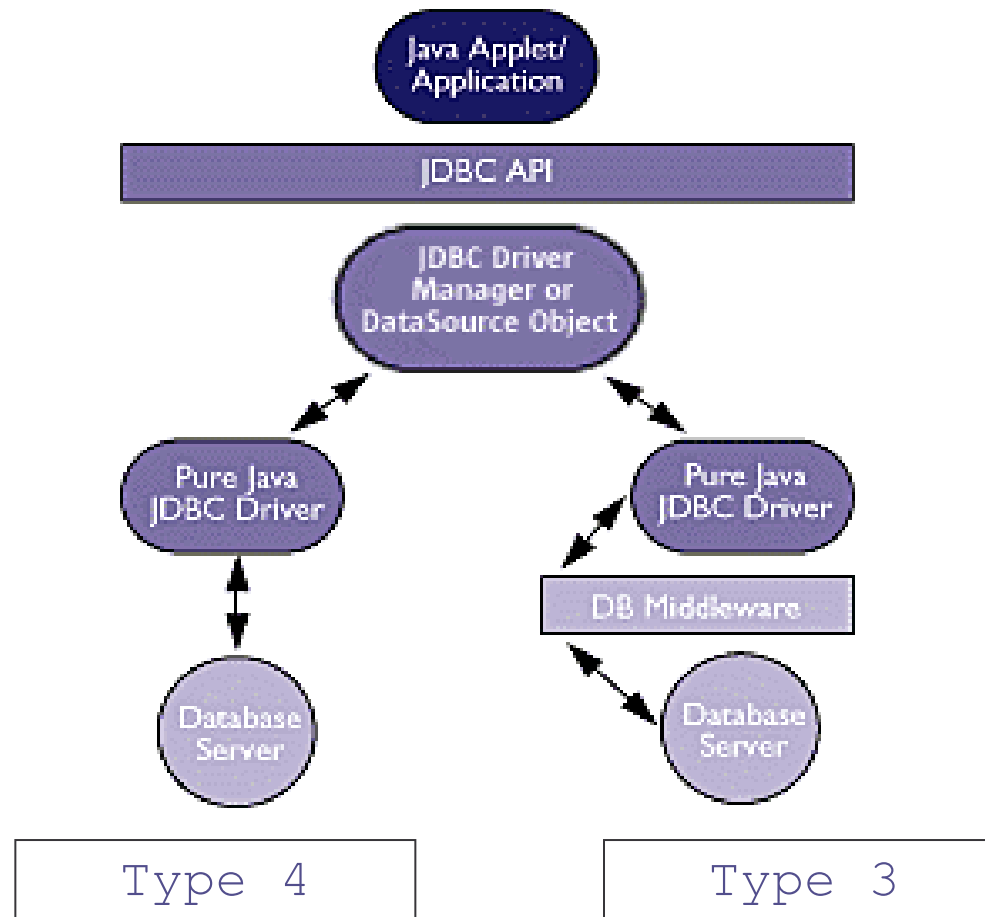
Disadvantage:

1) not optimized for server operating system, so the driver can't take advantage of operating system features

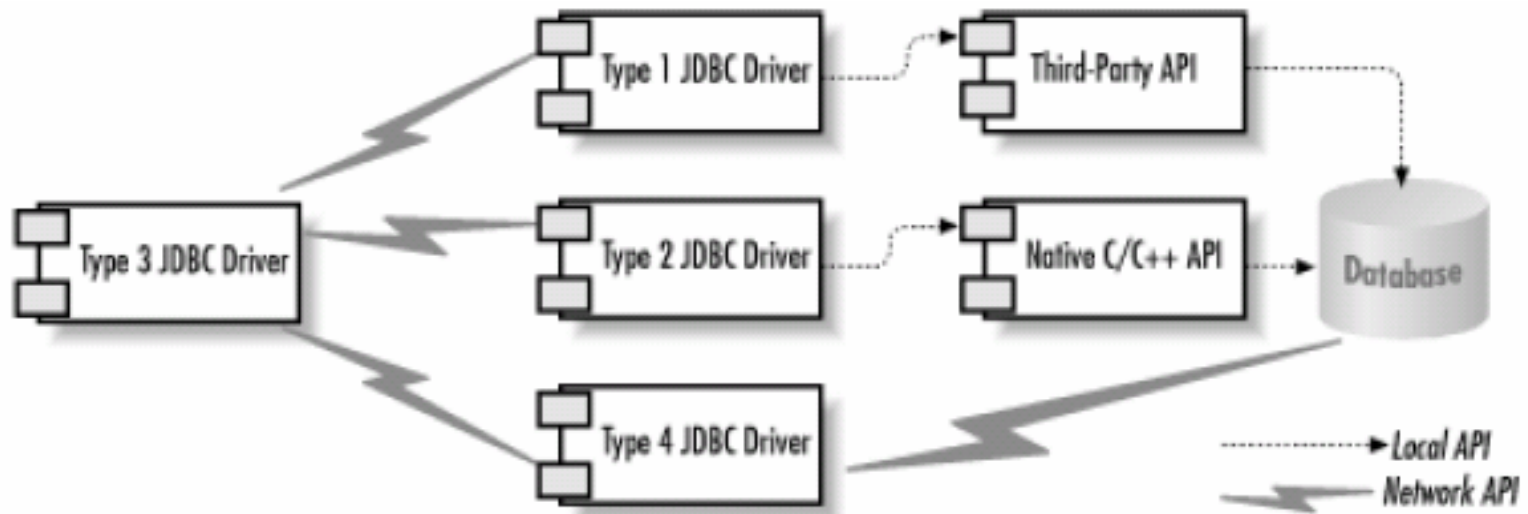
JDBC Driver Type 2



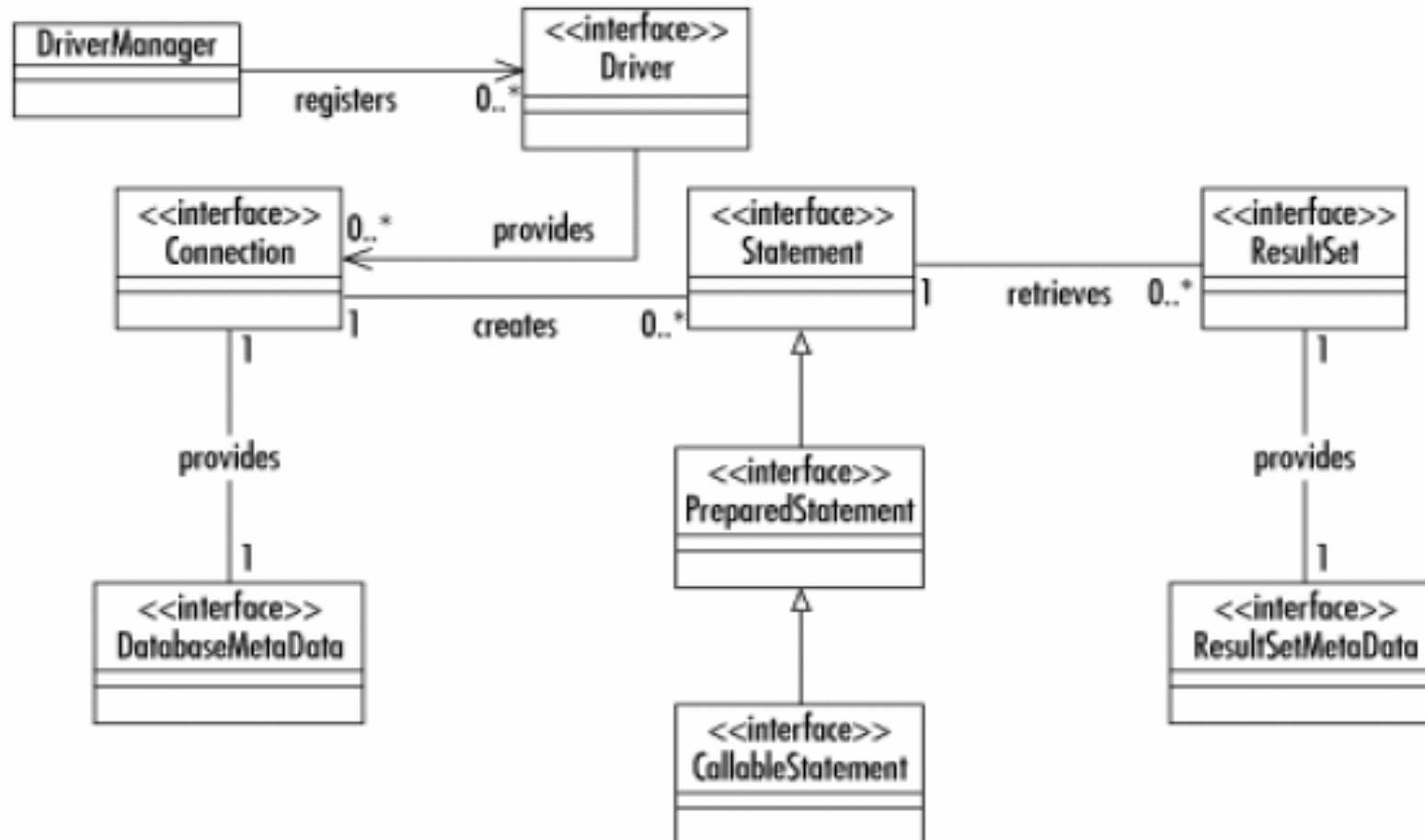
JDBC Driver Type 3



JDBC Drivers

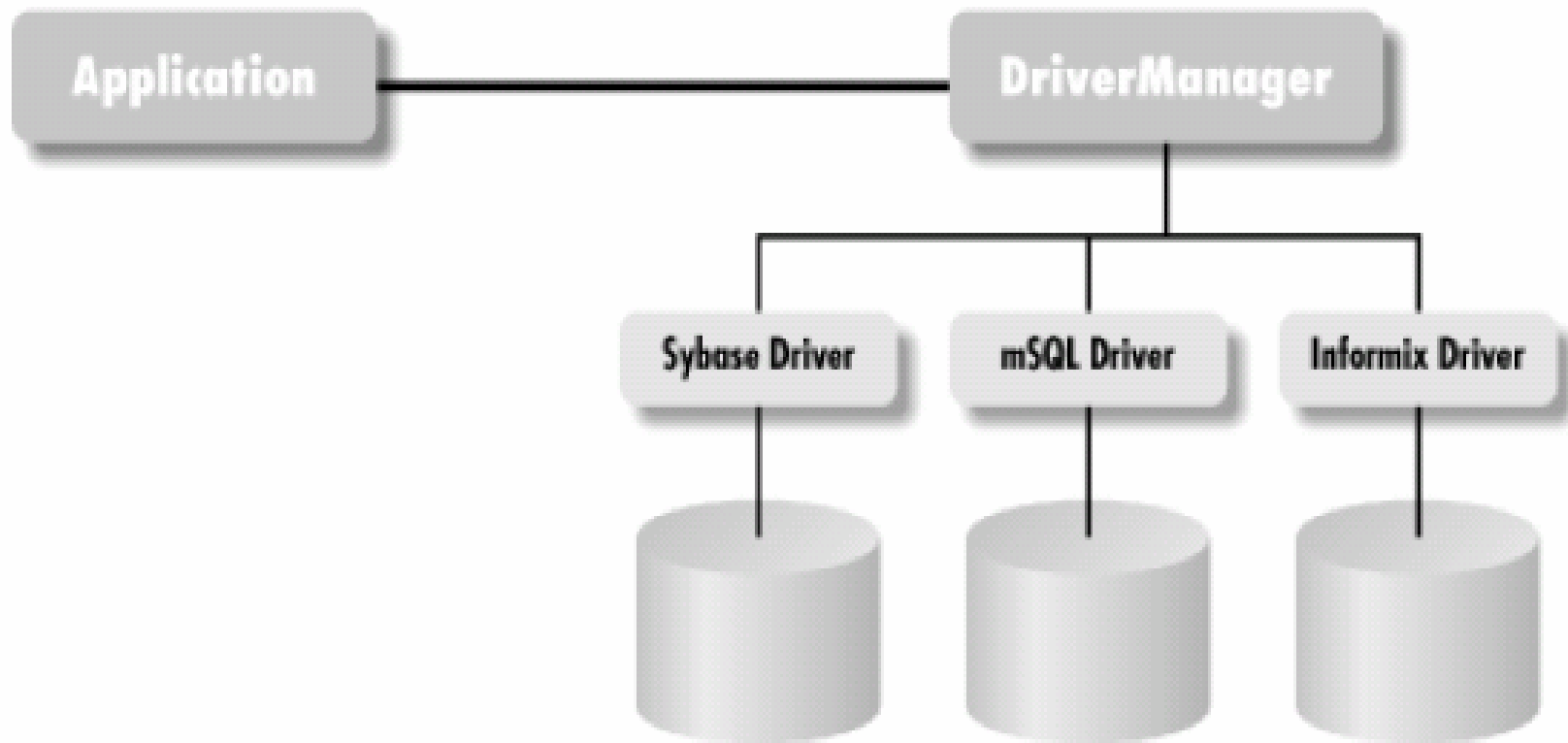


JDBC Class Diagram



Connecting to Database

JDBC shields an application from the specifics of individual database implementation.



Connection Troubles

The JDBC Connection process is the most difficult part of JDBC to get right.

There are generally two basic connection problems:

1) Connection fails with the message "Class not found"

Solution: Set your JDBC driver in your `CLASSPATH`

2) Connection fails with the message "Driver not found"

Solution: register the JDBC driver with the `DriverManager` class

Connection Process 1

When you write a Java database applet or application, the only driver-specific information JDBC requires from you is the database URL.

You can even have your application derive the URL at runtime—based on user input or applet parameters.

What happens when the URL and whatever properties the JDBC driver requires (generally a user ID and password) is passed?

- 1) the application will first request a `java.sql.Connection` implementation from the `DriverManager`
- 2) the `DriverManager` in turn will search through all of the known `java.sql.Driver` implementations for the one that connects with the URL you provided

Connection Process 2

- 3) if it exhausts all the implementations without finding a match, it throws an exception back to the application
- 4) once a `Driver` recognizes the URL, it creates a database connection using the properties specified
- 5) it then provides the `DriverManager` with a `java.sql.Connection` implementation representing that database connection
- 6) the `DriverManager` then passes that `Connection` object back to the application
- 7) the entire database connection process is handled by these two lines

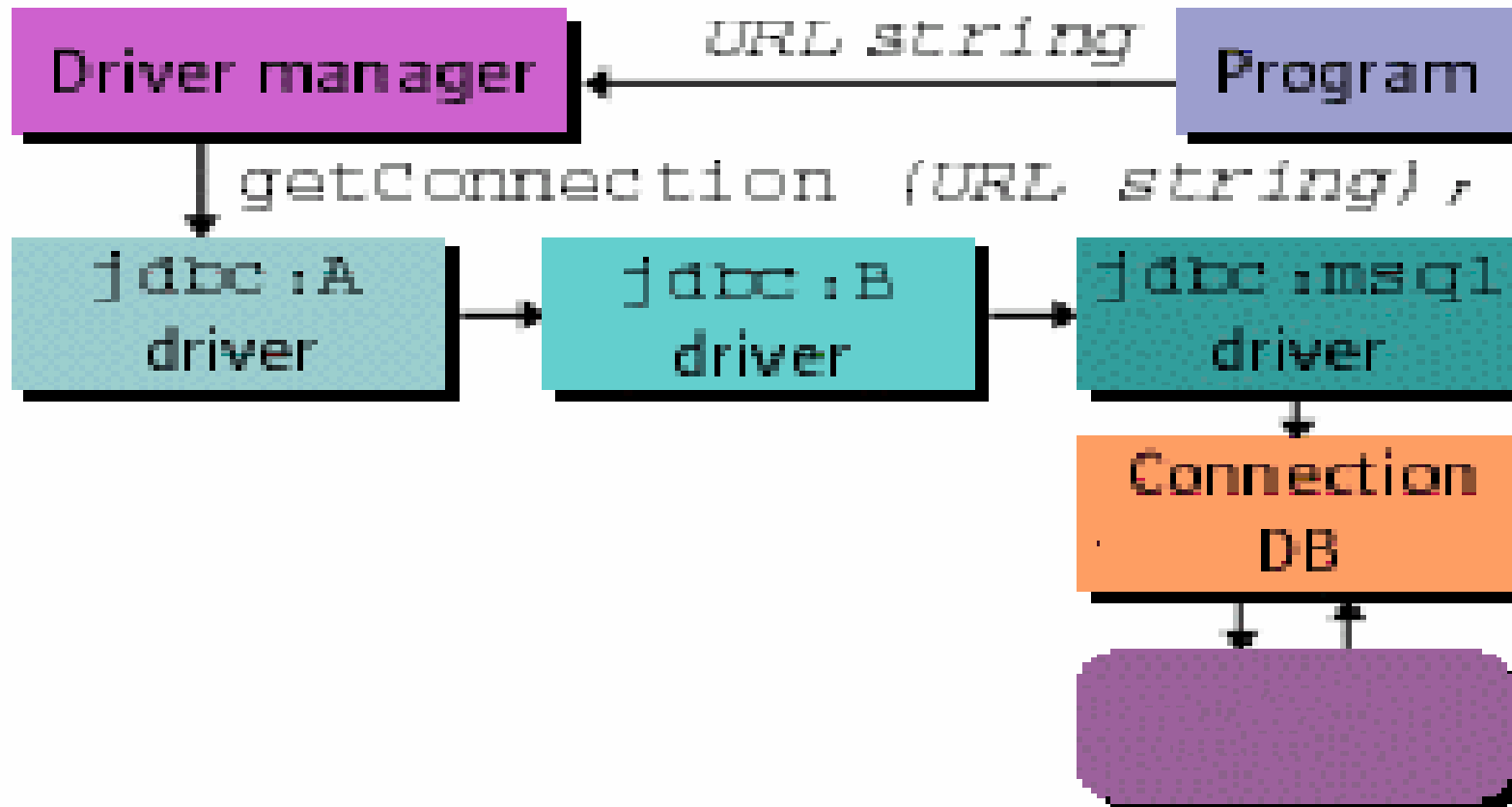
```
Connection con = null;  
con = DriverManager.getConnection(url, uid, password);
```

Connection Process 3

How does the JDBC `DriverManager` learn about a new driver implementation?

- 1) the `DriverManager` actually keeps a list of classes that implement the `java.sql.Driver` interface
- 2) `Driver` implementations has to be registered for any potential database drivers it might require with the `DriverManager`
- 3) The act of instantiating a `Driver` class thus enters it in the `DriverManager`'s list
- 4) The process is called **Driver Loading**

Connection Process 4



Loading JDBC Drivers

There are three basic ways of loading the drivers:

1) explicitly call new to load your driver's implementation of `Driver`

2) use the `jdbc.drivers` property

```
>java -Djdbc.drivers=jdbc.odbc.JdbcOdbcDriver queryDB
```

3) load the class using `Class.forName`

```
Class.forName("com.mysql.jdbc.Driver").newInstance();
```

Class for Creating a Connection 1

A class and two Interfaces are used for creating a connection to a database:

1) `java.sql.Driver`

- a) unless you are writing your own JDBC implementation, you should never have to deal with this class from your application
- b) a launching point for database connectivity by responding to `DriverManager` connection requests and providing information about the implementation in question

Class for Creating a Connection 2

2) `java.sql.DriverManager`

- a) Its main responsibility is to maintain a list of `Driver` implementations and present an application with one that matches a requested URL.
- b) has two methods `registerDriver()` and `deregisterDriver()`
- c) the methods allow `Driver` implementation to register and unregister itself with the `DriverManager`
- d) You can get an enumeration of registered drivers through the `getDrivers()` method

3) `java.sql.Connection`

- a) The `Connection` class represents a single logical database connection.

Example: Simple Connection 1

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class SimpleConnection {
    static public void main(String args[]) {
        Connection connection = null;
        // Process the command line
        if( args.length != 4 ) {
            System.out.print("Syntax: java SimpleConnection ");
            System.out.println("DRIVER URL UID PASSWORD");
            return;
        }
    }
}
```


Example: Simple Connection 2

```
try { // load the driver
    Class.forName(args[0]).newInstance( );
}catch( Exception e ) {
    e.printStackTrace( );
    return;
}
try {
    connection = DriverManager.getConnection(args[1],
                                              args[2], args[3]);

    System.out.println("Connection successful!");
    // Do whatever queries or updates you want here!!!
}catch( SQLException e ) {
    e.printStackTrace( );
}
```

Example: Simple Connection 3

```
`finally {  
    if( connection != null ) {  
        try { connection.close( );  
        }catch( SQLException e ) {  
            e.printStackTrace( );  
        }  
    }  
}  
}
```

Lab Work: Creating Connection

- 1) Open `LicenseApp.java` file stored on the server
- 2) The program creates an interface for you to enter your license information into the database you created. Study the code and understand what it does.
- 3) Import appropriate packages into the program. Locate `connectToDB()`
- 4) Write a code to connect to the database you have created in MySQL using the following parameters

```
url = "jdbc:mysql://localhost/emacao"  
username = root, Password = ""  
Driver = "com.mysql.jdbc.Driver"
```

Note: set your `classpath` to the jar files provided along with the code. Print out the `Connection` object.

Database Access

The most basic kind of database access involves writing

- 1) updates- `INSERT`, `UPDATE`, or `DELETE`
- 2) queries – `SELECT`

With these you know ahead of time the type of statements you are sending to the database.

Database Access Steps

Accessing database involves:

- 1) creating a `Connection` object
- 2) generating implementation of `java.sql.Statement` tied to the database
- 3) use the statement to rollback or commit the statement object associated with that `Connection`
- 4) with the `Statement` object you can execute updates and queries
- 5) The result of executing queries and update is `java.sql.ResultSet`
- 6) `ResultSet` provides you with access to the data retrieved by a query.

Basic JDBC Classes

JDBC's most fundamental classes are :

- 1) `java.sql.Connection`
- 2) `java.sql.Statement`
- 3) `java.sql.ResultSet`

We have discussed (1), we now consider (2) and (3)

Statement

`Statement` class represents SQL statements.

It has three generic forms of statement execution methods:

1) `executeQuery(String query)`

Usage: for any SQL calls that expect to return data from database

2) `executeUpdate(String query)`

Usage: when SQL calls are not expected to return data from database

It returns the number of row affected by `query`

3) `execute()`

Usage: when you cannot determine whether SQL is an update or query
return `true` if row is returned, use `getResultset()` to get the row
otherwise returns `false`

Submitting a Query 1

Submitting a query involves

1) create a `Statement` object

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

SQL exceptions occur when there is a database access error.

Errors are detected when a connection is broken or the database server goes down.

Submitting a Query 2

- 2) use the one the statement query method to submit the SQL statement to the database depending on the type of the SQL.

JDBC does not attempt to interpret queries.

Example:

```
ResultSet rs = null;  
rs = stmt.executeQuery("select * from license");
```

Example: Statement 1

```
import java.sql.*;
public class Update {
    public static void main(String args[]) {
        Connection connection = null;
        if( args.length != 2 ) {
            System.out.print("Syntax: <java Update [number]");
            System.out.println("[string]>");
            return;
        }
        try {
            String driver = "com.mysql.jdbc.Driver";
            Class.forName(driver).newInstance( );
            String url ="jdbc:mysql://localhost/emacao";
            con = DriverManager.getConnection(url, "root", "");
```

Example: Statement 2

```
Statement s = con.createStatement( );
String test_id = args[0];
String test_val = args[1];
int update_count =
s.executeUpdate("INSERT INTO test (test_id, test_val)
" + "VALUES(" + test_id + ", '" + test_val + "')");
System.out.println(update_count + " rows inserted.");
s.close( );
}catch( Exception e ) {
    e.printStackTrace( );
}
```

Example: Statement 3

```
finally {  
    if( con != null ) {  
        try {  
            con.close( );  
        }catch( SQLException e ) {  
            e.printStackTrace( );  
        }  
    }  
}
```

Lab Work: Statement

- 1) Using the `LicenseApp.java` program stored on the server insert records into `license` table in `emacao` database

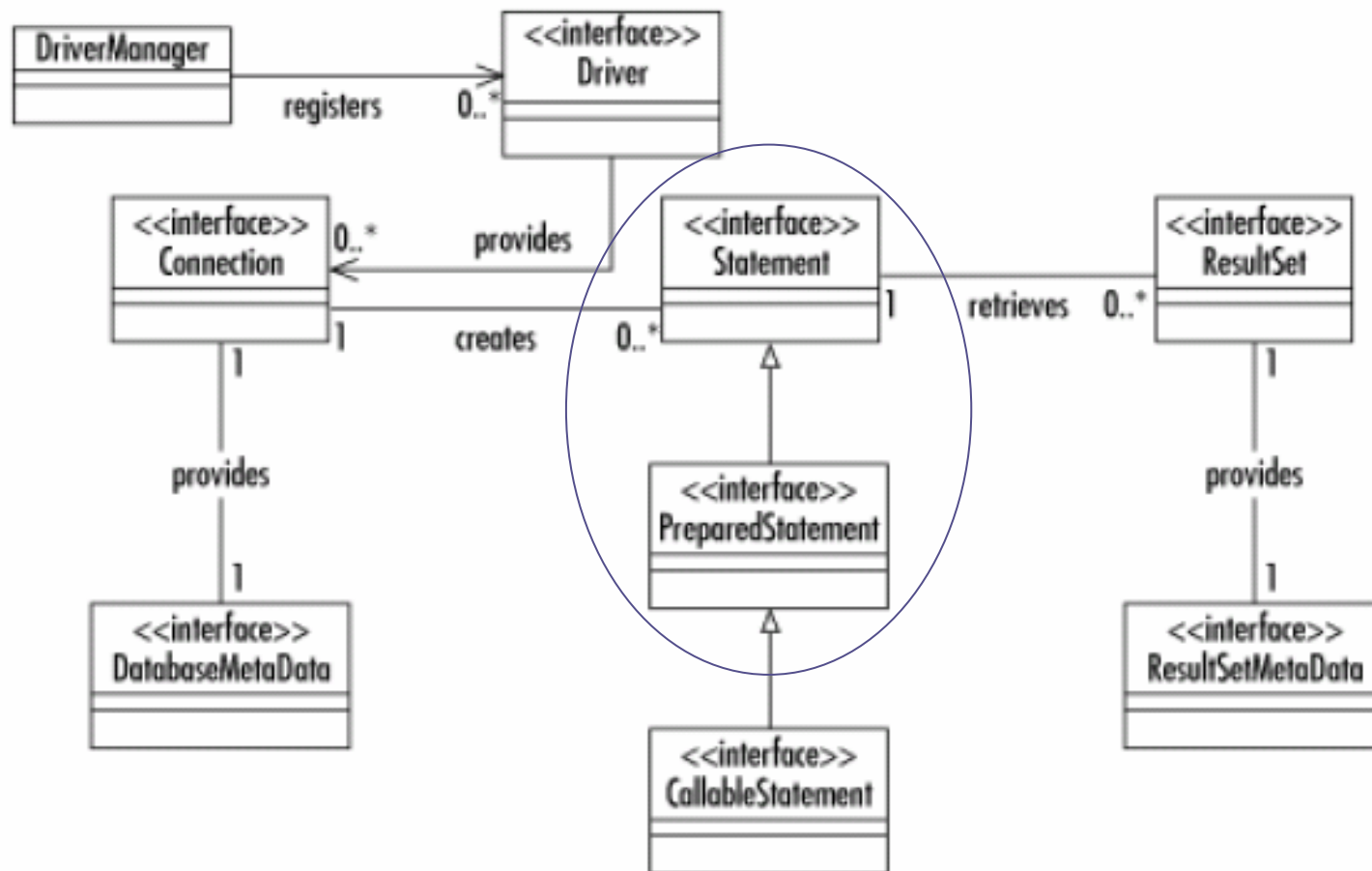
PreparedStatement

PreparedStatement is a precompiled SQL statement.

It is more efficient than calling the same SQL statement over and over.

The `PreparedStatement` class extends the `Statement` class by adding the capability of setting parameters inside of a statement.

PreparedStatement Inheritance



setXXX Methods 1

The `PreparedStatement` class extends the `Statement` class by adding the capability of setting parameters inside of a statement.

The `setXXX` methods are used to set SQL IN parameters values.

Must specify the types that are compatible with the defined SQL input type parameters.

For example, if the IN parameter has SQL type Integer, then you should use the `setInt` method

setXXX Methods 2

Method	SQL Types
setArrayLocator	Locator(<array>)
setASCIIStream	Uses an American Standards 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
setByte	TINYINT
setBytes	VARBINARY or LONGVARBINARY (depending upon the size relative to the limits on VARBINARY)
setCharacterStream	Uses 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 ("Javaobject") is converted to the target SQL Type before sent
setShort	SMALLINT
setString	VARCHAR OR LONGVARCHAR (depending upon the size relative to the driver's limits on VARCHAR)
setStructLocator	LOCATOR(<structure_type>)
setTime	TIME
setTimeStamp	TIMESTAMP
setUnicodeStream	Uses a Unicode stream to produce a LONGVARCHAR

Example: PreparedStatement

```
public boolean prepStatement(String name, String sex){
    String query = null;
    PreparedStatement prepStmt = null;
    query = "update license set name = ?, sex = ? where
                                                    id= 126";

    prepStmt = con.prepareStatement (query);
    prepStmt.setFloat(1, name);
    prepStmt.setString(2, sex);
    Int rowsUpdate = prepStmt.executeUpdate();
    return (rowsUpdate > 0);
}
```

CallableStatement

`CallableStatement` allows non-SQL statements (such as stored procedures) to be executed against the database.

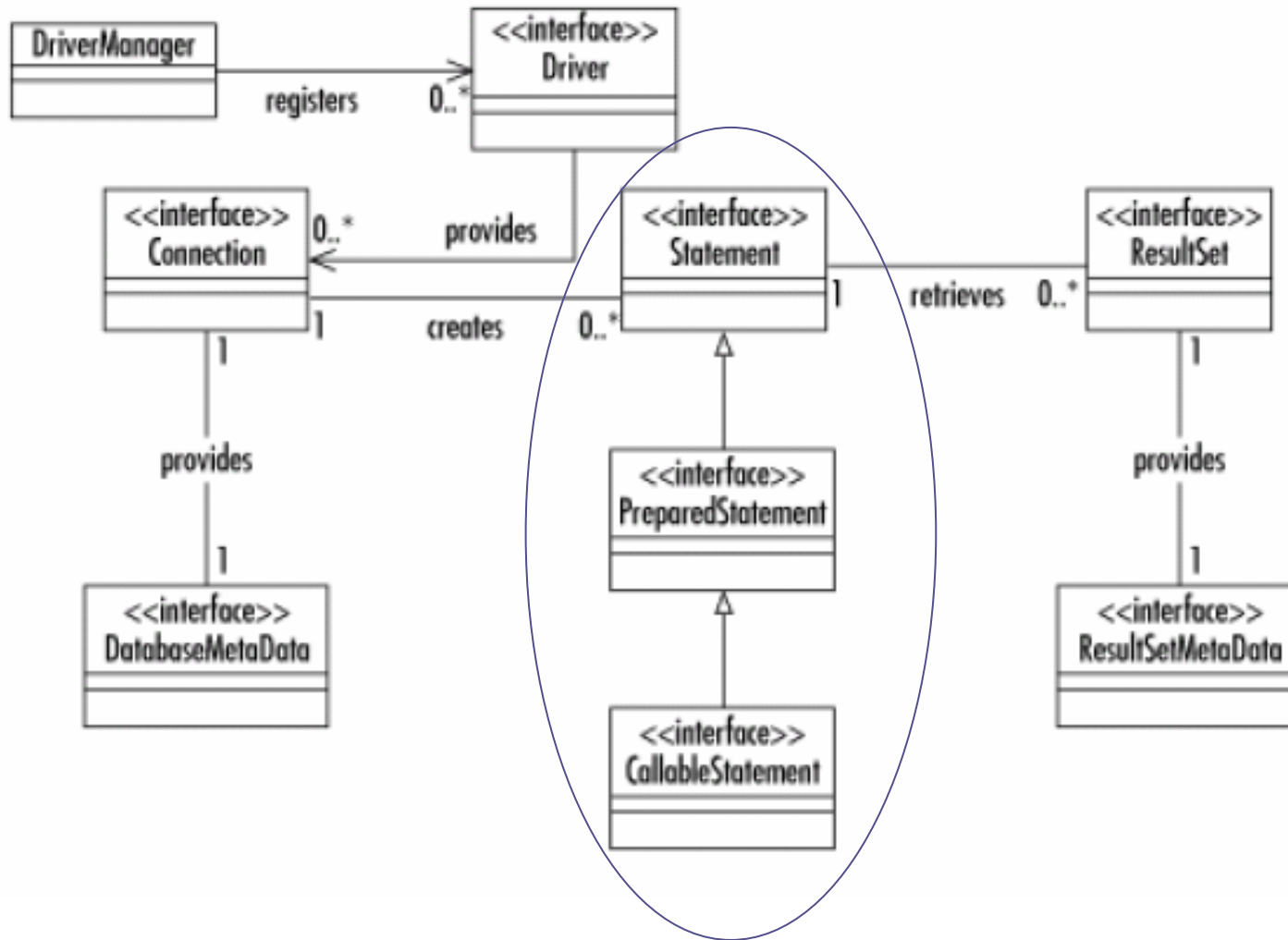
`CallableStatement` class extends the `PreparedStatement` class, which provides the methods for setting IN parameters.

Methods for retrieving multiple results with a stored Procedure are supported with the `Statement.getMoreResults()` method.

Example: CallableStatement

```
int id= 126;
CallableStatement callStm = null;
String storProcName="{?=call return_license(?)}"
querySales = con.prepareCall(storProcName);
try {
    callStm.registerOutParameter(1, Type.VARCHAR);
    callStm.setInt(2, id);
    callStm.execute();
    String license = callStm.getString(1);
} catch (SQLException e) {
    e.printStackTrace();
}
```

CallableStatement Inheritance



Transaction Management

A transaction is a set of one or more statements that are executed together as a unit.

Either all of the statements are executed, or none of the statements is executed.

There are times when you do not want one statement to take effect unless another one also succeeds.

This is achieved through the `setAutoCommit()` method of `Connection` object.

Transaction Management

A transaction is a set of one or more statements that are executed together as a unit.

Either all of the statements are executed, or none of the statements is executed.

There are times when you do not want one statement to take effect unless another one also succeeds.

This is achieved through the `setAutoCommit()` method of `Connection` object.

The method takes a `boolean` value as a parameter.

Disabling Auto-commit Mode

When a connection is created, it is in auto-commit mode.

Each individual SQL statement is treated as a transaction and will be automatically committed right after it is executed.

The way to allow two or more statements to be grouped into a transaction is to disable auto-commit mode.

Example:

```
con.setAutoCommit(false);
```


Committing a Transaction

Once auto-commit mode is disabled, no SQL statements will be committed until you call the method `commit` explicitly.

This is achieved through the `commit()` method of connection objects.

All statements executed after the previous call to the `commit()` method will be included in the current transaction and will be committed together as a unit.

If you are trying to execute one or more statements in a transaction and get an `SQLException`, you should call the `rollback()` method to abort the transaction and start the transaction all over again.

Example: Transaction Commit

```
con.setAutoCommit(false);
PreparedStatement updateName = null;
String query = null;
Query="UPDATE license SET name = ? WHERE id = 126"
updateName= con.prepareStatement(query);
updateName.setString(1, name);
updateName.executeUpdate();
PreparedStatement updateSex = null;
query = "UPDATE test SET test_value =?"
updateSex = con.prepareStatement(query);
updateSex.setString(1, "Male");
updateSex.executeUpdate();
con.commit();
con.setAutoCommit(true);
```

ResultSet

A `ResultSet` is one or more rows of data returned by a database query.

The class simply provides a series of methods for retrieving columns from the results of a database query

General form:

```
type gettype(int | String)
```

in which the argument represents either the column number or column name desired

can store values in the database as one type and retrieve them as a completely different type

ResultSet getXXX() Methods

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(< structure-type >)
getTime	java.sql.Time
getTimestamp	java.sql.Timestamp
getUnicodeStream	java.io.InputStream or Unicode characters

SQL and Java Type Mapping

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

Getting the Next Record

`ResultSet` class handles only a single row from the database at any given time.

The class provides the `next ()` method for making it reference the next row of a result set.

If `next ()` returns `true`, you have another row to process and any subsequent calls you make to the `ResultSet` object will be in reference to that next row.

If there are no rows left, it returns `false`.

Example: ResultSet

```
String query = "select * from license";
Statement stm = null;
stm = con.createStatement();
ResultSet rs = stm.executeQuery(query);
while(rs.next( )) {
    int a;
    String str;
    a = rs.getInt("id");
    if( rs.isNull( ) ) {
        a = -1;
    }
    str = rs.getString("name");
    if( rs.isNull( ) ) {
        str = null;
    }
}
```

SQL Null Versus Java null

SQL and Java have a serious mismatch in handling null values.

Java `ResultSet` has no way of representing a SQL NULL value for any numeric SQL column.

After retrieving a value from a `ResultSet`, it is therefore necessary to ask the `ResultSet` if the retrieved value represents a SQL NULL.

To avoid running into database oddities, however, it is recommended that you always check for SQL NULL.

Checking for SQL NULL involves a single call to the `wasNull()` method in your `ResultSet` after you retrieve a value.

Example: wasNull()

```
rs.afterLast( );
while(rs.previous( )) {
    int a;
    String str;
    a = rs.getInt("test_id");
    if( rs.wasNull( ) ) {
        a = -1;
    }
    str = rs.getString("test_val");
    if( rs.wasNull( ) ) {
        str = null;
    }
}
```

Scrollable ResultSet 1

The single most visible addition to the JDBC API in its 2.0 specification is support for scrollable result sets.

Using scrollable result sets starts with the way in which you create statements.

The `Connection` class actually has two versions of `createStatement()`

1) the zero parameter version

Example:

```
Statement stm = con.createStatement();
```

Scrollable ResultSet 2

- 2) a two parameter version that supports the creation of `Statement` instances that generate scrollable `ResultSet` objects.

```
createStatement(int rsType,int rsConcurrency)
```

Parameters:

`rsType` - a result set type; one of `ResultSet.TYPE_FORWARD_ONLY`, `ResultSet.TYPE_SCROLL_INSENSITIVE`, or `ResultSet.TYPE_SCROLL_SENSITIVE`

`rsConcurrency` - a concurrency type; one of `ResultSet.CONCUR_READ_ONLY` or `ResultSet.CONCUR_UPDATABLE`

ResultSet Constants

JDBC defines three types of result sets:

- 1) `TYPE_FORWARD_ONLY`
- 2) `TYPE_SCROLL_SENSITIVE`
- 3) `TYPE_SCROLL_INSENSITIVE`

Out of these three `TYPE_FORWARD_ONLY` is the only type that is not scrollable.

The other two types are distinguished by how they reflect changes made to them.

`TYPE_SCROLL_INSENSITIVE ResultSet` is unaware of in-place edits made to modifiable instances.

`TYPE_SCROLL_SENSITIVE`, on the other hand, means that you can see changes made to the results if you scroll back to the modified row at a later time.

Result Set Navigation 1

When `ResultSet` is first created, it is considered to be positioned before the first row.

Positioning methods such as `next()` point a `ResultSet` to actual rows.

Your first call to `next()`, for example, positions the cursor on the first row.

Subsequent calls to `next()` move the `ResultSet` ahead one row at a time.

With a scrollable `ResultSet`, however, a call to `next()` is not the only way to position a result set.

Result Set Navigation 2

The method `previous()` works in an almost identical fashion to `next()`.

While `next()` moves one row forward, `previous()` moves one row backward.

If it moves back beyond the first row, it returns `false`. Otherwise, it returns `true`.

Because a `ResultSet` is initially positioned before the first row, you need to move the `ResultSet` using some other method before you can call `previous()`.

Example: Result Set Navigation 1

```
import java.sql.*;
import java.util.*;
public class ReverseSelect {
    public static void main(String argv[]) {
        Connection con = null;
        try {
            String url = "jdbc:mysql://localhost/emacao";
            String driver = "com.mysql.jdbc.Driver";
            Statement stmt;
            ResultSet rs;
            Class.forName(driver).newInstance( );
            con = DriverManager.getConnection(url, "root", "");
            stmt =con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                                     ResultSet.CONCUR_READ_ONLY);
            rs = stmt.executeQuery("SELECT * from license ORDER BY id");

            System.out.println("Got results:");
```

Example: Result Set Navigation 2

```
rs.afterLast( );
while(rs.previous( )) {
    int a;
    String str;
    a = rs.getInt("id");
    a = rs.isNull( ) ? -1 : a;
    str = rs.getString("name");
    str = rs.isNull( ) ? null : str;
    System.out.print("\tid= " + a);
    System.out.println("/str= '" + str + "'");
}
System.out.println("Done.");
}catch( Exception e ) {
    e.printStackTrace( );
}
```


Example: Result Set Navigation 3

```
finally {  
    if( con != null ) {  
        try {  
            con.close( );  
        } catch( SQLException e ) {  
            e.printStackTrace( );  
        }  
    }  
}  
}
```

Other Navigation Methods

JDBC 2.0 provides new methods to navigate around rows in result sets:

- 1) `beforeFirst()`
- 2) `first()`
- 3) `last()`
- 4) `isBeforeFirst()`
- 5) `isFirst()`
- 6) `isLast()`
- 7) `isAfterLast()`
- 8) `getRow()`
- 9) `relative()`
- 10) `absolute()`

Except for `absolute()` and `relative()`, the names of the methods say exactly what they do. Each take integer arguments.

absolute() 1

For `absolute()`, the argument specifies a row to navigate to.

Example:

A call to `absolute(5)` moves the `ResultSet` to row 5 unless there are four or fewer rows in the `ResultSet`.

A call to `absolute()` with a row number beyond the last row is therefore identical to a call to `afterLast()`

absolute() 2

You can also pass negative numbers to `absolute()`.

A negative number specifies absolute navigation backwards from the last row

Example:

`absolute(1)` is identical to `first()`, `absolute(-1)` is identical to `last()`

Similarly, `absolute(-3)` is the third to last row in the `ResultSet`. If there are fewer than three rows in the `ResultSet`.

relative()

The `relative()` method handles relative navigation through a `ResultSet`.

In other words, it tells the `ResultSet` how many rows to move forward or backward.

Example:

A value of 1 behaves just like `next()` and a value of -1 just like `previous()`.

Clean Up

The `Connection`, `Statement`, and `ResultSet` classes all have `close()`.

It is always a good idea to close any instance of these objects when you are done with them.

It is useful to remember that closing a `Connection` implicitly closes all `Statement` instances associated with the `Connection`.

Similarly, closing a `Statement` implicitly closes `ResultSet` instances associated with it.

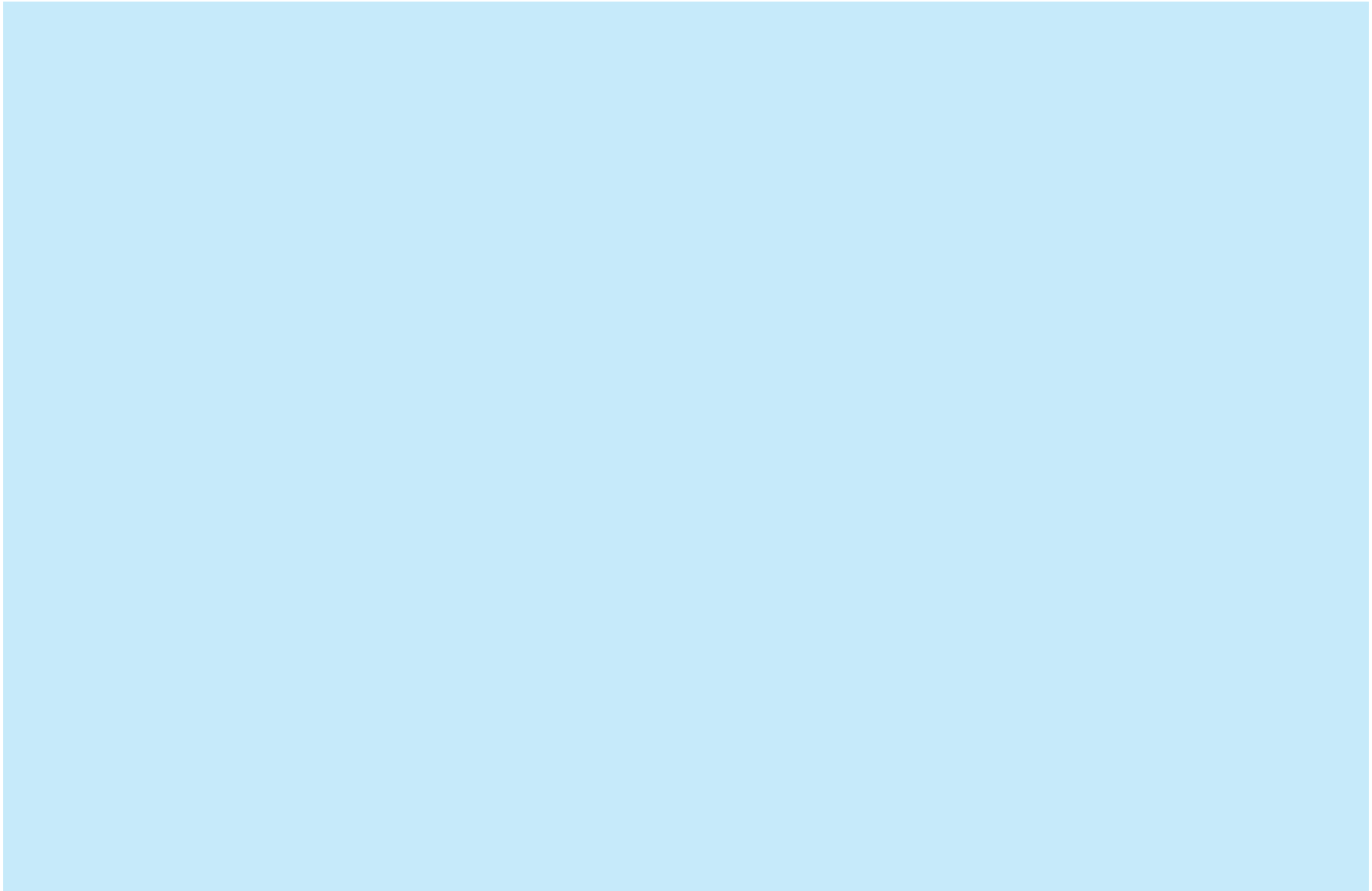
Example: Clean Up

```
try{
    // Connection, Statements here
}catch(SQLException ex){
    ex.printStackTrace();
}finally {
    if( con != null ) {
        try {
            con.close( );
        }catch( SQLException e ) {
            e.printStackTrace( );
        }
    }
}
```

Lab Work: ResultSet

- 1) Using the `LicenseApp.java`, before you save, check if the data you are saving exists, if it is update with the new values else insert a new record.

Exercise: JDBC



Message-Orientation

Course Outline

- 1) introduction
- 2) streams
- 3) networking
- 4) database connectivity
- 5) architectures
 - a) message-orientation
 - 1) **javamail**
 - 2) jms
 - b) distributed objects
 - 1) rmi
 - 2) corba
 - 3) JavaIDL
- 6) summary

JavaMail

Course Outline

- 1) introduction
- 2) streams
- 3) networking
- 4) database connectivity
- 5) message-orientation
 - a) **javamail**
 - b) jms
- 6) distributed objects
 - a) rmi
 - b) corba
 - c) Javaidl
- 7) summary

Overview

Email was the Internet's first killer application and still generates more Internet traffic than any protocol except HTTP.

One of the most frequently asked questions about Java is how to send email from a Java applet or application or how to send asynchronous messages between a Java application and homo-sapiens?

We shall be considering:

- 1) Introduction to JavaMail API
- 2) Protocols – SMTP, POP, IMAP MIME
- 3) Installation and configuration
- 4) Core Classes – Session, Message, Address, Authenticator, Transport, Store and Folder
- 5) Usage – sending and receiving email, processing HTML messages etc

What Is the JavaMail API?

The JavaMail API is a standard extension to Java that provides a class library for email clients.

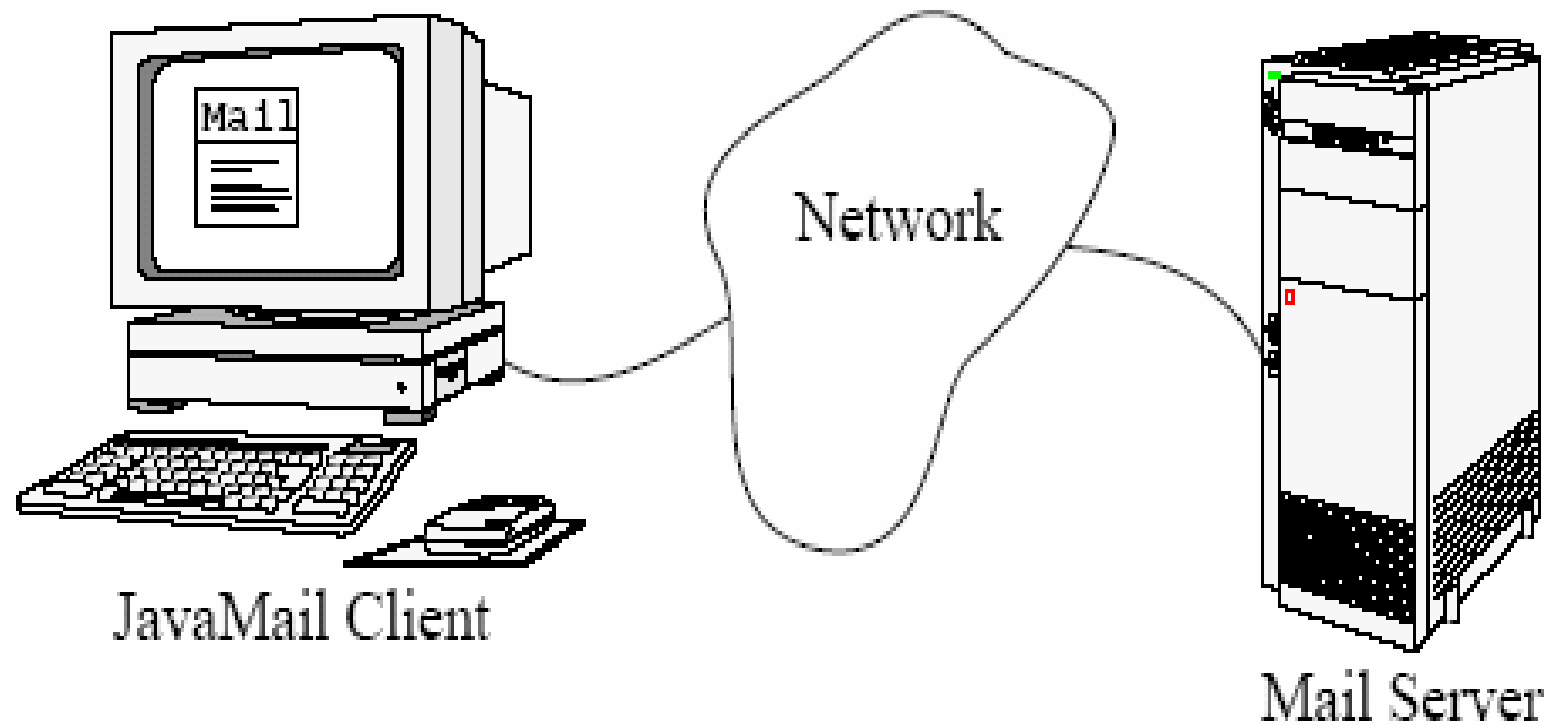
Is an optional package (standard extension) for reading, composing, and sending electronic messages.

You use the package to create **Mail User Agent (MUA)** type programs, similar to Eudora, Pine, and Microsoft Outlook.

Purpose:

- 1) transporting
- 2) delivering and
- 3) Forwarding messages like sendmail or other Mail Transfer Agents

Mail Client and Server



Why Mail?

There are situation in which an application may need to send an email

- 1) an error situation occurs
- 2) when the next step in some workflow must be started
- 3) or in response to some events that has occurred

JavaMail Applications

There are several areas in which JavaMail is useful.

Some are discussed below:

- 1) A server-monitoring application such as Whistle Blower can periodically load pages from a web server running on a different host and email the webmaster if the web server has crashed.
- 2) An applet can use email to send data to any process or person on the Internet that has an email address, in essence using the web server's SMTP server as a simple proxy to bypass the usual security restrictions about whom an applet is allowed to talk to. In reverse, an applet can talk to an IMAP server on the applet host to receive data from many hosts around the Net.
- 3) A newsreader could be implemented as a custom service provider that treats NNTP as just one more means of exchanging messages.

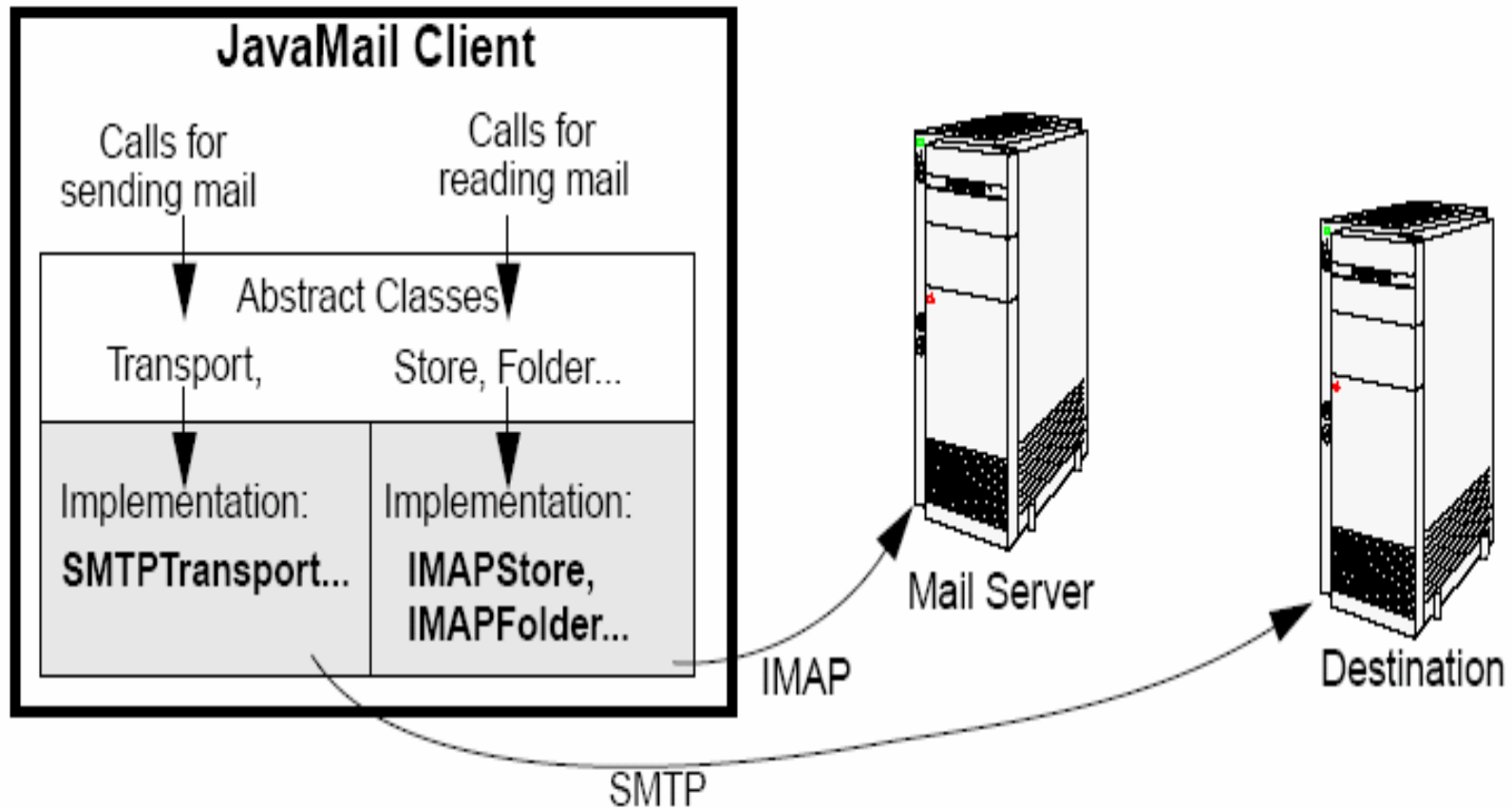
Related Protocols 1

There are four protocols are commonly used with the API:

- 1) Simple Mail Transfer Protocol (SMTP)
- 2) Post Office Protocol (POP)
- 3) Internet Message Access Protocol (IMAP)
- 4) Multipurpose Internet Mail Extensions (MIME)

Each will be considered.

Related Protocols 2



SMTP

The Simple Mail Transfer Protocol (SMTP) is the mechanism for delivery of email.

In the context of JavaMail,

- JavaMail-based program will communicate with company or Internet Service Provider's (ISP's) SMTP server.
- 2) The SMTP server will relay the message on to the SMTP server of the recipient to be acquired eventually by the user through POP or IMAP

POP

Post Office Protocol (POP) is the mechanism most people on the Internet use to get their mail.

It defines support for a single mailbox for each user.

Currently in version 3, also known as POP3

The ability to see how many new mail messages you have, are not supported by POP at all.

These capabilities are built into programs like Eudora or Microsoft Outlook, which remember things like the last mail received and calculate how many are new for you. So, when using the JavaMail API, if you want this type of information, you have to calculate it yourself.

IMAP

Internet Message Access Protocol (IMAP) more advanced protocol for receiving messages.

Currently in version 4, also known as IMAP4

Your mail server must support the protocol before you can use it.

You can't just change your program to use IMAP instead of POP and expect everything in IMAP to be supported.

Assuming your mail server supports IMAP, your JavaMail-based program can take advantage of users having multiple folders on the server and these folders can be shared by multiple users.

IMAP Drawbacks

It places a much heavier burden on the mail server requiring the server to receive the new messages, deliver them to users when requested, *and* maintain them in multiple folders for each user.

While this does centralize backups, as users' long-term mail folders get larger and larger, everyone suffers when disk space is exhausted.

But with POP, saved messages get offloaded from the mail.

MIME

MIME stands for Multipurpose Internet Mail Extensions

It is not a mail transfer protocol.

Instead, it defines the content of what is transferred.

For example:

- 1) format of the messages
- 2) attachments, and
- 3) etc

Installation

There are three versions of the JavaMail API commonly used today:

- 1) version 1.1.3
- 2) version 1.2
- 3) version 1.3.2

Version 1.3.2 is the latest.

The version of the JavaMail API you want to use affects what you download and install.

Installing JavaMail 1.3.2

- 1) Download `javamail-1_3_2.zip` from
<http://java.sun.com/products/javamail>
- 2) Extract the zip file into a folder
- 3) set it in the `CLASSPATH` environment variable
- 4) Include the following archive files in the `CLASSPATH`
 - a) `imap.jar`
 - b) `mailapi.jar`
 - c) `pop3.jar`
 - d) `smtp.jar`

JavaMail needs a framework in order to complete its functions.

This framework is known as **JavaBeans Activation Framework (JAF)**.

JAF

JavaBeans Activation Framework (JAF) is a standard extension that enables developers who use Java technology to take advantage of standard services:

- 1) to determine the type of an arbitrary piece of data,
- 2) encapsulate access to it,
- 3) discover the operations available on it,
- 4) and to instantiate the appropriate bean to perform the said operation(s).

It is the basic MIME-type support found in many browsers and mail tools.

Example: JAF

If a browser obtained a JPEG image JAF:

- 1) enables the browser to identify that stream of data as a JPEG image
- 2) and from that type, the browser could locate and instantiate an object that could manipulate, or view that image
- 3) discover the operations available on it,
- 4) and to instantiate the appropriate bean to perform the said operation(s).

Installing JAF

- 1) Download `jaf-1_0_2-upd.zip` from <http://java.sun.com/products/javabeans/glasgow/jaf.html>
- 2) extract the zip file into a folder
- 3) set it in the `CLASSPATH` environment variable
- 4) include `activation.jar` in the `CLASSPATH`

Installing JavaMail Using J2EE

JavaMail is bundled with J2EE

There is nothing special you have to do to use the basic JavaMail API.

Just make sure the `j2ee.jar` file is in your `CLASSPATH` and you are set.

Note: This will be deferred to J2EE courses!

Other Referencing Options

If you don't want to change the `CLASSPATH` environment variable:

- 1) copy the JAR files to your `lib/ext` directory under the Java Runtime environment (JRE) directory
- 2) for instance, `%JAVA_HOME%\lib\ext` on a Windows platform

Exercise

- 1) Download the latest version of the JavaMail API implementation.
- 2) Download the latest version of the JavaBeans Activation Framework.
- 3) Extract the zip files to a folder
- 4) Install the archive files.

Core Classes

There are seven core classes that make JavaMail API:

- 1) `Session`
- 2) `Message`
- 3) `Address`
- 4) `Authenticator`
- 5) `Transport`
- 6) `Store`
- 7) `Folder`

Each will be considered.

Session

It defines a basic mail session.

It is through this session that everything else works.

The `Session` object takes advantage of a `java.util.Properties` object to get information like mail server, username, password, and other information that can be shared across your entire application.

`Session` class is singleton

Session: Singleton 1

The constructors for the class are **private**.

An instance of the class can be created in four ways by calling the following methods of the class:

- 1) `getDefaultInstance(Properties props)`
- 2) `getDefaultInstance(Properties props,
Authenticator authenticator)`
- 3) `getInstance(Properties props)`
- 4) `getInstance(Properties props,
Authenticator authenticator)`

Session: Singleton 2

Each method returns either a default or new `Session` object.

The (1) and (2) methods get the default instance if one exists and if not a new session object is created.

The (3) and (4) create a new instance.

`props` is the `Properties` object that holds relevant properties

`authenticator` is `Authenticator` object used to call back to the application when a user name and password is needed.

Session: Usage

1) Get a default instance

```
Properties props = new Properties();  
// fill props with any information  
Session session = Session.getDefaultInstance(props,  
                                              null);
```

2) Create a unique session

```
Properties props = new Properties();  
// fill props with any information  
Session session = Session.getInstance(props, null);
```

In both cases here the `null` argument is an `Authenticator` object.

Message 1

This class models an email message. It is an abstract class.

Subclasses provide actual implementations.

Characteristics:

- Message implements the `Part` interface.
- Direct subclass is `MimeMessage`

2) Message contains a set of attributes and a "content".

3) Messages within a folder also have a set of flags that describe its state within the folder.

Message 2

Message defines some new attributes in addition to those defined in the Part interface.

These attributes specify meta-data for the message - i.e., addressing and descriptive information about the message.

Message objects are obtained either from a Folder or by constructing a new Message object of the appropriate subclass.

Messages that have been received are normally retrieved from a folder named "INBOX".

Message is an abstract class, you cannot work with it. Use the subclasses.

MimeMessage

`MimeMessage` is the direct subclass of `Message`

It is an email message that understands MIME types and headers.

Message headers are restricted to US-ASCII characters only, though non-ASCII characters can be encoded in certain header fields.

Once you have your `Session` object, then you can create the message to send.

Creating a Message

- 1) pass along the `Session` object to the `MimeMessage` constructor.

```
MimeMessage message = new MimeMessage(session);
```

- 2) set its parts, as `Message` implements the `Part` interface (with `MimeMessage` implementing `MimePart`).

```
message.setContent("Hello", "text/plain");
```

- 3) If, however, you know you are working with a `MimeMessage` and your message is plain text, then use `setText()` method

```
message.setText("Hello");
```

- 4) set the subject using the `setSubject()` method

```
message.setSubject("First");
```

Simple Message

Message Class

Header Attributes

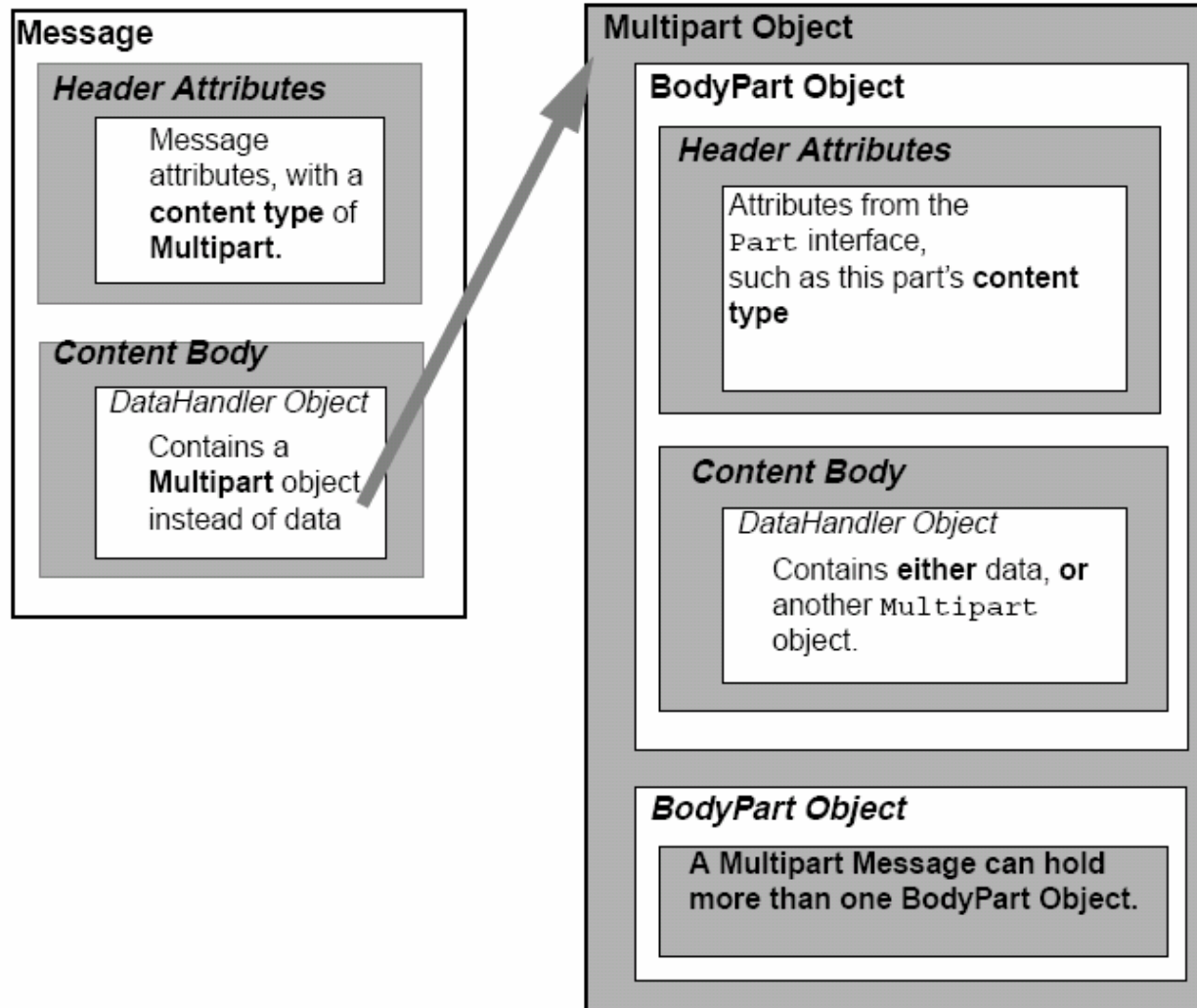
Attributes, such as
Content-Type.

Content Body

DataHandler Object

Contains data that conforms
to the Content-Type attribute

Multipart Message



Address

Once you've created the `Session` and the `Message`, as well as filled the message with content, it is time to address your letter with an `Address`.

This is done using `Address Class`.

Characteristics:

- 1) like `Message`, `Address` is an abstract class, hence use the subclass
- 2) you use the `javax.mail.internet.InternetAddress` class

Creating an Address 1

- 1) To create an address with just the email address, pass the email address to the constructor

```
Address address = new InternetAddress("xx@server.com");
```

- 2) If you want a name to appear next to the email address

```
Address address = new InternetAddress(xx@server.com,  
                                     "Mr. Gabriel");
```

Creating an Address 2

Once you have created the Addresses you connect them to a message in one of two ways:

- 1) For identifying the sender, you use the `setFrom()` and `setReplyTo()` methods.

```
message.setFrom(address);
```

or If your message needs to show multiple from addresses, use the `addFrom()` method

```
Address address[] = ...;  
message.addFrom(address);
```

Creating an Address 3

2) For identifying the message recipients, you use the `addRecipient()` method.

This requires a `Message.RecipientType` besides the address.

The three predefined types of address are:

- a) `Message.RecipientType.TO`
- b) `Message.RecipientType.CC`
- c) `Message.RecipientType.BCC`

Creating an Address 4

...

```
Address toAddress = new
    InternetAddress ("president@server.com");

Address ccAddress = new
    InternetAddress ("first.lady@server.com");

message.addRecipient (Message.RecipientType.TO,
                      toAddress);
message.addRecipient (Message.RecipientType.CC,
                      ccAddress);
```

...

Authenticator

`Authenticator` Class provide access to protected resources (mail server) via a username and password

To use the `Authenticator`, you subclass the abstract class and return a `PasswordAuthentication` instance from the `getPasswordAuthentication()` method.

Example:

```
Properties props = new Properties();  
// fill props with any information  
Authenticator auth = new MyAuthenticator();  
Session session = Session.getDefaultInstance(props,  
                                              auth);
```

Transport 1

The final part of sending a message is to use the `Transport` class.

This class speaks the protocol-specific language for sending the message (usually SMTP).

It's an abstract class and works something like `Session`.

There are two ways of sending a message:

- 1) You can use the default version of the class by just calling the static `send()` method:

```
Transport.send(message);
```

Transport 2

- 2) You can get a specific instance from the session for your protocol, pass along the username and password (blank if unnecessary), send the message, and close the connection:

```
message.saveChanges(); // implicit with send()
Transport transport = session.getTransport("smtp");
transport.connect(host, username, password);
transport.sendMessage(message, message.getAllRecipients(
    ));
transport.close();
```

Transport 3

This latter way is better when you need to send multiple messages.

It will keep the connection with the mail server active between messages.

The basic `send()` mechanism makes a separate connection to the server for each method call.

Store and Folder 1

Getting messages starts similarly to sending messages:

- 1) Get a Session Object
- 2) You connect to a `Store`, quite possibly with a username and password or Authenticator.
- 3) Like Transport, you tell the `Store` what protocol to use

```
//Store store = session.getStore("imap");  
Store store = session.getStore("pop3");  
store.connect(host, username, password);
```

Store and Folder 2

- 4) Get a `Folder`, which must be opened before you can read messages from it:

```
Folder folder = store.getFolder("INBOX");  
folder.open(Folder.READ_ONLY);  
Message message[] = folder.getMessages();
```

- 5) Get its content with `getContent()` or write its content to a stream with `writeTo()`. The `getContent()` method only gets the message content, while `writeTo()` output includes headers.

```
System.out.println(((MimeMessage)message).getContent());
```

Store and Folder 3

- 6) Once you're done reading mail, close the connection to the folder and store.

```
folder.close(aBoolean);  
store.close();
```

The boolean passed to the `close()` method of folder states whether or not to update the folder by removing deleted messages.

Using JavaMail API

We are going to demonstrate the usage of the API with the following:

- 1) sending messages
- 2) fetching messages
- 3) deleting Messages and Flags
- 4) authenticating Yourself
- 5) replying to Messages
- 6) forwarding Messages
- 7) working with attachments – sending and getting
- 8) processing HTML Messages – sending and including images

Sending Messages

This involves three steps:

- 1) getting a session
- 2) creating and filling a message
- 3) Send the message using the static `Transport.send()` method

You can specify your SMTP server by setting the `mail.smtp.host` property for the `Properties` object passed when getting the Session

Example: Sending Messages 1

```
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;
...
String host = pop3.iist.unu.edu;
String from = gab@iist.unu.edu;
String to = milton@iist.unu.edu;

// Get system properties
Properties props = System.getProperties();

// Setup mail server
props.put("mail.smtp.host", host);
```

Example: Sending Messages 2

```
// Get session
Session session = Session.getDefaultInstance(props,
                                             null);

// Define message
MimeMessage message = new MimeMessage(session);
message.setFrom(new InternetAddress(from));
message.addRecipient(Message.RecipientType.TO,
                     new InternetAddress(to));
message.setSubject("Hello JavaMail");
message.setText("Welcome to JavaMail");

// Send message
Transport.send(message);
```

Lab Work: Sending Messages

- 1) Starting with the skeleton code, get the system Properties.
- 2) Add the name of your SMTP server to the properties for the `mail.smtp.host` key.
- 3) Get a Session object based on the Properties.
- 4) Create a MimeMessage from the session.
- 5) Set the from field of the message.
- 6) Set the to field of the message.
- 7) Set the subject of the message.
- 8) Set the content of the message.
- 9) Use a Transport to send the message.
- 10) Compile and run the program, passing your SMTP server, from address, and to address on the command line.

Lab Work : Skeleton Code 1

```
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;

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

        Exception {
        String host = args[0];
        String from = args[1];
        String to = args[2];
        // Get system properties
        // Setup mail server
        // Get session
        // Define message
        // Get the from address
```

Lab Work : Skeleton Code 2

```
        // Set the to address
        // Set the subject
        // Set the content
        // Send message
    }
}
```

Fetching Messages

Reading messages involves five steps:

- 1) getting a session
- 2) get and connect to an appropriate store for your mailbox
- 3) open the appropriate folder
- 4) get your message(s)
- 5) and close the connection when done.

Example: Fetching Messages 1

```
import java.util.Properties;
import javax.mail.*;
import javax.mail.Internet.*;
...
String host = ...;
String username = ...;
String password = ...;

// Create empty properties
Properties props = new Properties();

// Get session
Session session = Session.getDefaultInstance(props,
                                             null);
```

Example: Fetching Messages 2

```
// Get the store
Store store = session.getStore("pop3");
store.connect(host, username, password);
// Get folder
Folder folder = store.getFolder("INBOX");
folder.open(Folder.READ_ONLY);

// Get directory
Message message[] = folder.getMessages();

for (int i=0, n=message.length; i<n; i++) {
    System.out.print(i + ": " + message[i].getFrom()[0]);
    System.out.println("\t" + message[i].getSubject());
}
```

Example: Fetching Messages 3

```
// Close connection  
folder.close(false);  
store.close();
```

This code snippet displays the subjects of the messages.

To display the whole message:

- 1) you can prompt the user after seeing the from and subject fields,
- 2) and then call the message's `writeTo()` method if they want to see it

Example: Displaying Content 1

```
BufferedReader reader = new BufferedReader (  
    new InputStreamReader(System.in));  
  
// Get directory  
Message message[] = folder.getMessages();  
for (int i=0, n=message.length; i<n; i++) {  
    System.out.print(i + ": " + message[i].getFrom()[0]);  
    System.out.println("\t" + message[i].getSubject());  
  
    System.out.print("Do you want to read message? ");  
    System.out.println("[YES to read/QUIT to end]");  
    String line = reader.readLine();  
}
```

Example: Displaying Content 2

```
if ("YES".equals(line)) {  
    message[i].writeTo(System.out);  
} else if ("QUIT".equals(line)) {  
    break;  
}  
}
```

Lab Work: Fetching Messages 1

- 1) Starting with the skeleton code, get or create a Properties object.
- 2) Get a Session object based on the Properties.
- 3) Get a Store for your email protocol, either pop3 or imap.
- 4) Connect to your mail host's store with the appropriate username and password.
- 5) Get the folder you want to read. More than likely, this will be the INBOX.
- 6) Open the folder read-only.
- 7) Get a directory of the messages in the folder. Save the message list in an array variable named message.
- 8) For each message, display the from field and the subject.
- 9) Display the message content when prompted.

Lab Work: Fetching Messages 2

- 10) Close the connection to the folder and store.
- 11) Compile and run the program, passing your mail server, username, and password on the command line. Answer YES to the messages you want to read. Just hit ENTER if you don't. If you want to stop reading your mail before making your way through all the messages, enter QUIT.

Lab Work : Skeleton Code 1

```
import java.io.*;
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;

public class GetMessageExample {
    public static void main (String args[]) throws
        Exception{
        String host = args[0];
        String username = args[1];
        String password = args[2];
        // Create empty properties
        // Get session
```


Lab Work : Skeleton Code 2

```
// Get the store
// Connect to store
// Get folder
// Open read-only
BufferedReader reader = new BufferedReader ( new
    InputStreamReader(System.in));
// Get directory
for (int i=0, n=message.length; i<n; i++) {
    // Display from field and subject
    System.out.print("Do you want to read message?");
    System.out.println("[YES to read/QUIT to
end]");
    String line = reader.readLine();
```

Lab Work : Skeleton Code 3

```
    if ("YES".equals(line)) {  
        // Display message content  
    } else if ("QUIT".equals(line)) {  
        break;  
    }  
} // Close connection  
}
```

Flags

The `Flags` class represents the set of flags on a `Message`. `Flags` are composed of predefined system flags, and user defined flags.

A System flag is represented by the `Flags.Flag` inner class.

- 1) `Flags.Flag.ANSWERED`
- 2) `Flags.Flag.DELETED`
- 3) `Flags.Flag.DRAFT`
- 4) `Flags.Flag.FLAGGED`
- 5) `Flags.Flag.RECENT`
- 6) `Flags.Flag.SEEN`
- 7) `Flags.Flag.USER`

Use the `getPermanentFlags()` method of `Folder` class to find out what flags are supported

A User defined flag is represented as a `String`.

Deleting Messages

To delete messages, you set the message's DELETED flag:

```
message.setFlag(Flags.Flag.DELETED, true);
```

Open up the folder in READ_WRITE mode first though:

```
folder.open(Folder.READ_WRITE);
```

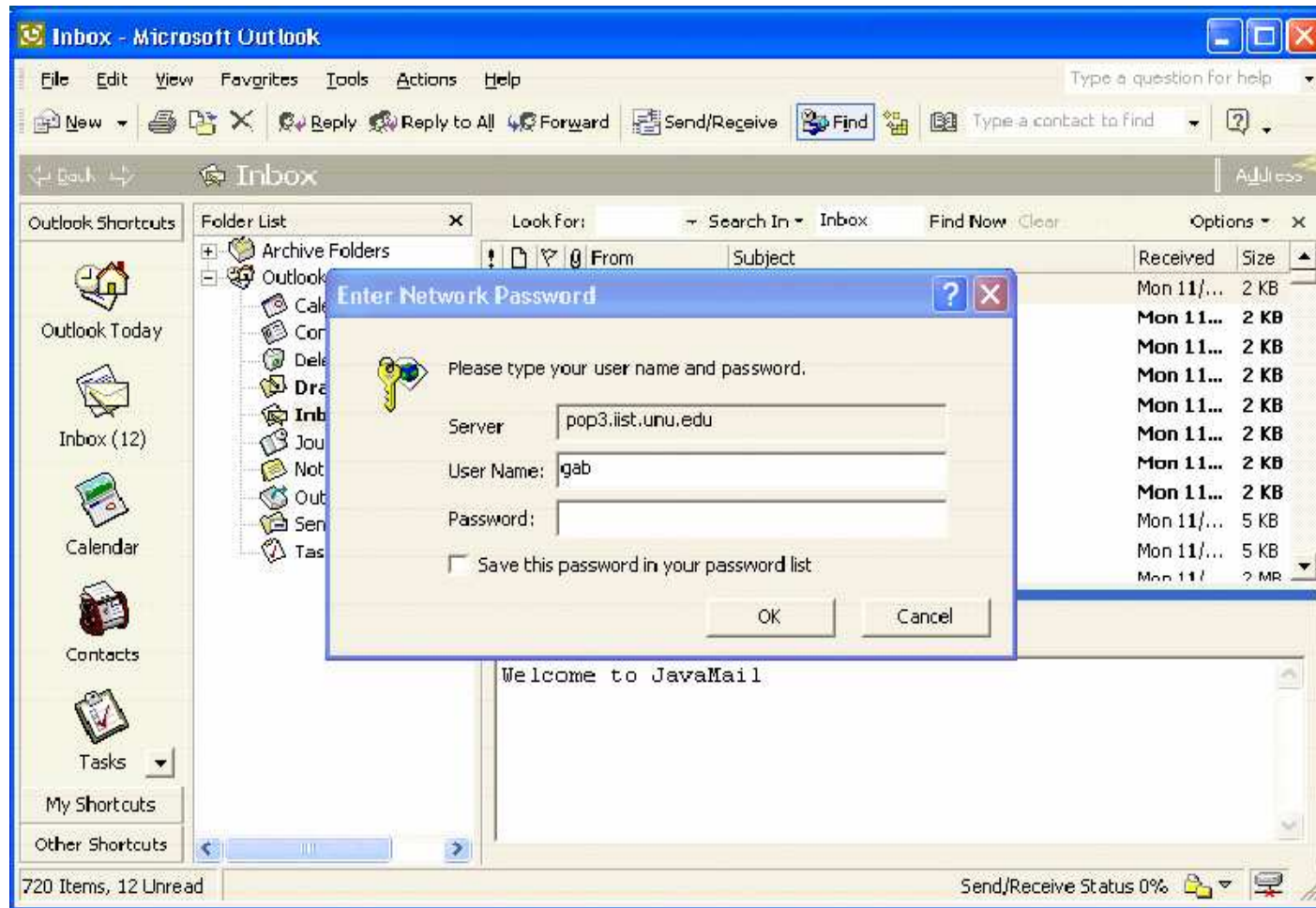
Then, when you are done processing all messages, close the folder, passing in a `true` value to expunge the deleted messages.

To unset a flag, just pass `false` to the `setFlag()` method.

To see if a flag is set, check with `isSet()`.

Authentication 1

How do you achieve something like this using JavaMail?



Authentication 2

Use an `Authenticator` to prompt for username and password when needed.

Instead of connecting to the `Store` with the host, username, and password, you configure the `Properties` to have the host, and tell the `Session` about your custom `Authenticator` instance.

Example:

```
Properties props = System.getProperties();  
props.put("mail.pop3.host", host);
```

```
// Setup authentication, get session  
Authenticator auth = new PopupAuthenticator();  
Session session = Session.getDefaultInstance(props,  
                                             auth);
```

Authentication 3

```
// Get the store  
Store store = session.getStore("pop3");  
store.connect();
```


PopupAuthenticator 2

```
username = st.nextToken();
password = st.nextToken();
return new PasswordAuthentication(username,
                                   password);
}
}
```

Replying to Messages

The `Message` class includes a `reply()` method to configure a new message with the proper recipient and subject, adding "Re: " if not already there.

This does not add any content to the message, only copying the **from** or **reply-to** header to the new recipient.

The method takes a `boolean` parameter indicating whether to reply to only the sender (`false`) or reply to all (`true`).

Example:

```
MimeMessage reply = (MimeMessage)message.reply(false);  
reply.setFrom(new InternetAddress("xxx@server.com"));  
reply.setText("Thanks");  
Transport.send(reply);
```

Lab Work: Replying to Messages

- 1) The skeleton code already includes the code to get the list of messages from the folder and prompt you to create a reply.
- 2) When answered affirmatively, create a new MimeMessage from the original message.
- 3) Set the from field to your email address.
- 4) Create the text for the reply. Include a canned message to start. When the original message is plain text, add each line of the original message, prefix each line with the "> " characters.
- 5) Set the message's content, once the message content is fully determined. Send the message.
- 6) Compile and run the program, passing your mail server, SMTP server, username, password, and from address on the command line. Answer YES to the messages you want to send replies. Just hit ENTER if you don't. If you want to stop going through your mail before making your way through all the messages, enter QUIT.

Lab Work: Skeleton Code 1

```
import java.io.*;
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;
public class ReplyExample {
    public static void main (String args[]) throws
                                                                    Exception {

        String host = args[0];
        String sendHost = args[1];
        String username = args[2];
        String password = args[3];
        String from = args[4];
```

Lab Work: Skeleton Code 2

```
// Create empty properties
Properties props = System.getProperties();
props.put("mail.smtp.host", sendHost);

// Get session
Session session = Session.getDefaultInstance
                        (props, null);

// Get the store
Store store = session.getStore("pop3");
store.connect(host, username, password);

// Get folder
Folder folder = store.getFolder("INBOX");
folder.open(Folder.READ_ONLY);
```

Lab Work: Skeleton Code 3

```
BufferedReader reader = new BufferedReader
    ( new InputStreamReader(System.in));
// Get directory
Message message[] = folder.getMessages();
for (int i=0, n=message.length; i<n; i++) {
    System.out.println(i + ": ") +
        message[i].getFrom()[0] + "\t" +
        message[i].getSubject());
    System.out.println("Do you want to reply to
        the message? [YES to reply/QUIT to
                                end]");
    String line = reader.readLine();
```

Lab Work: Skeleton Code 4

```
    if ("YES".equals(line)) {  
        // Create a reply message  
        // Set the from field  
        // Create the reply content, copying  
        //over the original if text  
        // Set the content  
        // Send the message  
    }else if ("QUIT".equals(line)) {  
        break;  
    }  
}
```

Lab Work: Skeleton Code 5

```
// Close connection  
folder.close(false);  
store.close();  
}  
}
```


Message Parts

A mail message can be made up of multiple parts

Each part is a `BodyPart`, or more specifically, a `MimeBodyPart` when working with MIME messages.

The different body parts get combined into a container called `Multipart` or, again, more specifically a `MimeMultipart`.

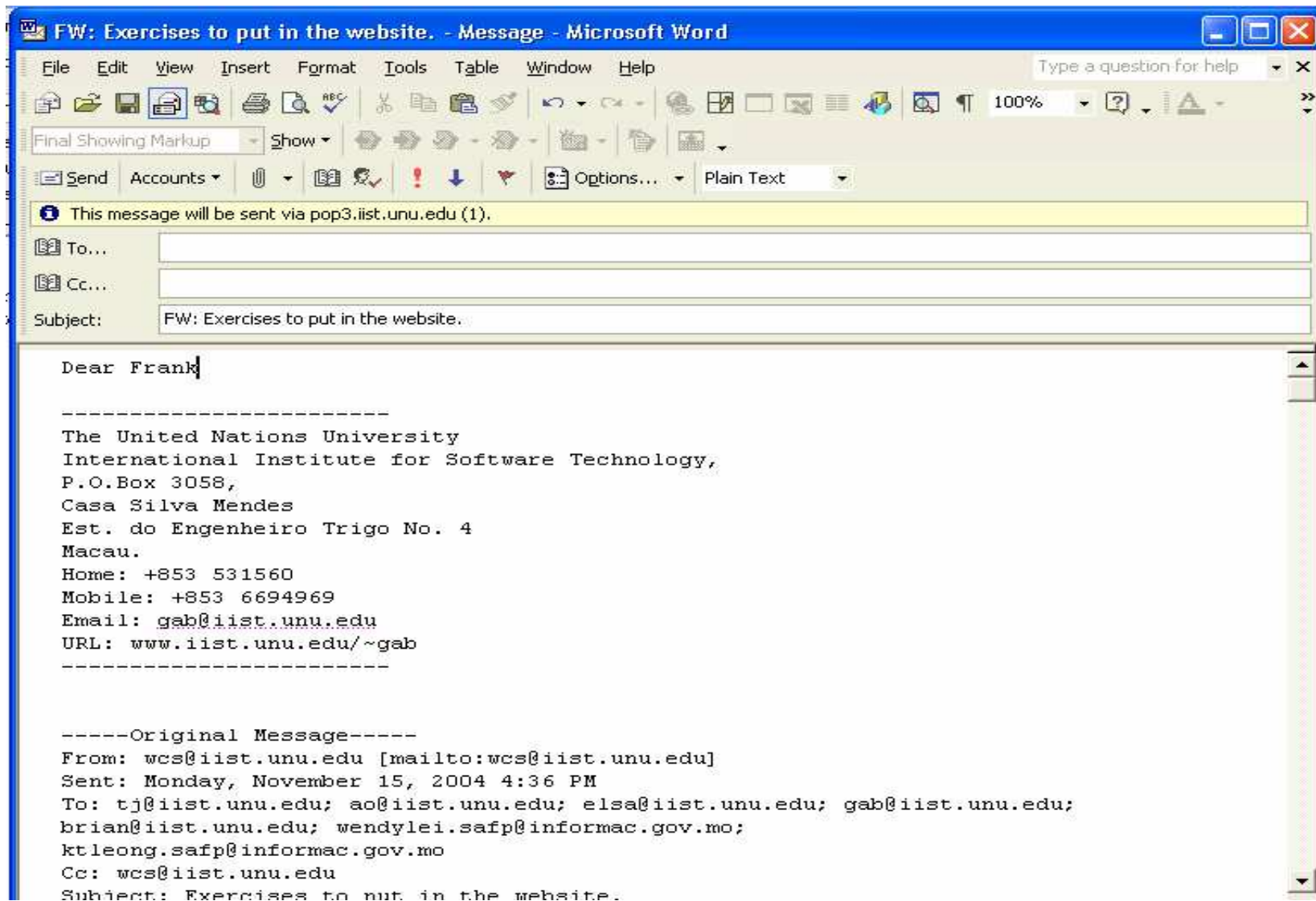
Forwarding Message 1

To forward a message:

- 1) you create one part for the text of your message
- 2) and a second part with the message to forward,
- 3) and combine the two into a multipart.
- 4) Then you add the multipart to a properly addressed message and send it.

To copy the content from one message to another, just copy over its `DataHandler`, a class from the JavaBeans Activation Framework.

Forwarding Message 2



Example: Forwarding Message 1

```
// Create the message to forward
Message forward = new MimeMessage(session);

// Fill in header
forward.setSubject("Fwd: " + message.getSubject());
forward.setFrom(new InternetAddress(from));
forward.addRecipient(Message.RecipientType.TO,
                    new InternetAddress(to));

// Create your new message part
BodyPart messageBodyPart = new MimeBodyPart();
messageBodyPart.setText("Here you go with the original
                        message:\n\n");
```

Example: Forwarding Message 2

```
// Create a multi-part to combine the parts
Multipart multipart = new MimeMultipart();
multipart.addBodyPart(messageBodyPart);

// Create and fill part for the forwarded content
messageBodyPart = new MimeBodyPart();
messageBodyPart.setDataHandler(message.getDataHandler());
// Add part to multi part
multipart.addBodyPart(messageBodyPart);

// Associate multi-part with message
forward.setContent(multipart);
// Send message
Transport.send(forward);
```

Working with Attachments

Attachments are resources associated with a mail message, usually kept outside of the message like a text file, spreadsheet, or image.

With JavaMail you can:

- 1) attach resources to your mail message with the JavaMail API
- 2) and get those attachments when you receive the message

Sending Attachments

To send an attachment with your mail

- 1) Create a new `MimeBodyPart`
- 2) Create a `DataSource` object. A `DataSource` object is part of JAF defined in `javax.activation` package.
- 3) Wrap the `DataSource` object in a `DataHandler`. This will allow us to pass the `DataHandler` to the body part object.

Example: Sending Attachments 1

```
// Define message
Message message = new MimeMessage(session);
message.setFrom(new InternetAddress(from));
message.addRecipient(Message.RecipientType.TO,
                    new InternetAddress(to));
message.setSubject("Hello JavaMail Attachment");

// Create the message part
BodyPart messageBodyPart = new MimeBodyPart();

// Fill the message
messageBodyPart.setText("Pardon Ideas");

Multipart multipart = new MimeMultipart();
multipart.addBodyPart(messageBodyPart);
```


Example: Sending Attachments 2

```
// Part two is attachment
messageBodyPart = new MimeBodyPart();
DataSource source = new FileDataSource(filename);
messageBodyPart.setDataHandler(new
                                DataHandler(source));
messageBodyPart.setFileName(filename);
multipart.addBodyPart(messageBodyPart);

// Put parts in message
message.setContent(multipart);

// Send the message
Transport.send(message);
```

Lab Work: Sending Attachment 1

- 1) The skeleton code already includes the code to get the initial mail session.
- 2) From the session, get a Message and set its header fields: to, from, and subject.
- 3) Create a BodyPart for the main message content and fill its content with the text of the message.
- 4) Create a Multipart to combine the main content with the attachment. Add the main content to the multipart.
- 5) Create a second BodyPart for the attachment.
- 6) Get the attachment as a DataSource.
- 7) Set the DataHandler for the message part to the data source. Carry the original filename along.
- 8) Add the second part of the message to the multipart.

Lab Work: Sending Attachment 2

- 9) Set the content of the message to the multipart.
- 10) Compile and run the program, passing your SMTP server, from address, to address, and filename on the command line. This will send the file as an attachment.

Lab Work: Skeleton Code 1

```
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;
import javax.activation.*;
public class AttachExample {
    public static void main (String args[]) throws
    Exception {
        String host = args[0];
        String from = args[1];
        String to = args[2];
        String filename = args[3];
        // Get system properties
        Properties props = System.getProperties();
```

Lab Work: Skeleton Code 2

```
// Setup mail server
    props.put("mail.smtp.host", host);
// Get session
    Session session = Session.getInstance(props,
                                           null);

// Define message
// Create the message part
// Fill the message
// Create a Multipart
// Add part one //
// Part two is attachment //
// Create second body part
```

Exercise : Skeleton Code 3

```
        // Get the attachment
    // Set the data handler to the attachment
    // Set the filename
    // Add part two
        // Put parts in message
    // Send the message
    }
}
```

Getting Attachments

The content of your message is a `Multipart` object when it has attachments.

You then need to process each `Part`, to get the main content and the attachment(s).

Parts marked with a disposition of `Part.ATTACHMENT` from `part.getDisposition()` are clearly attachments.

However, attachments can also come across with no disposition (and a non-text MIME type) or a disposition of `Part.INLINE`.

Just get the original filename with `getFileName()` and the input stream with `getInputStream()`.

Example: Getting Attachments

```
Multipart mp = (Multipart)message.getContent();

for (int i=0, n=multipart.getCount(); i<n; i++) {
    Part part = multipart.getBodyPart(i);
    String disposition = part.getDisposition();

    if ((disposition != null) &&

        ((disposition.equals(Part.ATTACHMENT) ||

            (disposition.equals(Part.INLINE))) {
        saveFile(part.getFileName(),
            part.getInputStream());
    }
}
```


Writing Attachments

The `saveFile()` method just creates a `File` from the filename, reads the bytes from the input stream, and writes them off to the file.

In case the file already exists, a number is added to the end of the filename until one is found that doesn't exist.

```
// from saveFile()  
File file = new File(filename);  
for (int i=0; file.exists(); i++) {  
    file = new File(filename+i);  
}
```

Attachment: General Case

The code above covers the simplest case where message parts are flagged appropriately.

To cover all cases, handle when the disposition is `null` and get the MIME type of the part to handle accordingly.

```
if (disposition == null) {  
    // Check if plain  
    MimeBodyPart mbp = (MimeBodyPart)part;  
    if (mbp.isMimeType("text/plain")) {  
        // Handle plain  
    } else {  
        // Special non-attachment cases here of  
        // image/gif, text/html, ...  
    }... }
```

Sending HTML Messages

To send a HTML file as the message and let the mail reader worry about fetching any embedded images or related pieces

- 1) use the `setContent()` method of `Message`
- 2) passing along the content as a `String` and setting the content type to `text/html`.

Example:

```
String htmlText = "<H1>Hello</H1>" +  
    "<imgsrc=\"http://www.jguru.com/images/logo.gif\">";  
message.setContent(htmlText, "text/html");
```

Including Images in HTML

if you want your HTML content message to be complete, with embedded images included as part of the message:

- 1) you must treat the image as an attachment
- 2) and reference the image with a special **cid URL**, where the **cid** is a reference to the **Content-ID header** of the image attachment.
- 3) tell the `MimeMultipart` that the parts are related by setting its **subtype** in the constructor (or with `setSubType()`)
- 4) and set the **Content-ID header** for the image to a random string which is used as the **src** for the image in the `` tag.

Example: Including Images 1

```
String file = ...;

// Create the message
Message message = new MimeMessage(session);

// Fill its headers
message.setSubject("Embedded Image");
message.setFrom(new InternetAddress(from));
message.addRecipient(Message.RecipientType.TO,
                     new InternetAddress(to));

// Create your new message part
BodyPart messageBodyPart = new MimeBodyPart();
String htmlText = "<H1>Hello</H1>" + "<img  
src=\"cid:memememe\">";
```

Example: Including Images 2

```
messageBodyPart.setContent(htmlText, "text/html");

// Create a related multi-part to combine the parts
MimeMultipart multipart = new MimeMultipart("related");
multipart.addBodyPart(messageBodyPart);

// Create part for the image
messageBodyPart = new MimeBodyPart();

// Fetch the image and associate to part
DataSource fds = new FileDataSource(file);
messageBodyPart.setDataHandler(new DataHandler(fds));
messageBodyPart.setHeader("Content-ID", "<memememe>");
```

Example: Including Images 3

```
// Add part to multi-part
multipart.addBodyPart(messageBodyPart);

// Associate multi-part with message
message.setContent(multipart);
```

Lab Work: Sending HTML

- 1) The skeleton code already includes the code to get the initial mail session, create the main message, and fill its headers (to, from, subject).
- 2) Create a BodyPart for the HTML message content.
- 3) Create a text string of the HTML content. Include a reference in the HTML to an image () that is local to the mail message.
- 4) Set the content of the message part. Be sure to specify the MIME type is text/html.
- 5) Create a Multipart to combine the main content with the attachment. Be sure to specify that the parts are related. Add the main content to the multipart.
- 6) Create a second BodyPart for the attachment.
- 7) Get the attachment as a DataSource, and set the DataHandler for the message part to the data source.

Lab Work:

- 8) Set the Content-ID header for the part to match the image reference specified in the HTML.
- 9) Add the second part of the message to the multipart, and set the content of the message to the multipart.
- 10) Send the message.
- 11) Compile and run the program, passing your SMTP server, from address, to address, and filename on the command line. This will send the images as an inline image within the HTML text.

Lab Work: Skeleton Code 1

```
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;
import javax.activation.*;

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

        String host = args[0];
        String from = args[1];
        String to = args[2];
        String file = args[3];
```

Lab Work: Skeleton Code 2

```
// Get system properties
Properties props = System.getProperties();

// Setup mail server
props.put("mail.smtp.host", host);

// Get session
Session session = session.getDefaultInstance(props,
null);

// Create the message
Message message = new MimeMessage(session);

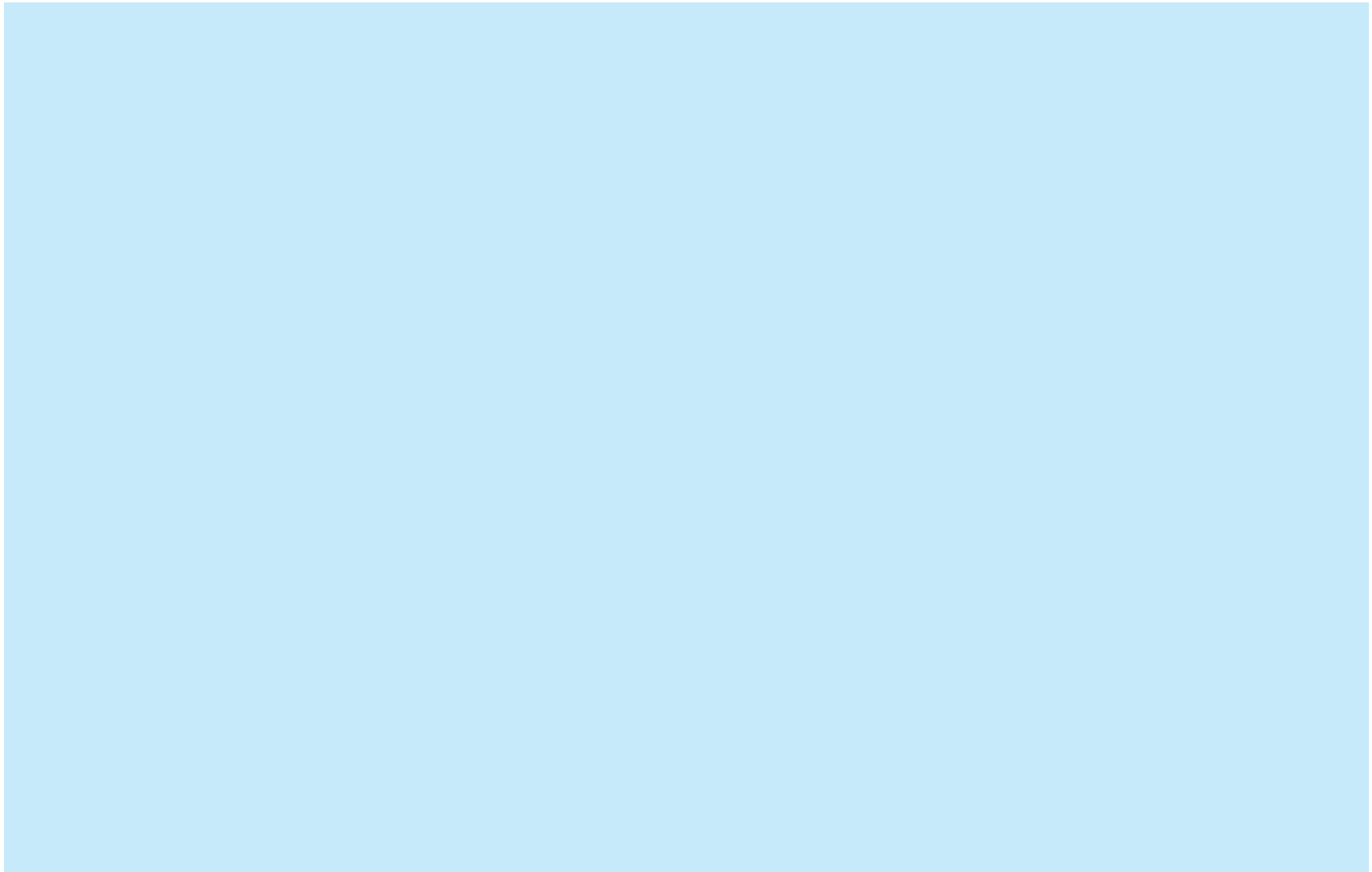
// Fill its headers
message.setSubject("Embedded Image");

message.setFrom(new InternetAddress(from));
```

Lab Work: Skeleton Code 3

```
message.addRecipient(Message.RecipientType.TO,  
    new InternetAddress(to));  
// Create your new message part  
// Set the HTML content, be sure it references  
//the attachment  
// Set the content of the body part  
// Create a related multi-part to combine the  
parts  
// Add body part to multipart  
// Create part for the image  
// Fetch the image and associate to part  
// Add a header to connect to the HTML  
// Add part to multi-part  
// Associate multi-part with message  
// Send message    } }
```

Exercise: JavaMail



Java Message Service

Course Outline

- 1) introduction
- 2) streams
- 3) networking
- 4) database connectivity
- 5) architectures
 - a) message-orientation
 - 1) javamail
 - 2) **jms**
 - b) distributed objects
 - 1) rmi
 - 2) corba
 - 3) JavaIDL
- 6) summary

Overview

- 1) introduction
- 2) JMS Messaging Model
- 3) JMS programming model and implementation
- 4) advance configuration
- 5) summary

Introduction 1

Information systems are increasingly based on distributed architectures

Needs for integrating existing stand-alone systems are increasing

Middleware is an attempt to ease distributed system development, and try to embed complexity of communication between programs such as:

- a) Different data representations & encodings
- b) Different transport protocols
- c) Different programming languages, ...

Introduction 2

Types of middleware

1) Procedure-oriented

a) Client/Server e.g. RPC

2) Object-oriented

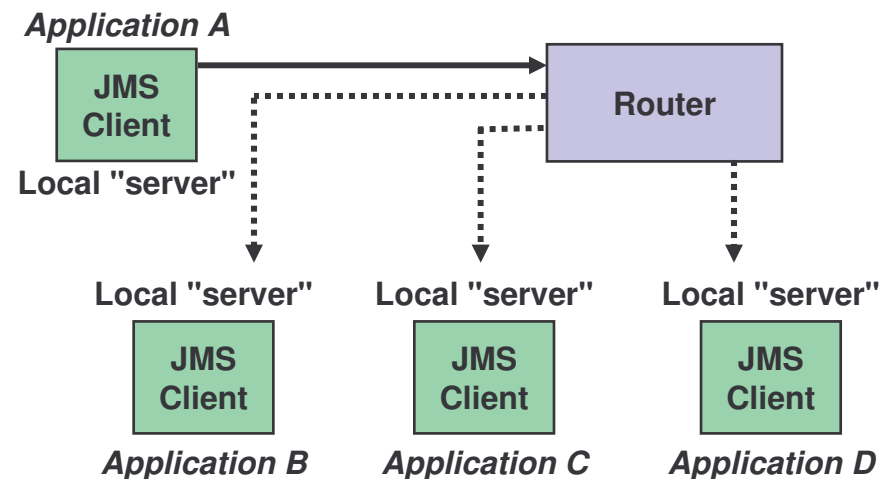
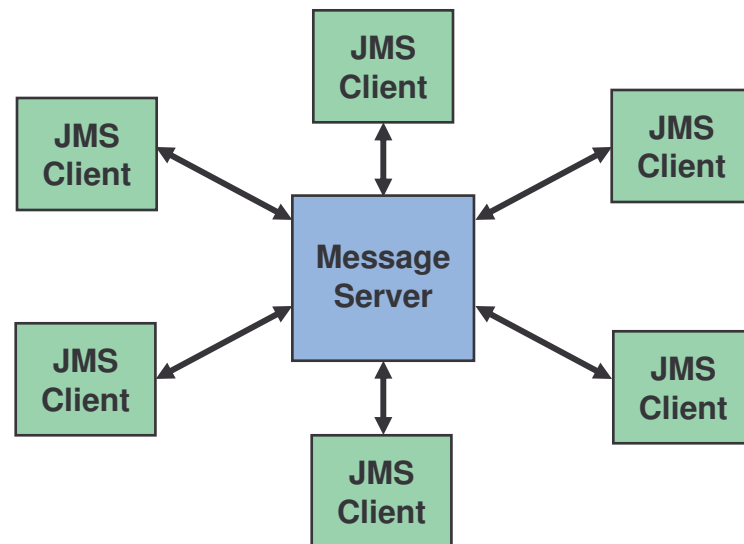
a) Distributed Objects e.g. CORBA, RMI

3) Message Oriented Middlewares(MOMs)

a) Asynchronous messaging e.g. JMS

What is Messaging ?

- 1) A method of peer-to-peer communication between software components or applications.
- 2) Enables distributed communication that is loosely coupled; differs from tightly coupled technologies, such as Remote Method Invocation (RMI), which require an application to know a remote application's methods.

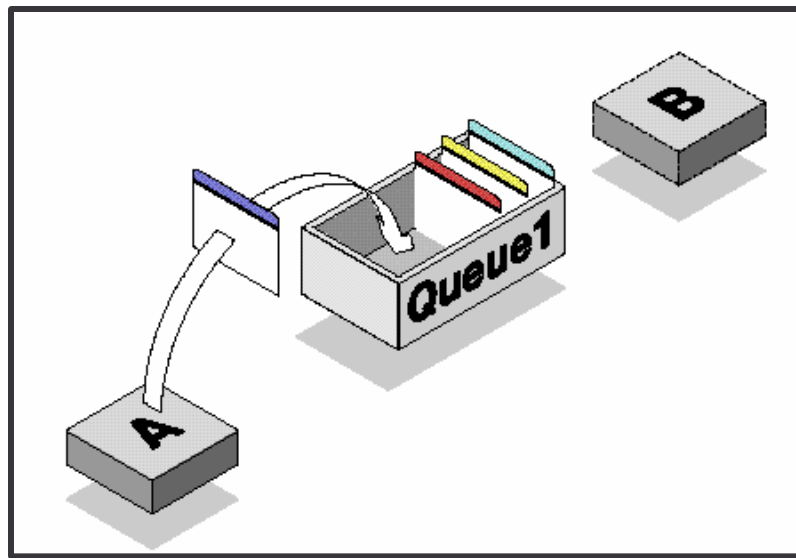


Reliable Messaging With Queues

MOMs provide asynchronous messaging

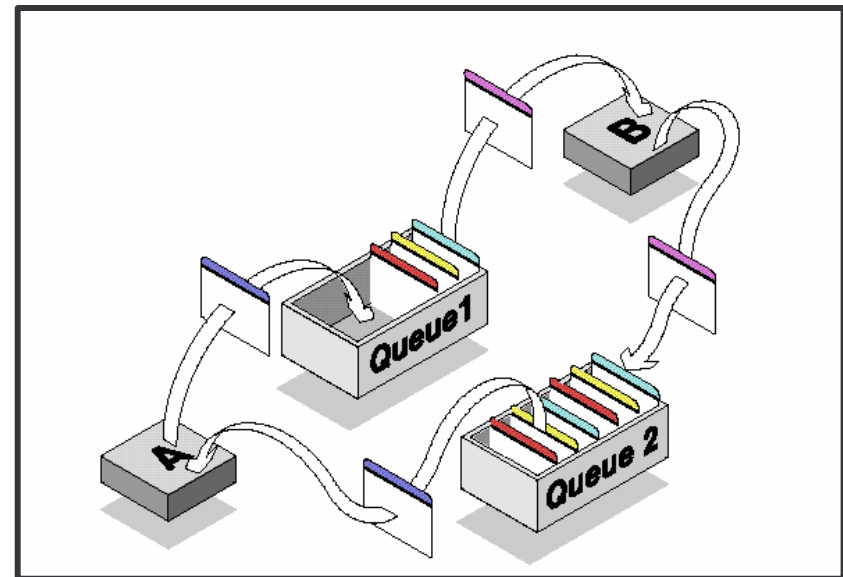
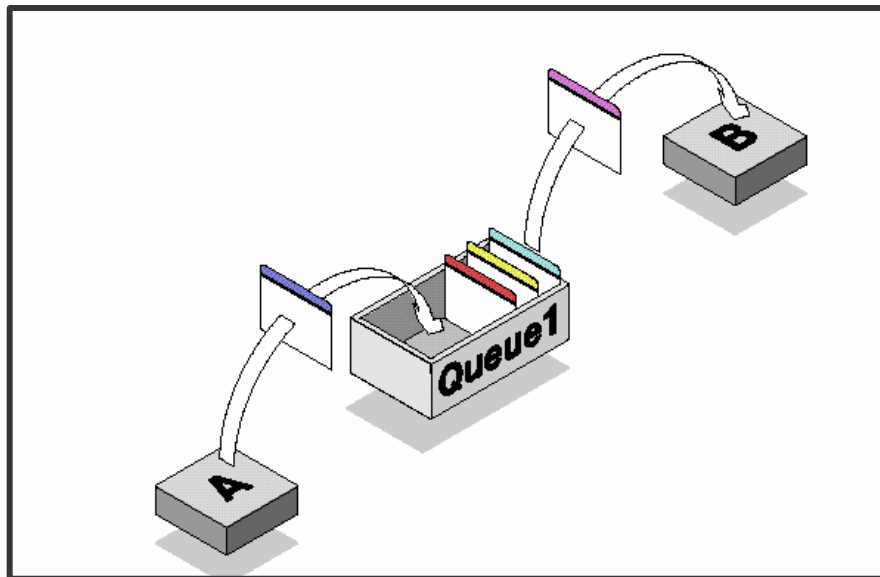
If one party is unavailable, messaging subsystem is still available and functional

Queues exist independent from the applications



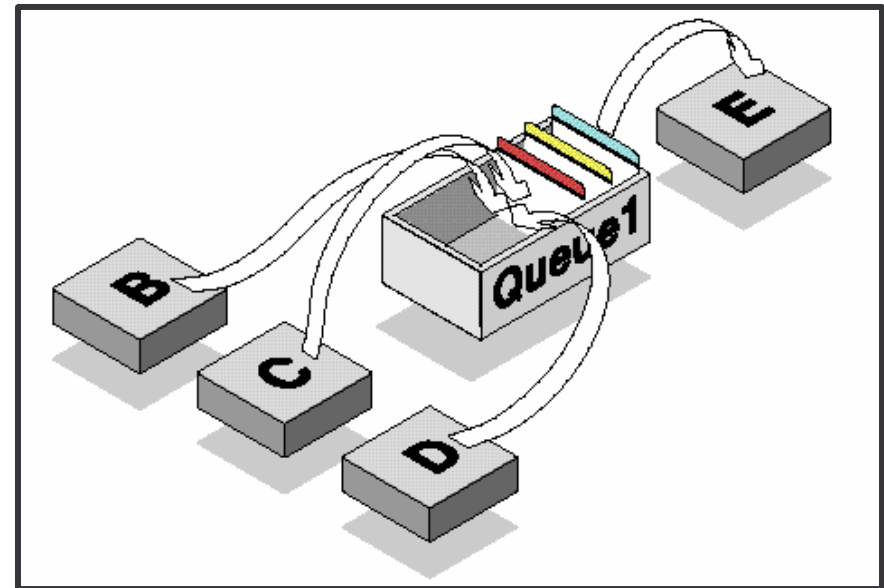
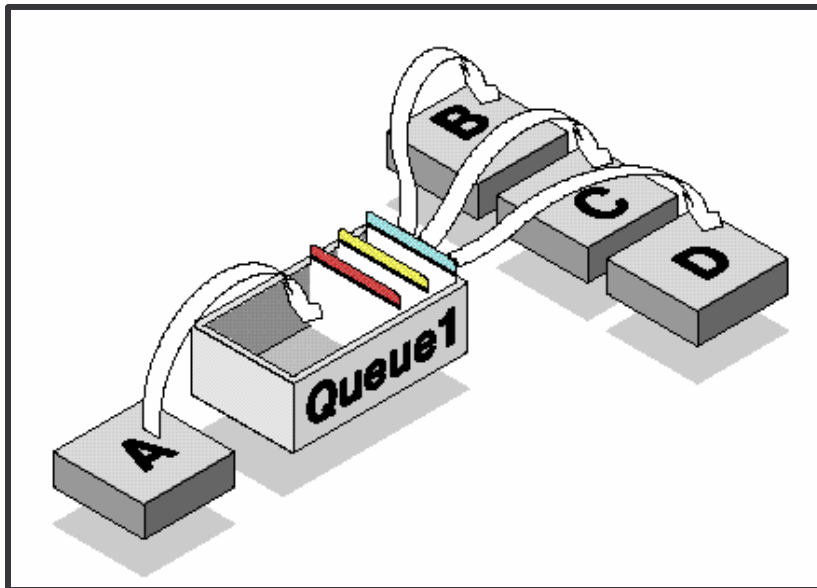
Queuing Basics 1

Queues are uni-directional, but multiple queues may be used to provide bi-directional messaging

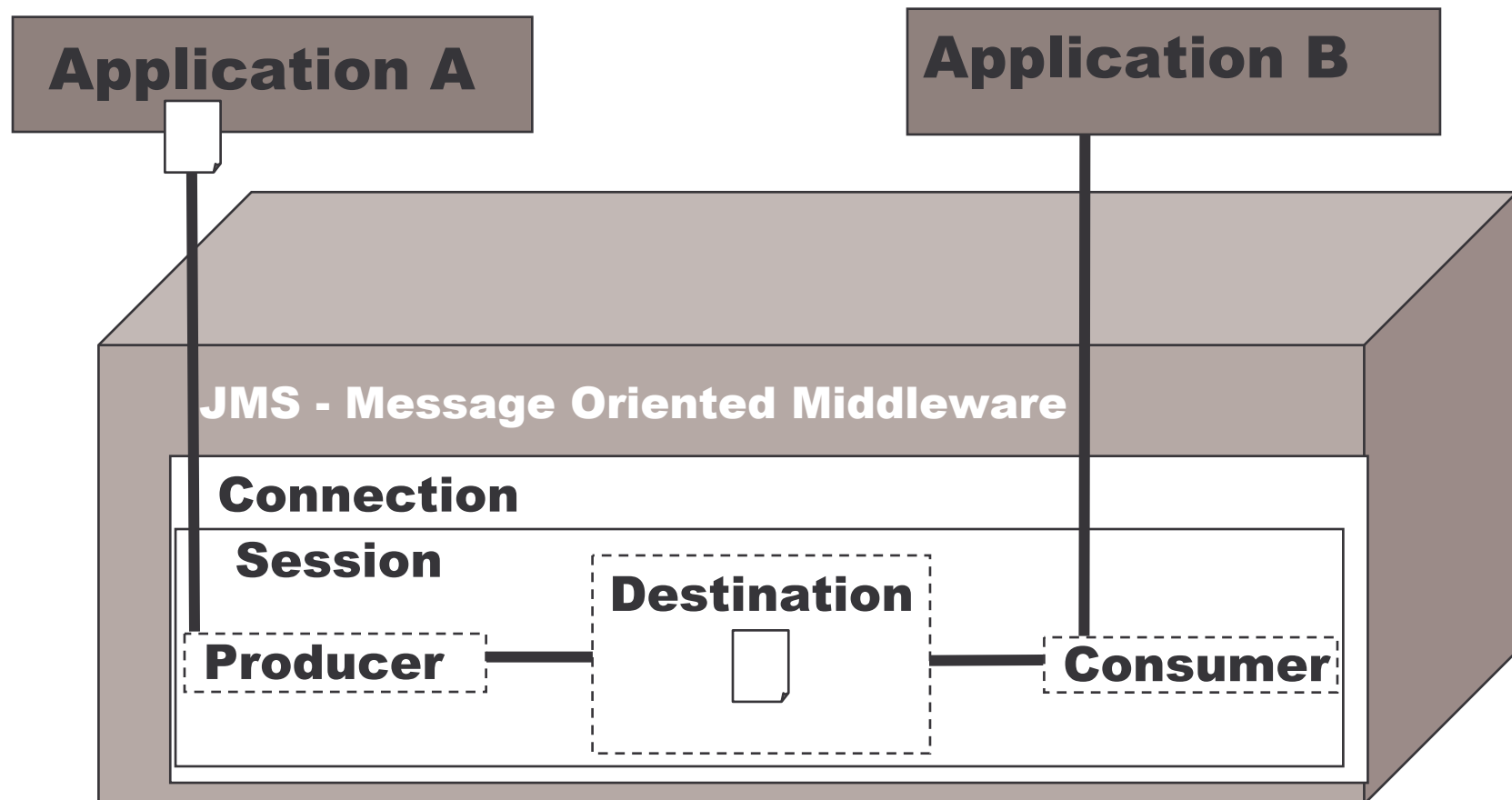


Queuing Basics 2

Queues can work in different models and make one to many and many to one relations possible



Producer and Consumer



The Java Messaging Service 1

A J2EE API to access MOM products from Java

Vendor-neutral API for higher-interoperability

Has two models:

1) Publish and Subscribe

- a) 0 or more recipients

- b) Messages passed between publishers and subscribers via topics

- c) Message can be subscribed to in a durable manner

- d) Messages are consumed at least once

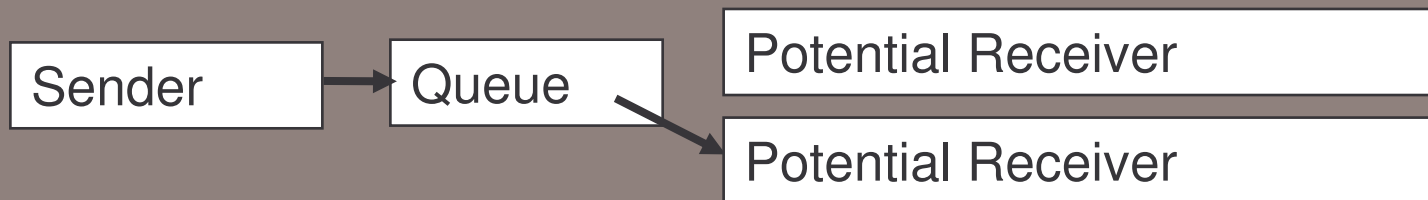
2) Point-to-Point

- a) One recipient only

- b) Messages are consumed at most once and only once

The Java Messaging Service 2

Point to Point (1 to 1)



Publish and Subscribe (1 to Many)



The Promises of JMS

- 1) “Messaging for the masses”
 - a) Could have similar impact that SQL had on databases
 - b) Similar to JDBC (which all vendors now support)
- 2) First enterprise messaging API to achieve wide industry support (standard)
- 3) Simplifies development of enterprise applications (ease of use)
- 4) Leverages existing enterprise-proven messaging systems (implementation)
- 5) Easy to write portable messaging based business applications (write once, run anywhere)

The Promises of JMS

- 1) “Messaging for the masses”
 - a) Could have similar impact that SQL had on databases
- 2) First enterprise messaging API to achieve wide industry support
- 3) Simplifies development of enterprise applications
- 4) Leverages existing enterprise-proven messaging systems
- 5) Easy to write portable messaging based business applications

Limitations of the JMS

JMS does not address

- a) Security
- b) Load Balancing
- c) Fault Tolerance
- d) Error Notification (apart from Exceptions)
- e) Administration API
- f) Transport protocol for messaging

Overview

- 1) introduction
- 2) **JMS messaging model**
- 3) JMS programming model and implementation
- 4) advance configuration
- 5) summary

JMS API Concepts 1

A JMS application is composed of the following parts:

1) JMS Provider

- a) messaging system that implements JMS and administrative functionality, e.g. IBM's MQSeries and JBossMQ message server.

2) JMS Clients

- a) Java programs that send/receive messages

3) Messages

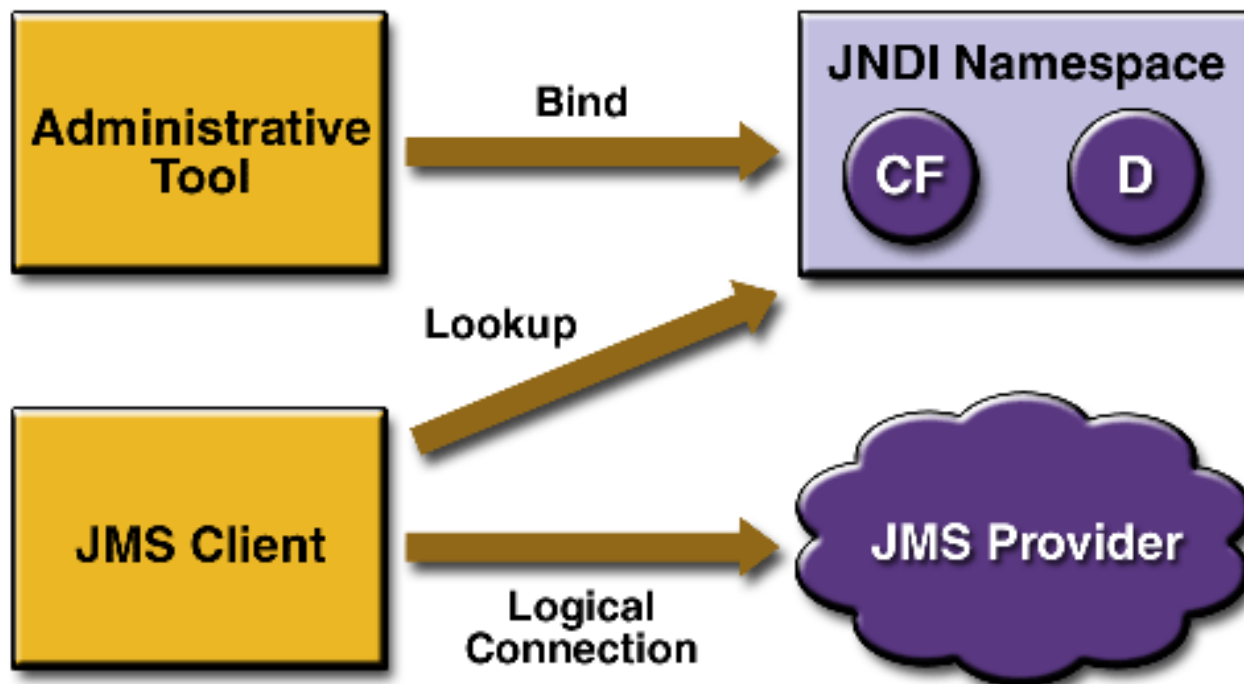
- a) Items of information sent between JMS clients.

4) Administered Objects

- a) preconfigured JMS objects created by an admin for the use of clients
- b) `ConnectionFactory`, `Destination` (queue or topic)

JMS API Concepts 2

Interaction between different parts of JMS:



JMS Messaging Domains

Point-to-Point (PTP)

- a) built around the concept of message queues
- b) each message has only one consumer

Publish-Subscribe systems

- a) uses a “topic” to send and receive messages
- b) each message has multiple consumers

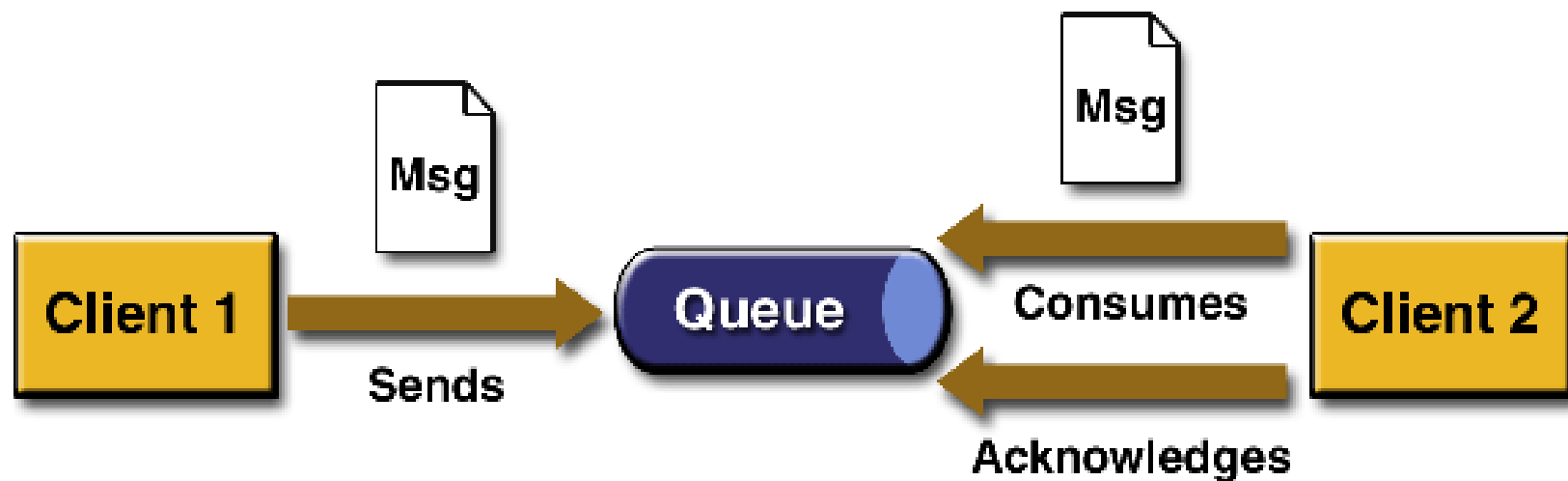
Point-to-Point Messaging 1

Each message is addressed to a specific queue

Receiving clients extract messages from the queue(s)

Queues retain all messages sent to them until:

- a) the messages are consumed
- b) the messages expire



Point-to-Point Messaging 2

Characteristics of Point-to-Point Messaging :

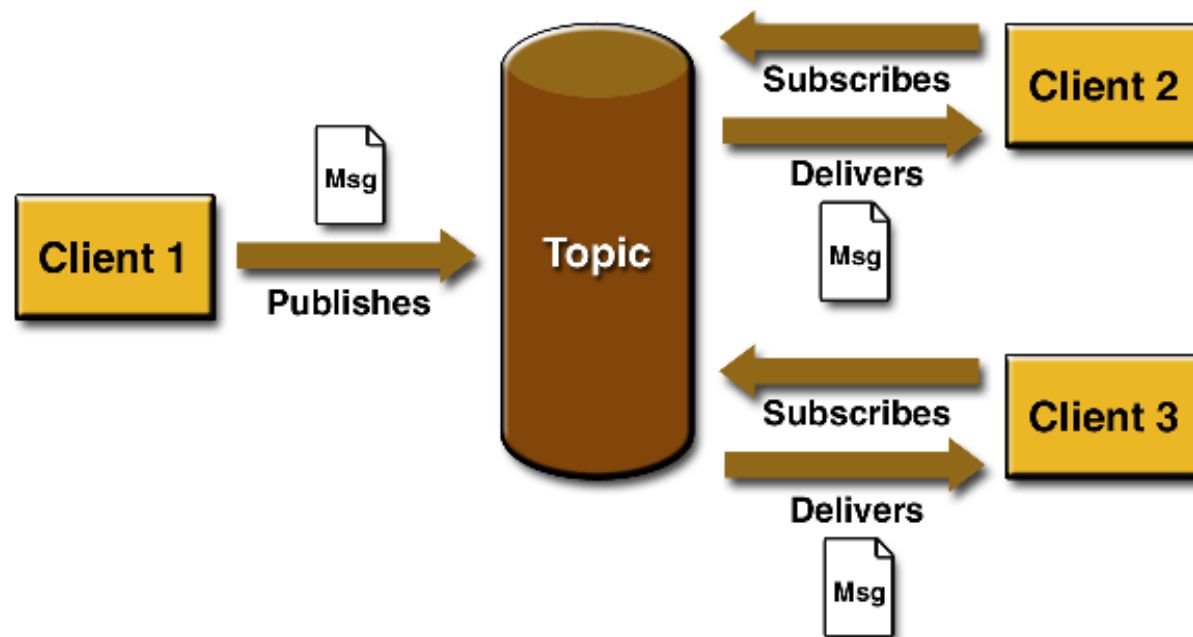
- a) Each message has only one consumer.
- b) A sender and a receiver of a message have no timing dependencies.
- c) The receiver acknowledges the successful processing of a message.
- d) Should be used when every message send must be processed successfully by one consumer.

Publish/Subscribe Messaging 1

Clients address messages to a topic.

The system takes care of distributing the messages.

Topics retain messages only as long as it takes to distribute them to current subscribers.



Publish/Subscribe Messaging 2

Characteristics of Pub/Sub Messaging :

- a) Each message may have multiple consumers.
- b) A client that subscribes to a topic can consume only messages published after the client has created a subscription.
- c) The subscriber must continue to be active in order for it to consume messages.
- d) Exception for time dependency is allowed for durable subscription.
(Will be discussed later)

JMS Message

A JMS message has three parts:

- 1) Message Header
 - a) used for identifying and routing messages
 - b) contains vendor-specified values, but could also contain application-specific data
 - c) typically name/value pairs
- 2) Message Properties (optional)
 - a) act like additional headers
- 3) Message Body(optional)
 - a) contains the data
 - b) five different message body types in the JMS specification

JMS Header

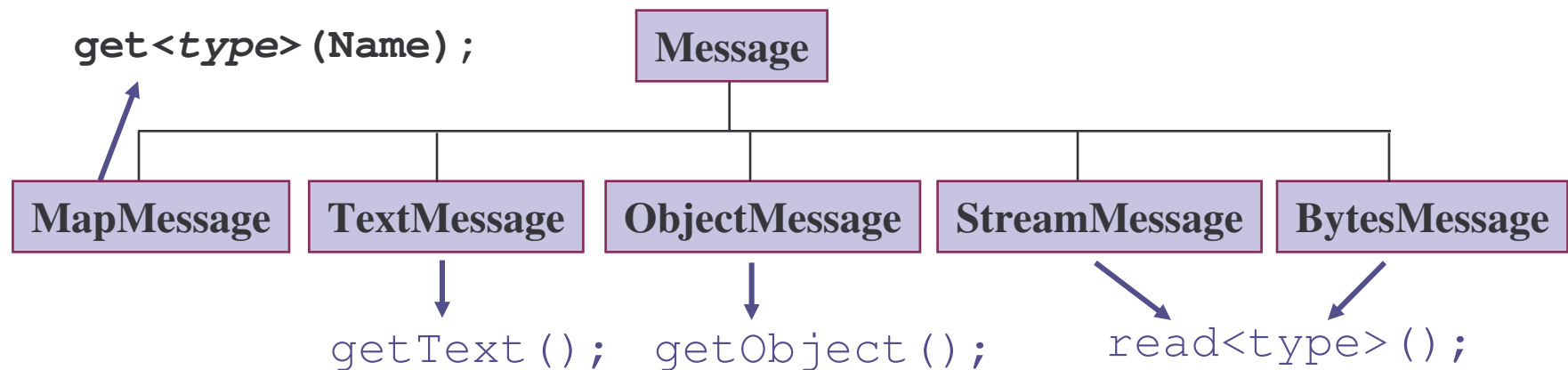
Automatically assigned headers
<code>JMSDestination</code>
<code>JMSDeliveryMode</code>
<code>JMSMessageID</code>
<code>JMSTimestamp</code>
<code>JMSExpiration</code>
<code>JMSRedelivered</code>
<code>JMSPriority</code>

Developer-assigned headers
<code>JMSReplyTo</code>
<code>JMSCorrelationID</code>
<code>JMSType</code>

JMS Message Types

Message Type	Contains	Some Methods
TextMessage	String	getText, setText
MapMessage	set of name/value pairs	setString, setDouble, setLong, getDouble, getString
BytesMessage	stream of uninterpreted bytes	writeBytes, readBytes, writeString, readString
StreamMessage	stream of primitive values	writeString, writeDouble, writeLong, readString
ObjectMessage	serialize object	setObject, getObject

Accessing JMS Message



Messages Consumption

In the JMS Specification, messages can be consumed in either of two ways:

Synchronously

- a) A subscriber or a receiver explicitly fetches the message from the destination by calling the receive method.
- b) The receive method can *block* until a message arrives or can time out if a message does not arrive within a specified time limit.

Asynchronously

- a) A client can register a *message listener* with a consumer.
- b) Whenever a message arrives at the destination, the JMS provider delivers the message by calling the listener's `onMessage()` method.

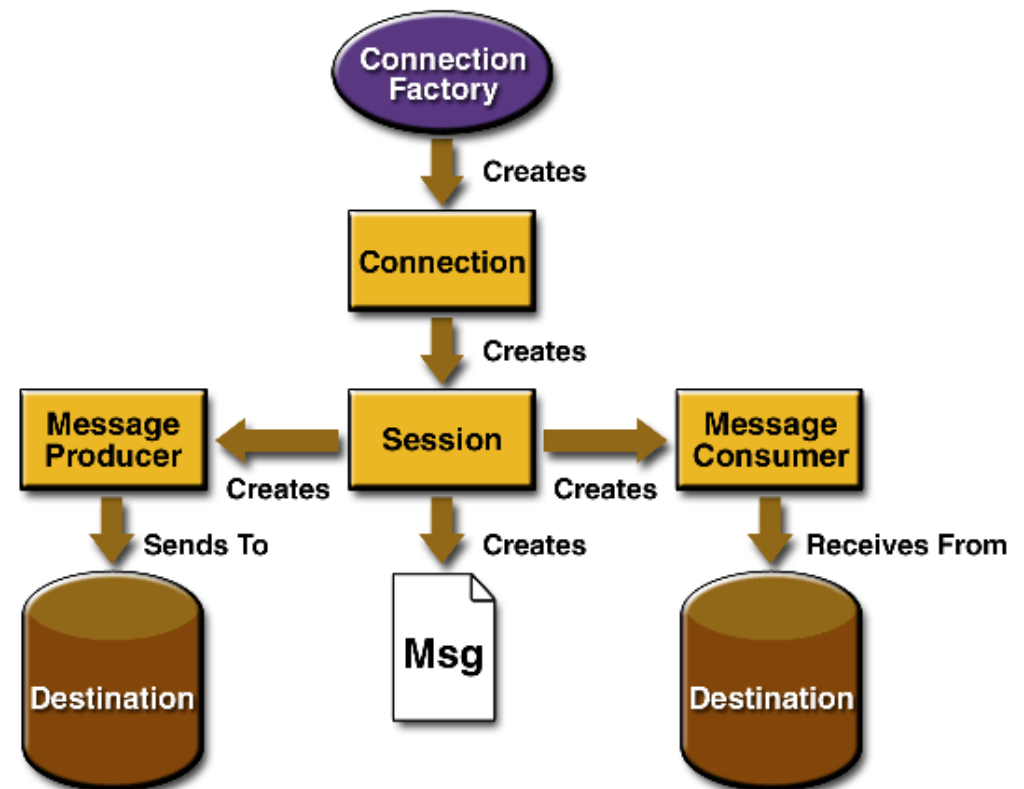
Overview

- 1) introduction
- 2) JMS Messaging Model
- 3) **JMS programming model and implementation**
- 4) advance configuration
- 5) summary

JMS API Programming Model

The basic building blocks of a JMS application:

- 1) Administered objects
- 2) Sessions
- 3) Message producers
- 4) Message consumers
- 5) Messages



JMS Client Setup Procedure

A typical pub/sub JMS client executes the following setup procedure:

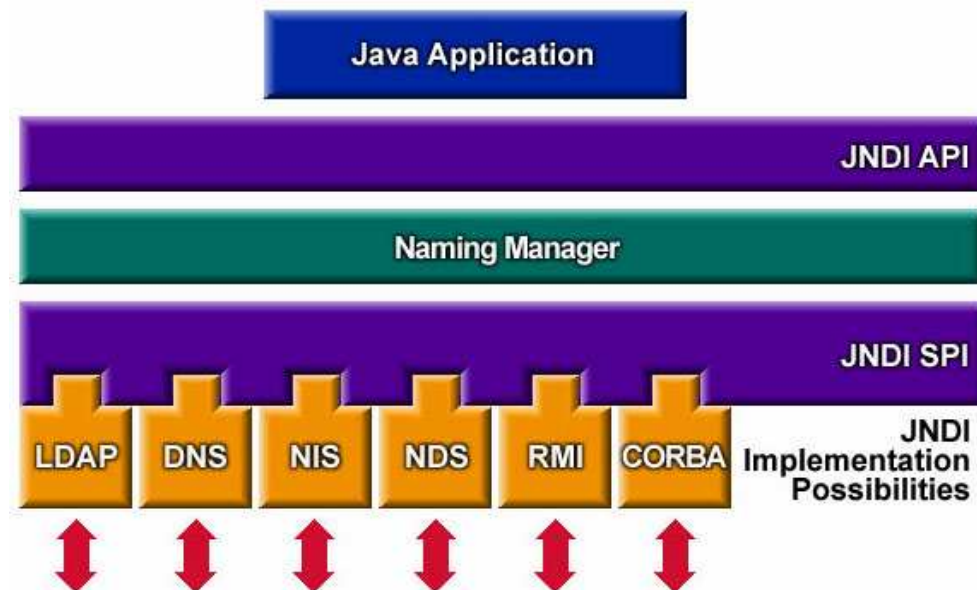
- 1) Use `JNDI` to find a `ConnectionFactory` object
- 2) Use `JNDI` to find one or more `Destination` objects
- 3) Use the `ConnectionFactory` to create a JMS Connection
- 4) Use the Connection to create one or more JMS Sessions
- 5) Use a Session and the Destinations to create the `TopicPublisher` and `TopicSubscriber` needed
- 6) Enable the Connection to start delivering messages to `TopicSubscriber`

What is JNDI

Java Naming and Directory Interface ([JNDI](#)) is an integral component of J2EE technology

[JNDI](#) is an application programming interface (API) that provides directory and naming services to Java applications.

[JNDI](#) is defined to be independent of any specific naming or directory service implementation. A variety of services can be accessed in a common way.



JNDI package

Following are the JNDI packages:

- 1) `javax.naming`
- 2) `javax.naming.directory`
- 3) `javax.naming.event`
- 4) `javax.naming.ldap`
- 5) `javax.naming.spi`

Obtain JNDI Connection 1

- 1) Instantiate an Properties object:

```
Properties env = new Properties();
```

- 2) Specify the JNDI properties specific to the vendor:

```
env.put("java.naming.factory.initial",  
        "org.jnp.interfaces.NamingContextFactory");  
env.put("java.naming.provider.url",  
        "jnp://localhost:1099");  
env.put("java.naming.factory.url.pkgs",  
        "org.jboss.naming:org.jnp.interfaces");
```

- 3) Obtain JNDI Connection

```
Context jndi=new InitialContext(env);
```

Obtain JNDI Connection 2

If a file named `jndi.properties` is in the classpath of the client program, you can use the following setting:

```
Context jndi = new  
    InitialContext (System.getProperties());
```

This can remove the vendor specific code from the client program.

Setup Using JNDI

- 1) Use JNDI to find a `ConnectionFactory` object :

```
TopicConnectionFactory conFactory =  
    (TopicConnectionFactory) jndi.lookup  
        ("ConnectionFactory");
```

- 2) Use JNDI to find one or more `Destination` objects :

```
Topic myTopic =  
    (Topic) jndi.lookup(topicName);
```

remark: In JBoss, the topic name can be found in the file :

```
<Jboss_Home>\server\default\deploy\jms\jbossmq-  
destinations-service.xml
```

Setup Connection and Session

1) Use `ConnectionFactory` to create a JMS Connection

```
TopicConnection connection =  
    conFactory.createTopicConnection();
```

2) Use the Connection to create one or more JMS Sessions

```
TopicSession pubSession =  
    connection.createTopicSession(false,  
        Session.AUTO_ACKNOWLEDGE);  
  
TopicSession subSession =  
    connection.createTopicSession(false,  
        Session.AUTO_ACKNOWLEDGE);
```

Message Publisher

1) Creating producer

```
MessageProducer producer=  
    pubSession.createProducer(myTopic);
```

2) Send a message

```
TextMessage m=pubSession.createTextMessage();  
m.setText("just another message");  
publisher.publish(m);
```

3) Closing the connection

```
connection.close();
```

Message Subscriber

1) Creating subscriber

```
TopicSubscriber subscriber =  
    subSession.createSubscriber(myTopic);
```

2) Set a JMS message listener

```
subscriber.setMessageListener(<Message Listener>);
```

Message Listener 1

A Message Listener is a class implements interface `javax.jms.MessageListener` and has to implement the `onMessage(javax.jms.Message message)` method

Example of `onMessage` method:

```
public void onMessage(Message message) {  
    TextMessage msg = null;  
    try {  
        if (message instanceof TextMessage) {  
            msg = (TextMessage) message;  
            System.out.println("Reading message: " +  
                msg.getText());  
        }  
    }  
}
```

Message Listener 2

```
else {  
    System.out.println("Message of wrong type: "  
+ message.getClass().getName());  
}  
  
} catch (JMSEException e) {  
    System.out.println("JMSEException in  
onMessage(): " + e.toString());  
}  
  
} catch (Throwable t) {  
    System.out.println("Exception in onMessage(): "  
+ t.getMessage());  
}  
  
}
```

Start and Close the Connection

Enable the Connection to start delivering messages to `TopicSubscriber`

```
connection.start();
```

Stop the Connection before ending the client program.

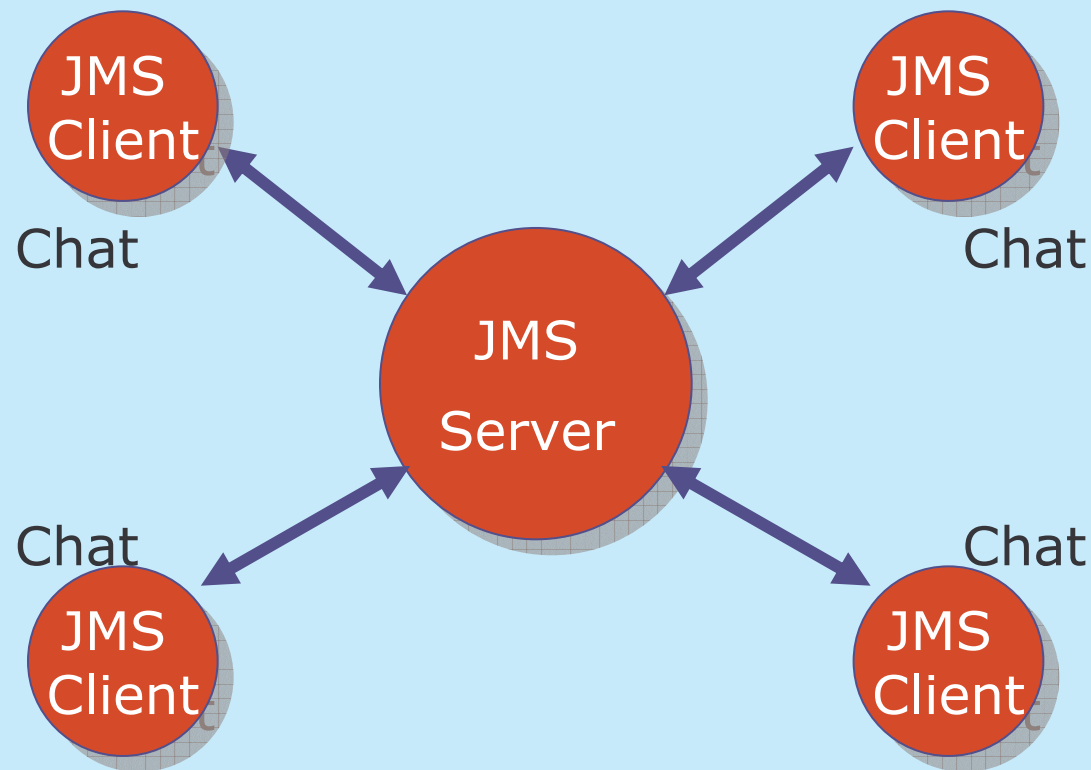
```
connection.close();
```

Both methods throws `javax.jms.JMSException`

Lab Work: A JMS Chat Client 1

- 1) According to the procedure we discussed, write a pub/sub chatting program using JMS.
 - a) Use JBoss as the JMS server.
 - b) Create a topic “emacao” in JBoss. You can modify the file `<JBoss Home>\server\default\deploy\jms\jbossmq-destinations-service.xml` for creating a topic.
 - c) Execute the program from the command line:
 1. `Java Chat topic/emacao username`
 2. Note: for JBoss, the default JNDI name for a topic is `topic/<topic name>`
 3. and is `queue/<queue name>` for a queue.

Lab Work: A JMS Chat Client 2

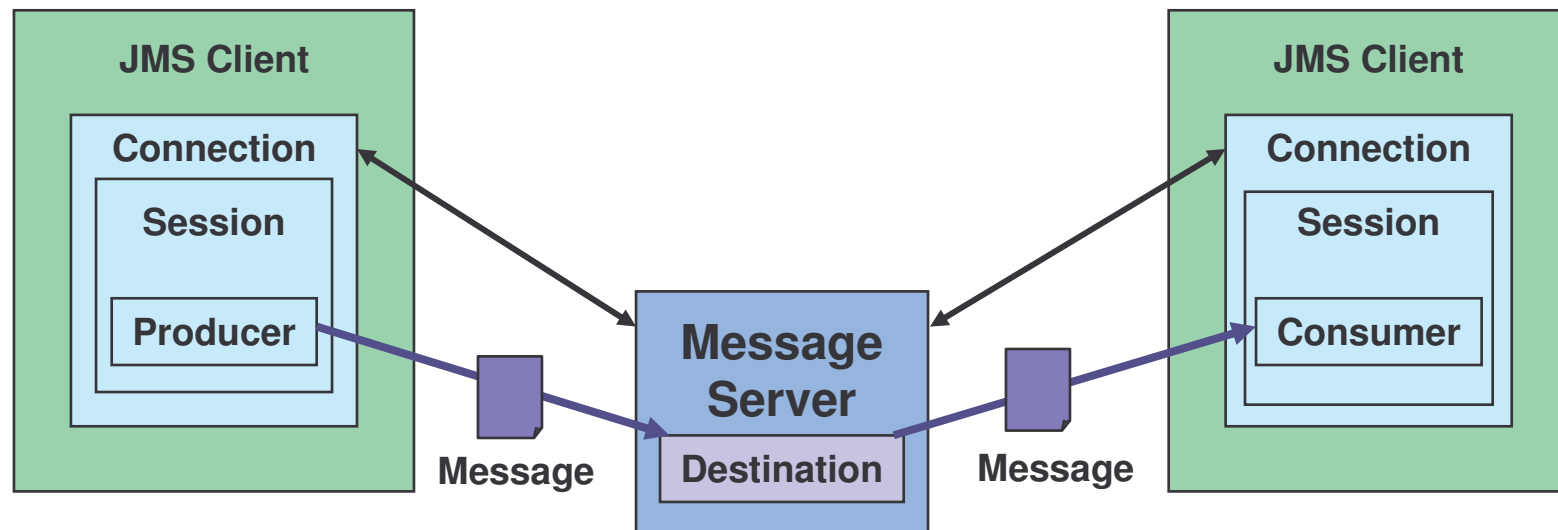


Point-to-Point Messaging 1

Point-to-Point (PTP) application is built around the concept of message queues, sender and receivers.

- a) Each message is addressed to a specific queue and the receiving clients extract messages from the queues established to hold their messages.
- b) Each message has only one consumer.
- c) A sender and receiver have no time dependencies.
- d) The receiver acknowledges the successful processing of a message.
- e) Use PTP when every message you send must be processed successfully by one consumer

Point-to-Point Messaging 2



Message Queue Sender

- 1) Performs a Java Naming and Directory Interface (JNDI) API lookup of the `QueueConnectionFactory` and `Queue`.
- 2) Creates a `QueueConnection` and a `QueueSession`.

- 3) Creating Queue Sender

```
javax.jms.QueueSender
```

```
sender=session.createSender(<queue name>);
```

- 4) Send a message

```
Message m=session.createTextMessage();
```

```
m.setText("just another message");
```

```
sender.send(m);
```

- 5) Closing the connection

```
connection.close();
```

Message Queue Receiver

- 1) Performs a Java Naming and Directory Interface (JNDI) API lookup of the `QueueConnectionFactory` and `Queue`.
- 2) Creates a `QueueConnection` and a `QueueSession`.

- 3) Creating Queue Receiver

```
javax.jms.QueueReceiver
```

```
queueReceiver=session.createReceiver(<queue name>);
```

- 4) Starts the connection, causing message delivery to begin
- 5) Receives the messages sent to the queue until the end-of-message-stream

- `Message m = queueReceiver.receive();`
- `Message m = queueReceiver.receive(0);`

- 6) Closing the connection

```
connection.close();
```

Timed Synchronous Receive

If you do not want your program to consume system resources unnecessarily, do one of the following:

- 1) Call the receive method with a timeout argument greater than 0:

```
Message m = queueReceiver.receive(1); // 1 ms
```

- 2) Call the receiveNoWait method, which receives a message only if one is available:

```
Message m = queueReceiver.receiveNoWait();
```

- 3) The `receive()` method is also available for the `TopicSubscriber` and will negate the use of the `onMessage()` callback.

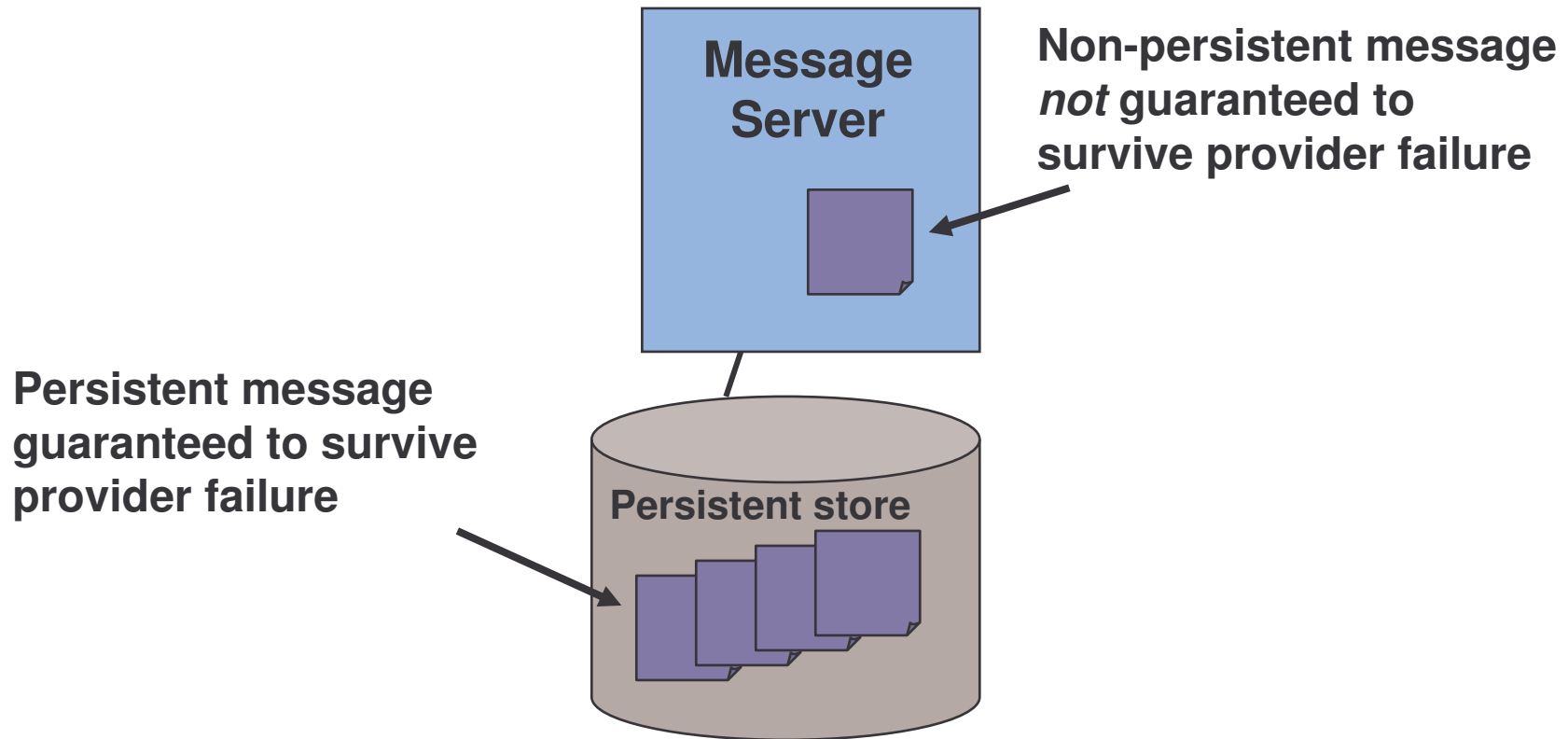
Basic Reliability Mechanisms

- 1) Specifying message persistence.
 - a) You can specify that messages are persistent, meaning that they must not be lost in the event of a provider failure.
- 2) Controlling message acknowledgment.
 - a) You can specify various levels of control over message acknowledgment.
- 3) Setting message priority levels.
- 4) Allowing messages to expire.
 - a) You can specify an expiration time for messages

Persistent Messages

Messages can be marked as persistent.

The implementation of the storage mechanism is up to the JMS provider.



Exercise: Message Queue

- 1) Create Point-to-Point messaging program
 - a) Create a queue in JBoss with a name qex.
 - b) According to our discussion, please create a JMS client for sending message to the qex queue.
 - c) Create a Queue Receiver for the qex queue.
 - d) Try to stop the receiver and use the sender to send some message. Restart the receiver and check if it receive the message.

Overview

- 1) introduction
- 2) JMS Messaging Model
- 3) JMS programming model and implementation
- 4) **advance configuration**
- 5) summary

Temporary Topics 1

Is a topic that is dynamically created by the JMS provider, using the `createTemporaryTopic()` of the `TopicSession` object.

Is a topic associated with the connection that belongs to the `TopicSession` that created it.

It lasts only as long as its associated client connection is active.

Topic identity is transferred using the `JMSReplyTo` header.

Temporary Topics 2

Procedure to create a temporary topic:

- 1) After a session, `mySession`, is created, the client can create a dynamic topic:

```
javax.jmx.Topic tempTopic =  
    mySession.createTemporaryTopic();
```

- 2) Create a message for the subscriber to reply to:

```
javax.jmx.TextMessage message =  
    mySession.createTextMessage();
```

- 3) Set up the `JMSReplyTo` destination

```
message.setJMSReplyTo(tempTopic);
```

Temporary Topics 3

When a client needs to respond to the message, it can use the `JMSReplyTo` Destination:

```
public void onMessage(javax.jms.Message amessage) {  
    ...  
    TextMessage message = (TextMessage) amessage;  
    javax.jms.Topic tempTopic =  
        (javax.jms.Topic) message.getJMSReplyTo();  
}
```

Durable Subscriptions

By default a subscriber gets only messages published on a topic while a subscriber is alive

Durable subscription retains messages until they are received by a subscriber or expire

You can use the `createDurableSubscriber` method of the `java.jms.TopicSession` to create a durable subscription:

...

```
javax.jms.TopicSubscriber subscriber =  
    session.createDurableSubscriber(tempTopic, "subsc  
ription name");
```

...

Unsubscribing

In order to explicitly unsubscribe a subscription, you can use the following methods:

For nondurable subscription:

```
...  
subscriber.close();
```

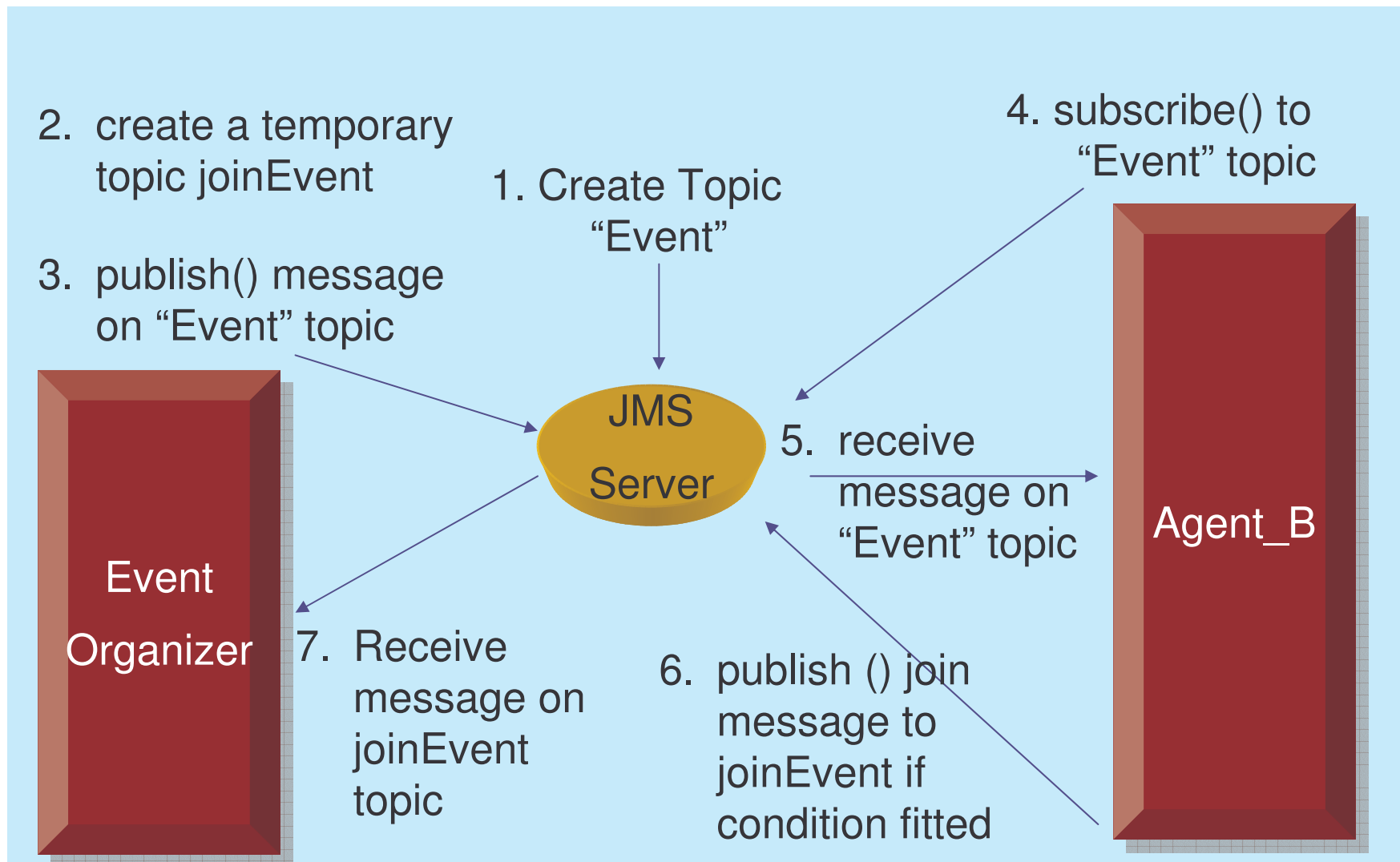
For durable subscription:

```
...  
session.unsubscribe  
("<subscription name>");
```

Lab Work: Temporary Topic 1

- 1) Write two JMS clients to simulate the following scenario :
 - a) An event organizer is constantly promoting events for its agents. It will publish the event message to and deliver to all the subscribed agents.
 - b) After received the message, the Agents' program will evaluate the event according to certain criteria and decide to either joining the event or not. In the exercise, you can make up your own criteria such as cost or date.
 - c) If the agent decided to join the event, it's program will automatically send a message back to the organizer.
 - d) Your organizer's program required to create a temporary topic and attached it as the destination for the agents to reply to.

Lab Work: Temporary Topic 2



Overview

- 1) introduction
- 2) JMS Messaging Model
- 3) JMS programming model and implementation
- 4) advance configuration
- 5) **summary**

Summary

In this session, we cover the followings:

- 1) Concepts of Message Oriented Middleware
- 2) Concepts of Messaging
- 3) Design model of Java Message Service
- 4) Programming Model of Java Message Service
- 5) Programming publisher/subscriber JMS application
- 6) Programming Point-to-Point JMS application

Distributed Objects

Course Outline

- 1) introduction
- 2) streams
- 3) networking
- 4) database connectivity
- 5) architectures
 - a) message-orientation
 - 1) javamail
 - 2) jms
 - b) distributed objects
 - 1) rmi
 - 2) corba
 - 3) JavaIDL
- 6) summary

Overview

What options do I have for distributed application development?

Developers who program using the Java programming language can choose several solutions for creating distributed applications programs.

- 1) Java RMI technology
- 2) Java IDL technology (for CORBA programmers)
- 3) Enterprise JavaBeans technology

In this section we shall be talking about Java RMI and IDL technologies.

Java RMI

Course Outline

- 1) introduction
- 2) streams
- 3) networking
- 4) database connectivity
- 5) architectures
 - a) message-orientation
 - 1) javamail
 - 2) jms
 - b) distributed objects
 - 1) rmi
 - 2) corba
 - 3) JavaIDL
- 6) summary

Overview

- 1) introduction
- 2) RMI architecture
- 3) implementing and running RMI system
- 4) Implementing activatable RMI server
- 5) summary

Introduction 1

Remote Method Invocation (RMI) technology was first introduced in JDK1.1.

RMI allows programmers to develop distributed Java programs with the same syntax and semantics used for non-distributed programs.

RMI is based on a similar, earlier technology for procedural programming called remote procedure call (RPC)

Introduction 2

Disadvantages of RPC

- a) RPC supports a limited set of data types

Therefore it is not suitable for passing and returning Java Objects

- b) RPC requires the programmer to learn a special interface definition language (IDL) to describe the functions that can be invoked remotely

Introduction 3

The RMI architecture defines

- a) How objects behave.
 - b) How and when exceptions can occur.
 - c) How memory is managed.
 - d) How parameters are passed to, and returned from, remote methods.
- The remote object model for Enterprise JavaBeans (EJB) is RMI-based.

Introduction 4

RMI is designed for Java-to-Java distributed applications.

RMI is simpler and easier to maintain than using socket.

Other options for creating Java-to-non-Java distributed applications are:

- a) Java Interface Definition Language (IDL)
- b) Remote Method Invocation (RMI) over Internet Inter-ORB Protocol (IIOP) -- RMI-IIOP.

Overview

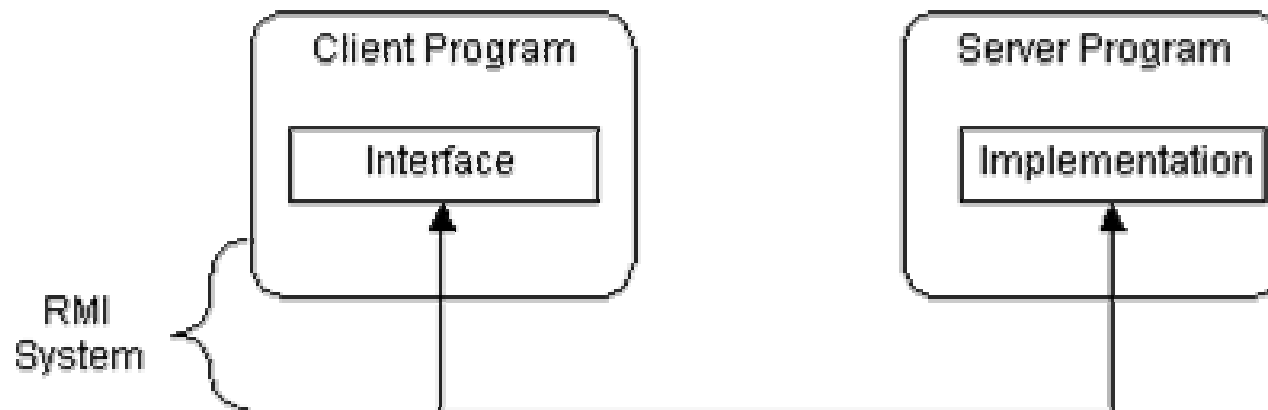
- 1) introduction
- 2) **RMI architecture**
- 3) implementing and running RMI system
- 4) Implementing activatable RMI server
- 5) summary

Architecture 1

RMI allows the code that defines the behavior and the code that implements the behavior to remain separate and to run on separate JVMs.

At client side, RMI uses interfaces to define behavior.

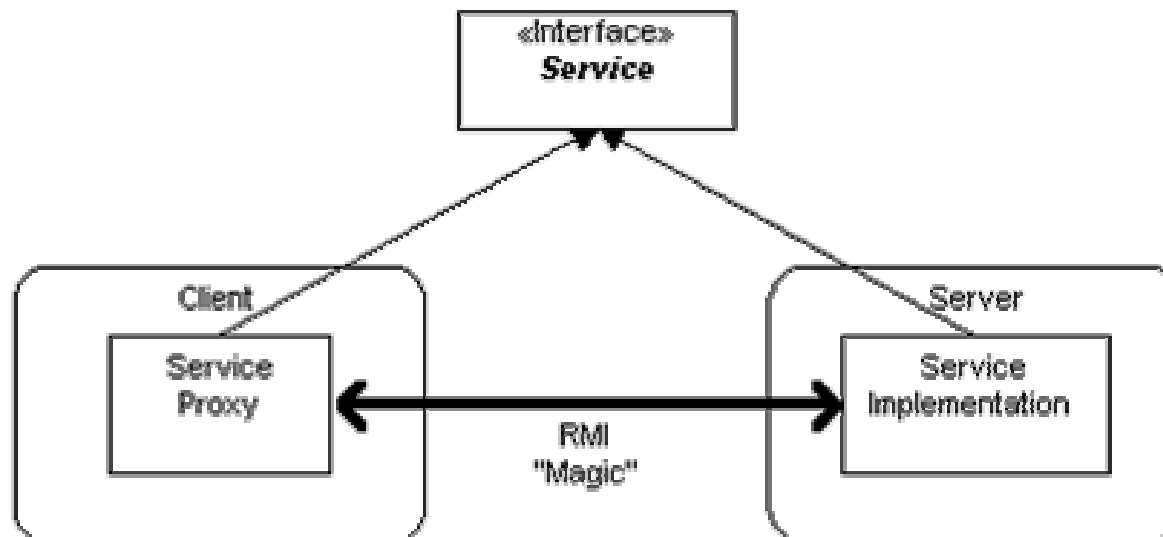
At server side, RMI uses classes to define implementation.



Architecture 2

The service interface is implemented by two classes.

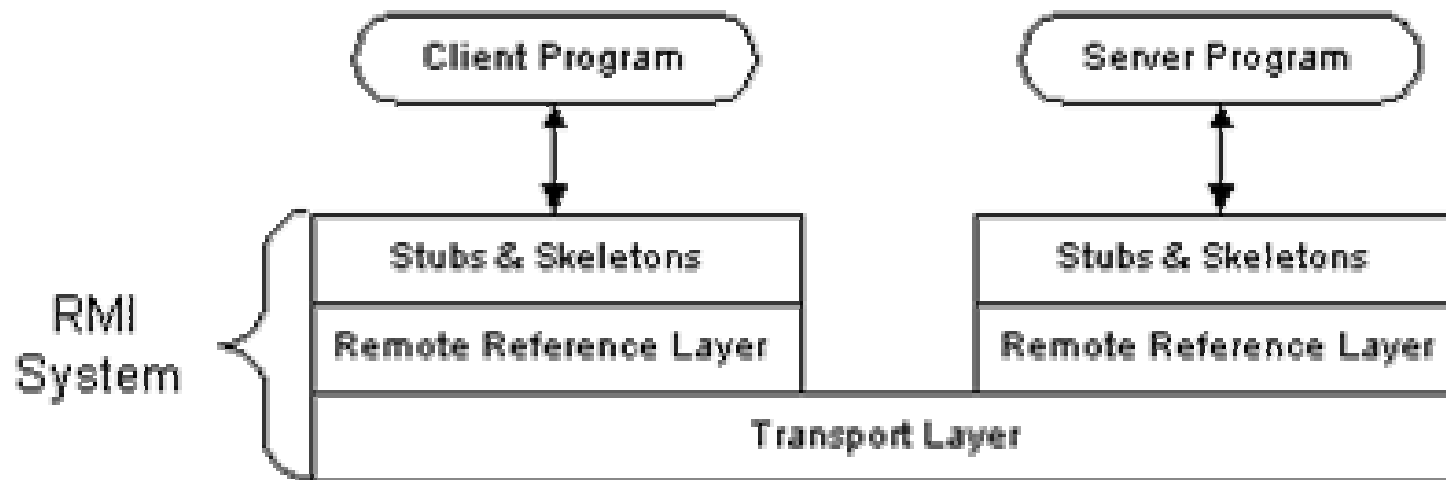
- The first one is at the server side which implements the behavior.
- The second one is at the client side which acts as a proxy.



Layers

The RMI implementation is built from three abstraction layers.

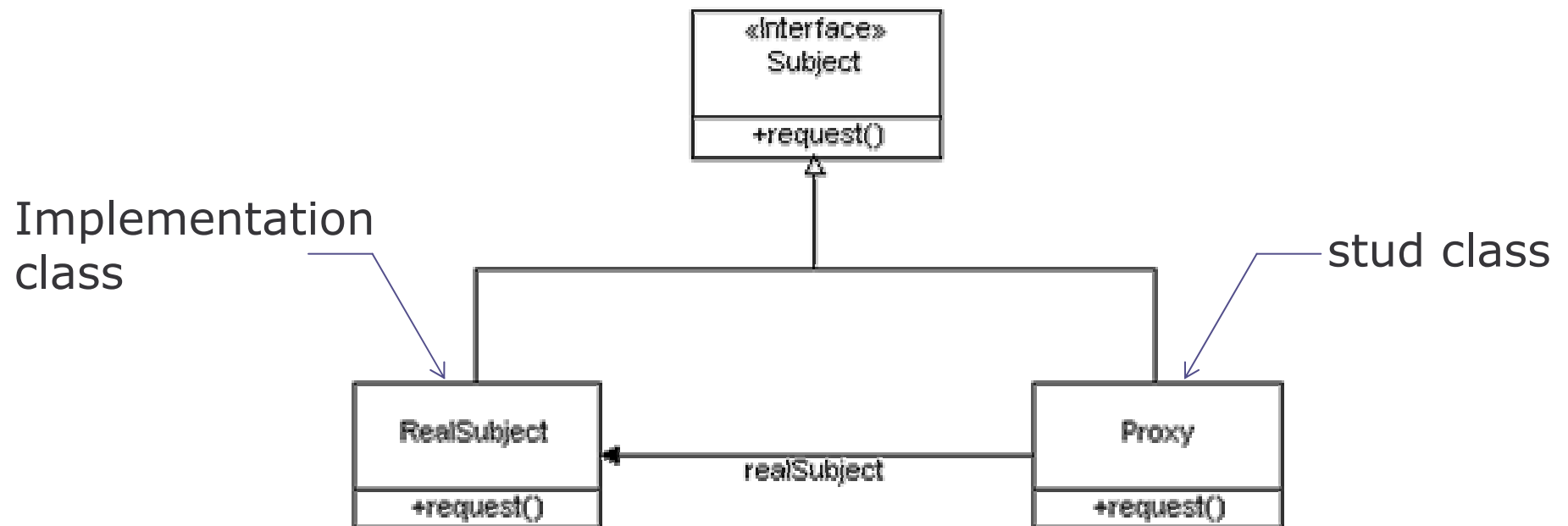
- a) The Stub and Skeleton layer
- b) The Remote Reference Layer
- c) The transport layer



Stub and Skeleton Layer 1

The first layer lies beneath the view of the developer intercepts method calls made by the client to the interface reference variable and redirects these calls to a remote RMI service.

RMI uses the Proxy design pattern in this layer.



Stub and Skeleton Layer 2

A skeleton is a helper class that is generated for to communicate with the stub across the RMI link

In the Java 2 SDK implementation of RMI, the new wire protocol has made skeleton classes obsolete.

You only have to worry about skeleton classes and objects in JDK 1.1 and JDK 1.1 compatible system implementations.

Remote Reference Layer

This layer provides a RemoteRef object that represents the link to the remote service implementation object.

In JDK 1.1, only unicast, point-to-point connection is supported. Before a client can use a remote service, the remote service must be instantiated on the server and ran all the time.

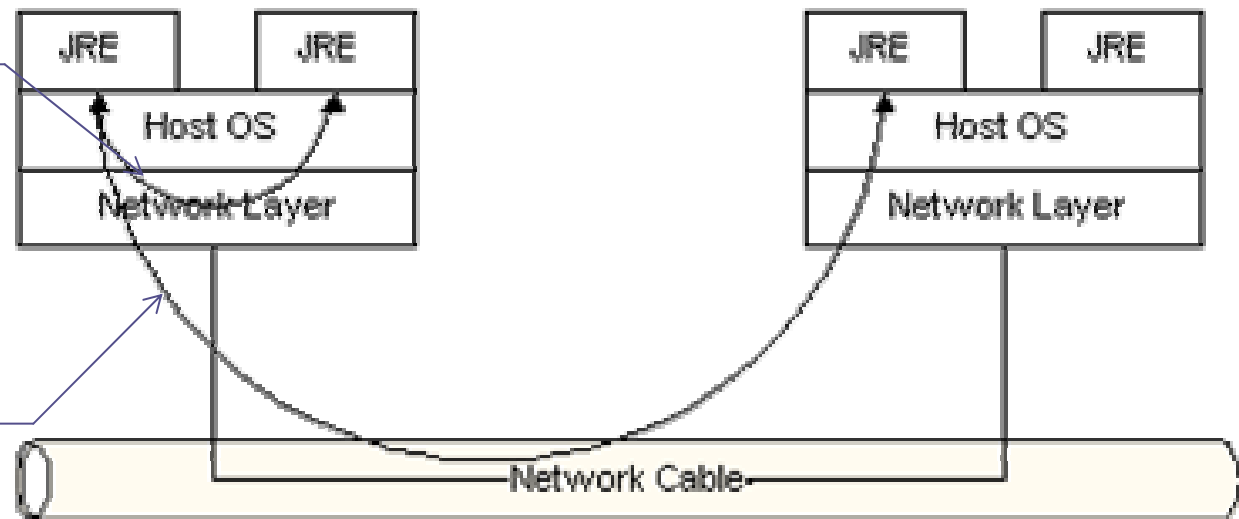
In Java 2 SDK, client-server connection is added and activatable remote objects is supported . With the introduction of the RMI daemon, rmid, remote objects can be created and execute "on demand," rather than running all the time.

Transport Layer 1

The Transport Layer makes the connection between JVMs. All connections are stream-based network connections that use TCP/IP.

Communication
between the
same host using
TCP/IP

Communication
between different
host using
TCP/IP



Transport Layer 2

On top of TCP/IP, RMI uses a wire level protocol called Java Remote Method Protocol (JRMP).

JRMP is a proprietary, stream-based protocol.

Sun and IBM have jointly worked on another version of RMI, called RMI-IIOP(Remote Method Invocation over Internet Inter-ORB Protocol), which combines RMI-style ease of use with CORBA cross-language interoperability.

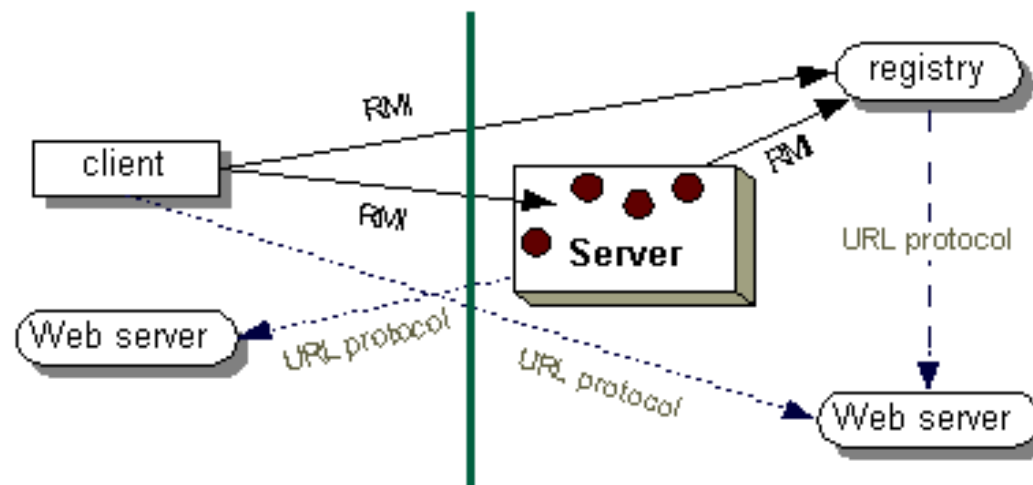
The remote object model for Enterprise Java Beans(EJBs) is RMI-based.

Naming Remote Objects

In RMI, clients find remote services by using a naming or directory service.

RMI can use many different directory services, including the Java Naming and Directory Interface (JNDI).

RMI itself includes a simple service called the RMI Registry, `rmiregistry`. The RMI Registry runs on each machine that hosts remote service objects and accepts queries for services, by default on port 1099.



Overview

- 1) introduction
- 2) RMI architecture
- 3) implementing and running RMI system
- 4) Implementing activatable RMI server
- 5) summary

Example : build a RMI system

In this example, we shall build a simple remote calculator service and use it from a client program.

A working RMI system is composed of several parts.

- a) Interface definitions for the remote services
- b) Implementations of the remote services
- c) Stub and Skeleton files
- d) A server to host the remote services
- e) An RMI Naming service that allows clients to find the remote services
- f) A class file provider (an [HTTP](#) or [FTP](#) server)
- g) A client program that needs the remote services

Interface 1

The first step is to write and compile the Java code for the service interface.

All the interface has to extend the `java.rmi.Remote` interface and all the methods has to declare that it may throw a `RemoteException` object.

Interface 2

The interface may look like the following:

```
public interface Calculator extends
    java.rmi.Remote
{ public long add(long a, long b) throws
    java.rmi.RemoteException;
  public long sub(long a, long b) throws
    java.rmi.RemoteException;
  public long mul(long a, long b) throws
    java.rmi.RemoteException;
  public long div(long a, long b) throws
    java.rmi.RemoteException;
}
```

Implement 1

The second step is to write the implementation for the remote service.

The implementation class may extend from the `java.rmi.server.UnicastRemoteObject` to link into the RMI system.

It must also provide an explicit default constructor throwing `RemoteException`. When this constructor calls `super()`, it activates code in `UnicastRemoteObject` that performs the RMI linking and remote object initialization.

Implement 2

```
public class CalculatorImpl extends
    java.rmi.server.UnicastRemoteObject implements
    Calculator {
    // Implementations must have an
    // explicit default constructor
    // in order to declare the
    // RemoteException exception
    public CalculatorImpl() throws
        java.rmi.RemoteException
    { super(); }
    public long add(long a, long b) throws
        java.rmi.RemoteException
    { return a + b; }
    ...
}
```

Lab Work: Implementation

- 1) Please write the rest of implementation for the Calculator interface.

Note: If you must extend some other classes other than extending from `UnicastRemoteObject`, the implementation may use the static `exportObject()` method of the `UnicastRemoteObject` instead.

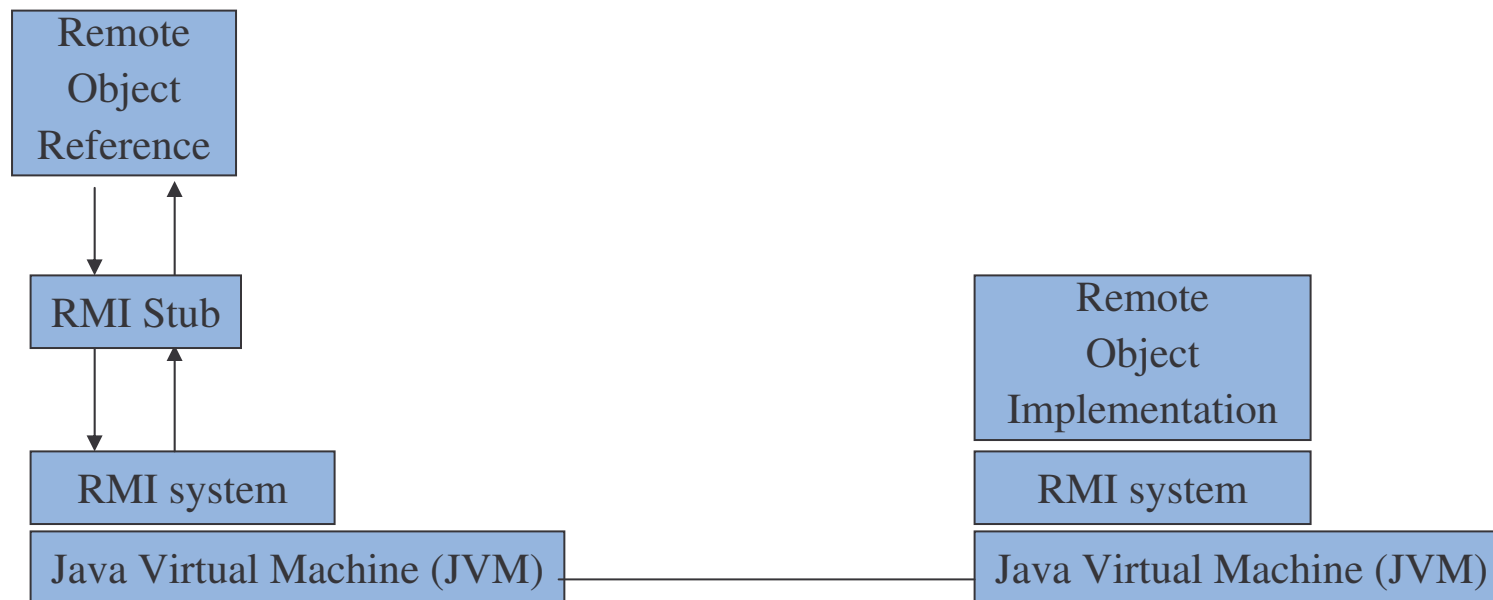
Be careful that you may need to synchronize some portions of your remotely available method. But it is not necessary for this example.

Stubs and Skeletons

To generate the Stub and Skeleton files, use the RMI compiler, `rmic` as the following:

```
>rmic CalculatorImpl
```

The default option will create stubs/skeletons compatible with both JDK 1.1 and Java 2.



Host Server 1

Remote RMI services must be hosted in a server process. The following code is a very simple server that provides the bare essentials for hosting.

```
import java.rmi.Naming;

public class CalculatorServer {
    public CalculatorServer() {
        try {
            Calculator c = new CalculatorImpl();

            Naming.rebind("rmi://localhost:1099/Calculator
Service", c);
        } catch (Exception e) {
            System.out.println("Trouble: " + e);
        }
    }
}
```


Host Server 2

```
public static void main(String args[]) {  
    new CalculatorServer();  
}
```

Client 1

- 1) In the client's code, all you need to do is to lookup the object and use its methods as local methods.
- 2) The client's code may look like the following:

```
import java.rmi.Naming;  
import java.rmi.RemoteException;  
import java.net.MalformedURLException;  
import java.rmi.NotBoundException;  
  
public class CalculatorClient {  
    public static void main(String[] args) {  
        try {  
            Calculator c = (Calculator)Naming.lookup (  
                "rmi://localhost/CalculatorService");
```

Client 2

```
        System.out.println( c.sub(4, 3) );
        System.out.println( c.add(4, 5) );
        System.out.println( c.mul(3, 6) );
        System.out.println( c.div(9, 3) );
    }

    catch (MalformedURLException murle) {}
    catch (RemoteException re){}
    catch (NotBoundException nbe){}
    catch (java.lang.ArithmeticException ae)
    {}

}

}
```

Running the RMI System

- 1) Start up three consoles, one for the server, one for the client, and one for the RMIRegistry.
- 2) Type `rmiregistry` in the directory that contains the classes you have written.
- 3) In the server's console, type `java CalculatorServer` to start the server.
- 4) In the client's console, type `java CalculatorClient` to start the client program.
- 5) The output should look like:

`1`

`9`

`18`

`3`

Lab Work: RMI System

- 1) Please follow what we have discussed to develop a RMI server which hosts a service for calculating the square root of a number.
- 2) Compile your RMI server and generate the corresponding stub class.
- 3) Create a client to test the RMI service.

Passing Parameters

All parameters passed from an RMI client to an RMI server must either be serializable or be a remote object.

For serializable:

- a) Data is extracted from the local object and sent across the network to the remote server.
- b) Object is then reconstructed in the remote server.
- c) Any changes to the object in the RMIServer will not be reflected in the object held in the RMI client and vice versa.

For a remote object:

- a) Stub information, not a copy of data, is actually sent over RMI.
- b) Any call made to the parameter object become a remote calls back to the actual object.
- c) Changes made in one JVM are reflected in the original JVM.

Conditions for serializability

If an object is to be serialized:

- a) The class must be declared as public
- b) The class must implement Serializable
- c) The class must have a default (no-argument) constructor
- d) All fields of the class must be serializable: either primitive types or serializable objects

Remote interfaces and class

A Remote class has two parts:

- a) The interface (used by both client and server):
 - 1. Must be public
 - 2. Must extend the interface `java.rmi.Remote`
 - 3. Every method in the interface must declare that it throws `java.rmi.RemoteException` (other exceptions may also be thrown)
- b) The class itself (used only by the server):
 - 1. Must implement a `Remote` interface
 - 2. Should extend `java.rmi.server.UnicastRemoteObject`
 - 3. May have locally accessible methods that are not in its `Remote` interface

Security

Your program should guarantee that the classes that get loaded do not perform operations that they are not allowed to perform.

A more conservative security manager than the default should be installed. The following code should be added to the main method of the server and client program:

```
if (System.getSecurityManager() == null){  
    System.setSecurityManager(new  
        RMISecurityManager());  
}
```

Overview

- 1) introduction
- 2) RMI architecture
- 3) implementing and running RMI system
- 4) **Implementing activatable RMI server**
- 5) summary

Activatable Server

Enable server programs to wake up and start to run when they are needed.

Java RMI Activation System Daemon (`rmid`) is introduced to handle this task.

When a client requests a reference to the server from the `rmiregistry`, the `rmid` program, which holds the servers details, will be requested to start up the server and return the reference to the client. After that, the `rmiregistry` will be able to provide the reference of the server directly.

Activatable Server:Implementation

1) Subcalss the `java.rmi.activation.Activatable` class and implement the remote interface.

2) Implement the following constructor:

```
public Server(ActivationID id, MarshalledObject
    data) throws RemoteException{
    super(id,0); //register activatable object and
                // export on anonymous port
}
```

3) Create an activation description used by the `rmid` program.

4) Register the activation description with the `rmid` program.

5) Compile the activatable server with `javac` and `rmic` compiler.

6) The client program needs no modification.

Activatable Server: Setup 1

Before we can use the activatable server, you need to generate the activation description used by the `rmid` and register the description with the `rmid` program. We will group these processes into a utility program for illustration.

The structure of the utility program may look like this:

```
//1. Make the appropriate imports
import java.rmi.*;
import java.rmi.activation.*;
import java.util.Properties;
public class SetupServer{
    public static void main(String args[]){
        try{
```

Activatable Server: Setup 2

```
//2. Declare for a security policy file
System.setSecurityManager(new RMISecurityManager());
Properties props =
    (Properties)System.getProperties();
props.put("java.security.policy", <location of
    security policy file>);
//3. Create an activation group description even there
// is only one server
ActivationGroupDesc agd = new ActivationGroupDesc
    (props, null);
//4. Create a new activation group
ActivationGroupID agid =
    ActivationGroup.getSystem().registerGroup(agd);
```

Activatable Server:Setup 3

```
//5. Create the actual activation description
//    Don't miss the trailing slash (/)
String codebase = "file:/<location of server
    implementation file>/";
ActivationDesc desc = new ActivationDesc(agid,
    "<name of the server>",codebase, null);
//6. Register the activation description to the rmid
//    program. Suppose the remote interface of the
//    server is RemoteInterface, the code will look
//    like this:
RemoteInterface ref =
    (RemoteInterface)Activatable.register(desc);
```

Activatable Server:Setup 4

```
//7. Bind the server in rmiregistry
Naming.rebind("Server", ref);
//8. Exit the setup program
System.exit(0);
}catch (Exception e){}
}
}
```


Compile and Run 1

- 1) Compile all the classes use `javac`.
- 2) Run `rmic` on the implementation class
- 3) Start rmi registry use `rmiregistry`.
 - a) make sure that the shell or window in which you will run the registry, either has no `CLASSPATH` set or has a `CLASSPATH` that does not include the path to any classes that you want downloaded to your client, including the stubs for your remote object implementation classes.
 - b) If you start the `rmiregistry`, and it can find your stub classes in its `CLASSPATH`, it will ignore the server's `java.rmi.server.codebase` property, and as a result, your client(s) will not be able to download the stub code for your remote object.

Compile and Run 2

- 4) Start the activation daemon, `rmid`. Use `-J` option for a runtime flag.

```
rmid -J-Djava.security.policy=rmid.policy
```

The policy file may look like this:

```
grant{  
  permission com.sun.rmi.rmid.ExecOptionPermission  
    "-Djava.*";  
  permission com.sun.rmi.rmid.ExecOptionPermission  
    "-Dsun.*";  
  permission com.sun.rmi.rmid.ExecOptionPermission  
    "-Dfile.*";
```

Compile and Run (3)

```
permission com.sun.rmi.rmid.ExecOptionPermission "-  
    Dpath.separator=*";  
permission com.sun.rmi.rmid.ExecOptionPermission "-  
    Duser.*";  
permission com.sun.rmi.rmid.ExecOptionPermission "-  
    Dos.*";  
permission com.sun.rmi.rmid.ExecOptionPermission "-  
    Dline.separator=*";  
permission com.sun.rmi.rmid.ExecOptionPermission "-  
    Dawt.*";
```

Compile and Run (4)

5) Running the setup program.

```
java  
-Djava.security.policy  
    =<full path of the policy file>  
-Djava.rmi.server.codebase  
    =file:/<location of the implementation stubs>/  
<class name of the setup program>
```

Compile and Run (5)

6) Running the client program.

```
java  
-Djava.security.policy  
    =<full path of the policy file>  
<client name>
```

For testing purpose, use the following `security.policy`:

```
grant {  
    permission java.security.AllPermission "", "";  
};
```

Exercise: Activatable RMI

- 1) Write a remote interface called `HelloInterface`.
- 2) Define a method `getMessage (String s)` in it. This method has a return type as a `String`. Don't forget to throw the proper exception.
- 3) Create a class named `Server` which has to be a subclass of the `java.rmi.activation.Activatable` class.
- 4) Implement the `getMessage` method which will append "Hello" the argument and return it as a `String`.
- 5) Create the Setup program for the server.
- 6) Create a client program which should look up the activatable server and use the `getMessage` method of it.
- 7) Compile and generate the corresponding files.
- 8) Run the client and check the result.

Overview

- 1) introduction
- 2) RMI architecture
- 3) implementing and running RMI system
- 4) **summary**

Summary

In this session, we cover the followings:

- 1) Architecture of RMI
- 2) Building RMI system including both client and server
- 3) Implementation for activatable RMI server

Lab Work: Activatable RMI 1

- 1) Build an activatable RMIServer for a chatting system.

As our focus will be on building an activatable RMIServer, all the codes for the client program and interfaces will be provided. You only need to implement the activatable RMIServer.

The interface for the server is provided as `chat.interface.ChatServer.java`. You need to write an implementation class for it naming `chat.server.ChatServerImpl.java`.

Lab Work: Activatable RMI 2

- 2) Write a setup program for the server.
- 3) Write a class `chat.Message.java` to represent the message sending between the server and client. It is required to keep the information of the sender and the message content.

Lab Work: Activatable RMI 3

4) Test the program

- a) For testing the dynamic class downloading, please download a basic HTTP server from the following address:

```
java.sun.com/products/jdk/rmi/class-server.zip
```

- b) Extract the files and compile them using the following command:

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

- c) After starting the `rmid`, you can start the HTTP server using the following command:

```
java examples.classServer.ClassFileServer  
<port number> <path for server's download  
directory>
```

Lab Work: Activatable RMI 4

- d) The directory structure and .class files of the exercise should be copied to the server's download directory.
- e) Start the setup program of your server. And for the codebase option, you should use http protocol instead of file this time.

```
-Djava.rmi.server.codebase =  
    http://localhost:port/
```

- f) You can use the security policy file in the example for testing purpose.
- 6) Examine the files in chat.client.packages. These are the classes for the client program. You will notice that the client needs no modification for dealing with the activatable RMIServer.

Lab Work: Activatable RMI 5

ChatServer interface:

```
package interfaces;
import java.rmi.*;
import chat.Message;

public interface ChatServer extends Remote {
    // register new ChatClient with ChatServer
    // In implementation, you may need to choose a
    // collection type for storing the connected client
    public void registerClient (ChatClient
    client)throws RemoteException;
    public void unregisterClient(ChatClient
    client)throws RemoteException;
```

Lab Work: Activatable RMI 6

```
// post new message to ChatServer
public void postMessage(Message message) throws
RemoteException;
// sending message to the clients that the server
// is stopping
public void stopServer()throws RemoteException;
}
```

Lab Work: Activatable RMI 7

ChatClient interface:

```
package interfaces;
import java.rmi.*;
import chat.Message;
public interface ChatClient extends Remote{
    //call back method allows the server to send
    //message to client
    public void deliverMessage(Message message)throws
    RemoteException;

    // method called when server shutting down
    public void serverStopping()throws
    RemoteException ;
}
```

Lab Work: Activatable RMI 8

MessageManager interface:

```
package interfaces;

public interface MessageManager {

    //connect to the server. Check the code in
    //RMIMessageManager for implementation
    public void connect (MessageListener listener)
    throws Exception;

    //disconnect to the server. Check the code in
    //RMIMessageManager for implementation
    public void disconnect (MessageListener
    listener)throws Exception;
```


Lab Work: Activatable RMI 9

```
//send message to the server. Check the code in
//RMIMessageManager for implementation
public void sendMessage(String from, String
                        message)throws Exception;

//Registers a DisconnectListener to be notified
//when the ChatServer disconnects the client.
//Each ClientGUI will do the registration when
//connection is made to the server.
public void setDisconnectListener
(DisconnectListener listener);
}
```

Lab Work: Activatable RMI 10

MessageListenr interface:

```
package interfaces;
```

```
public interface MessageListener {  
    //The inner class MyMessageListener defined inside  
    //the ClientGUI implements this interface for  
    //handling the message received.  
    public void messageReceived (String from, String  
        message);  
}
```

Lab Work: Activatable RMI 11

DisconnectHandler interface:

```
package interfaces;
//An inner class DisconnectHandler actually
//implements this interface.
//The DisconnectHandler will update GUI in thread
//safe manner after received disconnect notification.
public interface DisconnectListener {

    public void serverDisconnected(String message);
}
```

Overview

- 1) introduction
- 2) RMI architecture
- 3) implementing and running RMI system
- 4) **summary**

Summary

In this session, we cover the followings:

- 1) Architecture of RMI
- 2) Building RMI system including both client and server
- 3) Implementation and environment set up procedure for activatable RMI server

CORBA

Course Outline

- 1) introduction
- 2) streams
- 3) networking
- 4) database connectivity
- 5) architectures
 - a) message-orientation
 - 1) javamail
 - 2) jms
 - b) distributed objects
 - 1) rmi
 - 2) corba
 - 3) JavaIDL
- 6) summary

The OMG

- 1) Object Management Group
 - a) Founded in 1989
 - b) Not-for-Profit organization
 - c) Vendor neutral
 - d) ~800 member companies
- 2) Key Specifications
 - a) UML
 - b) CORBA

What is CORBA?

Defines a framework for object-oriented distributed applications.

Defined by a consortium of vendors under the direction of OMG.

Allows distributed programs in different languages and different platforms to interact as though they were in a single programming language on one computer.

Brings advantages of OO to distributed systems.

Allows you design a distributed application as a set of cooperating objects and to reuse existing objects.

Key CORBA Features

- 1) Application Development Transparencies
 - a) Hardware/Language neutral
 - b) Vendor neutral
 - c) Object oriented paradigm
- 2) CORBA Interface Definition Language (IDL)
- 3) CORBAServices
 - a) Naming
 - b) Event
 - c) Transaction
 - d) Security
- 4) Interoperability

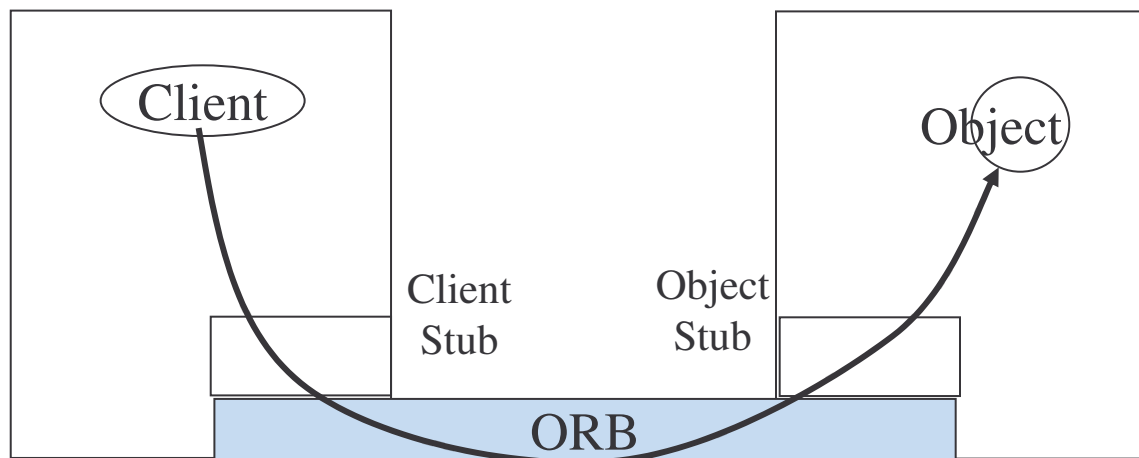
Object Request Broker (ORB)

- 1) A software component that mediates transfer of messages from a program to an object located on a remote host.
- 2) Hides underlying network communications from a programmer.
- 3) ORB allows you to create software objects whose member functions can be invoked by client programs located anywhere.
- 4) A server program contains instances of CORBA objects.

ORB: Conceptual View

- 1) When a client invokes a member function on a CORBA object, the ORB intercepts the function call.
- 2) ORB directs the function call across the network to the target object.
- 3) The ORB then collects the results from the function call returns these to the function call.

Implementation Details

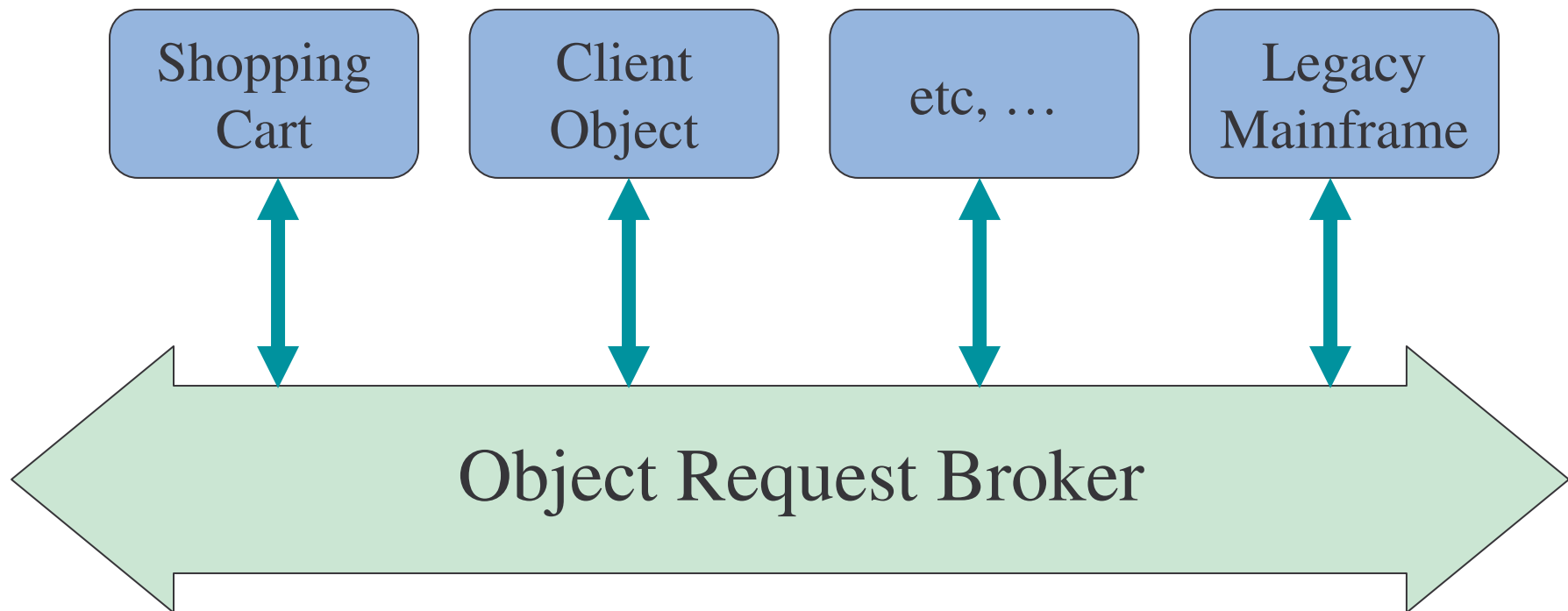


Access to the services provided by an Object

ORB : (Object-oriented middleware) Object Request Broker
ORB mediates transfer between client program and server object.

CORBA: A “Software Bus”

All CORBA objects connect to each other via ORB.



CORBA IDL

- 1) Interface Definition Language
 - a) used to generate application code (stubs/skeletons)
 - b) language neutral (Ada, C++, Java, ...)

- 2) IDL is NOT a programming language
 - a) lacks control structures
 - b) provides no implementation details
 - c) a specification

CORBA Objects and IDL

- 1) These are standard software objects implemented in any supported language including Java, C++ and Smalltalk.
- 2) Each CORBA object has a clearly defined interface specified in CORBA `interface definition language` (IDL).
- 3) The interface definition specifies the member functions available to the client without any assumption about the implementation of the object.

Client and IDL

- 1) To call a member function on a CORBA object the client needs only the object's IDL.
- 2) Client need not know the object's implementation, location or operating system on which the object runs.

Interface and Implementation

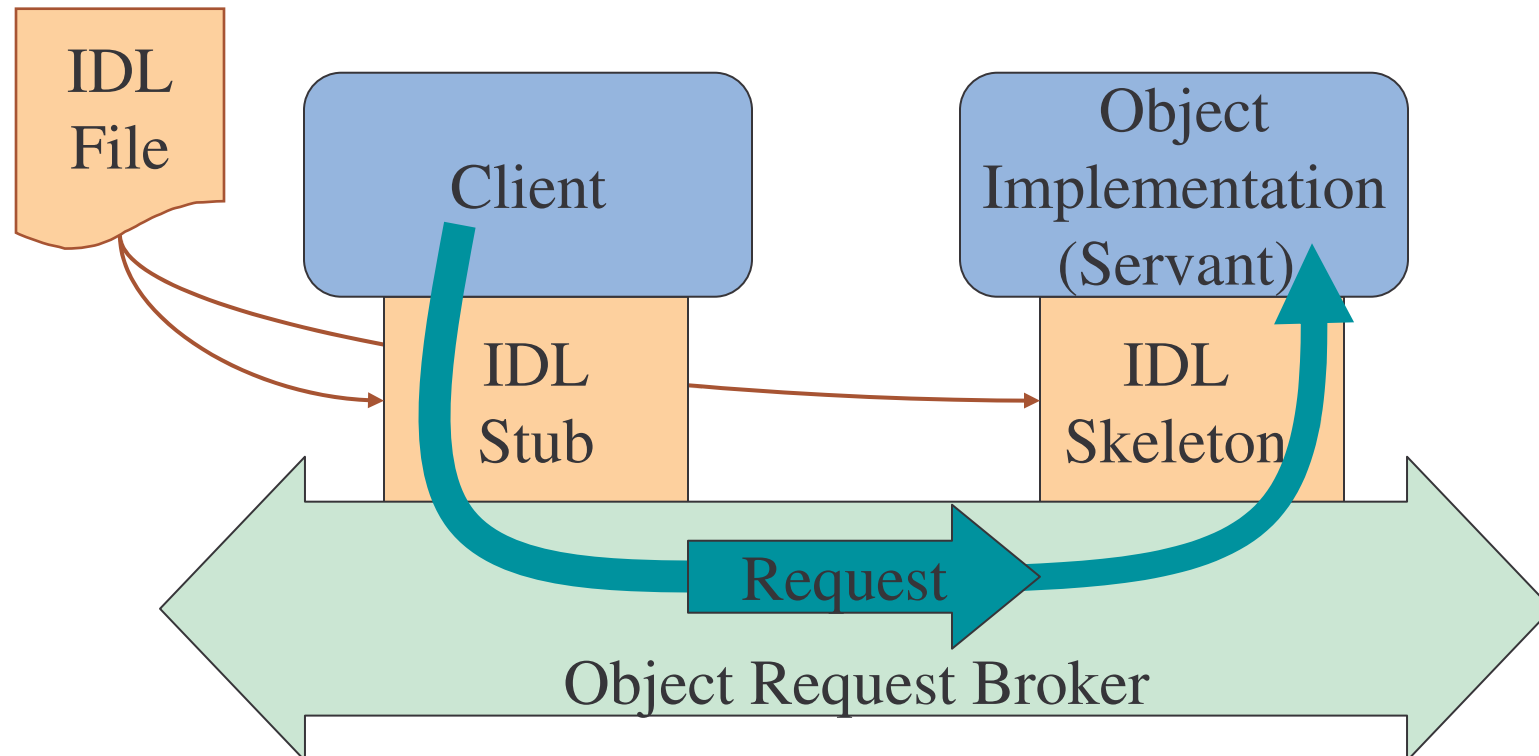
- 1) Interface and implementation can be in two different languages.
- 2) Interface abstracts and protects details (trade secrets) from client
- 3) Interface offers a means of expressing design without worrying about implementation.
- 4) Interface is separated from implementation

Example: CORBA IDL

```
module BankExample {  
    interface Account {  
        exception BadCheck {  
            float fee;  
        };  
        float deposit(in float amount);  
        float writeCheck(in float amount)  
            raises (BadCheck);  
    };  
    interface AccountManager {  
        Account openAccount(in string name);  
    };  
};
```

CORBA Application Diagram

Objects are identified by Interoperable Object References (IORs)



CORBA Development Steps

- 1) Design the Application
- 2) IDL Specification
- 3) IDL Compilation (Code Generation)
- 4) Write the Client & Server implementation specific code
- 5) Compile the source code
- 6) Run the application

JavaIDL

Course Outline

- 1) introduction
- 2) streams
- 3) networking
- 4) database connectivity
- 5) architectures
 - a) message-orientation
 - 1) javamail
 - 2) jms
 - b) distributed objects
 - 1) rmi
 - 2) corba
 - 3) **JavaIDL**
- 6) summary

Modules and Interfaces

- IDL modules

```
module MyStuff  
{  
  ...  
};
```

- Provide a namespace to group a set of interfaces. Names are scoped using the “::” operator.

- IDL interface

```
interface Foo { };
```

- Java packages

```
package MyStuff;  
...
```

- Provide Internet-wide namespaces. Scoped using the “.” operator.

- Java interface

```
public interface Foo  
{...};
```


IDL to Java: Parameters

- 1) Java uses pass-by-value for parameters (including parameters that are references)
- 2) IDL has `in`, `out` and `inout` types of parameters
- 3) The `in` parameter type maps to a normal Java parameter since it does not need to be changed
- 4) `out` and `inout` parameter types are passed via instances of Java `Holder` classes

Holder Classes

- 1) `Holder` classes encapsulate the real value of a parameter which can then be reassigned to
 - a) a member “`value`”

```
// user code
```

```
// select a target object
```

```
Example.Modes target = ...;
```

```
// prepare to receive out
```

```
IntHolder outHolder = new IntHolder();
```

```
// set up the in side of the inout
```

```
IntHolder inoutHolder = new IntHolder (131);
```

```
// make the invocation
```

```
int result = target.operation (  
                                outHolder, inoutHolder);
```

```
// use the values of holders
```

```
...outHolder.value...
```

```
...inoutHoulder.value...
```

```
// generated java
```

```
package Example;
```

```
public interface Modes {
```

```
    int operation (IntHolder outArg,  
                  IntHolder inoutArg);
```

```
}
```

Helper Classes

- 1) all user-defined IDL types have a Helper Java class
- 2) insert and extract *Any*
- 3) get CORBA::TypeCode of the type
- 4) narrow (for interfaces only)

IDL to Java: Attributes

- IDL attributes

```
attribute long assignable;  
readonly attribute long  
    fetchable;
```

- Java “get” and “set” methods

```
public int assignable();  
public void assignable(int  
    val);  
public int fetchable();
```

Basic Types

IDL Type	Java Type	Exception
boolean	boolean	
char	char	CORBA::DATA_CONVERSION
octet	byte	
string	java.lang.String	CORBA::MARSHAL...
short	short	
unsigned short	short	
long	int	
unsigned long	int	
long long	long	
unsigned long long	long	
float	float	
double	double	

IDL to Java: Basic Types

- IDL char

```
const char MyChar = 'A';
```
- Java char

```
final public class MyChar  
{  
    final public static char  
    value = (char)'A';  
}
```

IDL to Java: Basic Types

- IDL octet
`void foo(in octet x);`
- Java byte
`public void foo(byte x);`

IDL to Java: Basic Types

- IDL boolean

```
const boolean truth =  
TRUE;
```
- IDL constants `TRUE` and `FALSE`
- Java boolean

```
final public class truth  
{  
    final public static  
    boolean value = true;  
}
```
- Java constants `true` and `false`

IDL to Java: Basic Types

- IDL string

```
const string MyString =  
"Hello World";
```
- Java java.lang.String

```
final public class MyString  
{  
    final public static String  
        value = "Hello World";  
}
```

IDL to Java: Basic Types

- IDL integers
 - (unsigned) short
 - (unsigned) long
 - (unsigned) long long?

```
const unsigned short  
MyUnsignedShort = 1580;
```

- Java integers
 - short
 - int
 - long

```
final public class  
MyUnsignedShort  
{  
    final public static  
    short value =  
        (short)1580;  
}
```

IDL to Java: Basic Types

- IDL floating-point float, double

```
const double MyDouble =  
1.23456789;
```

- Java floating-point *float*, *double*

```
final public class  
MyDouble  
{  
    final public static  
    double value =  
        (double)1.23456789;  
}
```

IDL to Java: Constructed Types

- IDL Enum

```
enum MyEnum  
{none, first, second};
```

- Java class

```
final public class MyEnum  
{  
    final public static int  
none = 0;  
    final public static int  
first = 1;  
    final public static int  
second = 2;  
    final public static int  
narrow(int i) throws  
CORBA.BAD_PARAM {...};  
}
```

- the narrow method is for checking enum values

IDL to Java: Constructed Types

- IDL *struct*

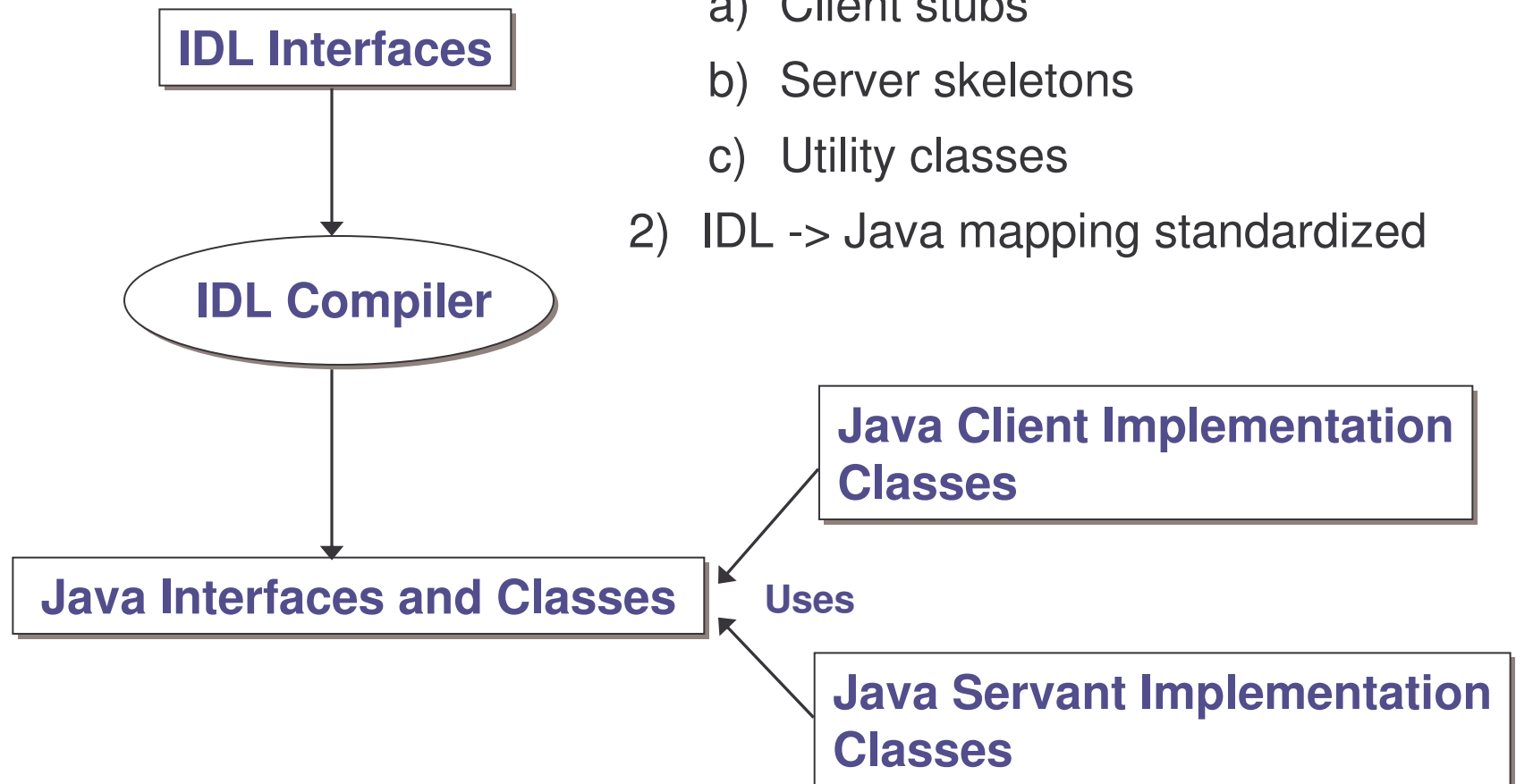
```
struct MyStruct
{
    long mylong;
    string mystring;
};
```

- Java *class*

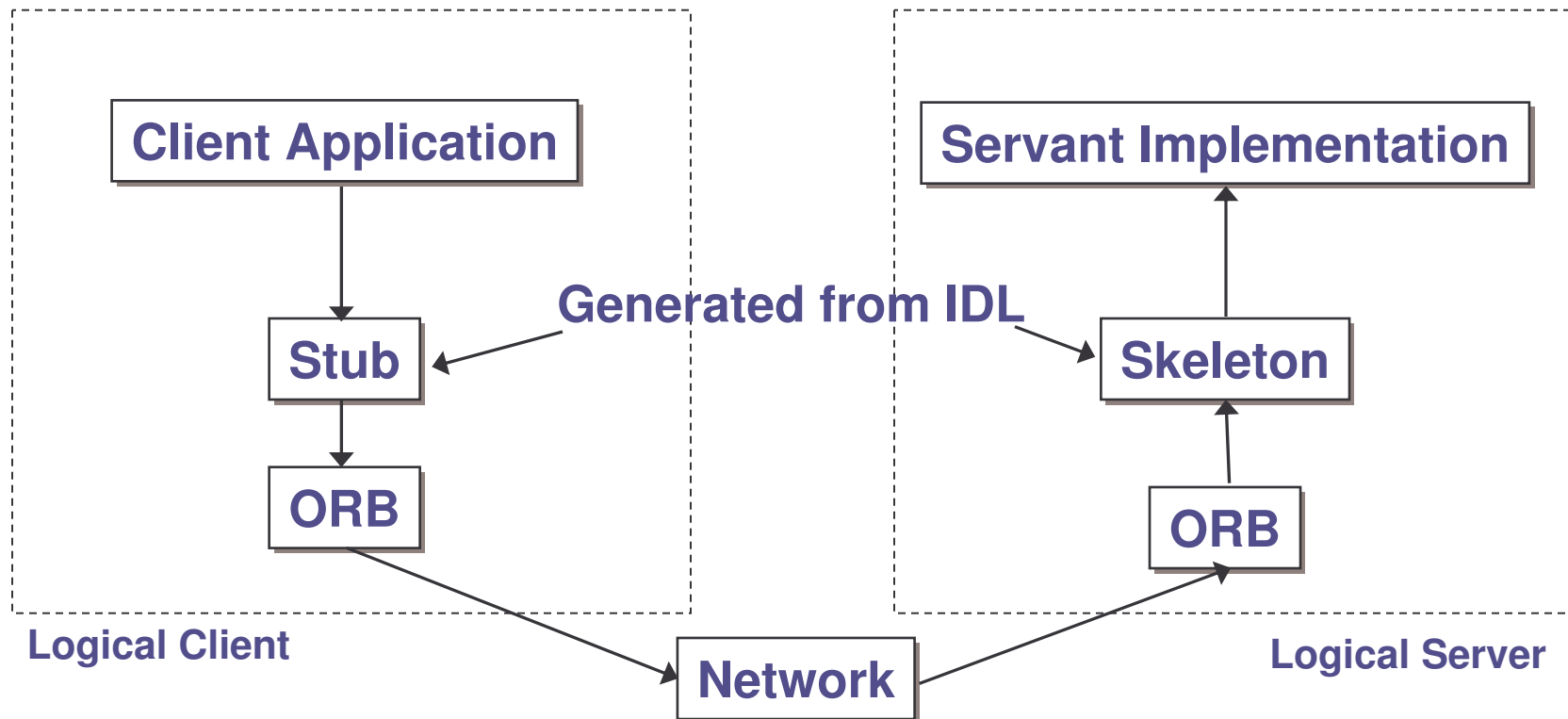
```
final public class
MyStruct
{
    public MyStruct(int
_mylong, String
_mystring) {...};
    public MyStruct() {...};

    public int mylong;
    public String mystring;
}
```

The Big Picture



Big Picture (Invocation)



Example: File Transfer

This presents a file download CORBA application

The client request for a file and the server in turn sends the file to the client which then saves it on the local machine.

There are a number of steps involved:

- 1) Define an interface in IDL
- 2) Map the IDL interface to Java (done automatically)
- 3) Implement the interface
- 4) Develop the server
- 5) Develop a client
- 6) Run the naming service, the server, and the client.

Step 1: Define the IDL Interface 1

The first thing to do is to determine the operation that the server will support.

In this application, the client will invoke a method to download a file.

Here is the code.

```
interface FileInterface {  
    typedef sequence<octet> Data;  
    Data downloadFile(in string fileName);  
};
```

Save this file as `FileInterface.idl`

Step 1: Define the IDL Interface 2

`Data` is a new type introduced using the `typedef` keyword.

A `sequence` in IDL is similar to an array except that a sequence does not have a fixed size

An `octet` is an 8-bit quantity that is equivalent to the Java type `byte`

The `downloadFile` method takes one parameter of type `string` that is declared `in`.

IDL defines three parameter-passing modes: `in` (for input from client to server), `out` (for output from server to client), and `inout` (used for both input and output).

Step 2: Map IDL to Java

Once you finish defining the IDL interface, you are ready to map the IDL interface to Java.

Java comes with the `idlj` compiler, which is used to map IDL definitions into Java declarations and statements.

The `idlj` compiler accepts options that allow you to specify if you wish to generate client stubs, server skeletons, or both.

let's compile the `FileInterface.idl` and generate both client and server-side files.

Step 3: Compile the IDL Interface

1) Compile the IDL Interface using:

```
prompt> idlj -oldImplBase -fall FileInterface.idl
```

2) IDL compilation produces many java constructs (interfaces and classes).

3) Each one is placed with a `<filename>.java`

Files Generated by IDL Compiler

- 1) Each file generated contains a Java interface or class scoped within a package.
- 2) This package is physically located in a directory of the same name according to Java conventions.

Client Side Files

- 1) `FileInterface.java` - an interface to provide a client a view of the methods in the IDL.
- 2) `_FileInterfaceStub.java` - a Java class that implements the methods defined in interface Grid. Provides functionality that allows client method invocations to be forwarded to a server.

Server Side Files

- 1) `_FileInterfaceImplBase.java` - an abstract Java class that allows server-side developers to implement the `FileInterface` interface.
- 2) Other files: `FileInterfaceHelper.java`,
`FileInterfaceHolder.java`,
`FileInterfaceOperations.java`,

Step 4: Implement the Interface 1

Provide an implementation to the `downloadFile()` method. This implementation is known as a servant.

```
import java.io.*;

public class FileServant extends _FileInterfaceImplBase
{
    public byte[] downloadFile(String fileName) {
        File file = new File(fileName);
        byte buffer[] = new byte[(int)file.length()];
        try {
            BufferedInputStream input = new
            BufferedInputStream(new
                                FileInputStream(fileName));
            input.read(buffer, 0, buffer.length);
            input.close();
        }
    }
}
```


Step 4: Implement the Interface 2

```
    } catch (Exception e) {  
        System.out.println("FileServant Error:  
                           "+e.getMessage());  
        e.printStackTrace();  
    }  
    return (buffer);  
}  
}
```

Step 5: Develop the Server 1

The next step is developing the CORBA server.

Write `FileServer` class that implements a CORBA server that does the following:

- 1) Initializes the ORB
- 2) Creates a `FileServant` object
- 3) Registers the object in the CORBA Naming Service (COS Naming)
- 4) Prints a status message
- 5) Waits for incoming client requests

Step 5: Develop the Server 2

```
import java.io.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;

public class FileServer {
    public static void main(String args[]) {
        try{
            // create and initialize the ORB
            ORB orb = ORB.init(args, null);
            // create the servant and register it with ORB
            FileServant fileRef = new FileServant();
            orb.connect(fileRef);
        }
    }
}
```

Step 5: Develop the Server 3

```
// get the root naming context
org.omg.CORBA.Object objRef =
    orb.resolve_initial_references("NameService");
NamingContext ncRef =
    NamingContextHelper.narrow(objRef);
// Bind the object reference in naming
NameComponent nc = new
    NameComponent("FileTransfer", " ");
NameComponent path[] = {nc};
ncRef.rebind(path, fileRef);
System.out.println("Server started....");
```

Step 5: Develop the Server 4

```
// Wait for invocations from clients
java.lang.Object sync = new java.lang.Object();
synchronized(sync) {
    sync.wait();
}
} catch (Exception e) {
    System.err.println("ERROR: " + e.getMessage());
    e.printStackTrace(System.out);
}
}
}
```

Step 6: Develop the Client 1

The next step is developing the CORBA client.

Write `FileClient` class that implements a CORBA client that does the following:

- 1) Initializes the ORB
- 2) Retrieve the `FileTransfer` service from the naming server
- 3) Call the `downloadFile` method.

Step 6: Develop the Client 2

```
import java.io.*;
import java.util.*;
import org.omg.CosNaming.*;
import org.omg.CORBA.*;

public class FileClient {
    public static void main(String argv[]) {
        try {
            // create and initialize the ORB
            ORB orb = ORB.init(argv, null);
            // get the root naming context
            org.omg.CORBA.Object objRef =
                orb.resolve_initial_references("NameService");
```

Step 6: Develop the Client 2

```
NamingContext ncRef =
    NamingContextHelper.narrow(objRef);
NameComponent nc = new
    NameComponent("FileTransfer", " ");
// Resolve the object reference in naming
NameComponent path[] = {nc};
FileInterfaceOperations fileRef =
    FileInterfaceHelper.narrow(ncRef.resolve(path));

if(argv.length < 1) {
    System.out.println("Usage: java FileClient
                        filename");
}
```


Step 6: Develop the Client 2

```
// save the file
File file = new File(argv[0]);
byte data[] = fileRef.downloadFile(argv[0]);
BufferedOutputStream output = new
BufferedOutputStream(new FileOutputStream(argv[0]));
output.write(data, 0, data.length);
output.flush();
output.close();
}catch(Exception e) {
    System.out.println("FileClient Error: " +
                        e.getMessage());
    e.printStackTrace();
}
}}
```

Step 7: Run the Application

1) Running the the CORBA naming service.

```
prompt> tnameserv -ORBInitialPort 2500
```

2) Start the server

```
prompt> java FileServer -ORBInitialPort 2500
```

3) Run the client

```
prompt> java FileClient c:\hello.txt -ORBInitialHost  
mycomputerName -ORBInitialPort 2500
```

Summary

- 1) We introduced general operation of CORBA.
- 2) Also details of specifying a client, server application, compiling them and registering and running.
- 3) You will have to configure your system before you try to do these steps.

Project Exercise

- 1) Implement the controller of your project as a distributed object by using either RMI or JavaIDL.
- 2) Device an approach through which the controller would locate a requested business object to handle a particular request and also invoke the appropriate operation requested for.
- 3) Persist your data using mySQL database engine.
- 4) See how you can make use of Java Message Service in your project.