Advanced Java Programming
Objectives

To introduce

- Concepts of Threads, Streams, Files, Persistence of objects, Serialization
- Java Database Connectivity
- Network Programming
- Remote Method Invocation
- Concepts of the Java Security Model
- Concepts of Java Naming and Directory Services
References

- Web Site : [http://java.sun.com/docs/books/tutorial/](http://java.sun.com/docs/books/tutorial/) available on Knowledge Shop also
Session Plan

Multithreading

- Creating and managing threads
- Priority management
- Thread groups and daemon threads
Multithreading
What are Threads?

- A unit of execution, can be considered a code fragment
- Helps in introducing software parallelism
How Threads are useful?

Multithreaded applications are most prevalent today

- Better utilization of system resources
- Multiple threads solve numerous problems better

Libraries of classes for programming multithreaded applications are available
Threads and Java

1. Most Java programs are threaded, may be implicitly
2. Threading systems depend on the implementation on that platform
Thread States

- **Runnable**
  - New
  - Start()

- **Blocked**
  - Wait
  - Notify
  - Suspend
  - Resume
  - Sleep
  - Sleep finished
  - IO block

- **Terminated or Dead**
  - IO complete
  - Run() exits
  - Some exception

Copyright © 2005, Infosys Technologies Ltd
Creating the Thread

Two ways:

- Extending the Thread class
- Implementing the Runnable interface
The “Thread” class

- By sub-classing the Thread class
- Overriding its run() method
- Other methods of Thread class can also be used
The “Runnable” Interface

- Implemented by classes whose instances are intended to be executed by a thread
- Need to implement the `run()` method
- Create a thread object using your `Runnable` object to perform thread operations
  ```java
  public Thread(Runnable target);
  ```
- Can also be implemented anonymously
Starting the Thread

- Using the `start()` method
- Placing the thread in runnable state
Thread Operations

- sleep()
- interrupt()
- yield()
- isAlive()
- join()
Race Conditions

- In multithreaded environment
- Two threads simultaneously contend for the same object
- Could result in an undefined state of the object, operated on
- Use of Java’s `synchronized` keyword avoids these problems
- Implemented within the language
Synchronized - Putting it Together

All access to delicate data should be synchronized.

Delicate data protected by synchronized should be private.
Inter-thread Communications

- Threads talk to each other
- Threads wait for each other
- Two ways of communication:
  - through shared data
  - through thread-control methods
wait() and notify()

- Defined in the Object class
- Should compulsorily be defined within a synchronized block
- The monitor defined the wait queue the thread should enter
- Replace `notify()` with `notifyAll()` to notify all the waiting threads on this monitor
public synchronized String retrieveMessage() {
    while(request == false) {
        try{
            wait();
        } catch(InterruptedException e){}
    }
    request = false;
    notify();
    return message;
}
Thread Scheduling

There are two approaches of scheduling:

– Preemptive scheduling
– Cooperative scheduling
Thread Priorities

- Provides ten priority levels for threads
- Maps to the native OS priorities
  - In NT there are 7 levels
  - In Solaris, there are $2^{31}$ levels
- Use defined constants to set priorities:
  
  MAX_PRIORITY, NORM_PRIORITY, MIN_PRIORITY
- Priority boosting cannot be controlled from within Java
Yielding

scheduler

Running

Runnable

Yield()
Polite Threads/Selfish Threads

- Polite yields() to the CPU while selfish does not
Daemon Threads

- Daemon threads are defined with respect to the JVM
- The JVM automatically exits when all users threads are dead
- Invoke the setDaemon() method before the call to start the thread
- Can query thread status using isDaemon()
Thread Groups

- Represents a set of threads
- Can also contain other thread groups, creating a hierarchy of thread groups
- Provides a single-point control on the threads belonging to the thread group
- Creation time association is for the life time of the thread
Some Finer Points

- Java’s threads are preemptable threads
- Threads have the overhead of a thread-context. In Java, it is 500 bytes.
- Multiple processors may not guarantee parallelism in execution
- Invoke the yield() method to provide for cooperative multithreading
Review

- Java understands thread both at language and runtime level
- The Thread class and Runnable interface are used for thread implementations
- Threads can be daemon, belong to a group, have priorities etc.
- Monitors are used for synchronizing shared data among threads
Serialization
Need for Serialization

Persistence:

– The capability of an object to exist beyond the execution of the program which created it.

– In other words: saving the state of the program in some permanent storage device, such as file
Use of Serialization

- Lightweight persistence
- communication via sockets
- Remote Method Invocation (RMI)
Serialization Mechanism

- Serializable objects are converted into stream of bytes and are stored in a file (in other words objects are stored in encoded form).
- Serializable objects implements java.io.Serializable interface.
De-Serialization Mechanism

- Serialized object is restored into its original form.
- Information for restoring
  - kept in Serialized form of object itself.
import java.io.*;

public class SerializationDemo{

    public static void main(String args[]){
        try{              //Object Seriliazation
            MyClass object1 = new MyClass ("hello",-7, 2.7);
            System.out.println("object1:" + object1);
            FileOutputStream fos = new FileOutputStream("seril");
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            oos.writeObject(object1);
            oos.writeObject(object1);
            oos.flush();
            fos.close();
        }catch(Exception e) {
            System.exit(0);
        }
    }
}
Object Serialization

//Object Deserialization

try{
    MyClass object2;
    FileInputStream fis = new FileInputStream("serial");
    ObjectInputStream ois = new ObjectInputStream(fis);
    object2 = (MyClass)ois.readObject();
    ois.close();
    System.out.println("Object2: " + object2);
} catch(Exception e) {
    System.exit(0);
}
Object Serialization

class MyClass implements Serializable{

    String s;
    int i;
    double d;

    public MyClass (String s, int i, double d){
        this.s = s;
        this.i = i;
        this.d = d;
    }

    public String toString(){
        return "s=" + s + ";i=" + i + ";d= " + d ;
    }
}

Externalization Interface

Class which implements this interface

- can control the encoding of its instances
- can also control the storage of the attributes of the superclasses in the stream
- methods for reading \texttt{(readExternal())} and writing \texttt{(writeExternal())} from the stream must be defined in this case
Security: an issue in serialization

- Serialized objects can be sent over network
- Can be accidentally or deliberately modified
- Also sensitive data can be read

Solution

- Encrypt the object during serialization using Security API
- Ensure that sensitive objects do not implement Serializable or Externalizable
Session Plan

Java Data Base Connectivity

- List JDBC API & use a JDBC driver
- Set up a connection to a database from Java
- Create an application to execute DML from Java
- Create an applet to execute DML.
Java Data Base Connectivity
• DataBase Driver used by Java Applications and Applets is JDBC driver

• JDBC: Not an acronym for anything but is associated with Java Database Connectivity
Connection

- A connection object represents a connection with a database.
- A connection session includes the SQL statements that are executed and the results that are returned over that connection.
- A single application can have one or more connections with a single database, or it can have many connections with many different databases.
A statement object is used to send SQL statements to a database.

Three kinds:
- Statement
- Prepared Statement
- Callable Statement
Transaction

- A new connection is in auto-commit mode by default
- If autocommit mode has been disabled, a transaction will not terminate until the method `commit` or `rollback` is called explicitly
- Most database drivers support transactions
- Transaction-isolation levels can be set
JDBC

JDBC is a Java API for executing SQL statements (A single program will be able to send SQL statements to the appropriate database)

The API consists of classes and interfaces to send SQL statements to any (relational) database (JDBC is a low level API that supports basic SQL functionality)
JDBC (contd.)

- JDBC makes it possible to do three things:
  - establish a connection with a database
  - send SQL statements
  - process the results
Two-tier & Three-tier Models

JDBC supports both

- two-tier: (client server configuration)
- three-tier model: commands are sent to a middle tier, which then send SQL statements to the database. The database processes the SQL statements and sends the result back to the middle tier (provides performance advantage)
JDBC products

JavaSoft provides three components as part of JDK

- the JDBC driver manager
  Connects java applications to the correct JDBC driver
- the JDBC driver test suite
  Provides the entry SQL functionality for JDBC functionality
- the JDBC-ODBC bridge
  Allows ODBC drivers to be used as JDBC drivers
JDBC Architecture

Java Application

JDBC Driver Manager

JDBC/ODBC Bridge

ODBC Driver

Vendor-supplied JDBC driver

Database

JDBC API

JDBC Driver API

Copyright © 2005, Infosys Technologies Ltd

48
JDBC- ODBC

Why do we not use ODBC from Java?

Why do we need JDBC?

- ODBC uses C interface (security, implementation, robustness, portability)
- ODBC is complex to learn for simple queries, JDBC is easier to use
- JDBC code is automatically installable and portable from network computers to mainframes

You can use ODBC from Java, but this is best done with the help of JDBC in the form of JDBC-ODBC bridge
JDBC-ODBC Bridge

- Application
- JDBC API
- JDBC Manager
- JDBC-ODBC Bridge
- ODBC Manager
- ODBC Driver
- DBMS Server

Provides JDBC access via most ODBC drivers
**JDBC - classes and interfaces**

**DriverManager class** - manages the JDBC drivers that are installed on the system.

- getConnection() : to establish a connection to a database.
  - Connection getConnection(String url)
  - Connection getConnection(String url, String userID, String password)
JDBC - classes and interfaces

- **Connection interface** - defines methods for interacting with the database via the established connection.

- The different methods are:
  - `close()` - closes the database connection
  - `createStatement()` - creates an SQL Statement object
  - `prepareStatement()` - creates an SQL PreparedStatement object.
    (PreparedStatement objects are precompiled SQL statements)
  - `prepareCall()` - creates an SQL CallableStatement object using an SQL string. (CallableStatement objects are SQL stored procedure call statements)
JDBC - classes and interfaces..

**Statement interface** - defines methods that are used to interact with database via the execution of SQL statements.

The different methods are:

- **executeQuery()** - executes an SQL statement (SELECT) that queries a database and returns a ResultSet object.

- **executeUpdate()** - executes an SQL statement (INSERT, UPDATE, or DELETE) that updates the database and returns an int, the row count associated with the SQL statement.

- **execute()** - executes an SQL statement that is written as String object.

- **getResultSet()** - used to retrieve the ResultSet object.
JDBC - classes and interfaces..

**ResultSet Interface** - maintains a pointer to a row within the tabular results. The `next()` method is used to successively step through the rows of the tabular results.

The different methods are:

- `getBoolean(int)` - Get the value of a column in the current row as a Java boolean.
- `getByte(int)` - Get the value of a column in the current row as a Java byte.
- `getDouble(int)` - Get the value of a column in the current row as a Java double.
- `getInt(int)` - Get the value of a column in the current row as a Java int.
JDBC - classes and interfaces..

- **ResultSetMetaData Interface** - holds information on the types and properties of the columns in a ResultSet. Provides information about the database as a whole. Constructed from the Connection object.

- The different methods are:
  - `getColumnName()`
  - `getColumnType()`
  - `getColumnLabel(count)`
import java.util.*;
import java.sql.*;
class StatementTest{
    public static void main(String args[]){
        try{
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            Connection c = DriverManager.getConnection ("jdbc: odbc: Emp");
            Statement sm = c.createStatement();
            int in = sm.executeUpdate("CREATE TABLE
            Emp(empno integer, ename
            varchar(20),deptno integer)");
JDBC - An Example..

int rm1 = sm.executeUpdate("INSERT INTO Emp
values(001, ‘Rahul’, 10)");
int rm2 = sm.executeUpdate("INSERT INTO Emp
values(002, ‘Anu’, 20)");
ResultSet r = sm.executeQuery("SELECT * FROM
Emp");
ResultSetMetaData rsmd = r.getMetaData();
int column = rsmd.getColumnCount();

for(int i=1;i<column;i++){
    if(i>1)
        System.out.print(" ");
    System.out.println(rsmd.getColumnLabel(i));
}

Copyright © 2005, Infosys
Technologies Ltd
JDBC - An Example..

```java
System.out.println("\n");
if(r.next()) {
    int ctr = 0;
    while(r.next()) {
        ctr++;
        for(int i = 1; i <= column; i++) {
            if(i>1)
                System.out.print(" ");
            System.out.println(r.getString(i));
        }
    }
}
```
JDBC - An Example..

```java
}catch(Exception e){
    System.out.println("Exception" + e);
}
```
Client program can connect to Database Server through JDBC Driver.

Since most of the Database servers support ODBC driver therefore JDBC-ODBC Bridge driver is commonly used.

Type of SQL statements which can be executed depends on the support provided by driver.

Transactions can be easily created in Java.
Some Terminologies

- **Node**: any device on the network
- **host**: a computer on the network
- **address**: computer-readable name for host
- **host name**: human-readable name for host
Some Terminologies..

- **IP Address**: a unique number assigned to a computer
- **Port number**: a communication point of an application
- **Blocking**: an action of continuous wait
Client-Server

- **Client** - initiates connection
  - retrieves data,
  - displays data,
  - responds to user input,
  - requests more data
- **Examples of Client:** Web Browser, Chat Program, PC accessing files
Client-Server..

- **Server** - responds to connection
  - receives request for data,
  - looks it up,
  - delivers it

- Examples of Server: Web Server, Database Server, Domain Name Server, etc
Client-Server--Difference

- Difference between client and server is semantic
- It's all just peers talking to each other
- Protocol - roles, vocabulary, rules for communication
Java and Networking

- Built into language
- Network ClassLoader
- java.net API
- Based on TCP/IP, the Internet Protocol
Networking Basics

Computer on Internet communicate to each other using any one of the following:

- Transmission Control Protocol (TCP)
- User Datagram Protocol (UDP)
TCP/IP: Internet Protocol

- Application Layer (HTTP, FTP, SMTP)
- Transport Layer (TCP, UDP)
- Internet Layer (IP)
- Physical Network
Transmission Control Protocol

- **Reliable** - When TCP segments, the smallest unit of TCP transmissions, are lost or corrupted, the TCP implementation will detect this and retransmit necessary segments.

- **Connection-oriented** - TCP sets up a connection before transmission of any data.

- **Continuous Stream** - TCP provides a communication medium that allows for an arbitrary number of bytes to be sent and received smoothly.
User Datagram Protocol

- **Unreliable** - UDP has no mechanism for detecting errors nor for retransmission of lost data
- **Connectionless** - UDP does not negotiate a connection before transmission of data
- **Message-oriented** - UDP allows application to send self-contained messages within UDP datagrams
Sockets & Ports
What is a Socket

- Originally a UNIX idea
- “The network is just like a file system”
- Read and write streams of data “to the network” via a socket.
What is a Socket..

- A socket is bound to a *port number* so that the TCP layer can identify the correct application for data.
Sockets and Ports

- Port: a meeting place on a host
  - one service per port
  - 1-1023 = well-known services
  - 1024+ = experimental services, temporary
- Socket: a two-way connection
Sockets and Ports (Diagram)

Client

Socket

Server

Port 13
Time Service

Port 80
Web Service
The Java Networking Model

Server

- ServerSocket(port #)
- ServerSocket.accept()
- Socket()
- OutputStream
- InputStream
- Socket.close()

Register with this service

Wait for a connection

Client

- Socket(host, port#) (attempt to connect)
- OutputStream
- InputStream
- Socket.close()
Inside java.net package
The net package provides several classes that support socket-based client/server communication.
Inside java.net..

The InetAddress class

- encapsulates Internet IP addresses
- supports conversion between dotted decimal addresses and host names
Inside java.net..

The

- Socket
- Server Socket
- DatagramSocket and
- MulticastSocket

Classes implement client and server sockets for connection-oriented and connectionless communication.
Inside java.net..

- The `DatagramPacket` class is used to construct UDP datagram packets.

- The `SocketImpl` and `DatagramSocketImpl` classes and the `SocketImplFactory` interface provide hooks for implementing custom sockets.
Inside java.net..

The FileNameMap interface is used to map filenames to the MIME types
Inside java.net

The

- URL
- URLConnection
- HTTPURLConnection and
- URLEncoder classes

implement high level browser server Web connections.
The

- **ContentHandler** and
- **URLStreamHandler** classes are abstract classes that have provided the basis for the implementation of Web content supported by

  **ContentHandlerFactory** and

  **URLStreamHandlerFactory** interfaces
Exploring java.net
Datagram Sockets

- The java.net package also supports communication using datagrams.
- Sending or receiving point for a packet delivery service.
- Multiple packets sent from one machine to another may be routed differently, and may arrive in any order.
DatagramSocket - methods

- close() - Closes this datagram socket.
- getLocalAddress() - Gets the local address to which the socket is bound.
- getLocalPort() - Returns the port number on the local host to which this socket is bound.
- receive(DatagramPacket) - Receives a datagram packet from this socket.
- send(DatagramPacket) - Sends a datagram packet from this socket.
Sockets class

- Encapsulated by the Socket object

- Use streams to communicate with the socket object
  
  ```java
  Socket conn = new Socket("www.javaworld.com", 13);
  BufferedReader in = new BufferedReader(new conn.getInputStream());
  String str = in.readLine();
  conn.close();
  System.out.println(str);
  ```
Server-side in Java

- The server part of the socket can also be programmed in Java
- The ServerSocket class helps to listen on a specified port
- Provides a Socket object upon connection from a client
ServerSocket class - methods

- Socket accept() - Listens for a connection to be made to this socket and accepts it.
- void close() - Closes this socket.
- InetAddress getInetAddress() - Returns the local address of this server socket.
- int getLocalPort() - Returns the port on which this socket is listening.
URL Operations

- URL(java.lang.String)
- String getFile();
- String getHost();
- int getPort();
- String getProtocol();
- String getRef();
Review

- Networking capabilities in Java are simple
- Can connect using
  - URLs
  - TCP-based sockets
  - UDP-based sockets
Session Plan (contd.)

Remote Method Invocation

- Need for RMI
- Access to Remote Objects
- RMI APIs
- Client-Server Demo
Remote Method Invocation
Remote Method Invocation

Examples of Use

- Database access
- Computations
- Any custom protocol
- Not for standard protocols (HTTP, FTP, etc.)
The goals for supporting distributed objects

- Support seamless remote invocation on objects in different virtual machines.
- Support callbacks from servers to applets.
- Integrate the distributed object model into the Java language.
- Make differences between the distributed object model and local Java object model apparent.
The goals for supporting distributed objects:

• Make writing reliable distributed applications as simple as possible.
• Preserve the type-safety provided by the Java runtime environment.
• The safe Java environment provided by security managers and class loaders.
Remote Objects

Java Virtual Machine

TCP

Remote Object

Client Object

Java Virtual Machine
Distributed Object Applications

RMI applications are often comprised of two separate programs:

• a server and

• a client.
Typical Server

– A *typical server* application

• creates a number of remote objects,

• makes references to those remote objects accessible

• waits for clients to invoke methods on those remote objects.
A *typical client* application

- gets a remote reference to one or more remote objects in the server and
- invokes methods on them.
Distributed object applications need to:

- Locate remote objects
- Communicate with remote objects
- Load class bytecodes for objects that are passed as parameters or return values
Remote Object

A remote object is one whose methods can be invoked from another Java virtual machine, potentially on a different host.
Remote Interfaces

An object of remote type is described by one or more remote interfaces, which are Java interfaces that declare the methods of the remote object.
Remote method invocation (RMI) is the action of invoking a method of a remote interface on a remote object.
RMI Layers

Java Virtual Machine

Client Object

Object ‘S’ Stub

Remote Reference Layer

TransportLayer

TCP

Java Virtual Machine

Server Object ‘S’

Object ‘S’ Skeleton

Remote Reference Layer

TransportLayer
stubs and skeletons

- RMI uses a standard mechanism (employed in RPC systems) for communicating with remote objects:
  - *stubs* and *skeletons*.
- Stubs and skeletons are generated by the `rmic` compiler.
stub

- lives on client
- pretends to be remote object

i.e. stub which is responsible for carrying out the method call on the remote object.
skeleton

• lives on server

• receives requests from stub

• talks to true remote object

• delivers response to stub

i.e The skeleton is responsible for dispatching the call to the actual remote object implementation.
Remote Reference Layer

- Local pointer's not good enough
- Figures out which remote object is being referenced
- Could span multiple virtual machines
- Communicates via TCP/IP
Transport Layer

- Deals with communications
- Connection management
- Dispatching messages between stub and skeleton
- Sits on top of java.net
The java.rmi Packages

The RMI API is implemented by the following five packages:

- java.rmi
- java.rmi.server
- java.rmi.registry
- java.rmi.activation
- java.rmi.dgc
java.rmi

Provides the \texttt{Remote} interface, class for accessing remote names, the \texttt{MarshalledObject} class, and a security manager for RMI.
java.rmi.server

Provides the classes and interfaces used

- to implement remote objects, stubs, and skeletons
- to support RMI communication.

This package implements the bulk of the RMI API
Creating Remote Objects

- Define a Remote Interface
  - extends java.rmi.Remote
The Remote Interface

package java.rmi;

public interface Remote {}

• The `java.rmi.Remote` interface serves to identify all remote interfaces.

• All remote objects must directly or indirectly implement this interface.
A remote interface may also extend another non-remote interface,

But all of the methods (if any) of the extended interface satisfy the requirements of a remote method declaration.
Class implementing Remote Interface

Implementation classes can implement any number of remote interfaces and can extend other remote implementation classes.

Remote object implementations can extend from these classes:

- java.rmi.server.UnicastRemoteObject
- java.rmi.activation.Activatable.
Compiling Remote Classes

- Compile the Java class
  - `javac`
    - reads .java file
    - produces .class file
- Compile the Stub and Skeleton using rmic
  - reads .class file
  - produces _Skel.class and _Stub.class
Compiling Remote Classes
(Diagram)

Adder.java
(interface)

Adder.class
(interface classfile)

AdderImpl.java
(remote class)

AdderImpl.class
(classfile)

AdderImpl_Stub.class
(stub classfile)

AdderImpl_Skel.class
(skeleton classfile)

javac

rmic
Demo

RMIClientDemo.java
RMIServerDemo.java
RMIServerImpl.java
RMI distributed application

RMI system uses an existing web server to load Java class bytecodes, from/to server and client for objects when needed.
Java Security Model
Why Security?

- Code obtained from network may contain
  - virus which may produce problems on the user computer
  - or some malicious executable programs which may steal away some confidential information (credit card number, bank account info., etc.) or alter them without the knowledge of users.

- Solution:
  security mechanisms which helps make Java suitable for networks because they establish a needed trust in the safety of network-mobile code.
Evolution of Java Security

Original security model (JDK1.0)

- applets: not trusted for accessing any system resource
- applications i.e. local code: full access to all vital system resources.

What is this Sandbox Model?
Sandbox

- an intrinsic part of Java's architecture
- a shell that surrounds a running Java program, protects the host system from malicious code

What does it restrict?
Sandbox Restricted access

- For running untrusted code obtained from network
- To prohibit many activities e.g.:
  - Reading or writing to the local disk
  - Making a network connection to any host, except the host from which the applet came
  - Creating a new process
  - Loading a new dynamic library and directly calling a native method
Security Architecture II

Modified security model (JDK1.1)

- provided restricted but somewhat flexible environment
  - applets: not trusted for accessing any system resource but digitally signed applets were considered as local code after verification
  - applications i.e. local code: full access to all vital system resources.

Is there any change in the model in JDK1.2?
Extended Sandbox Security Model (JDK 1.2)
Extended SandBox Model (contd.)

Domain

runtime system organizes code into individual domains, each of which encloses a set of classes whose instances are granted the same set of permissions
Extended SandBox Model (contd.)

Security Manager

Java.lang.SecurityManager is a class that allows applications to implement security policy

Methods inside this class are called by various methods in runtime libraries before those methods perform certain potentially sensitive operations
Extended SandBox Model (contd.)

Security Policy

- grants only those privileges which are allowed for the applets and applications
- specified in the form of an ASCII text file (system policy file) located at `<java.home>\lib\security\java.policy`
Policy Permissions

- The policy configuration files
  - that define the security policy i.e. it specifies what permissions are allowed for code from specified code sources.
  - Implemented by java.security.Permission classes & their sub-classes
- System policy is loaded first by Java byte code interpreter & then user policy
- If none is present then original sandbox policy is loaded.
Cryptography Support

- To ensure
  - confidentiality
  - authentication
Cryptography

- the study of algorithms and protocols for securing messages during transmission and storage.
  - **Encryption**: Process of transforming the text in such a way that it can not be read without authorization
  - **Decryption**: Encrypted data can be decrypted in its original form with a key (mark of authorization)
Java Cryptography Architecture (JCA)
Java Security API

- Provides a flexible framework for implementing cryptographic functions and other security controls
- includes standard algorithms to support these security features (e.g. MD5 and DSA)
- many different cryptographic packages can be plugged in and installed within the common Security API framework.
Cryptographic Engines

- Supported by Security API

- includes following classes
  - `MessageDigest` : supports computation of message digest
  - `Signature` : supports creation and verification of digital signature
  - `KeyPairGenerator` : used to access provider-furnished key generation algorithms
  - some more...

  - `service provider interface (SPI)` is an abstract class that defines the service provider interface methods that cryptographic service providers must implement.
Digital Signatures

- Uses a public key algorithm with little change i.e. private key is used for encryption and public key is used for decryption

Features of Digital Signature:
  - Unforgeability
  - Verifiablity
  - Single use
  - Non-repudiation
  - Sealing
Digital Certificates

- These basically work on the same principle of digital signatures.
- Digital certificates are messages signed by a certifying authority which certify the value of the particular public key.
- The X.509 are the well known digital certificate format.
Digital certificate working...

User → User’s info
User → User’s public key

Certification Authority
Signature algorithm

User’s certificate
Signed by CA
JAR files and Digital signatures

- Jar file are used to group all the related files into one .jar file.
- This helps in terms of performance as browser has to make just one request to the web server.
- Signed archived jar files can be used to make untrusted applets as trusted applets.
How do I create a jar file…..

The *jar* tool provided by the jdk is used to create the jar file

Syntax :

```
jar [c,f,x,t] filename.jar  filestobeadded
```
Execution of jar files…. 

To the html file through which the applet is invoked add the following applet tag option

archive = “filename.jar”

Execute the html file with the appletviewer

appletviewer –J-Djava.policy=Mypolicy my.html
Digital signatures for Applets

Steps:

1. Create the public/private key pair using `keytool`
2. Sign the jar file created using `jarsigner`
3. Assign the user created policy file to the java policy file during execution of the applet using `-J-D` option of appletviewer.
Creation of keys.

The `keytool` provided in the jdk1.2\bin. Use this command with the –genkey option.

Eg:

```bash
keytool -genkey -alias "key" -keystore "storekey"
```

This generates a key pair for the alias “key” in the keystore “storekey”

Key tool then prompts to enter the password, after that just fill in the additional information asked.
Signing the jar file

The `jarsigner` tool can be used for signature generation and to verify the signature.

Eg.

```
jarsigner –keystore storekey –storepass “storepassword”
- keypass “MyPassword” jarfilename alias
```

Verify the signature with –`verify` option of jarsigner
Execution of the program

Execute the .html file through appletviewer with the options –J-D and also assigning the policy file created to the java.policy.

Ex.

appletviewer –J-Djava.policy=test.policy my.html
Java Naming and Directory Interface (JNDI)
Introduction

Directory and Naming Services is used to organize information hierarchically to map human understanding of names and directory objects.
Naming Concept

Naming System

- associates names with addresses
  
  example: Phone book associates people’s name with phone number and addresses)

Naming Service

- software system which exposes the naming system to other softwares
Naming Services

- Maintains a set of binding that relates name to object
- Clients use naming service to locate objects by name
Directory Concept

Directory Service

- extended naming service
- allows not only name of the object to be stored but also its attributes
- helps in searching objects not only by name but also by one of its attributes
Directory Services..

- arrange the namespaces created in the Naming Services in a hierarchy
- provides operations for creating, adding, removing and modifying attributes associated with objects in a directory
Directory Services..

- Like the DOS file system; where the hierarchy starts from the root directory then the subdirectories and then the files.
- It also has attributes like the date, size of the file which gives us additional information.
Network Directory Services

- These provide information about the network, network services etc..
- Example of Network directory Services:
  - Novell Netware Directory Services (NDS)
  - Network Information Services Plus (NIS+)
- All these Network Directory Services adopt the ISO’s X.500 Directory service and are proprietary.
Lightweight Directory Access Protocol

- LDAP is a non-proprietary protocol.

- LDAP works over the TCP/IP

- It makes directory management simpler and also makes the directories globally accessible.
LDAP

- LDAP defines how the data should be accessed by the client.
- Most of the Directory Services have LDAP as their front-end.
## An LDAP Entry

<table>
<thead>
<tr>
<th>Entry</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Value</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Entry**
- **Attributes**
  - **Type**
  - **Value**
LDAP Entry

- The Entry is called the Distinguished Name which can have any number of attributes.

- Some of the LDAP Attributes:
  - cn: name
  - uid: userid
  - mail: email address

Objectclass: it tells what attributes can be allowed for a particular entry
Java Naming and Directory Interface (JNDI) is an API that provides directory and naming functionality to Java applications.

It is defined to be independent of any specific directory service implementation.

Thus, a variety of directories can be accessed in a common way.
JNDI Architecture

Java Application

JNDI Implementation Manager

JNDI API

JNDI SPI

JNDI-RMI

LDAP

NDS
The JNDI architecture consists of the JNDI API and the JNDI SPI (Service Provider Interface).

- JNDI API - allows Java applications to access different types of naming and directory services.
- JNDI SPI - designed to be used by arbitrary service providers including directory service providers.
JNDI API

This standard extension API consists of three packages:

- `javax.naming`
  - supports naming operations

- `javax.naming.directory`
  - supports directory operations

- `javax.naming.spi`
  - provide support for service provider interface
Summary

- RMI Interface can be used to invoke a method of an object on different machine or different JVM on same machine
- Java security model is critical as mostly this language is being used in Internet scenario
- JNDI API are used to interface Java Programs with directory services to simplify enterprise networking
Thank You!