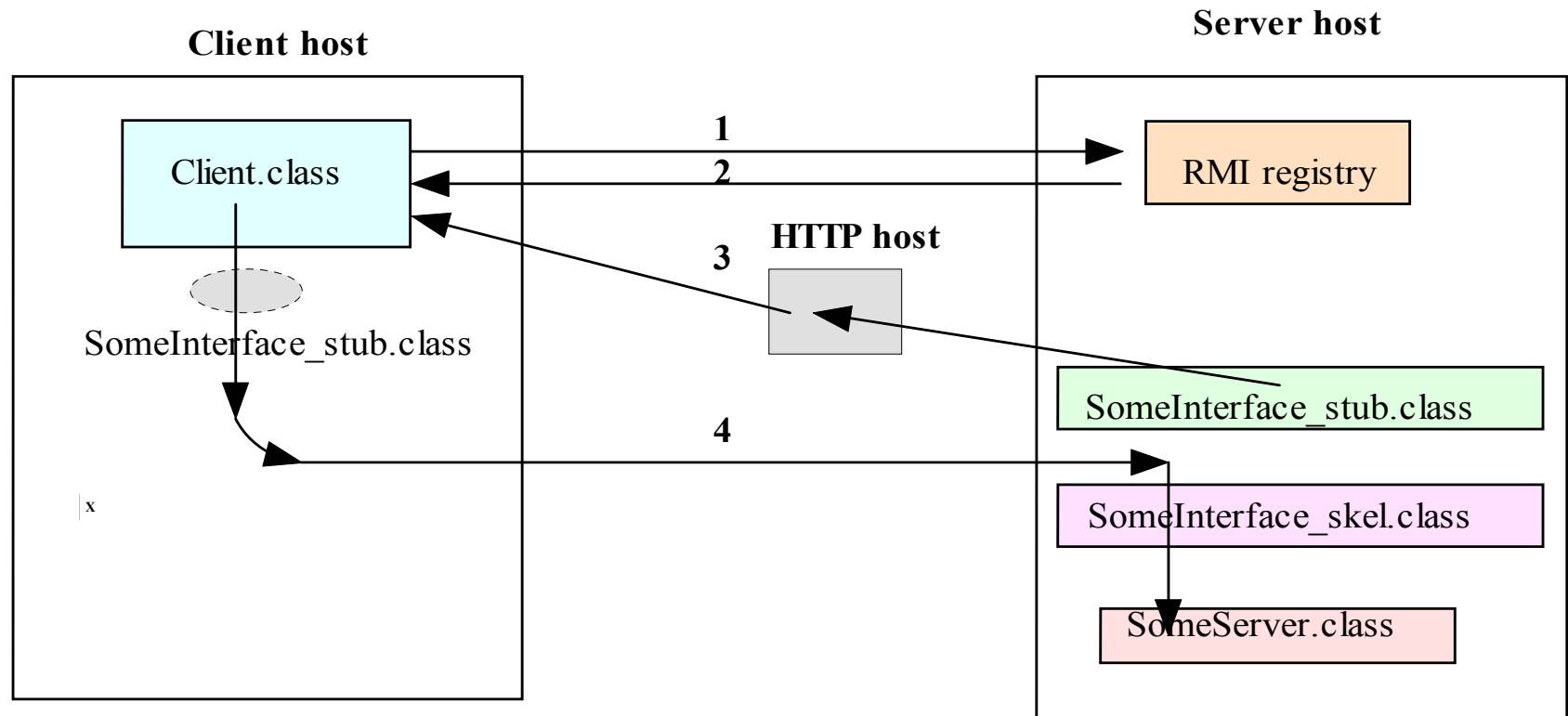


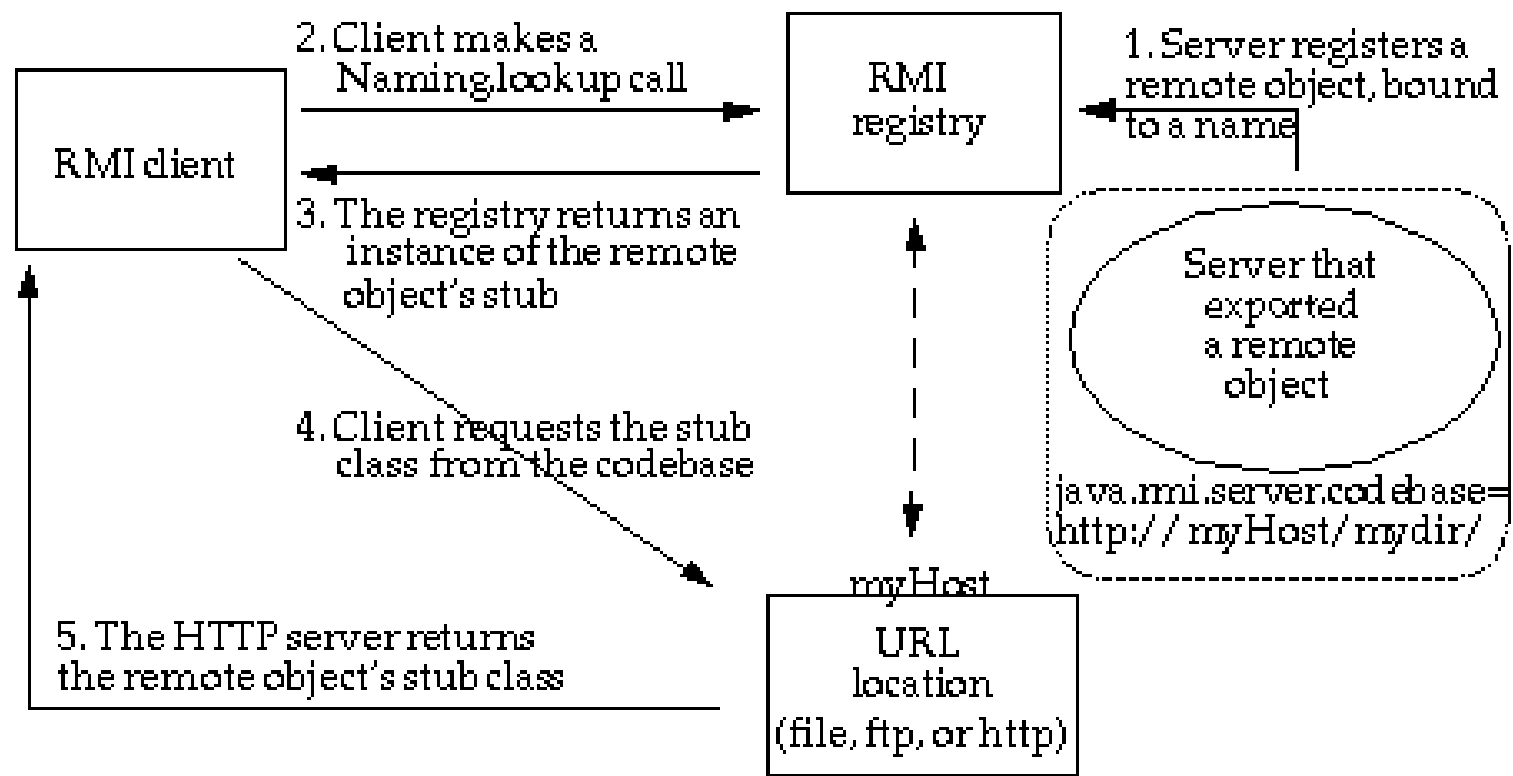
# Using Remote Method Invocation to Implement Callbacks

# Java RMI Client Server Interaction



1. Client looks up the interface object in the RMI registry on the server host.
2. The RMI Registry returns a remote reference to the interface object.
3. If the interface object's stub is not on the client host and if it is so arranged by the server, the stub is downloaded from an HTTP server.
4. Via the server stub, the client process interacts with the skeleton of the interface object to access the methods in the server object.

# CodeBase



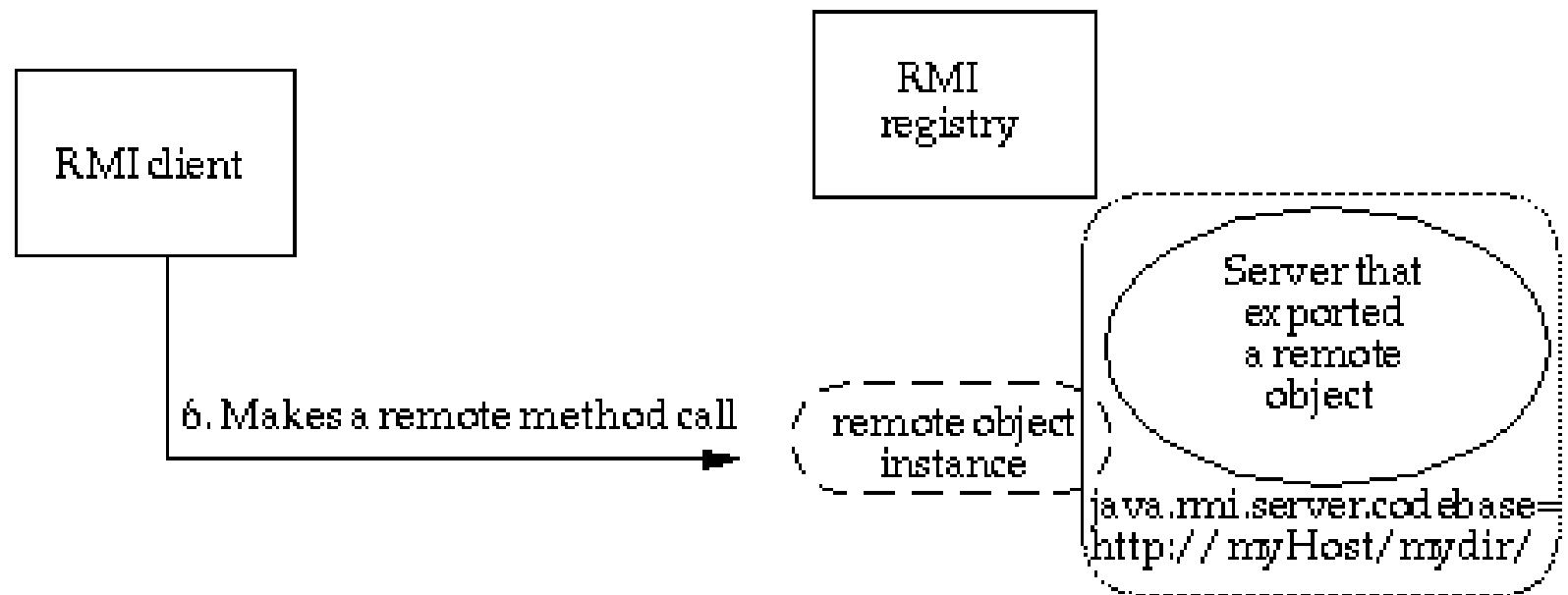
- A codebase can be defined as a source, or a place, from which to load classes into a virtual machine

# ► CodeBase

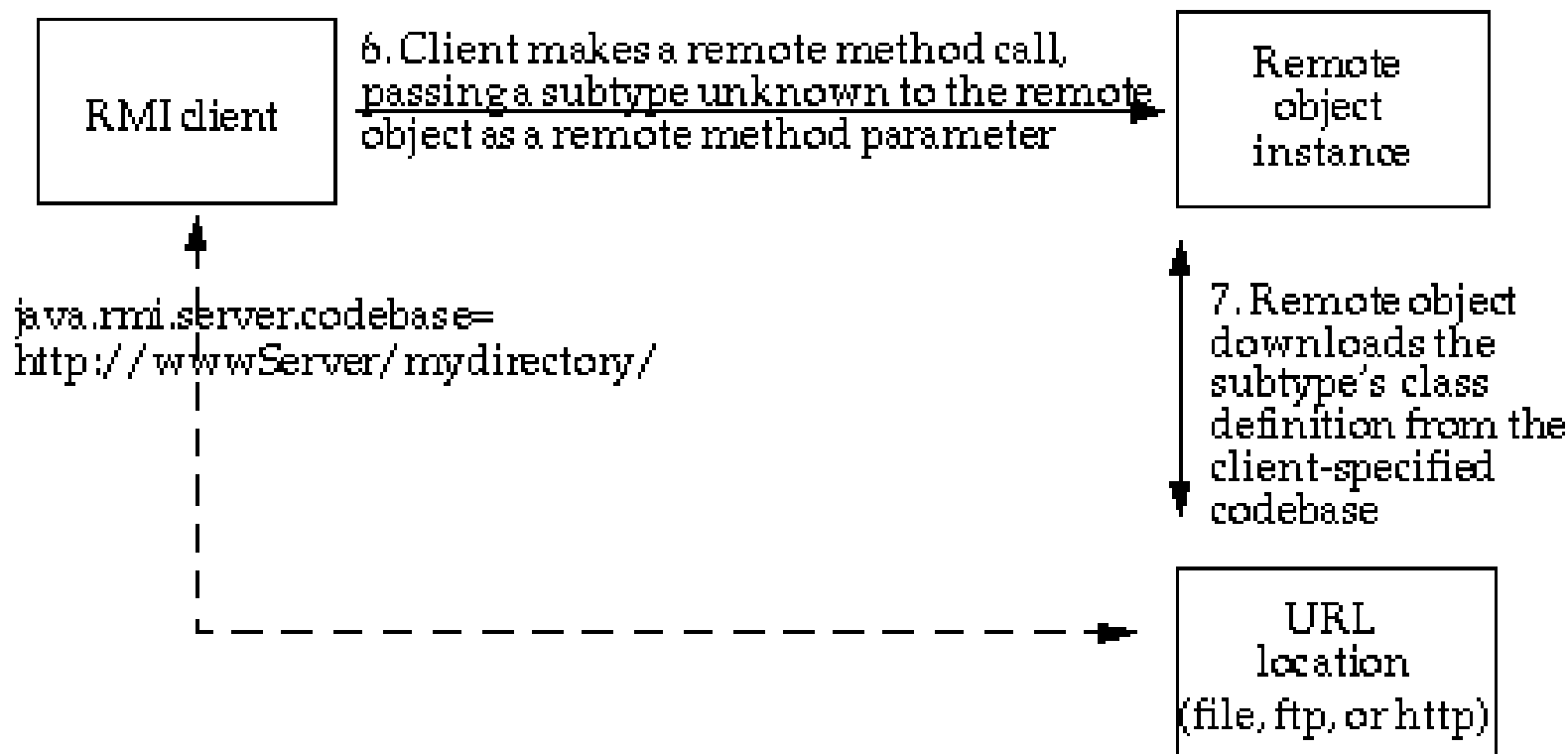
1. The remote object's **codebase** is specified by the remote object's server by setting the **java.rmi.server.codebase** property. The Java RMI server registers a remote object, bound to a name, with the Java RMI registry. The codebase set on the server VM is annotated to the remote object reference in the Java RMI registry.
2. The Java RMI client requests a reference to a named remote object. The reference (the remote object's stub instance) is what the client will use to make remote method calls to the remote object.
3. The Java RMI registry returns a reference (the stub instance) to the requested class. If the class definition for the stub instance can be found locally in the client's **CLASSPATH**, which is always searched before the codebase, the client will load the class locally. However, if the definition for the stub is not found in the client's **CLASSPATH**, the client will attempt to retrieve the class definition from the remote object's codebase.

# ► CodeBase

4. The client requests the class definition from the **codebase**. The codebase the client uses is the URL that was annotated to the stub instance when the stub class was loaded by the registry.
  5. The class definition for the stub (and any other class(es) that it needs) is downloaded to the client.
  6. Now the client has all the information that it needs to invoke remote methods on the remote object. The stub instance acts as a proxy to the remote object that exists on the server; so unlike the applet which uses a codebase to execute code in its local VM, the Java RMI client uses the remote object's codebase to execute code in another, potentially remote VM
- In addition to downloading stubs and their associated classes to clients, the `java.rmi.server.codebase` property can be used to specify a location from which any class, not only stubs, can be downloaded.



- **Java RMI client making a remote method call**



Java RMI client making a remote method call, passing an unknown subtype as a method parameter

# ► Preparing for Deployment

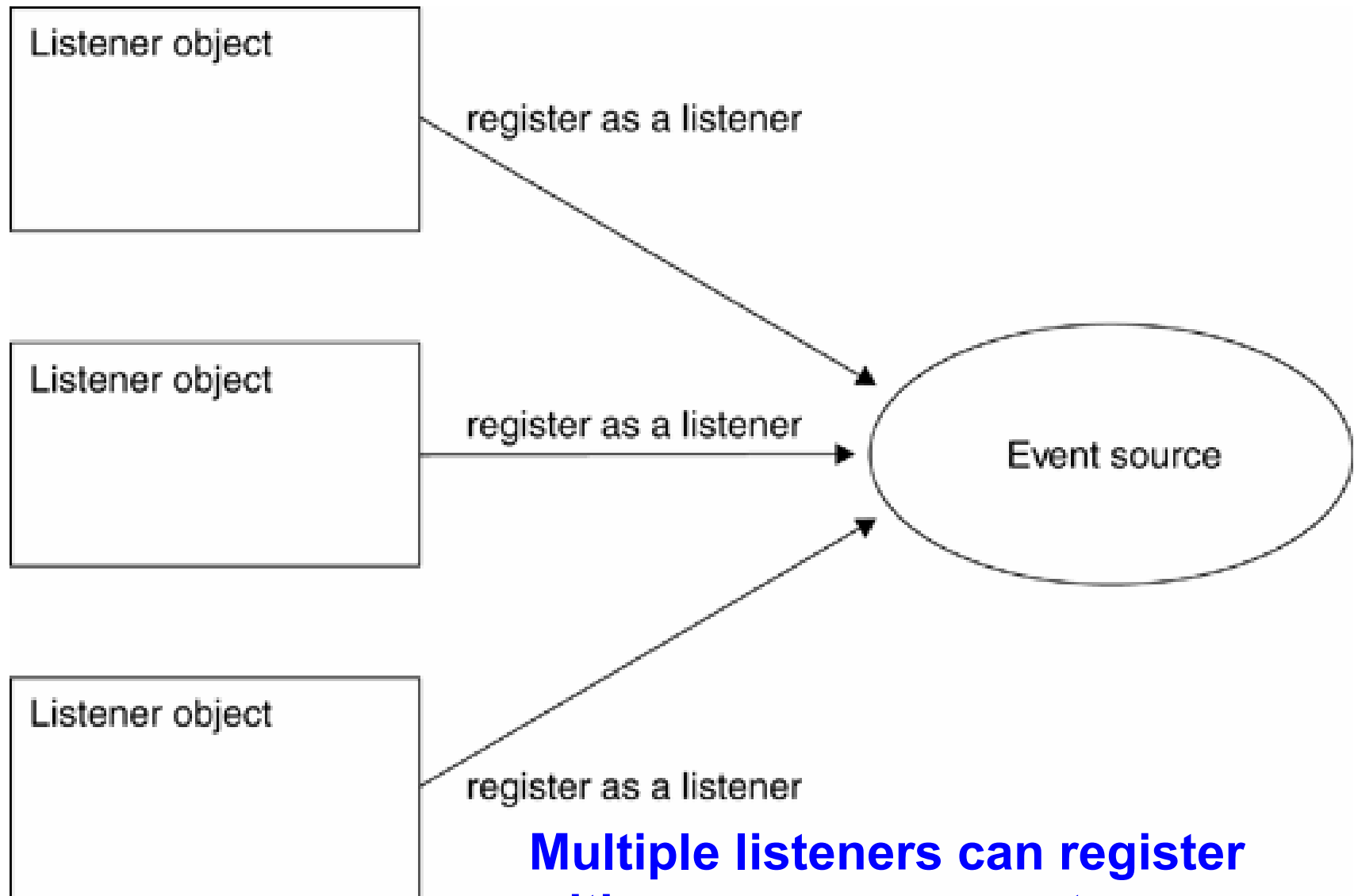
- Deploying an application that uses RMI can be tricky because so many things can go wrong and the error messages that you get when something does go wrong are so poor. Separate the class files into three subdirectories:
  - **server**
  - **download**
  - **client**
- **The server directory contains all files that are needed to run the server.** You will later move these files to the machine running the server process. In our example, the server directory contains the following files:
  - **server/**
    - ProductServer.class**
    - ProductImpl.class**
    - Product.class**



# ► Preparing for Deployment

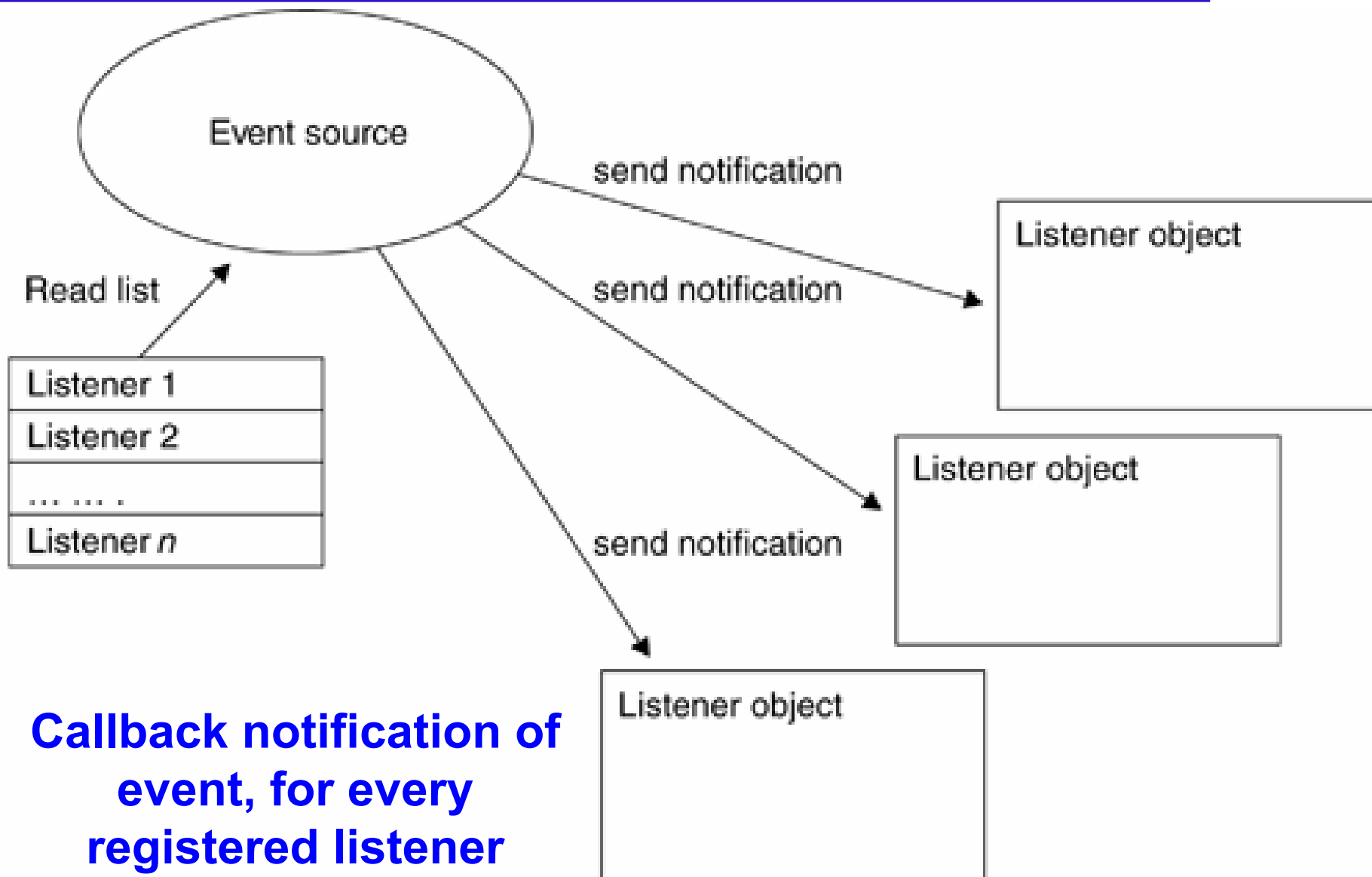
- The client directory contains the files that are needed to start the client. These are
    - `client/`
      - `ProductClient.class`
      - `Product.class`
      - `client.policy`
  - You will deploy these files on the client computer. Finally, the **download directory contains those class files needed by the RMI registry, the client, and the server, as well as the classes they depend on**. In our example, the download directory looks like this:
    - `download/`
      - `ProductImpl_Stub.class`
- java -Djava.rmi.server.codebase = http://localhost:8080/  
download/ ProductServer &**

# ► Using RMI to Implement Callbacks

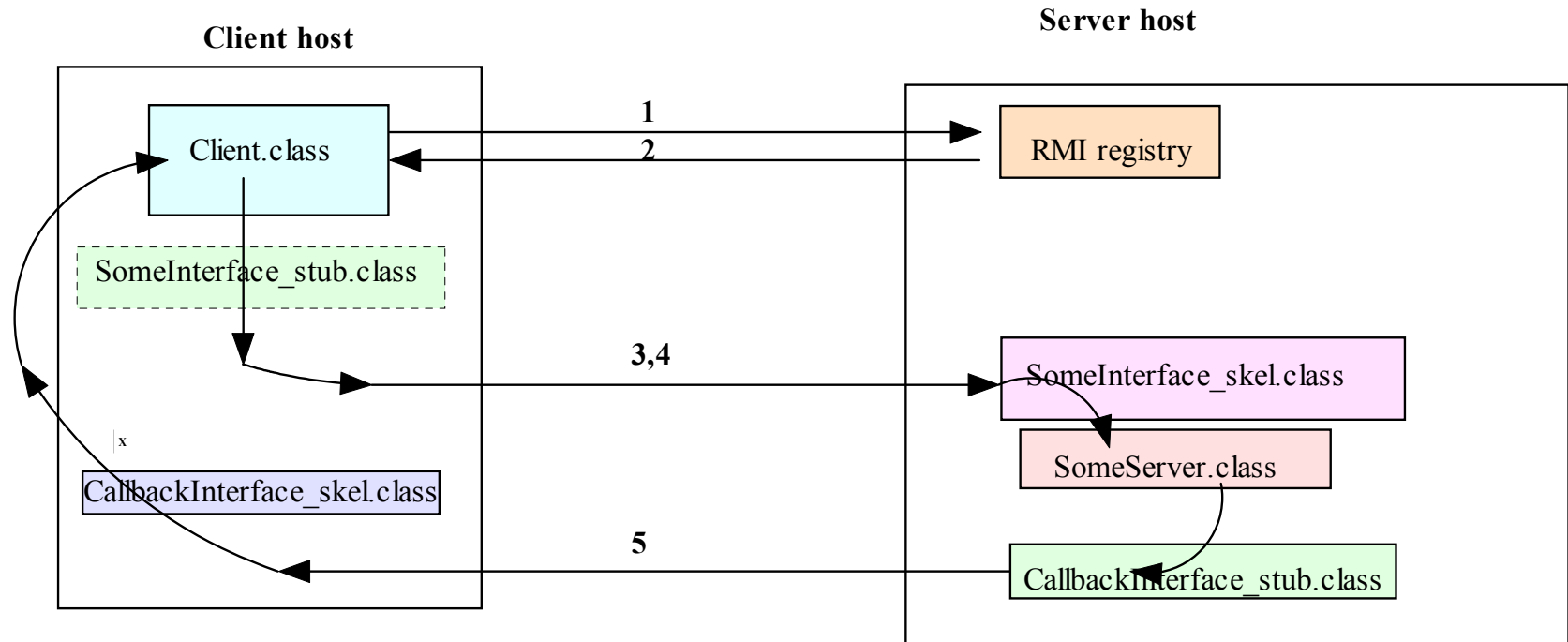


**Multiple listeners can register with one or more event sources.**

# ► Using RMI to Implement Callbacks



# ▶ Callback Client-Server Interactions



1. Client looks up the interface object in the RMI registry on the server host.
2. The RMI Registry returns a remote reference to the interface object.
3. Via the server stub, the client process invokes a remote method to register itself for callback, passing a remote reference to itself to the server. The server saves the reference in its callback list.
4. Via the server stub, the client process interacts with the skeleton of the interface object to access the methods in the interface object.
5. When the anticipated event takes place, the server makes a callback to each registered client via the callback interface stub on the server side and the callback interface skeleton on the client side.

# ► Defining the Listener Interface

- The listener interface defines a remote object with a single method. This method should be invoked by an event source whenever an event occurs, so as to act as notification that the event occurred. The method signifies a change in temperature, and allows the new temperature to be passed as a parameter.

```
interface TemperatureListener extends java.rmi.Remote {  
    public void temperatureChanged(double temperature)  
    throws java.rmi.RemoteException;  
}
```

# ► Defining the Event Source Interface

- The event source must allow a listener to be registered and unregistered, and may optionally provide additional methods. In this case, a method to request the temperature on demand is offered.

```
interface TemperatureSensor extends java.rmi.Remote{  
    public double getTemperature()  
        throws java.rmi.RemoteException;  
    public void addTemperatureListener  
        (TemperatureListener listener )  
        throws java.rmi.RemoteException;  
    public void removeTemperatureListener  
        (TemperatureListener listener )  
        throws java.rmi.RemoteException;  
}
```

# ► Implementing the Event Source Interface

- Once interfaces have been defined, the next step is to implement them. A **TemperatureSensorServerImpl** class is defined, which acts as an RMI server.
- To notify registered listeners as a client (**The server must extend **UnicastRemoteObject****, to offer a service, and **implement the **Temperature Sensor**** interface.
- To create an instance of the service and registering it with the **rmiregistry**
- **To launch a new thread, responsible for updating the value of the temperature**, based on randomly generated numbers. Both the amount (plus or minus 0.5 degrees) and the time delay between changes are generated randomly.
- As each change occurs, registered listeners are notified, by reading from a list of listeners stored in a **java.util.Vector** object. This list is modified by the remote **addTemperatureListener(TemperatureListener)** and **removeTemperatureListener(TemperatureListener)** methods.

# ► Implementing the Event Source Interface

```
public class TemperatureSensorImpl extends UnicastRemoteObject
    implements TemperatureSensor, Runnable {
    private volatile double temp;
    private Vector list = new Vector();
    public TemperatureSensorImpl() throws java.rmi.RemoteException {
        super();
        temp = 98.0; // Assign a default setting for the temperature
    }
    public double getTemperature() throws java.rmi.RemoteException {
        return temp;
    }
    public void addTemperatureListener(TemperatureListener listener)
        throws java.rmi.RemoteException {
        System.out.println("adding listener -" + listener);
        list.add(listener);
    }
    public void removeTemperatureListener(TemperatureListener listener)
        throws java.rmi.RemoteException {
        System.out.println("removing listener -" + listener);
        list.remove(listener);
    }
}
```



# ► Implementing the Event Source Interface

```
public void run() {  
    Random r = new Random();  
    for (; ; ) {  
        try {  
            // Sleep for a random amount of time  
            int duration = r.nextInt() % 10000 + 2000;  
            // Check to see if negative, if so, reverse  
            if (duration < 0) duration = duration * -1;  
            Thread.sleep(duration);  
        } catch (InterruptedException ie) {}  
        // Get a number, to see if temp goes up or down  
        int num = r.nextInt();  
        if (num < 0) temp += 0.5; else temp -= 0.5;  
        notifyListeners(); // Notify registered listeners  
    }  
}
```

# ► Implementing the Event Source Interface

```
private void notifyListeners() {  
    // Notify every listener in the registered list  
    for (Enumeration e = list.elements();  
         e.hasMoreElements();) {  
        TemperatureListener listener = (TemperatureListener)  
            e.nextElement();  
        // Notify, if possible a listener  
        try {  
            listener.temperatureChanged(temp);  
        } catch (RemoteException re) {  
            System.out.println("removing listener -" + listener);  
            // Remove the listener  
            list.remove(listener);  
        }  
    }  
}
```

# ► Implementing the Listener Interface

- The temperature monitor **client must implement the `TemperatureListener` interface, and register itself with the remote temperature sensor service**, by invoking the **`TemperatureSensor.addTemperatureListener(Temperature Listener)`** method.
- By registering as a listener, the monitor client will be notified of changes as they occur, using a remote callback. The client waits patiently for any changes, and though it does not ever remove itself as a listener, functionality to achieve this is supplied by the **`TemperatureSensor.removeTemperatureListener(TemperatureListener)`** method.

# ► Implementing the Listener Interface

```
import java.rmi.*;
import java.rmi.server.*;
public class TemperatureListenerImpl extends
UnicastRemoteObject implements TemperatureListener {
    // Default constructor throws a RemoteException
    public TemperatureListenerImpl() throws
        RemoteException {
        super();
    }

    public void temperatureChanged(double temperature)
        throws java.rmi.RemoteException {
        System.out.println("Temperature change event : " +
            temperature);
    }
}
```

# ► TemperatureSensorServer

```
public class TemperatureSensorServer{
    public static void main(String args[]) {
        System.out.println("Loading temperature service");
        try {
            // Load the service
            TemperatureSensorImpl sensor = new TemperatureSensorImpl();
            // Register with service so that clients can find us
            String registry = "localhost";
            String registration = "rmi://" + registry + "/TemperatureSensor";
            Naming.rebind(registration, sensor);
            // Create a thread, and pass the sensor server. This will activate the
            //run() method, and trigger regular temperature changes.
            Thread thread = new Thread(sensor);
            thread.start();
        } catch (RemoteException re) {
            System.err.println("Remote Error - " + re);
        } catch (Exception e) {
            System.err.println("Error - " + e);
        }
    }
}
```

# ► TemperatureMonitor

```
public static void main(String args[]) {
    System.out.println("Looking for temperature sensor");
    try {
        // Lookup the service in the registry, and obtain a remote service
        String registry = "localhost";
        String registration = "rmi://" + registry + "/TemperatureSensor";
        Remote remoteService = Naming.lookup(registration);
        // Cast to a TemperatureSensor interface
        TemperatureSensor sensor = (TemperatureSensor) remoteService;
        // Get and display current temperature
        double reading = sensor.getTemperature();
        System.out.println("Original temp : " + reading);
        // Create a new monitor and register it as a listener with remote sensor
        TemperatureListenerImpl monitor = new TemperatureListenerImpl();
        sensor.addTemperatureListener(monitor);
    } catch (RemoteException re) {
        System.out.println("RMI Error - " + re);
    } catch (Exception e) {
        System.out.println("Error - " + e);
    }
}
```

# ► Remote Object Activation

- One of the chief disadvantages of using RMI for large and complex systems is the amount of overhead generated by RMI server software. Suppose you are constructing a massive system, with many different RMI service interfaces and server implementations.
- **Each server would have to be running continually, even if it was seldom used by client software.** Servers must create and export remote objects that implement RMI service interfaces, which consumes memory and CPU time.
- **If each implementation is created as a separate server, this means that there will be as many JVM instances as there are services.**
- **Remote object activation is a technique that solves the problem of running a large number of idle RMI services.** It allows services to be registered with the rmiregistry, but not instantiated. Instead, they remain inactive, until called upon by a client, when they will awaken and perform their operation. **A special daemon process called the *remote method invocation activation system daemon (rmid)*** listens for these requests and instantiates RMI services on demand.

# ► Creating an Activatable Remote Object

- **Creation** of an activatable remote object is a little different from that of a **UnicastRemoteObject**, but the code is fairly similar.

```
public interface MyRemoteInterface extends  
                                java.rmi.Remote{  
    public void doSomething () throws  
                                java.rmi.RemoteException;  
}
```

- The next step is to **create an implementation** that extends the **java.rmi.activation.Activatable** class. It must implement our RMI interface and provide both a constructor and a **doSomething()** method. Unlike classes that extend the **UnicastRemoteObject** class, a different constructor signature is used. This constructor calls the parent constructor and exports the object on any available port. The service is then available for use by clients.



# ► Implementing an Activatable Remote Object

```
public class MyRemoteInterfaceImpl extends
    java.rmi.activation.Activatable
        implements MyRemoteInterface{

    public MyRemoteInterfaceImpl
        (java.rmi.activation.ActivationID
            activationID, java.rmi.MarshalledObject data) throws
            java.rmi.RemoteException {
        // call the Activatable(ActivationID activationID,
        // int port) parent constructor
        super (activationID, 0);
    }

    public void doSomething(){
        System.out.println ("Doing something....");
    }
}
```

# ► Registering an Activatable Remote Object

## • Step One

The first step is to create an activation group descriptor. This descriptor is associated with an activation group, and allows you to specify system properties. In this example, no system properties are required and a new instance of the Properties class is created.

- // Step one : create an ActivationGroupDesc instance

```
ActivationGroupDesc groupDescriptor = new  
ActivationGroupDesc(new Properties(), null);
```

## • Step Two

- The second step is to register the activation group descriptor with the RMI activation daemon process. This process is represented by the **Activation System** interface. Applications do not create or extend this interface. Instead, a reference to the activation daemon process is obtained by invoking the static **ActivationGroup.getSystem()** method.

- // Step two : register that activation group descriptor  
// with the activation system, and get groupID

```
ActivationSystem system = ActivationGroup.getSystem();
```

# ► Registering an Activatable Remote Object

- Once a reference to the **ActivationSystem** class is obtained, the group descriptor is registered using the **ActivationSystem.registerGroup** method, which returns an **ActivationGroupID** instance.
- // Register the group descriptor - without registering the // group, no execution  
**ActivationGroupID groupId = system.registerGroup(groupDescriptor);**
- **Step Three**
- Once a group descriptor has been registered with the activation system and a valid **ActivationGroupID** obtained, the next step is to actually create the activation group. The act of registering it is not sufficient—one must be created, by calling the static **ActivationGroup.createGroup( ActivationGroupID id, ActivationGroupDesc descriptor)** method. This creates and assigns an activation group for the JVM, and notifies the activation system that the group is active.
- // Step three: create an activation group, passing the // group ID and descriptor as parameters  
**ActivationGroup.createGroup(groupId, groupDescriptor, 0);**

# ► Registering an Activatable Remote Object

## • Step Four

- The fourth step is to create an activation descriptor, which describes an activatable remote object. It stores three key pieces of information: the class name of the remote object, the URL location of the remote object, and (optionally) a `MarshaledObject` that contains a serialized version of the remote object.

## • String strLoc = new

`File(".").getCanonicalFile().toURI().toString();`

// whose constructor requires the class name, codebase, //and an optional marshalled object

`ActivationDesc desc = new`

`ActivationDesc("RMIActivationLightBulbImpl", strLoc, null);`

## • Step Five

- The fifth step is to register the activation descriptor with the activation system. This instructs the activation system of the class name and the location of the class definition file. A remote stub will be returned, which can then be registered as an RMI service.

- // Step five : Register the object with the activation system  
// Returns a stub, which may be registered with rmiregistry

`Remote stub = Activatable.register(desc);`

# ► Registering an Activatable Remote Object

## • Step Six

- The sixth and final step is to add a registry entry for the RMI service, so that clients may locate the service. The code for this is similar to that of the previous RMI lightbulb server, except that a remote stub is registered, not an instance of a remote object.

- // Check to see if a registry was specified

**String registry = "localhost";**

**// Registration format //registry\_hostname:port/service**

**// Note the :port field is optional**

**String registration = "rmi://" + registry + "/RMILightBulb";**

**// Step six : Register the stub with the rmiregistry**

**Naming.rebind(registration, stub);**

**System.out.println("Service registered with rmid. Now  
terminating...");**

**System.exit(0);**

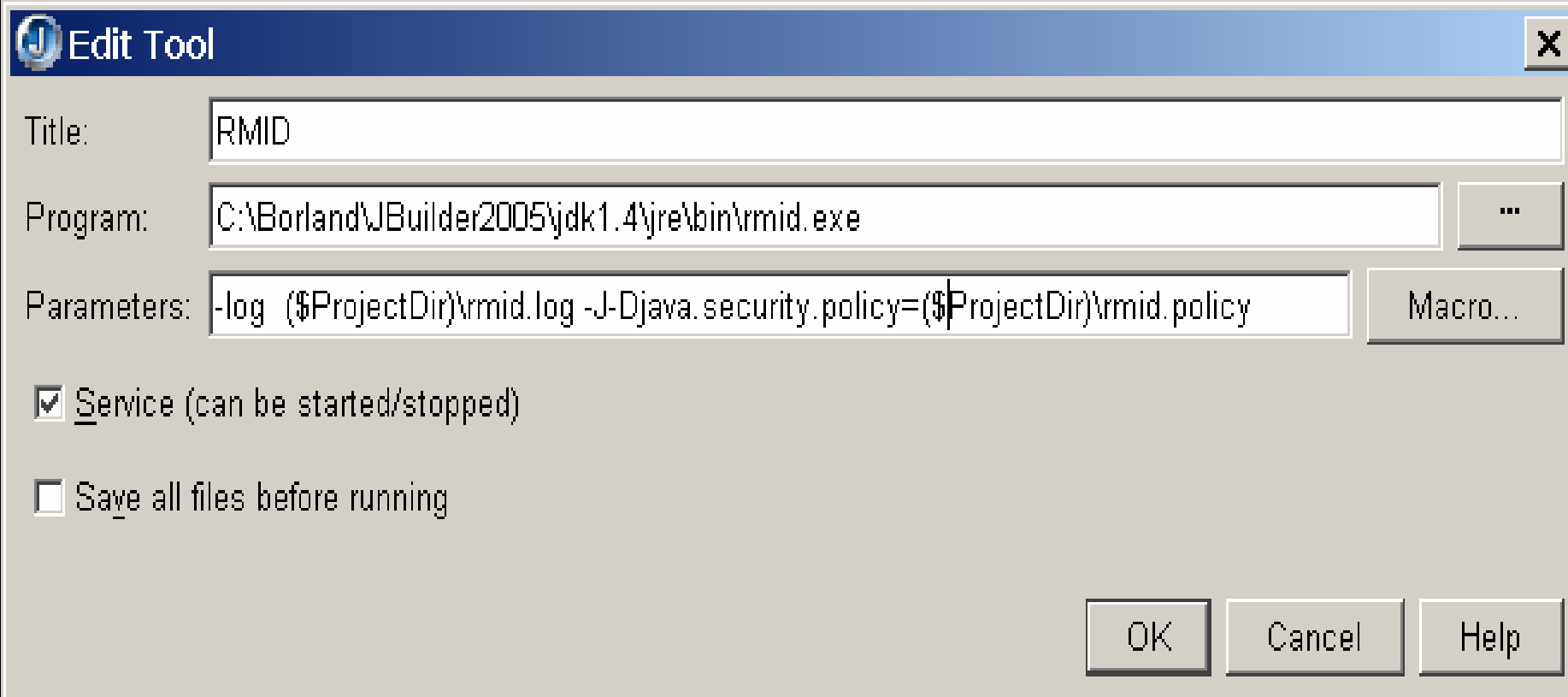
# ► Configuring a RMID

01-2004

Khoa CNTT

30/28

PHẠM VĂN TÍNH



The screenshot shows a dialog box titled "Edit Tool" with a close button (X) in the top right corner. The dialog contains the following fields and options:

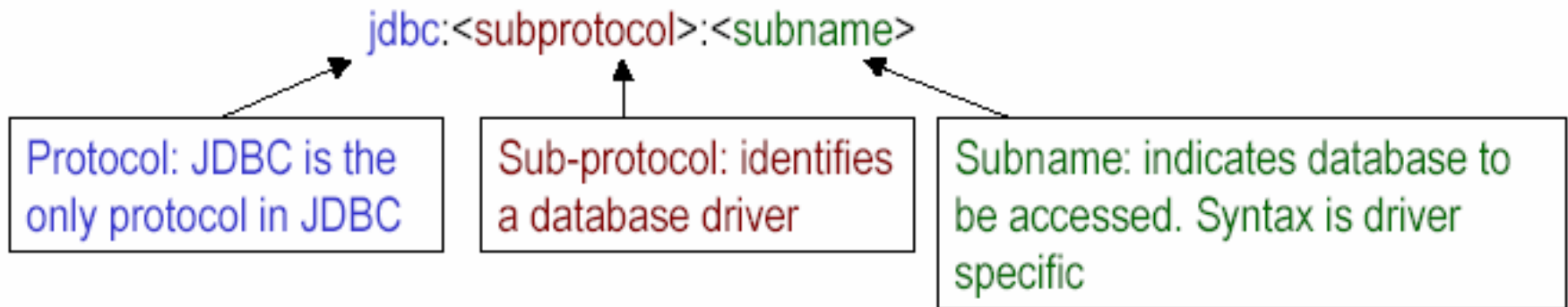
- Title:** A text field containing "RMID".
- Program:** A text field containing "C:\Borland\JBuilder2005\jdk1.4\jre\bin\rmid.exe", followed by a button with three dots "...".
- Parameters:** A text field containing "-log (\$ProjectDir)\rmid.log -J-Djava.security.policy=(\$ProjectDir)\rmid.policy", followed by a button labeled "Macro...".
- Service:** A checked checkbox labeled "Service (can be started/stopped)".
- Save:** An unchecked checkbox labeled "Save all files before running".
- Buttons:** "OK", "Cancel", and "Help" buttons at the bottom right.

# ► Basic JDBC Programming Concepts

## ► Database URLs



# Database URLs



- For databases on the Internet/intranet, the **subname** can contain the Net URL **//hostname:port/...** The **<subprotocol>** can be any name that a database understands. The `odbc` subprotocol name is reserved for ODBC-style data sources. A normal ODBC database JDBC URL looks like:  
`jdbc:odbc:<>;User=<>;PW=<>`
- `jdbc:postgresql://www.hcmuaf.edu.vn/ts`

# ► Making the Connection

- Loading a driver:

- `Class.forName(className)`  
`Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");`
  - `System.setProperty("jdbc.drivers", "className");`  
`System.setProperty("jdbc.drivers",  
"sun.jdbc.odbc.JdbcOdbcDriver");`

- Making a connection

- `Connection conn = DriverManager.getConnection(...);`  
`static Connection getConnection(String url);`  
`static Connection getConnection(String url,`  
`String user,String password);`  
`static int getLoginTimeout(); // seconds`  
`static void setLoginTimeout(int seconds);`  
`static void println(String message); // in ra logfile`  
`static void registerDriver(Driver d);`  
`static void deregisterDriver(Driver d);`

# ▶ Executing SQL Commands

# ▶ Executing SQL Commands

- The **Statement** object does all of the work to interact with the Database Management System in terms of SQL statements. You can create many **Statement** objects from one **Connection** object.
- JDBC supports three types of statements:
  - **Statement** : the SQL is prepared and executed in one step (from the application program point of view)
  - **PreparedStatement**: the driver stores the execution plan handle for later use (SQL command with a parameters).
  - **CallableStatement**: the SQL statement is actually making a call to a stored procedure
- The **Connection** object has the **createStatement()**, **prepareStatement()**, and **prepareCall()** methods to create these **Statement** objects.

# ► Creating and Using Direct SQL Statements

- A `Statement` object is created using the `createStatement()` method in the `Connection` object obtained from the call to `DriverManager.getConnection`
- `resultSet executeQuery (String sql)`
- `int executeUpdate (String sql)`
- `boolean execute(String sql)`
- The `executeUpdate` method can execute actions such as **INSERT**, **UPDATE**, and **DELETE** as well as data definition commands such as **CREATE TABLE**, and **DROPTABLE**. The `executeUpdate` method returns a count of the rows that were affected by the SQL command.
- The `executeQuery` method is used to execute **SELECT** queries. The `executeQuery` object returns an object of type `ResultSet` that you use to walk through the result a row at a time.
- `ResultSet rs = stat.executeQuery("SELECT * FROM Books")`

# ► Using executeUpdate method

```
try {  
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");  
    Connection connection =  
        DriverManager.getConnection(url,userName,password);  
    Statement statement = connection.createStatement();  
    sqlCommand = "CREATE TABLE Books(Title CHAR(60),  
        ISBN CHAR(13),Price CURRENCY)";  
    statement.executeUpdate(sqlCommand);  
    sqlCommand = " INSERT INTO Books VALUES (  
        'Beyond HTML', '0-07-882198-3', 27.95)";  
    statement.executeUpdate(sqlCommand);  
    statement.close();  
    connection.close();  
} catch (Exception e) {  
    System.out.println( e.getMessage() );  
}
```

# ► Using executeQuery method

```
try {
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    Connection connection =
        DriverManager.getConnection(url,userName,password);
    Statement statement = connection.createStatement();
    sqlCommand ="SELECT * FROM BOOKS";
    ResultSet rs = statement.executeQuery(sqlCommand);
    while (rs.next()) {
        title = rs.getString(1);
        isbn = rs.getString(2);
        price = rs.getFloat("Price");
        System.out.println(title + " , " + isbn + " , " + price );
    }
    rs.close();
    .....
} catch (Exception e) {....}
```

# ► SQL data types and their corresponding Java language types

01-2004	SQL data type	Java data type
Khoa CNTT	INTEGER or INT	int
	SMALLINT	short
	NUMERIC( <i>m,n</i> ), DECIMAL( <i>m,n</i> ) or DEC( <i>m,n</i> )	java.sql.Numeric
	FLOAT( <i>n</i> )	double
	REAL	float
10/18	DOUBLE	double
	CHARACTER( <i>n</i> ) or CHAR( <i>n</i> )	String
	VARCHAR( <i>n</i> )	String
PHẠM VĂN TÍNH	BOOLEAN	Boolean
	DATE	java.sql.Date
	TIME	java.sql.Time



# ► Creating and Using Compiles SQL Statements (PreparedStatement)

- The **PreparedStatement**, the driver actually sends only the execution plan ID and the parameters to the DBMS. This results in less network traffic and is well-suited for Java applications on the Internet. The **PreparedStatement should be used when you need to execute the SQL statement many times in a Java application**. But remember, even though the optimized execution plan is available during the execution of a Java program, the DBMS discards the execution plan at the end of the program. So, the DBMS must go through all of the steps of creating an execution plan every time the program runs. **The PreparedStatement object achieves faster SQL execution performance than the simple Statement object**, as the DBMS does not have to run through the steps of creating the execution plan. .

- Notice that the **executeQuery()**, **executeUpdate()**, and **execute()** methods do not take any parameters.

<i>Return Type</i>	<i>Method Name</i>	<i>Parameter</i>
ResultSet	<b>executeQuery</b>	( )
int	<b>executeUpdate</b>	( )
Boolean	<b>execute</b>	( )

# ► Creating and Using Compiles SQL Statements (PreparedStatement)

- One of the major features of a `PreparedStatement` is that it can handle `IN` types of `parameters`. The parameters are indicated in a SQL statement by placing the `?` as the parameter marker instead of the actual values. In the Java program, the association is made to the parameters with the `setXXXX()` methods. All of the `setXXXX()` methods take the parameter index, which is `1` for the first "`?`," `2` for the second "`?`," and so on.
- `void setBoolean (int parameterIndex, boolean x)`
- `void setByte (int parameterIndex, byte x)`
- `void setDouble (int parameterIndex, double x)`
- `void setFloat (int parameterIndex, float x)`
- `void setInt (int parameterIndex, int x)`
- `void setLong (int parameterIndex, long x)`
- `void setNumeric (int parameterIndex, Numeric x)`
- `void setShort (int parameterIndex, short x)`
- `void setString (int parameterIndex, String x)`

# ► Creating and Using Compiled SQL Statements (PreparedStatement)

```
try {
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    Connection connection =
        DriverManager.getConnection(url,userName,password);
    PreparedStatement = connection.prepareStatement("SELECT * FROM
                                                BOOKS WHERE ISBN = ?");
    PreparedStatement.setString(1,"0-07-882198-3");
    ResultSet rs = PreparedStatement.executeQuery();
    while (rs.next()) {
        title = rs.getString(1);
        isbn = rs.getString(2);
        price = rs.getFloat("Price");
        System.out.println(title + ", " + isbn + " , " + price );
    }
    rs.close();
    PreparedStatement.close();
    connection.close();
}
```

# ▶ **java.sql.DriverManager**

- static Connection getConnection(String url, String user, String password)  
establishes a connection to the given database and returns a Connection object.

## ► ***java.sql.Connection***

- **Statement createStatement()**  
creates a statement object that can be used to execute SQL queries and updates without parameters.
- **PreparedStatement prepareStatement(String sql)**  
returns a **PreparedStatement** object containing the precompiled statement. The string **sql** contains a SQL statement that can contain one or more parameter placeholders denoted by **?** characters.
- **void close()**  
immediately closes the current connection.

# ► **java.sql.Statement**

- **ResultSet executeQuery(String sql)**  
executes the SQL statement given in the string and returns a **ResultSet** to view the query result.
- **int executeUpdate(String sql)**  
executes the SQL **INSERT**, **UPDATE**, or **DELETE** statement specified by the string. Also used to execute Data Definition Language (DDL) statements such as **CREATE TABLE**. Returns the number of records affected, or -1 for a statement without an update count.
- **boolean execute(String sql)**  
executes the SQL statement specified by the string. Returns **true** if the statement returns a result set, **false** otherwise. Use the **getResultSet** or **getUpdateCount** method to obtain the statement outcome.
- **int getUpdateCount()**  
Returns the number of records affected by the preceding update statement, or -1 if the preceding statement was a statement without an update count. Call this method only once per executed statement.
- **ResultSet getResultSet()**  
Returns the result set of the preceding query statement, or null if the preceding statement did not have a result set. Call this method only once per executed statement.

# ▶ **java.sql.ResultSet**

- **boolean next()**  
makes the current row in the result set move forward by one. Returns **false** after the last row. Note that you must call this method to advance to the first row.
- **Xxx getXxx(int columnNumber)**
- **Xxx getXxx(String columnName)**
- (**Xxx** is a type such as **int**, **double**, **String**, **Date**, etc.) return the value of the column with column index **columnNumber** or with column names, converted to the specified type. Not all type conversions are legal. See documentation for details.
- **int findColumn(String columnName)**  
gives the column index associated with a column name.
- **void close()**  
immediately closes the current result set.

# ► **java.sql.PreparedStatement**

- **void setXxx(int n, Xxx x)**  
(**Xxx** is a type such as **int**, **double**, **String**, **Date**, etc.) sets the value of the **n-th** parameter to **x**.
- **void clearParameters()**  
clears all current parameters in the prepared statement.
- **ResultSet executeQuery()**  
executes a prepared SQL query and returns a **ResultSet** object.
- **int executeUpdate()**  
executes the prepared SQL **INSERT**, **UPDATE**, or **DELETE** statement represented by the **PreparedStatement** object. Returns the number of rows affected, or 0 for DDL statements.



# ▶ LẬP TRÌNH MẠNG 1

**Thời gian học : 45 Tiết**

**GV: TS. Phạm Văn Tính**

**Khoa CNTT – Đại học Nông Lâm**

**Email: [pvtinh@hcmuaf.edu.vn](mailto:pvtinh@hcmuaf.edu.vn)**

**Web: [www.hcmuaf.edu.vn/pvtinh/](http://www.hcmuaf.edu.vn/pvtinh/)**

# ▶ **PART 1 – INPUT/OUTPUT STREAMS**

- **Streams concepts**
- **Input Streams**
- **Output Streams**
- **Pipe Stream**
- **Reader**
- **Writer**
- **Object Serialization**
- **Object Input Stream**
- **Object Output Stream**

# ▶ **PART 2 – SOCKET PROGRAMMING**

- **Sockets and Interprocess Communication**
- **IP Addresses**
- **Internet Addressing with Java**
- **TCP Connection and UDP Connection**
- **Ports and Standard Protocols**
- **Client/Server Programming**
- **TCP Programming**
- **UDP Programming**
- **URLs**
- **Some of the Standard Protocols:**
  - **HTTP**
  - **FTP**
  - **SMTP**
  - **POP3**

# ▶ **PART 3 – JAVA DATABASE CONNECTIVITY**

- **JDBC concepts**
- **JDBC Driver Types**
- **Steps in creating a JDBC application**
- **JDBC key components**
- **Basic JDBC Programming Concepts**
  - **Making the Connection**
  - **Executing the SQL commands**
  - **Thực hiện lệnh SQL**
  - **Processing the ResultSet**
  - **Transactions**
  - **Scrollable & Updatable Result Sets**

# ▶ **PART 4 – REMOTE METHOD INVOCATION**

- **Distributed Object & Java RMI**
- **Java RMI architecture - Stub/Skeleton Layer**
- **RMI Remote Classes**
- **RMI's Naming System**
- **Remote Invocations**
- **CallBack**

- **Advanced JAVA Networking, Addison Wesley, 2002**
- **Java Network Programming and Distributed Computing, Addison Wesley, 2002**
- **Java Network Programming, 2nd Edition, OReilly, 2002**
- **Core Java 2 – Volume II, SunPress, 2001**

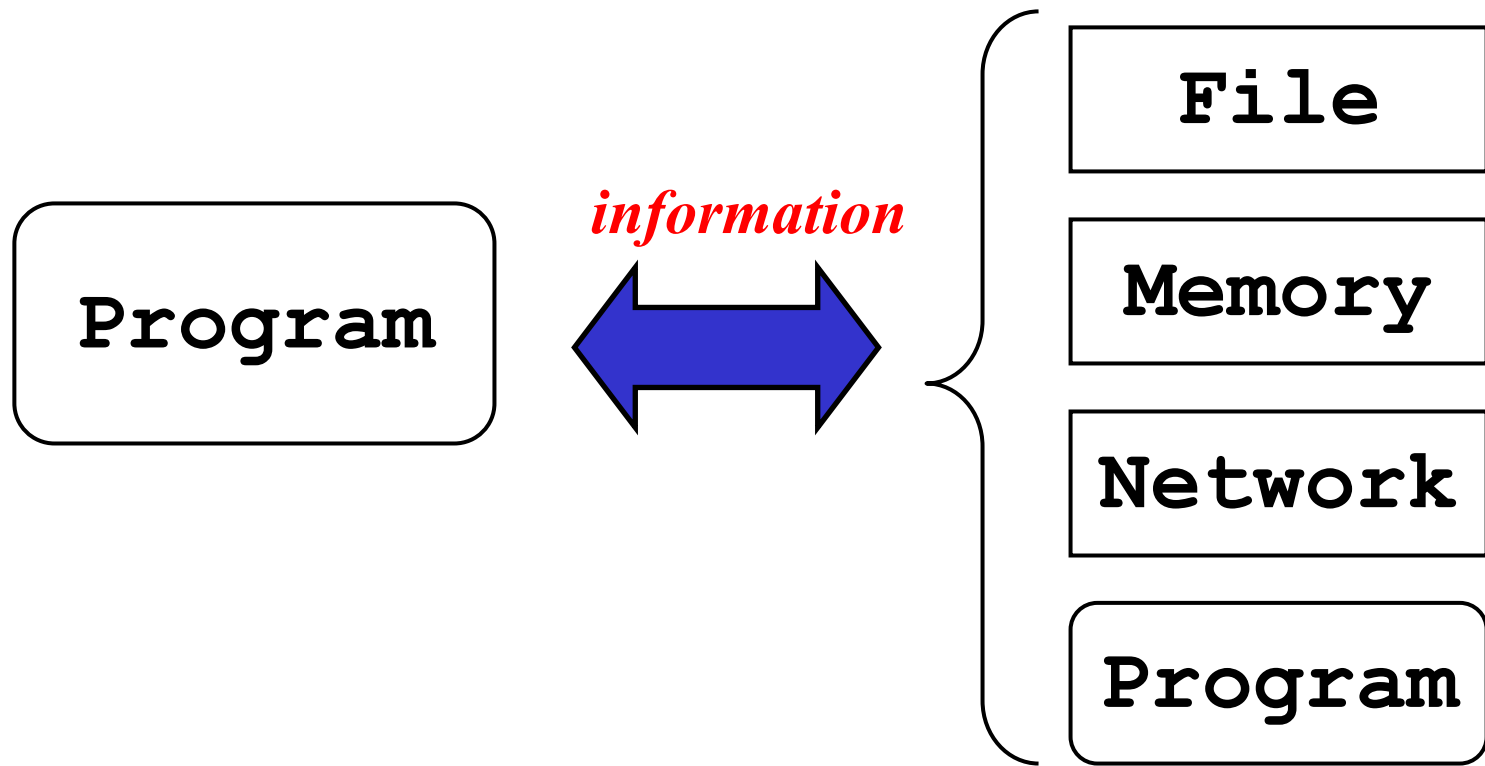
# ▶ **JAVA IO STREAMS**

# ▶ **PART 1 – INPUT/OUTPUT STREAMS**

- **Streams concepts**
- **Input Streams**
- **Output Streams**
- **Reader**
- **Writer**
- **Object Serialization**
- **Object Input Stream**
- **Object Output Stream**



# ▶ Stream concepts

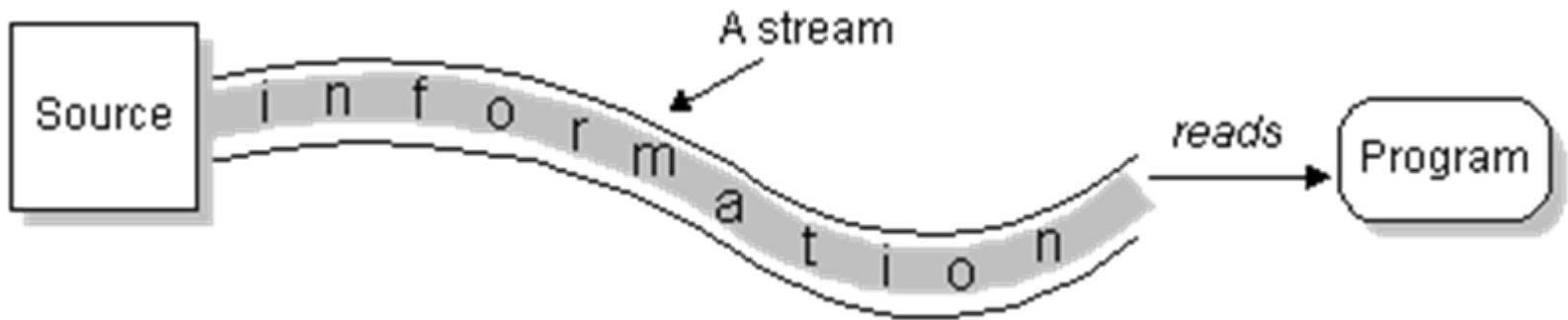


- **Data exchange**

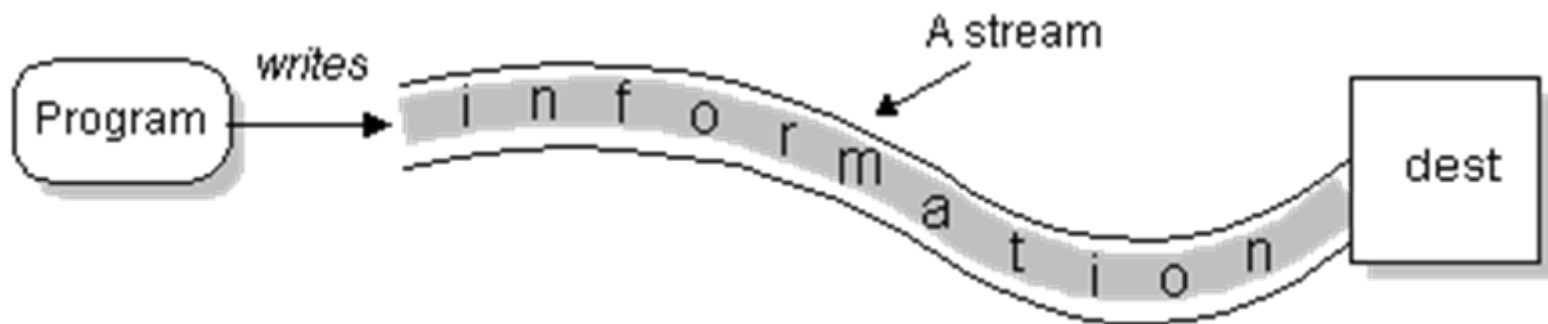
- **Data exchange type: Character, Object, voice, picture, audio, video...**

- **Stream:**
  - dòng thông tin giữa 2 tác nhân (mức cao)
  - một dãy tuần tự các byte (mức thấp)
- Một stream được gắn với một nguồn (source), hay một đích (destination)
- **Stream operations:**
  - open stream
  - close stream
  - read
  - write
  - seek
- **Input stream:** support reading functions
- **Output stream:** support writing functions
- **Filter stream:** buffer

# ► Input & Output Stream



Hình 1: *Chương trình xử lý dữ liệu từ 1 input stream*



Hình 2: *Chương trình ghi dữ liệu ra output stream*

# ▶ Trình tự đọc/ ghi dòng

```
open input stream
while (more information)
{
    read information
    process information
}
close input stream
```

Đọc thông tin  
từ input stream

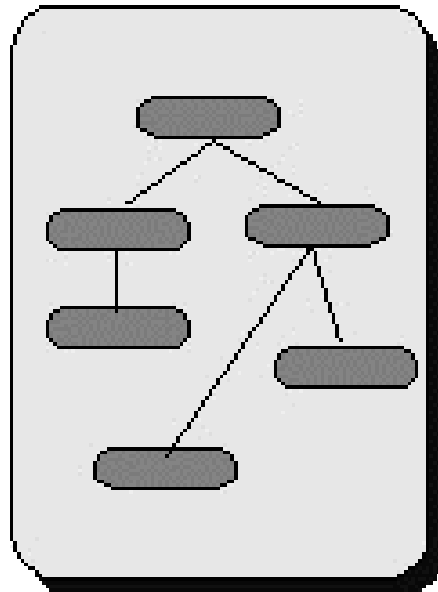
```
open output stream
while (more information)
{
    get information from ...
    write information
}
close output stream
```

Ghi thông tin  
vào output stream

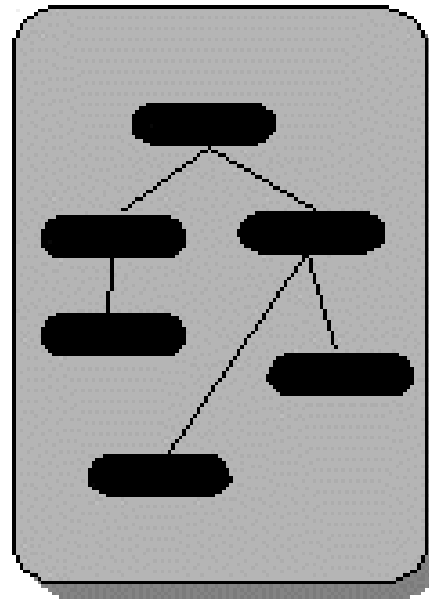
# ► Các loại stream trong package java.io

- các lớp trong gói **java.io** được thiết kế gồm 2 nhóm chính:
  - Nhóm **input/output** stream, hay nhóm hướng **byte**
  - Nhóm **reader/writer**, hay nhóm hướng ký tự (**unicode**)

Character Streams



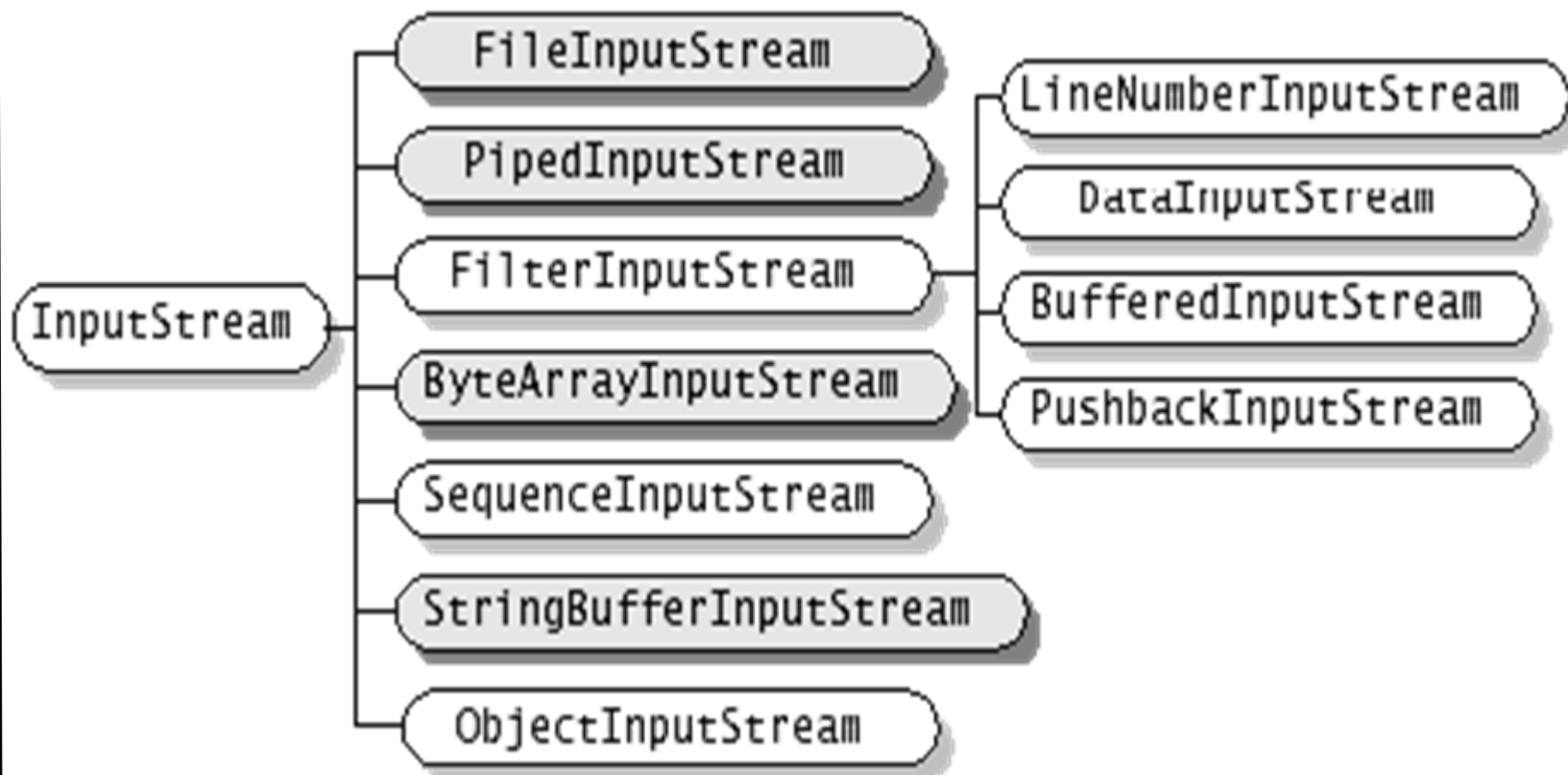
Byte Streams



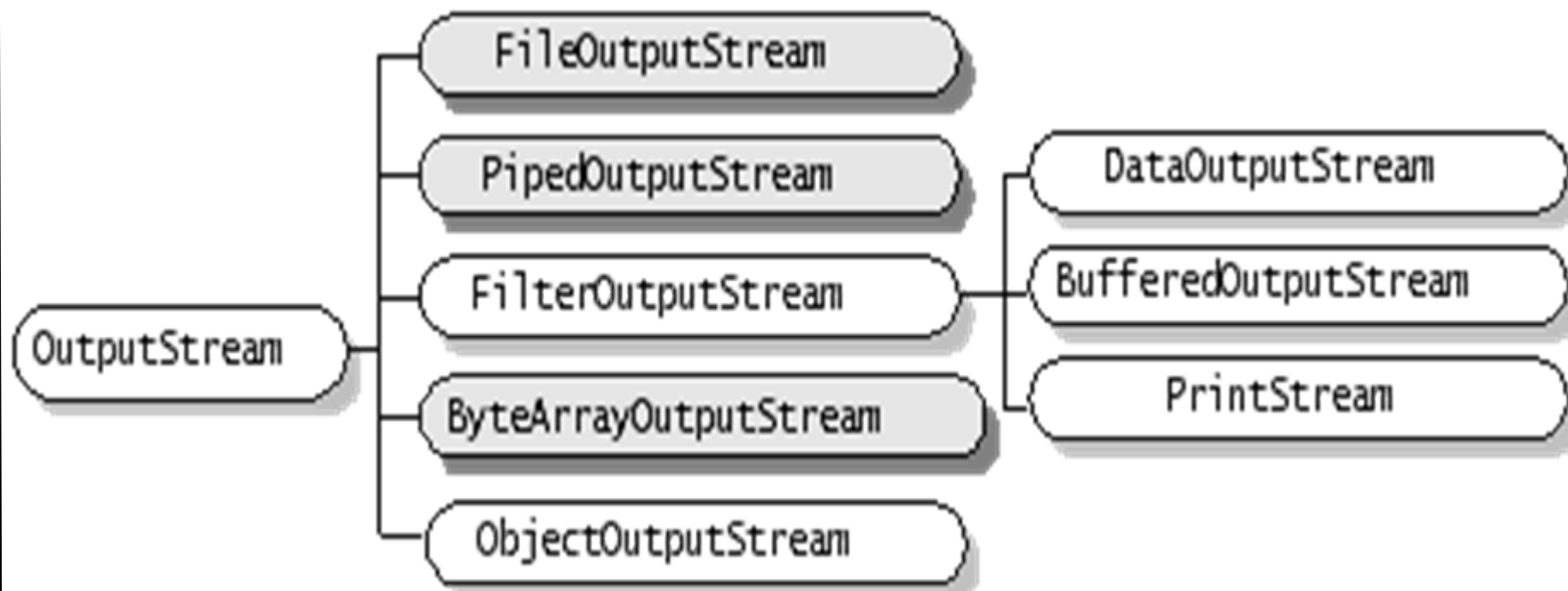
## ► Nhóm input/output stream

- được gọi là **nhóm hướng byte**, vì thao tác đọc/ghi áp dụng cho 1 hoặc nhiều byte
- chỉ giới hạn xử lý các byte 8 bits ISO-Latin-1.
- **rất thích hợp khi cần xử lý dữ liệu nhị phân như ảnh, âm thanh, binary files...**
- Các input stream được mở rộng từ lớp **InputStream**
- Các output stream được mở rộng từ lớp **OutputStream**

# ► Nhóm input stream



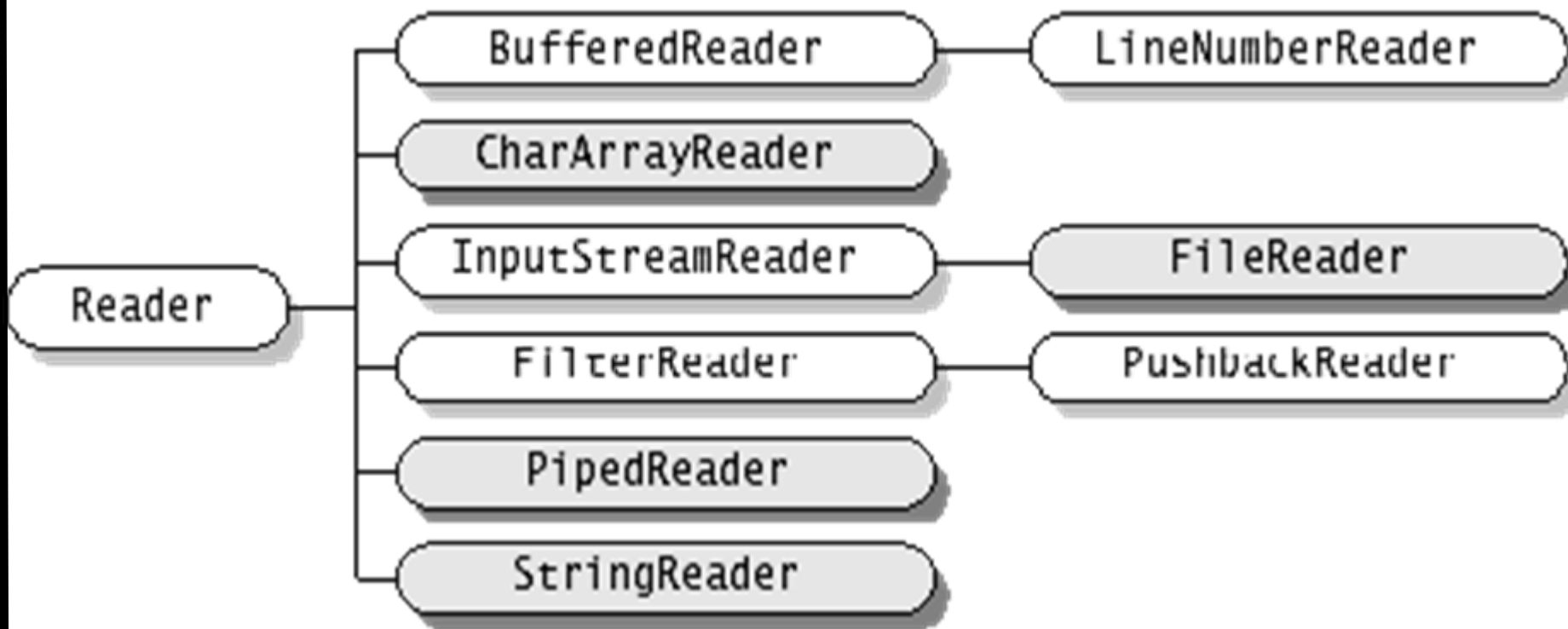
## ► Nhóm output stream



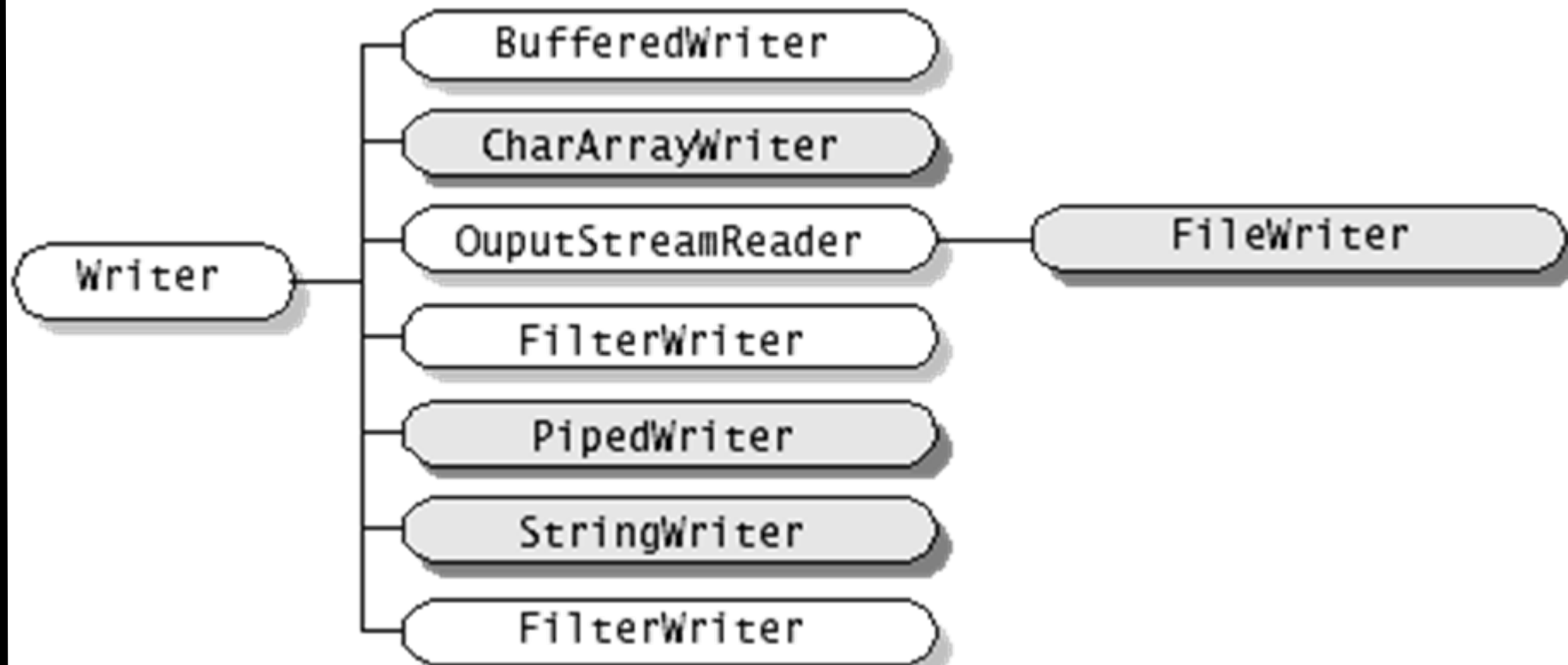


## ► Nhóm reader/writer

- được gọi là **nhóm hướng ký tự**, vì thao tác đọc/ghi áp dụng cho 1 hoặc nhiều ký tự **Unicode** (1 character = 2bytes)



# ► Nhóm Writer



## ► Các loại các stream (tt)

- java IO cũng cung cấp cách thức kết gắn stream với các loại tác nhân khác như bộ nhớ, file ...
- Các lớp `InputStreamReader` và `OutputStreamWriter` cung cấp sự chuyển đổi giữa `stream` và `reader/writer`
- Xem bảng phân loại tóm tắt để biết thêm chi tiết

# ► Tổng quát về các Streams

02-2004

Khoa CNTT

14/80

PHẠM VĂN TÍNH

I/O	Streams	Mô tả khái quát
Memory	<b>CharArrayReader</b> <b>CharArrayWriter</b> <b>ByteArrayInput-Stream</b> <b>ByteArrayOutput-Stream</b>	<b>Đọc/ghi từ/vào bộ nhớ.</b> <b>Tạo stream từ một mảng, tiếp theo dùng các phương thức đọc/ghi để đọc/ghi từ/vào mảng.</b> <b>Tác vụ đọc sẽ lấy dữ liệu từ mảng</b> <b>Tác vụ ghi sẽ ghi ra mảng</b>
	<b>StringReader</b> <b>StringWriter</b> <b>StringBuffer-InputStream</b>	<b>StringReader để đọc các ký tự từ một String trong bộ nhớ.</b> <b>StringWriter để ghi vào String.</b> <b>StringBufferInputStream tương tự như StringReader. Sự khác biệt chỉ là StringBufferInputStream đọc các bytes từ một đối tượng StringBuffer</b>

# ► Tổng quát về các Streams

02-2004

Khoa CNTT

15/80

PHẠM VĂN TÍNH

Pipe	<code>PipedReader</code> <code>PipedWriter</code> <code>PipedInputStream</code> <code>PipedOutputStream</code>	Hiện thực các thành phần input và output của một pipe. Pipes được dùng như một kênh truyền, nối output của một thread vào input của một thread khác.
File	<code>FileReader</code> <code>FileWriter</code> <code>FileInputStream</code> <code>FileOutputStream</code>	Được gọi là các file streams. File streams dùng để đọc/ghi từ/vào file trên file system.
Nối	<code>SequenceInput-Stream</code> <i>(concatenation)</i>	Nối nhiều input streams thành một input stream.

# ► Tổng quát về các Streams

02-2004

Khoa CNTT

16/80

PHẠM VĂN TÍNH

Object Serial- ization	<code>ObjectInputStream</code> <code>ObjectOutputStream</code>	Dùng khi cần lưu trữ, khôi phục, hoặc truyền toàn bộ đối tượng.
Chuyển đổi dạng dữ liệu ( <i>Data Conver- sion</i> )	<code>DataInputStream</code> <code>DataOutputStream</code>	Thuận tiện khi cần đọc/ghi các kiểu dữ liệu cơ bản ( <i>primitive data types</i> ) như <code>int</code> , <code>double</code> , ...
Counting	<code>LineNumberReader</code> <code>LineNumberInput- Stream</code>	Theo dõi số hàng trong khi đọc

# ► Tổng quát về các Streams

02-2004

Khoa CNTT

17/80

PHẠM VĂN TÍNH

Printing	<code>PrintWriter</code> <code>PrintStream</code>	Rất thuận tiện khi cần kết xuất, dễ đọc với người. <code>System.out</code> là một đối tượng thuộc lớp <code>PrintStream</code> .
Đệm ( <i>Buffering</i> )	<code>BufferedReader</code> <code>BufferedWriter</code> <code>bufferedInputStream</code> <code>BufferedOutputStream</code>	Đệm dữ liệu trong các thao tác đọc/ghi. Đệm dữ liệu cải thiện tốc độ đọc ghi vì giảm số lần truy xuất thiết bị.
Lọc dữ liệu ( <i>Filtering</i> )	<code>FilterReader</code> <code>FilterWriter</code> <code>FilterInputStream</code> <code>FilterOutputStream</code>	Các lớp abstract này định nghĩa các giao tiếp cho các filter streams lọc dữ liệu trong khi đọc/ghi.

# ► Tổng quát về các Streams

02-2004

Khoa CNTT

18/80

PHẠM VĂN TÍNH

Chuyển đổi byte ⇔ ký tự

(*Converting between Bytes and Characters*)

InputStreamReader  
OutputStreamWriter

Cặp reader/writer này là cầu nối giữa các byte streams và character streams.

Một **InputStreamReader** đọc các bytes từ một **InputStream** và chuyển các bytes đó thành các ký tự.

Một **OutputStreamWriter** chuyển các ký tự sang các bytes, và ghi các bytes đó vào một **OutputStream**.

Quá trình chuyển đổi sẽ sử dụng bộ mã mặc định nếu không được chỉ định rõ.

Gọi

**System.getProperty("file.encoding")** để lấy về tên bộ mã mặc định.



# ► Các lớp IO

- **InputStream, OutputStream, Reader và Writer** là các lớp abstract:
  - Các lớp input stream được mở rộng từ lớp **InputStream**
  - Các lớp reader được mở rộng từ lớp **Reader**
  - Các lớp output stream được mở rộng từ lớp **OutputStream**
  - Các lớp writer được mở rộng từ lớp **Writer**
- 2 lớp **InputStream** và **Reader** cung cấp những phương thức **read** tương đối giống nhau.
- 2 lớp **OutputStream** và **Writer** cung cấp những phương thức **write** tương đối giống nhau.

# ► InputStream

- |   | <b>Low-Level Input Stream</b>  | <b>Purpose of Stream</b>  |
|---|--------------------------------|---|
| • | <b>ByteArrayInputStream</b>    | Reads bytes of data from an in-memory array   |
| • | <b>FileInputStream</b>         | Reads bytes of data from a file on the local file system  |
| • | <b>PipedInputStream</b>        | Reads bytes of data from a thread pipe  |
| • | <b>StringBufferInputStream</b> | Reads bytes of data from a string   |
| • | <b>SequenceInputStream</b>     | Reads bytes of data from two or more low-level streams, switching from one stream to the next when the end of the stream is reached |
| • | <b>System.in</b>               | Reads bytes of data from the user console   |

# ► The `java.io.InputStream` Class

- **`int available()`** throws `java.io.IOException`— returns the number of bytes currently available for reading.
- **`void close()`** throws `java.io.IOException`— closes the input stream and frees any resources (such as file handles or file locks) associated with the input stream.
- **`int read()`** throws `java.io.IOException`— returns the next byte of data from the stream. When the end of the stream is reached, a value of `-1` is returned.
- **`int read(byte[] byteArray)`** throws `java.io.IOException`— reads a sequence of bytes and places them in the specified byte array. This method returns the number of bytes successfully read, or `-1` if the end of the stream has been reached.
- **`int read(byte[] byteArray, int offset, int length)`** throws `java.io.IOException`, `java.lang.IndexOutOfBoundsException`— reads a sequence of bytes, placing them in the specified array at the specified offset, and for the specified length, if possible.

# ► The `java.io.InputStream` Class

- **`long skip(long amount)`** throws `java.io.IOException`— reads, but ignores, the specified amount of bytes. These bytes are discarded, and the position of the input stream is updated. The skip method returns the number of bytes skipped over, which may be less than the requested amount.
- The following code fragment reads 10 bytes from the `InputStream` **`in`** and stores them in the byte array **`input`**. However, if end of stream is detected, the loop is terminated early:

```
byte[] input = new byte[10];  
for (int i = 0; i < input.length; i++) {  
    int b = in.read( );  
    if (b == -1) break;  
    input[i] = (byte) b;  
}
```

# ▶ The java.io.InputStream Class

- For example, you may try to **read 1 024 bytes** from a network connection, when **only 512 have actually** arrived from the server. The rest are still in transit. They'll arrive eventually, but they aren't available now.

```
byte[] input = new byte[1024];  
int bytesRead = in.read(input);
```

- It attempts to **read 1 024 bytes** from the InputStream into the array input. However, if **only 512 bytes are available, then bytesRead will be set to 512**. To guarantee that all the bytes you want are actually read, you must place the read in a loop that reads repeatedly until the array is filled.

```
int bytesRead = 0;  
int bytesToRead = 1024;  
byte[] input = new byte[bytesToRead];  
while (bytesRead < bytesToRead) {  
    bytesRead += in.read(input, bytesRead, bytesToRead  
        - bytesRead);  
}
```

# ► The java.io.File Class

- An abstract representation of file and directory pathnames.
- For **UNIX** platforms, the prefix of an **absolute pathname** is always **"/"**. **Relative pathnames** have **no prefix**.
- For **Microsoft Windows** platforms, the prefix of a pathname that contains a drive specifier consists of the **drive letter** followed by **":"** and possibly followed by **"\"** if the pathname is absolute (**D:\\myfolder\\t.txt**). A relative pathname that does not specify a drive has no prefix.
- **public File(File parent, String child)**
  - Creates a new File instance from a parent abstract pathname and a child pathname string.
- **public File(String parent, String child)**
  - Creates a new File instance from a parent pathname string and a child pathname string.

# ► The `java.io.File` Class

- **`public File(String pathname)`**
  - Creates a new `File` instance by converting the given `pathname` string into an abstract `pathname`. If the given string is the empty string, then the result is the empty abstract `pathname`.
- **`public String getPath()`**
  - Converts this abstract `pathname` into a `pathname` string.
- **`public boolean isAbsolute()`**
  - Tests whether this abstract `pathname` is absolute.
- **`public String getAbsolutePath()`**
  - Returns the absolute `pathname` string of this abstract `pathname`.
- **`public boolean canRead()`**
  - Tests whether the application can read the file denoted by this abstract `pathname`.

# ► The java.io.File Class

- **public boolean canWrite()**
  - Tests whether the application can modify the file denoted by this abstract pathname
- **public boolean exists()**
  - Tests whether the file or directory denoted by this abstract pathname exists.
- **public boolean isDirectory()**
  - Tests whether the file denoted by this abstract pathname is a directory.
- **public boolean isFile()**
  - Tests whether the file denoted by this abstract pathname is a normal file.
- **public boolean isHidden()**
  - Tests whether the file named by this abstract pathname is a hidden file.



# ► The **java.io.File** Class

- **public long length()**
  - Returns the length (the size in Kbyte) of the file denoted by this abstract pathname. The return value is unspecified if this pathname denotes a directory.
- **public boolean delete()**
  - Deletes the file or directory denoted by this abstract pathname. If this pathname denotes a directory, then the directory must be empty in order to be deleted.
- **public String[] list()**
  - Returns an array of strings naming the files and directories in the directory denoted by this abstract pathname.
- **public String[] list(FileNameFilter filter)**
  - Returns an array of strings naming the files and directories in the directory denoted by this abstract pathname that satisfy the specified filter. The behavior of this method is the same as that of the list() method, except that the strings in the returned array must satisfy the filter. If the given filter is null then all names are accepted.

# ► The `java.io.File` Class

- **`public File[] listFiles()`**
  - Returns an array of abstract pathnames denoting the files in the directory denoted by this abstract pathname.
- **`public File[] listFiles(FilenameFilter filter)`**
  - Returns an array of abstract pathnames denoting the files and directories in the directory denoted by this abstract pathname that satisfy the specified filter.
- **`public boolean mkdir()`**
  - Creates the directory named by this abstract pathname.
- **`public boolean mkdirs()`**
  - Creates the directory named by this abstract pathname, including any necessary but nonexistent parent directories. Note that if this operation fails it may have succeeded in creating some of the necessary parent directories.

# ► The java.io.File Class

//List of all files in D: with extension tgz

```
import java.io.*;
import java.util.*;
public class DLister {
    public static void main(String[] args) {
        File path = new File("D:\\");
        String[] list;
        list = path.list(new DirFilter(".tgz"));
        for(int i = 0; i < list.length; i++)
            System.out.println(list[i]);
    }
}

class DirFilter implements FilenameFilter {
    String afn;
    DirFilter(String afn) { this.afn = afn; }
    public boolean accept(File dir, String name) {
        String f = new File(name).getName();
        return f.indexOf(afn) != -1;
    }
}
```

# ► The java.io.File Class

//Directory operations

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

```
public class MakeDirectories {
```

```
private final static String usage =
```

```
"Usage: MakeDirectories path1 ...\n" +
```

```
"Creates each path\n" +
```

```
"Usage: MakeDirectories -d path1 ...\n" +
```

```
"Deletes each path\n" +
```

```
"Usage: MakeDirectories -r path1 path2\n" +
```

```
"Renames from path1 to path2\n";
```

```
private static void usage() {
```

```
    System.err.println(usage);
```

```
    System.exit(1);
```

```
}
```

# ► The java.io.File Class

```
private static void fileData(File f) {  
    System.out.println(  
        "Absolute path: " + f.getAbsolutePath() +  
        "\n Can read: " + f.canRead() +  
        "\n Can write: " + f.canWrite() +  
        "\n getName: " + f.getName() +  
        "\n getParent: " + f.getParent() +  
        "\n getPath: " + f.getPath() +  
        "\n length: " + f.length() +  
        "\n lastModified: " + f.lastModified());  
    If(f.isFile()) System.out.println("it's a file");  
    else if(f.isDirectory())  
        System.out.println("it's a directory");  
}
```

# ► The java.io.File Class

```
public static void main(String[] args) {
    if(args.length < 1) usage();
    if(args[0].equals("-r")) {
        if(args.length != 3) usage();
        File old = new File(args[1]), rname = new File(args[2]);
        old.renameTo(rname);
        fileData(old);
        fileData(rname);
        return; // Exit main
    }
    int count = 0;
    boolean del = false;
    if(args[0].equals("-d")) {
        count++;
        del = true;
    }
}
```

# ► The java.io.File Class

```
for( ; count < args.length; count++) {  
    File f = new File(args[count]);  
    if(f.exists()) {  
        System.out.println(f + " exists");  
        if(del) {  
            System.out.println("deleting..." + f);  
            f.delete();  
        }  
    } else { // Doesn't exist  
        if(!del) {  
            f.mkdirs();  
            System.out.println("created " + f);  
        }  
    }  
    fileData(f);  
}  
}}
```

# ► The **java.io.FileInputStream** Class

- A **FileInputStream** obtains input bytes from a file in a file system. What files are available depends on the host environment.
- **FileInputStream** is meant for reading streams of raw bytes such as image data. For reading streams of characters, consider using **FileReader**.
- **public FileInputStream(File file) throws FileNotFoundException**
  - Creates a **FileInputStream** by opening a connection to an actual file, the file named by the **File** object **file** in the file system. A new **FileDescriptor** object is created to represent this file connection.
- **public FileInputStream(String name) throws FileNotFoundException**
  - Creates a **FileInputStream** by opening a connection to an actual file, the file named by the path name **name** in the file system.



# ► The `java.io.FileInputStream` Class

- **`public int read()` throws `IOException`**
  - Reads a byte of data from this input stream. This method blocks if no input is yet available.
  - Returns: the next byte of data, or -1 if the end of the file is reached.
- **`public int read(byte[] b)` throws `IOException`  
`public int read(byte[] b, int off, int len)`**
  - Reads up to `b.length` bytes of data from this input stream into an array of bytes. This method blocks until some input is available.
  - **Parameters:**
    - **`b`** - the buffer into which the data is read.
    - **`off`** - the start offset of the data.
    - **`len`** - the maximum number of bytes read.
  - **Returns:** the total number of bytes read into the buffer, or -1 if there is no more data because the end of the file has been reached.

# ▶ The `java.io.FileInputStream` Class

- **public long skip(long n) throws IOException**
  - Skips over and discards n bytes of data from the input stream.
  - **Parameters:** n - the number of bytes to be skipped.
  - **Returns:** the actual number of bytes skipped.
  - **Throws:** IOException - if n is negative, or if an I/O error occurs.
- **public int available() throws IOException**
  - Returns the number of bytes that can be read from this file input stream without blocking.
- **public void close() throws IOException**
  - Closes this file input stream and releases any system resources associated with the stream.

# ► FileInputStreamDemo

- Below we examine a practical application of using a low-level input stream to display the contents of a file. A byte at a time is read from the file and displayed to the screen.

```
// Create an input stream, reading from the specified file
InputStream fileInput = new FileInputStream ( args[0] );
// Read the first byte of data
int data = fileInput.read();
// Repeat : until end of file (EOF) reached
while (data != -1){
    // Send byte to standard output
    System.out.write ( data );
    // Read next byte
    data = fileInput.read();
}
```

# ► FileInputStreamDemo

```
import java.io.*;
public class FileInputStreamDemo{
    public static void main(String args[]){
        if (args.length != 1){
            System.err.println ("Syntax - FileInputStreamDemo file");
            return;
        }
        try{
            InputStream fileInput = new FileInputStream( args[0] );
            int data = fileInput.read();
            while (data != -1){
                System.out.write ( data );
                data = fileInput.read();
            }
            // Close the file
            fileInput.close();
        }
        catch (IOException ioe){
            System.err.println ("I/O error - " + ioe);
        }
    }
}
```

# ► Output Streams

- **Low-Level Output Stream Purpose of Stream**
- **ByteArrayOutputStream** Writes bytes of data to an array of bytes.
- **FileOutputStream** Writes bytes of data to a local file.
- **PipedOutputStream** Writes bytes of data to a communications pipe, which will be connected to a `java.io.PipedInputStream`.
- **StringBufferOutputStream** Writes bytes to a string buffer (a substitute data structure for the fixed-length string).
- **System.err** Writes bytes of data to the error stream of the user console, also known as standard error. In addition, this stream is cast to a `PrintStream`.
- **System.out** Writes bytes of data to the user console, also known as standard output. In addition, this stream is cast to a `PrintStream`.

# ► The `java.io.OutputStream` Class

- **`void close()` throws `java.io.IOException`**— closes the output stream, notifying the other side that the stream has ended. Pending data that has not yet been sent will be sent, but no more data will be delivered.
- **`void flush()` throws `java.io.IOException`**— performs a "flush" of any unsent data and sends it to the recipient of the output stream. To improve performance, streams will often be buffered, so data remains unsent. This is useful at times, but obstructive at others. The method is particularly important for `OutputStream` subclasses that represent network operations, as flushing should always occur after a request or response is sent so that the remote side isn't left waiting for data.

# ► The `java.io.OutputStream` Class

- **`void write(int byte)` throws `java.io.IOException`**— writes the specified byte. This is an abstract method, overridden by `OutputStream` subclasses.
- **`void write(byte[] byteArray)` throws `java.io.IOException`**— writes the contents of the byte array to the output stream. The entire contents of the array (barring any error) will be written.
- **`void write(byte[] byteArray, int offset, int length)` throws `java.io.IOException`**— writes the contents of a subset of the byte array to the output stream. This method allows developers to specify just how much of an array is sent, and which part, as opposed to the `OutputStream.write(byte[] byteArray)` method, which sends the entire contents of an array.

# ► **Java.io.FileOutputStream**

- **public FileOutputStream(String name) throws FileNotFoundException**
  - Creates an output file stream to write to the file with the specified name.
  - Throws: **FileNotFoundException** - if the file exists but is a directory rather than a regular file, does not exist but cannot be created, or cannot be opened for any other reason
- **public FileOutputStream(String name, boolean append) throws FileNotFoundException**
  - Creates an output file stream to write to the file with the specified name. If the second argument is true, then bytes will be written to the end of the file rather than the beginning.
  - If the file exists but is a directory rather than a regular file, does not exist but cannot be created, or cannot be opened for any other reason then a **FileNotFoundException** is thrown.



# ► Java.io.FileOutputStream

- **public FileOutputStream(File file) throws FileNotFoundException**
  - Creates a file output stream to write to the file represented by the specified File object
- **public FileOutputStream(File file, boolean append) throws FileNotFoundException**
- **public void write(int b) throws IOException**
  - Writes the specified byte to this file output stream. Implements the write method of OutputStream.
- **public void write(byte[] b) throws IOException**
  - Writes b.length bytes from the specified byte array to this file output stream.
- **public void write(byte[] b, int off, int len) throws IOException**
  - Writes len bytes from the specified byte array starting at offset off to this file output stream.
- **public void close() throws IOException**
  - Closes this file output stream and releases any system resources associated with this stream. This file output stream may no longer be used for writing bytes.

# ► **FileOutputStream Demo (FileCopy)**

- The program copies a file by reading the contents of the file and writing it, one byte at a time, to a new file.
- To open a file for writing, a `FileOutputStream` is used. This class will create a file, if one does not already exist, or override the contents of the file (unless opened in append mode).

`// Output file for output`

`OutputStream output = new FileOutputStream(destination);`

- Once opened, it can be written to by invoking the `OutputStream.write()` method. This method is called repeatedly by the application, to write the contents of a file that it is reading.

`while ( data != -1){`

`// Write byte of data to our file`

`output.write (data);`

`// Read next byte`

`data=input.read();`

`}`

# ► **FileOutputStream Demo (FileCopy)**

```
import java.io.*;
public class FileOutputStreamDemo{
public static void main(String args[]){
    // Two parameters are required, the source and destination
    if (args.length != 2){
        System.err.println("Syntax - FileOutputStreamDemo src dest");
        return;
    }
    String source = args[0];
    String destination = args[1];
    try{
        // Open source file for input
        InputStream input = new FileInputStream( source );
        System.out.println ("Opened " +source + " for reading.");
        // Output file for output
        OutputStream output = new FileOutputStream(destination);
        System.out.println ("Opened " +destination + " for writing.");
    }
}
```

# ► FileOutputStream Demo

```
int data = input.read();
while ( data != -1){
    // Write byte of data to our file
    output.write (data);
    // Read next byte
    data=input.read();
}
// Close both streams
input.close();
output.close();
System.out.println ("I/O streams closed");
}
catch (IOException ioe){
    System.err.println ("I/O error - " + ioe);
}}}
```

# ► Filter Streams

- Filter streams add additional functionality to an existing stream, by processing data in some form (such as buffering for performance) or offering additional methods that allow data to be accessed in a different manner (for example, reading a line of text rather than a sequence of bytes).
- Filters make life easier for programmers, as they can work with familiar constructs such as strings, lines of text, and numbers, rather than individual bytes.
- Filter streams can be connected to any other stream, to a low-level stream or even another filter stream. Filter streams are extended from the **java.io.FilterInputStream** and **java.io.FilterOutputStream** classes.

# ► Filter Streams

- For example, suppose you wanted to connect a **PrintStream** (used to print text to an **OutputStream** subclass) to a stream that wrote to a file. The following code may be used to connect the filter stream and write a message using the new filter.
- `FileOutputStream fout = new FileOutputStream ( somefile );`  
`PrintStream pout = new PrintStream (fout);`  
`pout.println ("hello world");`
- This process is fairly simple as long as the programmer remembers two things:
  1. **Read and write operations must take place on the new filter stream.**
  2. **Read and write operations on the underlying stream can still take place, but not at the same time as an operation on the filter stream.**

# ► Filter Input Streams

•Filter Input Stream	Purpose of Stream
• <b>BufferedInputStream</b>	Buffers access to data, to improve efficiency.
• <b>DataInputStream</b>	Reads primitive data types, such as an int, a float, a double, or even a line of text, from an input stream.
• <b>LineNumberInputStream</b>	Maintains a count of which line is being read, based on interpretation of end-of-line characters. Handles both Unix and Windows end-of-line sequences.
• <b>PushBackInputStream</b>	Allows a byte of data to be pushed into the head of the stream.

# ► BufferedInputStream

- **BufferedInputStream (InputStream input)**— creates a buffered stream that will read from the specified InputStream object.
- **BufferedInputStream (InputStream input, int bufferSize)** throws java.lang.IllegalArgumentException— creates a buffered stream, of the specified size, which reads from the InputStream object passed as a parameter.
- BufferedInputStream does not declare any new methods of its own. It only overrides methods from InputStream.
- **Chaining Filters Together**  
DataOutputStream dout = **new DataOutputStream(  
    new BufferedOutputStream(  
        new FileOutputStream("data.txt")));**  
DataInputStream din = **new DataInputStream(  
    new BufferedInputStream(  
        new FileInputStream("data.txt")));**



# ► FileCopy using BufferedInputStream

```
public class BufferCopyFile {
    public static void main(String[] args) throws IOException {
        File inputFile = new File("DV.pdf");
        File outputFile = new File("DVcopy.pdf");
        BufferedInputStream in = new BufferedInputStream(new
                                                    FileInputStream(inputFile));
        BufferedOutputStream out = new BufferedOutputStream(new
                                                    FileOutputStream(outputFile));

        int c;
        Stopwatch stopwatch = new Stopwatch();
        stopwatch.start();
        while ((c = in.read()) != -1) out.write(c);
        stopwatch.stop();
        System.out.println("With buffering: " + stopwatch );
        in.close();
        out.close();
    }
}
```

# ► A Speed of new FileCopy

- Example runs of the following BufferCopyFile, with text files of various sizes, shows gains of ~3x. (In *Java Platform Performance* by Wilson and Kesselman, an example using a 370K JPEG file has a gain in execution speed of 83x!)
- **Size - 624 bytes :**  
With buffering: 10 ms  
Without buffering: 30 ms
- **Size - 10,610 bytes :**  
With buffering: 30 ms  
Without buffering: 80 ms
- **Size - 742,702 bytes :**  
With buffering: 180 ms  
Without buffering: 741 ms

# ► **DataInputStream Class**

- The **DataInputStream** and **DataOutputStream** classes provide methods for reading and writing Java's primitive data types and strings in a binary format. The binary formats used are primarily intended for exchanging data between two different Java programs whether through a network connection, a data file, a pipe, or some other intermediary. What a data output stream writes, a data input stream can read.
- **Constructors**
- **DataInputStream (InputStream in)**— creates a data input stream, reading from the specified input stream.

# ► **DataInputStream Class**

- **public final int read(byte[] b) throws IOException**
  - Reads some number of bytes from the contained input stream and stores them into the buffer array b. This method blocks until input data is available, end of file is detected, or an exception is thrown.
- **public final int read(byte[] b, int off, int len) throws IOException**
  - Reads up to len bytes of data from the contained input stream into an array of bytes. This method blocks until input data is available, end of file is detected, or an exception is thrown.
- **public final int skipBytes(int n)**
- **public final boolean readBoolean()**
- **public final byte readByte() : signed 8-bit byte**
- **public final int readUnsignedByte() : an unsigned 8-bit number**

# ► **DataInputStream Class**

- **public final short readShort()** : a signed 16-bit number
- **public final int readUnsignedShort()** : an unsigned 16-bit integer
- **public final char readChar()** : 2 bytes of this input stream as a Unicode character
- **public final int readInt()** : 4 bytes of this input stream, interpreted as an int
- **public final long readLong()** : eight bytes of this input stream, interpreted as a long
- **public final float readFloat()** : 4 bytes of this input stream, interpreted as a float.
- **public final double readDouble()** : 8 bytes of this input stream, interpreted as a double

## ► **PrintStream Class**

- The PrintStream class is the first filter output stream most programmers encounter because **System.out is a PrintStream**. However, other output streams can also be chained to print streams, using these two constructors:
- **public PrintStream(OutputStream out)**  
**public PrintStream(OutputStream out, boolean autoFlush)**
- By default, print streams should be explicitly flushed. However, **if the autoFlush argument is true, then the stream will be flushed every time** a byte array or linefeed is written or a `println( )` method is invoked.

# ► **PrintStream Class**

- **void print(boolean value)**— prints a boolean value.
- **void print(char character)**— prints a character value.
- **void print(char[] charArray)**— prints an array of characters.
- **void print(double doubleValue)**— prints a double value.
- **void print(float floatValue)**— prints a float value.
- **void print(int intValue)**— prints an int value.
- **void print(long longValue)**— prints a long value.
- **void print(Object obj)**— prints the value of the specified object's toString() method.
- **void print(String string)**— prints a string's contents.
- **void println()**— sends a line separator (such as '\n'). This value is system dependent and determined by the value of the system property "line.separator."

# ► Random Access File Stream

- **Lớp RandomAccessFile:**
  - Cung cấp cách thức đọc/ghi dữ liệu từ/ra file
  - cung cấp thêm thao tác seek → vị trí đọc/ghi là bất kỳ (random access)
- **một random access file chứa 1 file pointer chỉ đến vị trí sẽ được truy xuất:**
  - phương thức seek di chuyển file pointer đến vị trí bất kỳ
  - phương thức getFilePointer trả về vị trí hiện tại của file pointer



# ▶ Random Access File Stream

02-2004

Khoa CNTT

59/80

PHẠM VĂN TÍNH

<<Interface>>  
DataOutput  
(from io)

✦ write()

✦ write()

✦ write()

✦ writeBoolean()

✦ writeByte()

✦ writeShort()

✦ writeChar()

✦ writeInt()

✦ writeLong()

✦ writeFloat()

✦ writeDouble()

✦ writeBytes()

✦ writeChars()

✦ writeUTF()

RandomAccessFile  
(from io)

<<Interface>>  
DataInput  
(from io)

✦ readFully()

✦ readFully()

✦ skipBytes()

✦ readBoolean()

✦ readByte()

✦ readUnsignedByte()

✦ readShort()

✦ readUnsignedShort()

✦ readChar()

✦ readInt()

✦ readLong()

✦ readFloat()

✦ readDouble()

✦ readLine()

✦ readUTF()

# ► Java.io.RandomAccessFile

- A random access file behaves like a large array of bytes stored in the file system. There is a kind of cursor, or index into the implied array, called the *file pointer*; input operations read bytes starting at the file pointer and advance the file pointer past the bytes read
- The file pointer can be read by the `getFilePointer` method and set by the `seek` method
- `public RandomAccessFile(String name, String mode)`  
`public RandomAccessFile(File file, String mode)`
  - Creates a random access file stream to read from, and optionally to write to, a file with the specified name.
  - `"r"` Open for reading only.
  - `"rw"` Open for reading and writing. If the file does not already exist then an attempt will be made to create it.
  - `"rws"` Open for reading and writing, as with `"rw"`, and also require that every update to the file's content or metadata be written synchronously to the underlying storage device.
  - `"rwd"` Open for reading and writing, as with `"rw"`, and also require that every update to the file's content be written synchronously to the underlying storage device.

# ► Java.io.RandomAccessFile

- **public int read() throws IOException**

- Reads a byte of data from this file. The byte is returned as an integer in the range 0 to 255 (0x00-0x0ff). This method blocks if no input is yet available.

- **public int read(byte[] b, int off, int len) throws IOException**

- Reads up to len bytes of data from this file into an array of bytes. This method blocks until at least one byte of input is available.

- **public int read(byte[] b) throws IOException**

- Reads up to b.length bytes of data from this file into an array of bytes. This method blocks until at least one byte of input is available.

- **public long getFilePointer() throws IOException**

- Returns the current offset in this file.

# ► Java.io.RandomAccessFile

- **public void write(int b) throws IOException**
  - Writes the specified byte to this file. The write starts at the current file pointer.
- **public void write(byte[] b) throws IOException**
  - Writes b.length bytes from the specified byte array to this file, starting at the current file pointer.
- **public void write(byte[] b, int off, int len) throws ...**
  - Writes len bytes from the specified byte array starting at offset off to this file
- **public void seek(long pos) throws IOException**
  - Sets the file-pointer offset, measured from the beginning of this file, at which the next read or write occurs. The offset may be set beyond the end of the file. Setting the offset beyond the end of the file does not change the file length. The file length will change only by writing after the offset has been set beyond the end of the file.

# ► Java.io.RandomAccessFile

- **public long length() throws IOException**
  - Returns the length of this file.
- **public void setLength(long newLength) throws ....**
  - Sets the length of this file. If the present length of the file is greater than the newLength argument then the file will be truncated. In this case, if the file offset as returned by the getFilePointer method is greater than newLength then after this method returns the offset will be equal to newLength.
  - If the present length of the file as returned by the length method is smaller than the newLength argument then the file will be extended. In this case, the contents of the extended portion of the file are not defined.
- **public void close() throws IOException**
  - Closes this random access file stream and releases any system resources associated with the stream

# ► **Java.io.RandomAccessFile**

- **public final boolean readBoolean()**
- **public final byte readByte()**
- **public final int readUnsignedByte()**
- **public final short readShort()**
- **public final int readUnsignedShort()**
- **public final char readChar()**
- **public final int readInt()**
- **public final long readLong()**
- **public final float readFloat()**
- **public final double readDouble()**
- **public final String readLine()**

# ► Java.io.RandomAccessFile

- `public final void writeBoolean(boolean v)`
- `public final void writeByte(int v)`
- `public final void writeShort(int v)`
- `public final void writeChar(int v)`
- `public final void writeInt(int v)`
- `public final void writeLong(long v)`
- `public final void writeFloat(float v)`
- `public final void writeDouble(double v)`
- `public final void writeBytes(String s) throws IOException`
  - Writes the string to the file as a sequence of bytes. Each character in the string is written out, in sequence, by discarding its high eight bits. The write starts at the current position of the file pointer.
- `public final void writeChars(String s) throws IOException`
  - Writes a string to the file as a sequence of characters. Each character is written to the data output stream as if by the `writeChar` method. The write starts at the current position of the file pointer.

# ► New I/O

- The Java “new” I/O library, introduced in JDK 1.4 in the **java.nio.\*** packages, has one goal: speed.
- The speed comes by using structures which are closer to the operating system’s way of performing I/O: *channels* and *buffers*.
- The NIO APIs include the following features:
  - **Buffers for data of primitive types**
  - **Character-set encoders and decoders**
  - **Channels, a new primitive I/O abstraction**
  - **A file interface that supports locks and memory mapping**
  - **A multiplexed, non-blocking I/O facility for writing scalable servers**



# ▶ java.nio.ByteBuffer

- **public static ByteBuffer allocate(int capacity)**

- Allocates a new byte buffer. The new buffer's position will be zero, its limit will be its capacity

- **public abstract byte get()**

- Relative *get* method. Reads the byte at this buffer's current position, and then increments the position.

- **public abstract ByteBuffer put(byte b)**

- Relative *put* method (*optional operation*). Writes the given byte into this buffer at the current position, and then increments the position.

- **public abstract byte get(int index)**

- Absolute *get* method. Reads the byte at the given index.

- **public abstract ByteBuffer put(int index, byte b)**

- Absolute *put* method (*optional operation*). Writes the given byte into this buffer at the given index.

- **public ByteBuffer get(byte[] dst, int offset, int length)**

- Relative bulk *get* method. Otherwise, this method copies length bytes from this buffer into the given array, starting at the current position of this buffer and at the given offset in the array. The position of this buffer is then incremented by length.

# ▶ java.nio.ByteBuffer

- **public ByteBuffer get(byte[] dst)**

- Relative bulk *get* method. This method transfers bytes from this buffer into the given destination array.

- **public ByteBuffer put(ByteBuffer src)**

- Relative bulk *put* method (*optional operation*). This method transfers the bytes remaining in the given source buffer into this buffer. If there are more bytes remaining in the source buffer than in this buffer, that is, if `src.remaining() > remaining()`, then no bytes are transferred and a `BufferOverflowException` is thrown.

- **public ByteBuffer put(byte[] src, int offset, int length)**

- Relative bulk *put* method (*optional operation*). this method copies length bytes from the given array into this buffer, starting at the given offset in the array and at the current position of this buffer. The position of this buffer is then incremented by length.

- **public final ByteBuffer put(byte[] src)**

- Relative bulk *put* method (*optional operation*). This method transfers the entire content of the given source byte array into this buffer.

- **public final byte[] array()**

- Returns the byte array that backs this buffer (*optional operation*).

# ▶ java.nio.ByteBuffer

- **public abstract char getChar()**

- Relative *get* method for reading a char value. Reads the next two bytes at this buffer's current position, composing them into a char value according to the current byte order, and then increments the position by two.

- **public abstract ByteBuffer putChar(char value)**

- Relative *put* method for writing a char value (*optional operation*). Writes two bytes containing the given char value, in the current byte order, into this buffer at the current position, and then increments the position by two.

- **public abstract char getChar(int index)**

- Absolute *get* method for reading a char value. Reads two bytes at the given index, composing them into a char value according to the current byte order.

- **public abstract ByteBuffer putChar(int index, char value)**

- Absolute *put* method for writing a char value (*optional operation*). Writes two bytes containing the given char value, in the current byte order, into this buffer at the given index.

- **public abstract CharBuffer asCharBuffer()**

- Creates a view of this byte buffer as a char buffer.

# ▶ java.nio.ByteBuffer

- public abstract short getShort()
- public abstract ByteBuffer putShort(short value)
- public abstract short getShort(int index)
- public abstract ByteBuffer putShort(int index, short value)
- public abstract ShortBuffer asShortBuffer()
  - Creates a view of this byte buffer as a short buffer.
- public abstract int getInt()
- public abstract ByteBuffer putInt(int value)
- public abstract int getInt(int index)
- public abstract ByteBuffer putInt(int index, int value)
- public abstract IntBuffer asIntBuffer()
- Long, Float, Double

- A container for data of a specific primitive type:
- A buffer is a linear, finite sequence of elements of a specific primitive type. Aside from its content, the essential properties of a buffer are its capacity, limit, and position:
- A buffer's *capacity* is the number of elements it contains. The capacity of a buffer is never negative and never changes.
- A buffer's *limit* is the index of the first element that should not be read or written. A buffer's limit is never negative and is never greater than its capacity.
- A buffer's *position* is the index of the next element to be read or written. A buffer's position is never negative and is never greater than its limit.

# ▶ **java.nio.Buffer**

- **public final int capacity()**
  - Returns this buffer's capacity.
- **public final int position()**
  - Returns this buffer's position.
- **public final Buffer position(int newPosition)**
  - Sets this buffer's position. If the mark is defined and larger than the new position then it is discarded.
- **public final int limit()**
  - Returns this buffer's limit.
- **public final Buffer limit(int newLimit)**
  - Sets this buffer's limit. If the position is larger than the new limit then it is set to the new limit. If the mark is defined and larger than the new limit then it is discarded.

- **public final Buffer flip()**

- Flips this buffer. The limit is set to the current position and then the position is set to zero. If the mark is defined then it is discarded. After a sequence of channel-read or *put* operations, invoke this method to prepare for a sequence of channel-write or relative *get* operations. For example:
- `buf.put(magic);` // Prepend header  
`in.read(buf);` // Read data into rest of buffer  
`buf.flip();` // Flip buffer  
`out.write(buf);` // Write header + data to channel

- **public final Buffer rewind()**

- Rewinds this buffer. The position is set to zero and the mark is discarded. Invoke this method before a sequence of channel-write or *get* operations, assuming that the limit has already been set appropriately. For example:
- `out.write(buf);` // Write remaining data  
`buf.rewind();` // Rewind buffer  
`buf.get(array);` // Copy data into array

- **public final Buffer clear()**

- Clears this buffer. The position is set to zero, the limit is set to the capacity, and the mark is discarded. Invoke this method before using a sequence of channel-read or *put* operations to fill this buffer. For example:
  - **buf.clear();** // Prepare buffer for reading
  - **in.read(buf);** // Read data

- **public final int remaining()**

- Returns the number of elements between the current position and the limit.

- **public final boolean hasRemaining()**

- Tells whether there are any elements between the current position and the limit.



# ▶ java.nio.channels.FileChannel

- This class does not define methods for opening existing files or for creating new ones; such methods may be added in a future release. In this release a file channel can be obtained from an existing **FileInputStream**, **FileOutputStream**, or **RandomAccessFile** object by invoking that object's **getChannel** method, which returns a file channel that is connected to the same underlying file.
- **public abstract int read(ByteBuffer dst)**
- **public abstract long read(ByteBuffer[] dsts, int offset, int length)**
- **public abstract int write(ByteBuffer src)**
- **public abstract long write(ByteBuffer[] srcs, int offset, int length)**
- **public abstract long position()**
- **public abstract FileChannel position(long newPosition)**

# ▶ java.nio.channels.FileChannel

- **public abstract long size() throws IOException**
  - Returns the current size of this channel's file, measured in bytes
- **public abstract FileChannel truncate(long size) throws IOException**
  - Truncates this channel's file to the given size. If the given **size is less than** the file's **current size** then the **file is truncated**, discarding any bytes beyond the new end of the file. If the given **size is greater than or equal to** the file's **current size** then the **file is not modified**. In either case, if this channel's file position is greater than the given size then it is set to that size.
- **public abstract void force(boolean metaData) throws IOException**
  - Forces any updates to this channel's file to be written to the storage device that contains it.
- **public abstract long transferTo(long position, long count, WritableByteChannel target) throws IOException**
  - Transfers bytes from this channel's file to the given writable byte channel.

# ▶ java.nio.channels.FileChannel

- public abstract long transferFrom(ReadableByteChannel src, long position, long count) throws IOException
  - Transfers bytes into this channel's file from the given readable byte channel.
- public abstract int read(ByteBuffer dst, long position)
- public abstract int write(ByteBuffer src, long position)
- public abstract FileLock lock(long position, long size, boolean shared) throws IOException
- public abstract FileLock tryLock(long position, long size, boolean shared) throws IOException
  - Acquires a lock on the given region of this channel's file.
  - **position** : The position at which the locked region is to start; must be non-negative
  - **size** : The size of the locked region; must be non-negative, and the sum position + size must be non-negative
  - **shared** : true to request a shared lock, in which case this channel must be open for reading (and possibly writing); false to request an exclusive lock, in which case this channel must be open for writing (and possibly reading)

# ▶ java.nio.channels.FileChannel

- public abstract MappedByteBuffer  
map(FileChannel.MapMode mode, long position,  
long size) throws IOException
  - Maps a region of this channel's file directly into memory. A region of a file may be mapped into memory in one of three modes:
    - *Read-only*: Any attempt to modify the resulting buffer will cause a ReadOnlyBufferException to be thrown. (**MapMode.READ\_ONLY**)
    - *Read/write*: Changes made to the resulting buffer will eventually be propagated to the file; they may or may not be made visible to other programs that have mapped the same file. (**MapMode.READ\_WRITE**)
    - *Private*: Changes made to the resulting buffer will not be propagated to the file and will not be visible to other programs that have mapped the same file; instead, they will cause private copies of the modified portions of the buffer to be created. (**MapMode.PRIVATE**)
  - For a read-only mapping, this channel must have been opened for reading; for a read/write or private mapping, this channel must have been opened for both reading and writing.

# ▶ java.nio.channels.FileChannel

```
import java.io.*;
import java.nio.*;
import java.nio.channels.*;
public class ChannelCopy {
    private static final int BSIZE = 1024;
    public static void main(String[] args) throws Exception {
        FileChannel in = new FileInputStream("data.pdf").getChannel(),
            out = new FileOutputStream("data1.pdf").getChannel();
        ByteBuffer buffer = ByteBuffer.allocate(BSIZE);
        Stopwatch stopwatch = new Stopwatch();
        stopwatch.start();
        while(in.read(buffer) != -1) {
            buffer.flip(); // Prepare for writing
            out.write(buffer);
            buffer.clear(); // Prepare for reading
        }
        stopwatch.stop();
        System.out.println("With channel buffering: " + stopwatch );
        in.close();
        out.close();
    }
}
```

# ▶ java.nio.channels.FileChannel

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

import java.nio.channels.*;
public class TransferTo {
    public static void main(String[] args) throws Exception {
        FileChannel in = new FileInputStream("data.pdf").getChannel(),
            out = new FileOutputStream("data1.pdf").getChannel();
        Stopwatch stopwatch = new Stopwatch();
        stopwatch.start();
        in.transferTo(0, in.size(), out);
        stopwatch.stop();
        System.out.println("With channel Transfer: " + stopwatch );
        in.close();
        out.close();
    }
}
```

# ► CRC without Memory-Mapped file

```
import java.io.*;
import java.util.zip.*;
/* This program computes the CRC checksum of a file, using an input
stream.
*/
public class CRC {
    public static void main(String[] args) throws IOException {
        InputStream in = new FileInputStream("data.pdf");
        CRC32 crc = new CRC32();
        int c;
        long start = System.currentTimeMillis();
        while((c = in.read()) != -1)
            crc.update(c);
        long end = System.currentTimeMillis();
        System.out.println(Long.toHexString(crc.getValue()));
        System.out.println((end - start) + " milliseconds");
    }
}
```

# ► CRC with Memory-Mapped file

```
import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.util.zip.*;
//compute the CRC checksum of a file, using a memory-mapped file.
public class NIOCRC {
    public static void main(String[] args) throws Exception {
        FileInputStream in = new FileInputStream("data.pdf");
        FileChannel channel = in.getChannel();
        CRC32 crc = new CRC32();
        long start = System.currentTimeMillis();
        MappedByteBuffer buffer = channel.map(
            FileChannel.MapMode.READ_ONLY, 0, (int)channel.size());
        while (buffer.hasRemaining()) crc.update(buffer.get());
        long end = System.currentTimeMillis();
        System.out.println(Long.toHexString(crc.getValue()));
        System.out.println((end - start) + " milliseconds");
    }
}
```



# ► Readers and Writers

- While input streams and output streams may be used to read and write text as well as bytes of information and primitive data types, a better alternative is to use readers and writers. Readers and writers were introduced in JDK1.1 to better support Unicode character streams.
- The most important concrete subclasses of Reader and Writer are the **InputStreamReader** and the **OutputStreamWriter** classes. An **InputStreamReader** contains an underlying input stream from which it **reads raw bytes. It translates these bytes into Unicode characters according to a specified encoding.** An **OutputStreamWriter** receives **Unicode characters** from a running program. It then **translates those characters into bytes** using a specified encoding and writes the bytes onto an underlying output stream.

- Like OutputStream, the **Writer class is never used directly**, only polymorphically through one of its subclasses. It has five write( ) methods as well as a flush( ) and a close( ) method:
- protected Writer( )
- protected Writer(Object lock)
- **public abstract void write(char[] text, int offset, int length)**
- **throws IOException**
- **public void write(int c) throws IOException**
- **public void write(char[] text) throws IOException**
- **public void write(String s) throws IOException**
- **public void write(String s, int offset, int length) throws IOException**
- **public abstract void flush( ) throws IOException**
- **public abstract void close( ) throws IOException**

```
char[] network = {'N', 'e', 't', 'w', 'o', 'r', 'k'};  
w.write(network, 0, network.length);
```

- The same task can be accomplished with these other methods as well:

```
for (int i = 0; i < network.length; i++) w.write(network[i]);  
w.write("Network");  
w.write("Network", 0, 7);
```

- If it's using **big-endian Unicode**, then it will write these **14 bytes** (shown here in hexadecimal) in this order:

**00 4E 00 65 00 74 00 77 00 6F 00 72 00 6B**

- On the other hand, if w uses **little-endian Unicode**, this sequence of **14 bytes** is written:

**4E 00 65 00 74 00 77 00 6F 00 72 00 6B 00**

- If uses Latin-1, **UTF-8**, or MacRoman, this sequence of seven bytes is written:

**4E 65 74 77 6F 72 6B**

# ▶ java.io.OutputStreamWriter

- **public OutputStreamWriter(OutputStream out, String charsetName)** throws **UnsupportedEncodingException**
  - Create an OutputStreamWriter that uses the named charset.
- | Charset    | Description  |
|------------|--|
| US-ASCII   | Seven-bit ASCII, a.k.a. ISO646-US, a.k.a. the Basic Latin block of the Unicode character set |
| ISO-8859-1 | ISO Latin Alphabet No. 1, a.k.a. ISO-LATIN-1   |
| UTF-8      | Eight-bit UCS Transformation Format  |
| UTF-16BE   | Sixteen-bit UCS Transformation Format, big-endian byte order                                 |
| UTF-16LE   | Sixteen-bit UCS Transformation Format, little-endian byte order                              |
| UTF-16     | Sixteen-bit UCS Transformation Format, byte order identified by an optional byte-order mark  |

# ▶ java.io.OutputStreamWriter

- **public OutputStreamWriter(OutputStream out)**
  - Create an OutputStreamWriter that uses the **default character encoding**.
- **public OutputStreamWriter(OutputStream out, CharsetEncoder enc)**
  - Create an OutputStreamWriter that uses the given charset encoder.
- **public String getEncoding()**
  - Return the name of the character encoding being used by this stream.
- **public void write(int c) throws IOException**
  - Write a single character.
- **public void write(char[] cbuf, int off, int len) throws IOException**
  - Write a portion of an array of characters.

# ▶ **java.io.OutputStreamWriter**

- **public void write(String str, int off, int len)** throws **IOException**
  - Write a portion of a string.
- **public void flush()** throws **IOException**
  - Flush the stream.
- **public void close()** throws **IOException**
  - Close the stream.

# ► OutputStreamWriter demo

```
• import java.io.*;
// Chapter 4, Listing 4
public class OutputStreamToWriterDemo{
    public static void main(String args[]){
        try{
            OutputStream output = new FileOutputStream("utf8.txt");
            // Create an OutputStreamWriter
            OutputStreamWriter writer = new OutputStreamWriter
                (output,"UTF-8");

            // Write to file using a writer
            writer.write ("Phạm Văn Tính");
            // Flush and close the writer, to ensure it is written
            writer.flush();
            writer.close();
        } catch (IOException ioe){
            System.err.println ("I/O error : " + ioe);
        }
    }
}
```

# ▶ java.io.FileWriter

- **Convenience class for writing character files.** The constructors of this class assume that the **default character encoding** and the **default byte-buffer size** are acceptable. **To specify these values yourself, construct an OutputStreamWriter on a FileOutputStream.**
- **public FileWriter(String fileName)** throws IOException
  - Constructs a FileWriter object given a file name.
- **public FileWriter(String fileName, boolean append)** throws IOException
  - Constructs a FileWriter object given a file name with a boolean indicating whether or not to append the data written.
- **public FileWriter(File file)** throws IOException
  - Constructs a FileWriter object given a File object.
- **public FileWriter(File file, boolean append)** throws IOException
  - Constructs a FileWriter object given a File object. If the second argument is true, then bytes will be written to the end of the file rather than the beginning.
- **Methods inherited from class java.io.OutputStreamWriter:** **close, flush, getEncoding, write**



# ▶ java.io.InputStreamReader

- An `InputStreamReader` is a bridge from byte streams to character streams: It reads bytes and decodes them into characters using a specified charset. The charset that it uses may be specified by name or may be given explicitly, or the platform's default charset may be accepted.
- `public InputStreamReader(InputStream in)`
  - Create an `InputStreamReader` that uses the default charset.
- `public InputStreamReader(InputStream in, String charsetName)` throws `UnsupportedEncodingException`
  - Create an `InputStreamReader` that uses the named charset.
- `public String getEncoding()`
  - Return the name of the character encoding being used by this stream.

# ▶ java.io.InputStreamReader

- **public int read()** throws IOException
  - Read a single character.
- **public int read(char[] cbuf, int offset, int length)** throws IOException
  - Read characters into a portion of an array.
- **public boolean ready()** throws IOException
  - Tell whether this stream is ready to be read. An InputStreamReader is ready if its input buffer is not empty, or if bytes are available to be read from the underlying byte stream.
- **public void close()** throws IOException

# ► Charset Translation

```
public class InputStreamReaderDemo {
    public static void main(String args[]){
        try{
            OutputStream output = new FileOutputStream("utf8_16.txt");
            // Create an OutputStreamWriter
            OutputStreamWriter writer = new OutputStreamWriter(output,
                                                                "UTF-16");

            InputStream input = new FileInputStream("utf8.txt");
            InputStreamReader reader = new InputStreamReader(input,
                                                            "UTF-8");

            char[] buff = new char[100];
            // Write to file using a writer
            int rNumber = reader.read(buff);
            System.out.println("Number of char: "+rNumber);
            writer.write(buff,0,rNumber);
            // Flush and close the writer, to ensure it is written
            writer.flush();      writer.close();
            reader.close();
        } catch (IOException ioe){
            System.err.println ("I/O error : " + ioe);
        }
    }
}
```

# ► Complete example

02-2004

Khoa CNTT

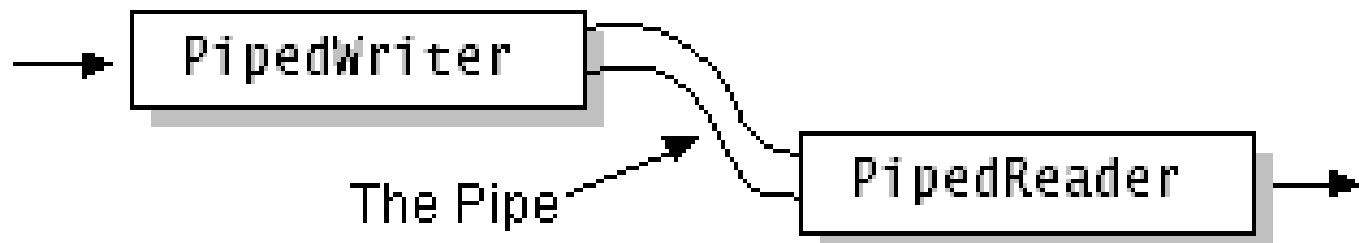
94/80

PHẠM VĂN TÍNH

- **Student List**

# ► Pipe stream

- Pipe được dùng làm một kênh truyền nối output của 1 thread với input của 1 thread khác.
  - Khi liên lạc với nhau bằng pipe, các thread không phải quan tâm đến vấn đề đồng bộ
- Tưởng tượng pipe là ống có 2 đầu, 1 nối với output thread, 1 nối với input thread. Input thread sẽ nhận được những gì output thread đã đưa ra pipe



# ► Pipe Stream (tt)

## **java.io.PipedInputStream**

```
public PipedInputStream()  
public PipedInputStream(PipedOutputStream src)  
public void connect(PipedOutputStream src)
```

## **java.io.PipedOutputStream**

```
public PipedOutputStream()  
public PipedOutputStream(PipedInputStream src)  
public void connect(PipedInputStream dest)
```

## **java.io.PipedReader**

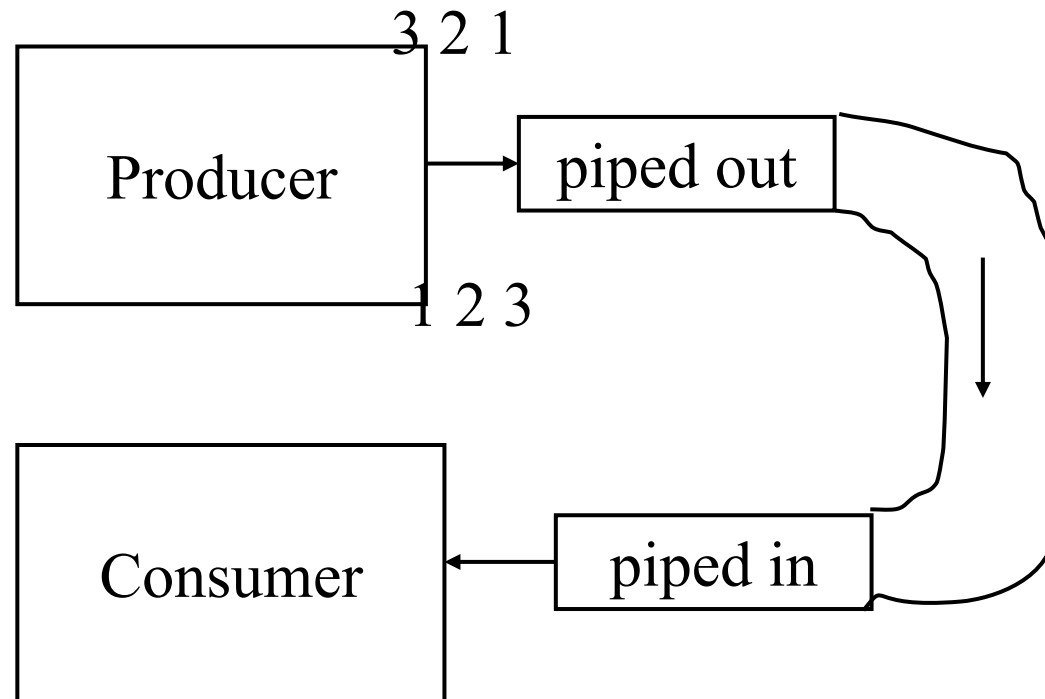
```
public PipedReader()  
public PipedReader(PipedWriter src)  
public void connect(PipedWriter src)
```

## **java.io.PipedWriter**

```
public PipedWriter()  
public PipedWriter(PipedReader dest)  
public void connect(PipedReader dest)
```

## ► Ví dụ pipe stream

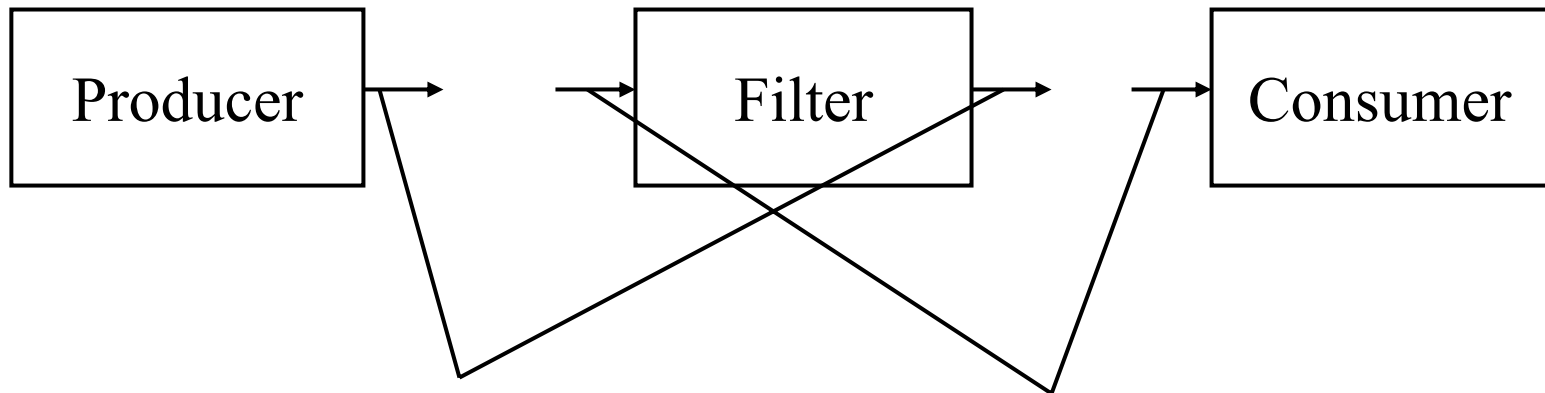
- Ví dụ 1: PipeTest gồm 2 thread:
  - 1 output thread phát sinh các số nguyên
  - 1 input thread đọc các giá trị phát sinh từ output thread và in ra



## ► Ví dụ pipe stream

- Ví dụ 2: PipeTest gồm 3 thread:

- producer thread phát sinh các số nguyên ra pipe
- filter thread nhận dữ liệu từ producer, tính trung bình, và đưa kết quả ra 1 pipe
- consumer thread đọc các giá trị phát sinh từ filter thread và in ra kết quả



*Piped output stream*

*Piped input stream*



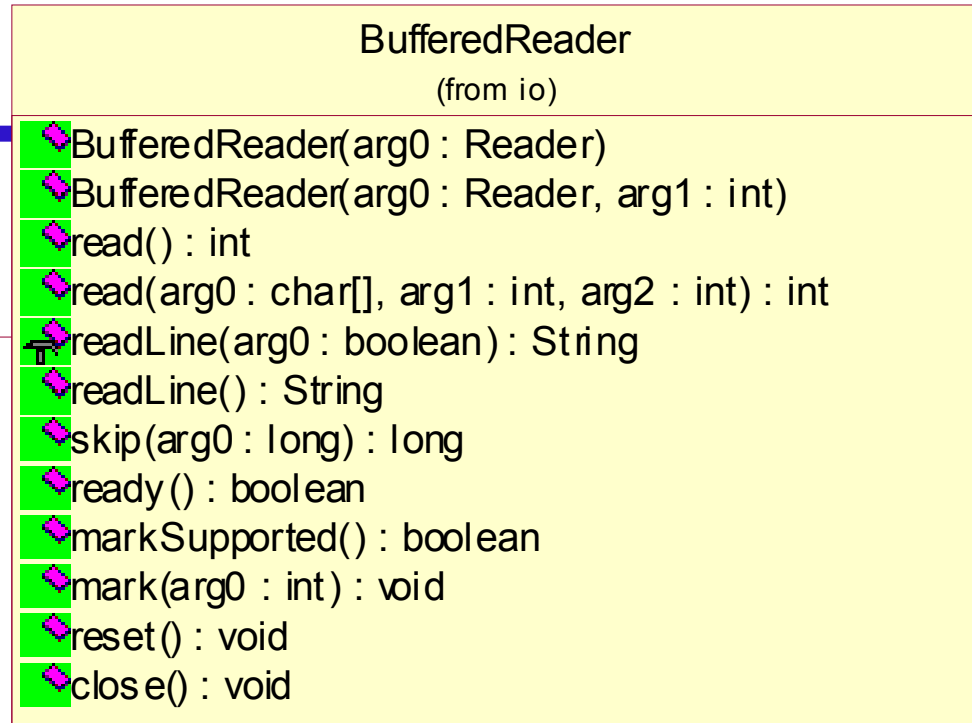
- Text stream cho phép user nhìn stream dưới dạng “đọc được” (readable)
  - `InputStreamReader`, `OutputStreamWriter` còn cung cấp thêm khả năng chuyển đổi stream  $\leftrightarrow$  reader/writer, khả năng làm việc với các bảng mã khác nhau
  - `BufferedReader` cung cấp cách đọc ra từng hàng từ một stream
  - `BufferedWriter` cung cấp cách thức ghi các chuỗi ra stream dưới dạng đọc được
  - `PrintWriter` cung cấp cách thức ghi các chuỗi, số nguyên, số thực, ... ra stream dưới dạng đọc được

## PrintWriter

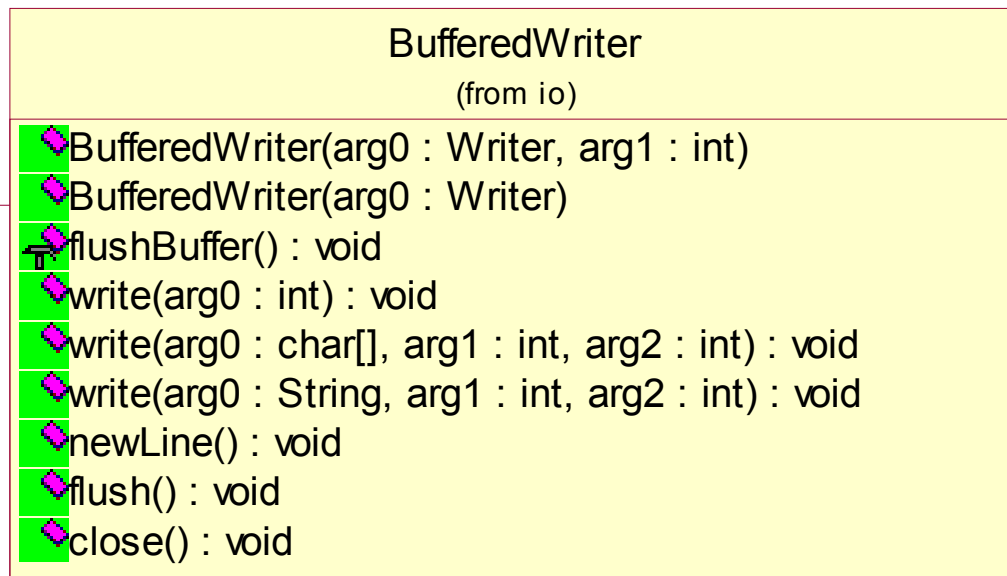
(from io)

- PrintWriter(arg0 : OutputStream, autoFlush : boolean)
- PrintWriter(arg0 : OutputStream)
- PrintWriter(arg0 : Writer, autoFlush : boolean)
- PrintWriter(arg0 : Writer)
- flush() : void
- close() : void
- checkError() : boolean
- setError() : void
- write(arg0 : int) : void
- write(arg0 : char[], arg1 : int, arg2 : int) : void
- write(arg0 : char[]) : void
- write(arg0 : String, arg1 : int, arg2 : int) : void
- write(arg0 : String) : void
- print(arg0 : boolean) : void
- print(arg0 : char) : void
- print(arg0 : int) : void
- print(arg0 : long) : void
- print(arg0 : float) : void
- print(arg0 : double) : void
- print(arg0 : char[]) : void
- print(arg0 : String) : void
- print(arg0 : Object) : void
- println() : void
- println(arg0 : boolean) : void
- println(arg0 : char) : void
- println(arg0 : int) : void
- println(arg0 : long) : void
- println(arg0 : float) : void
- println(arg0 : double) : void
- println(arg0 : char[]) : void
- println(arg0 : String) : void
- println(arg0 : Object) : void

*Reader*  
(from io)



*Writer*  
(from io)



# ► Object Streams

- Using a **fixed-length** record format is a good choice if you need **to store data of the same type**. However, **objects** that you create in an object-oriented program **are rarely all of the same type**.
- If we want to save files that contain this kind of **information**, we must first save the type of each object and then the data that defines the current state of the object. When we read this information back from a file, we must:
  - Read the object type;
  - Create a blank object of that type;
  - Fill it with the data that we stored in the file.
- It is entirely possible (if very tedious) to do this by hand. However, **Sun Microsystems developed a powerful mechanism called object serialization to read/write objects from/into the file**.

# ► Storing Objects of Variable Type

- To save object data, you first need to open an `ObjectOutputStream` object:
- `ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("student.dat"));`
- Now, to save an object, you simply use the `writeObject` method of the `ObjectOutputStream` class as in the following fragment:
- `//create objects`  
`Student hoa = new Employee("Trần Thị Hoa", 1980, "CD02");`  
`Student vinh = new Employee("Lương Thế Vinh", 1981, "DH03");`
- `//Storing objects into stream`  
`out.writeObject(hoa);`  
`out.writeObject(vinh);`

# ▶ Reading Objects back

- First get an `ObjectInputStream` object
- `ObjectInputStream in = new ObjectInputStream(new FileInputStream("employee.dat"));`
- Then, retrieve the objects in the same order in which they were written, using the `readObject` method.
- `Student st1 = (Student)in.readObject();`  
`Student st2 = (Student)in.readObject();`  
.....
- When reading back objects, you must **carefully keep track of the number of objects that were saved**, their order, and their types. Each call to `readObject` reads in another object of the type `Object`. You, therefore, will need to cast it to its correct type.

# ► **Serializable interface**

- you need to make to any class that you want to save and restore in an object stream. The class must implement the **Serializable** interface:
- class **Employee** implements **Serializable** { . . . }
- The **Serializable** interface has no methods, so you don't need to change your classes in any way.
- **To make a class serializable, you do not need to do anything else.**
- Writing an array is done with a single operation:
- **Student[] stList = new Student[3];**  
**out.writeObject(stList);**
- Similarly, reading in the result is done with a single operation. However, we must apply a cast to the return value of the **readObject** method:
- **Student[] newStList = (Student[])in.readObject();**

# ▶ Student List using Object Streams

```
public class SerialStudent implements Serializable{
    private String name;
    private int age;
    private String cl;

    public SerialStudent(String n, int a, String c){
        name = n;
        age = a;
        cl = c;
    }
    public String getName() { return name; }
    public int getAge() { return age; }
    public String getCl(){ return cl; }

    public String toString() {
        return getClass().getName() + "[Name=" + name
            + ",Age=" + age + ",Class=" + cl + "];"
    }
    public void exportData(PrintWriter out){
        out.println(name + "|" + age + "|" + cl);
    }
}
```



# ▶ Student List using Object Streams

```
public class SerialTest {
    public static void main(String[] args) {
        SerialStudent[] st = new SerialStudent[3];
        st[0] = new SerialStudent("Phạm Thị Mỹ Hạnh", 20, "TC02");
        st[1] = new SerialStudent("Trần Thị Hoa", 18, "CD02");
        st[2] = new SerialStudent("Nguyễn Văn Vệ", 19, "DH03");
        try {
            // save all students records to the file studentemployee.dat
            ObjectOutputStream out = new ObjectOutputStream(new
                FileOutputStream("SerialStudent.dat"));

            out.writeObject(st);
            out.close();
            // retrieve all records into a new array
            ObjectInputStream in = new ObjectInputStream(new
                FileInputStream("SerialStudent.dat"));

            try{
                SerialStudent[] newSt = (SerialStudent[])in.readObject();
                // print the newly read student records
                for (int i = 0; i < newSt.length; i++) System.out.println(newSt[i]);
            } catch (ClassNotFoundException e) {};
            in.close();
        }
    }
}
```

# ▶ **java.io.ObjectOutputStream**

- **ObjectOutputStream(OutputStream out)**  
creates an ObjectOutputStream so that you can write objects to the specified OutputStream.
- **void writeObject(Object obj)**  
writes the specified object to the ObjectOutputStream. This method saves the class of the object, the signature of the class, and the values of any non-static, non-transient field of the class and its superclasses.

# ▶ **java.io.ObjectInputStream**

- **ObjectInputStream(InputStream is)**  
creates an ObjectInputStream to read back object information from the specified InputStream.
- **Object readObject()**  
reads an object from the ObjectInputStream. In particular, this reads back the class of the object, the signature of the class, and the values of the nontransient and nonstatic fields of the class and all of its superclasses. It does deserializing to allow multiple object references to be recovered.

# ▶ **Java Database Connectivity (JDBC)**

# ► JDBC Driver Types

---

01-2004

Khoa CNTT

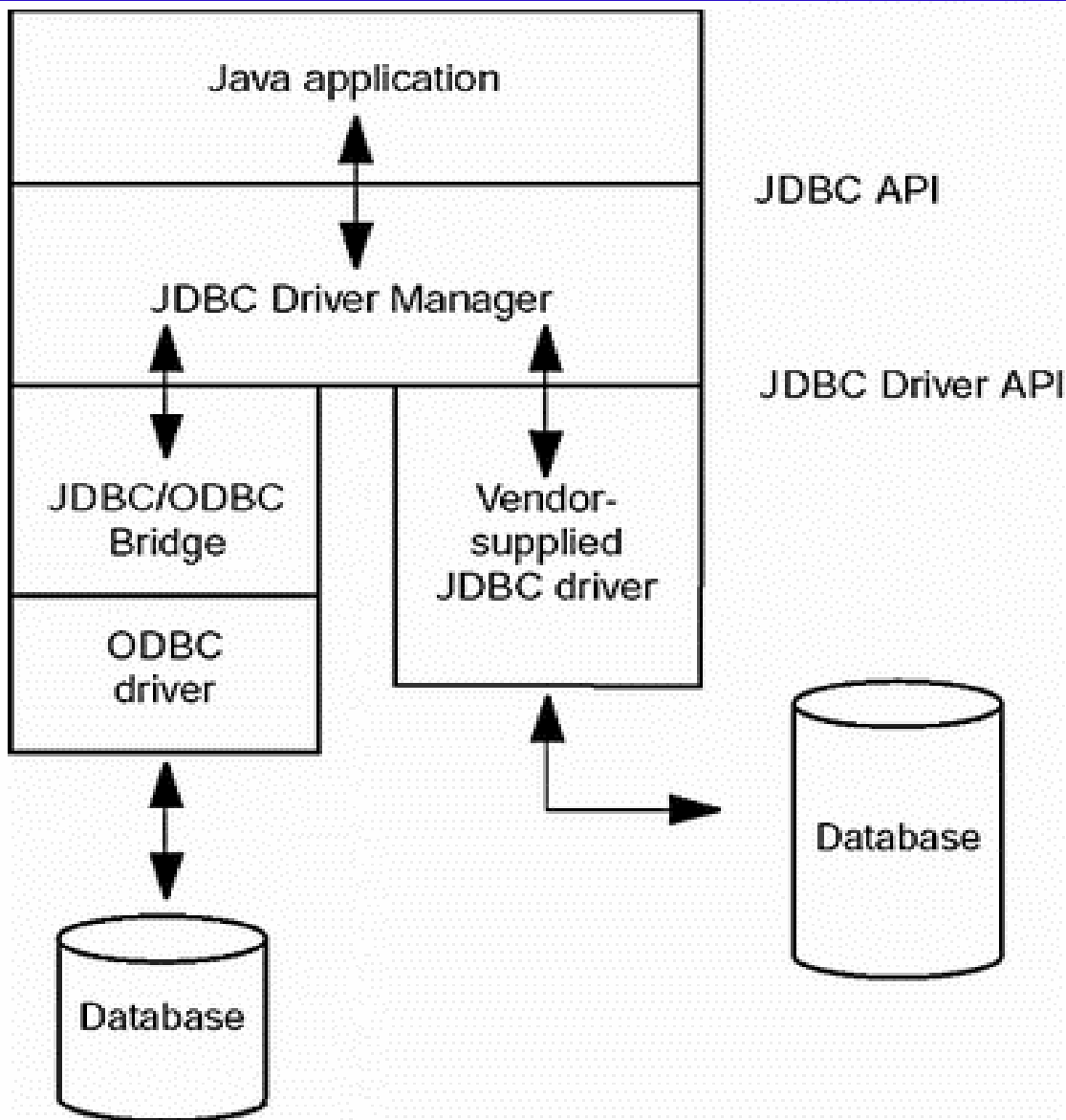
2/25

PHẠM VĂN TÍNH

- **Core Java: Chapter 4, Page: 329**
- **Advanced JAVA Prog: Chapter 4, Page 93**

- **JDBC (Java DataBase Connectivity) - provides access to *relational database systems***
- **JDBC is a vendor independent API for accessing relational data from different vendors (Microsoft Access, Oracle) in a consistent way**
- **The language SQL (Structured Query Language) is normally used to make queries on relational data**
- **JDBC API provides methods for executing SQL statements and obtaining results: *SELECT, UPDATE, INSERT, DELETE* etc.**
- **Provides portability (eliminates rewriting code for different databases and recompiling for different platforms) and faster, reusable object developing environment**
- **JDBC API is part of core Java; JDBC 1.0 is bundled with JDK 1.1 or higher (package: java.sql) and JDBC 2.0 (needs compliant driver; additional package: javax.sql;**

# ► JDBC-to-database communication path



# Vendor specific APIs - JDBC Drivers

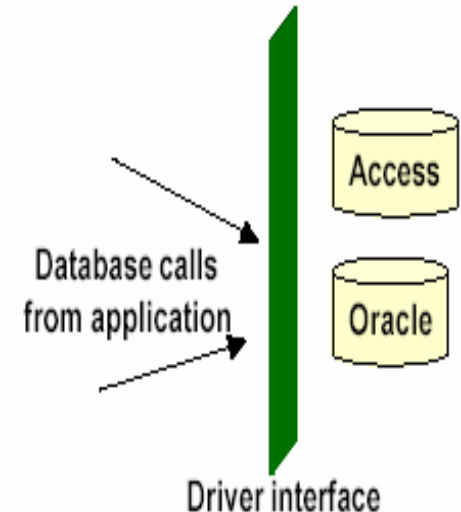
- Database vendors provide proprietary APIs for accessing data managed by the server
- Languages such as C/C++ can make use of these APIs to interface with the database
- JDBC aims at providing an API that eliminates vendor specific nature in accessing a database

- However, JDBC still requires a vendor-specific driver for accessing database from a particular vendor
- The driver provides interface between JDBC API and vendor database by converting calls from JDBC API to vendor's database calls

- With additional functionality of the driver, the same application may be reusable with a different vendor's database by simply switching to that driver

- Example drivers:

- JDBC/ODBC driver: [sun.jdbc.odbc.JdbcOdbcDriver](#) (used in this course)
- Oracle driver: [oracle.jdbc.driver.OracleDriver](#)

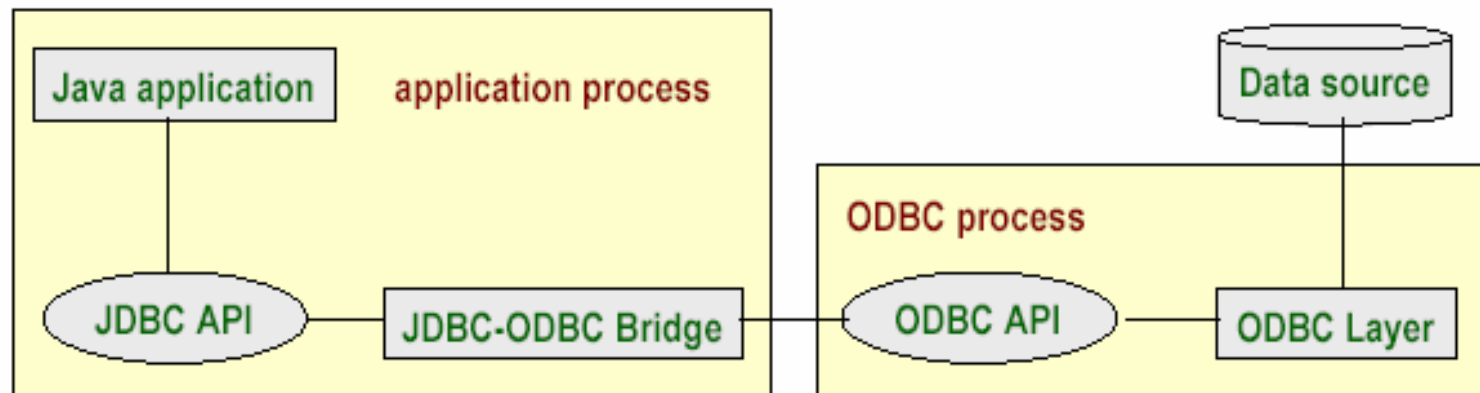




# ► JDBC Driver Types

## Type 1: JDBC-ODBC Bridge

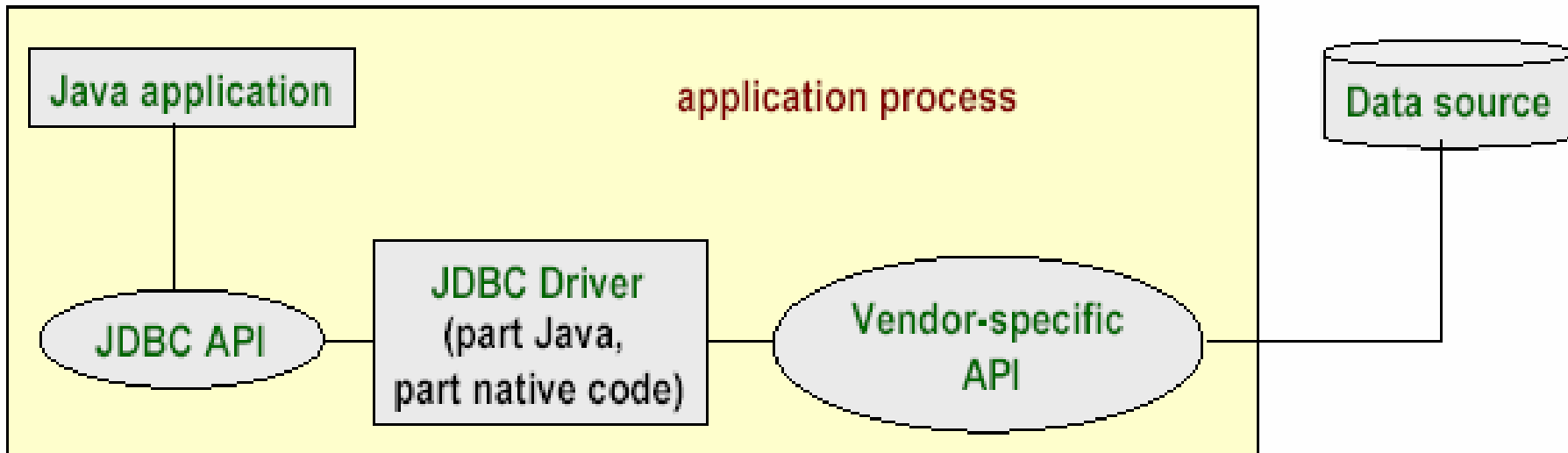
- ODBC (Open Database Connectivity) is Microsoft's API for SQL; popular on Windows platform
- ODBC API provides a set of functions for accessing a database
- JDBC drivers of this type translate calls from JDBC into corresponding ODBC calls



- The database access may be inefficient due to many layers of calls involved
- Limited to what ODBC API provides even if database vendor's API has superior functionality
- This is the only way Microsoft Access database can be accessed from Java

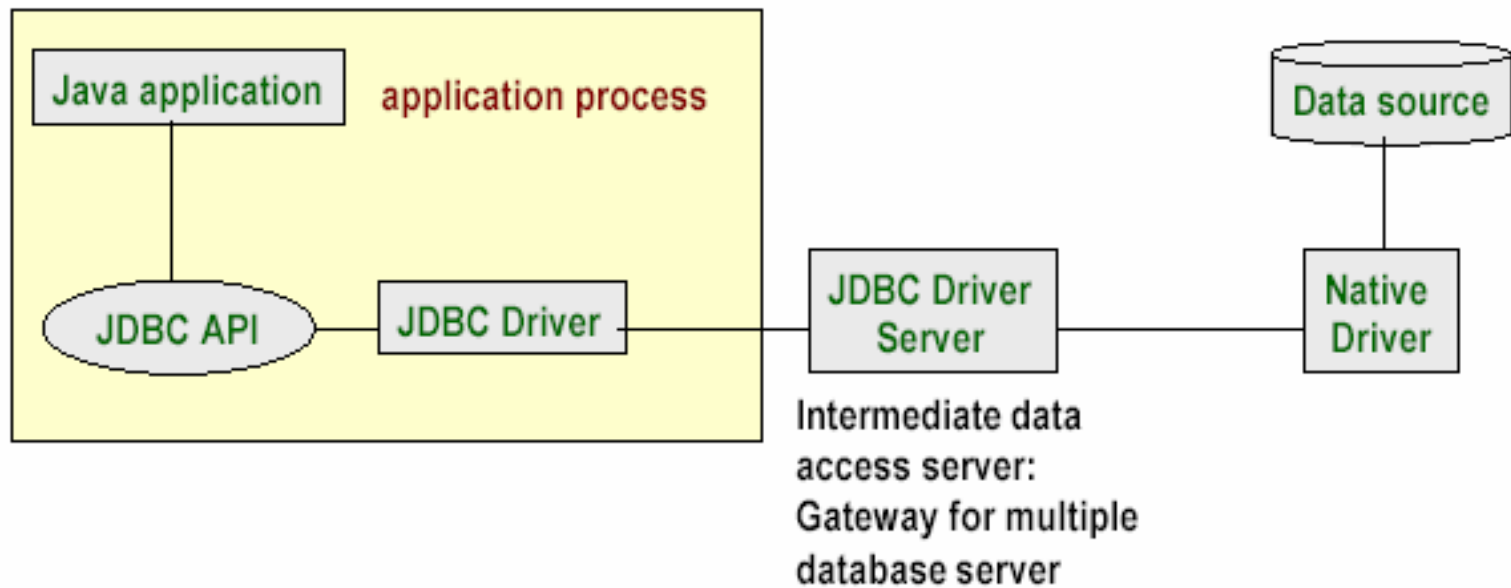
## ► Type 2 - Part Java, Part Native Driver

- JDBC driver consists of java code and native code which uses vendor-specific API for accessing databases
- More efficient than JDBC-ODBC bridge due to fewer layers of communication
- Typical of this type of driver is the driver provided by IBM for its DB2 Universal Database (UDB).



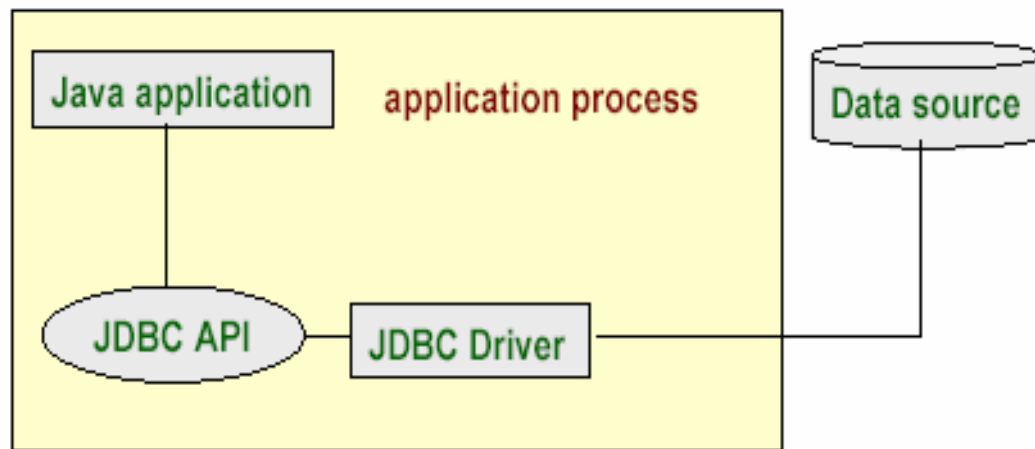
## ► Type 3 - Intermediate Database Access Server

- An intermediate access server between client application and the data source acts as a gateway to connect to multiple database servers
- application sends JDBC calls to the intermediate server via JDBC driver
- The intermediate server handles the request using a native driver



## ► Type 4 - Pure Java Driver

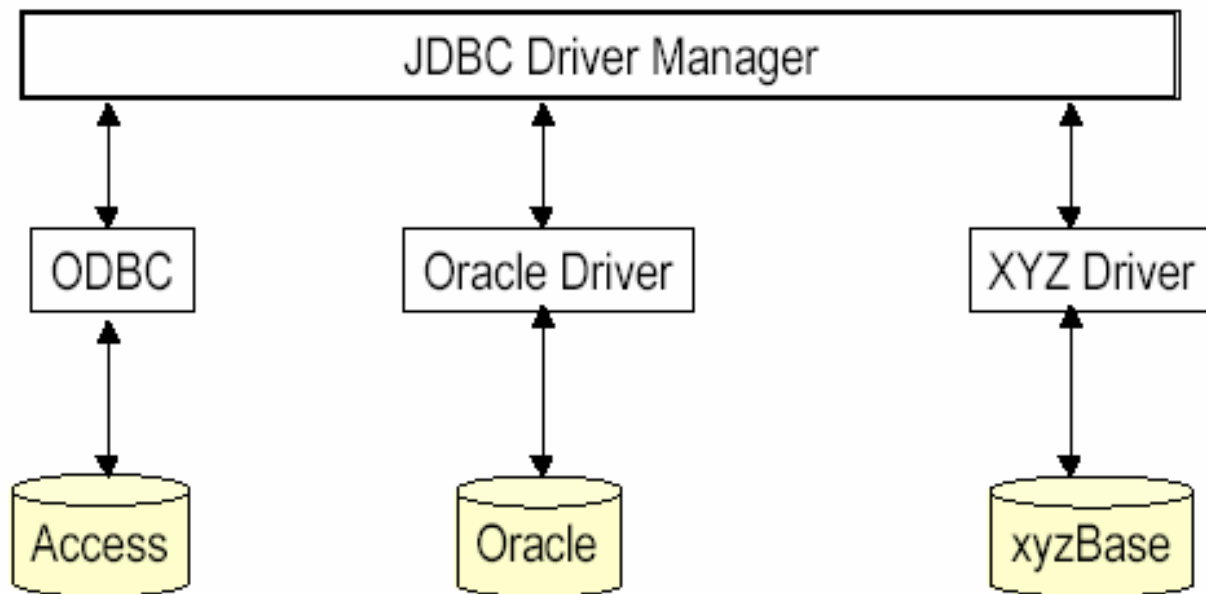
- JDBC calls are directly translated to database calls specific to vendor
- very efficient in terms of performance and development time



One of the types of drivers must be loaded before making a connection to the database and run SQL statements using JDBC

# ► JDBC Driver Manager

- JDBC key components: DriverManager, Connection, Statement, ResultSet
- DriverManager handles communication with different drivers that conform to JDBC Driver API.
  - The static class *DriverManager* manages the loaded drivers and contains methods for accessing connections to the databases



# ► Four steps in creating a database application

Step 1: load a database driver

Step 2: make a database connection

Step 3: create and execute SQL statement

Step 4: process the result set, if necessary

# ► Step 1: Loading a Driver

- A driver is always needed to obtain a connection
- Loading a driver requires class name of the driver.

For JDBC-ODBC driver the class name is:

**sun.jdbc.odbc.JdbcOdbcDriver**

Oracle driver is: **oracle.jdbc.driver.OracleDriver**

- The class definition of the driver is loaded using *forName* static method of the class *Class* (in package *java.lang*)

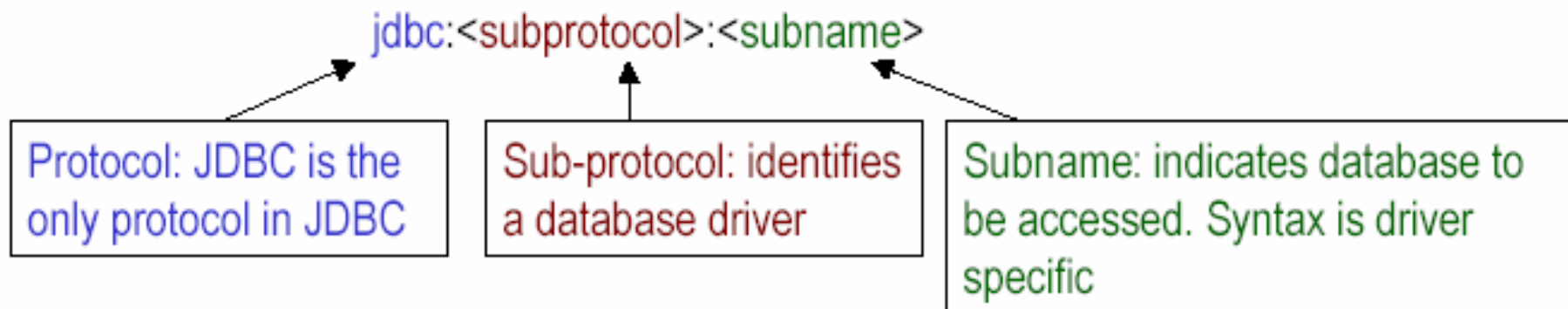
```
try {  
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");  
}  
catch (Exception e) {  
    out.println( e.getMessage() + "\n Class not found Exception.");  
}
```

- It is possible to load several drivers to access various databases (see for vendors list at [java.sun.com/products/jdbc](http://java.sun.com/products/jdbc) )
- The class *DriverManager* manages the loaded drivers

## ► Step 2: Opening a Database Connection

- **getConnection** method of **DriverManager** class returns a connection
- When there are several databases used by the same application, the driver required to access the database is uniquely identified using JDBC URLs

**JDBC URL:** Represents a driver and has following three-part syntax



Examples:

"**jdbc:odbc:books**" : specifies database *books* as ODBC data source (books is a logical name linked to actual database)



## ► Step 2: Opening a Database Connection(contd.)

- Getting a connection to ODBC data source *books* (MS Access database)

```
private String url = "jdbc:odbc:books";  
private String userName = "anonymous";  
private String password = "guest";  
Connection connection = DriverManager.getConnection(url, userName, password);
```

- **DriverManager** has other variants of **getConnection** method that accept different set of parameters
- **Connection** is an interface defined in **java.sql** package. A Connection object represents a connection with the database. The interface has methods to create statements which can be used to manipulate the database

Before the database *books* can be used here, it must be registered as an ODBC source

# Step 3: Creating Statement objects and executing SQL Statements

- Connection objects can be used to create statement objects.

**statement = connection.createStatement();**

- Statement is an interface that contains methods for executing SQL queries

```
String sqlStr = "INSERT INTO Authors VALUES('5', 'Walter', 'Lippman', 'Journalist')";  
int rows = statement.executeUpdate(sqlStr);
```

- **sqlStr** contains a string which is an SQL statement for inserting a new record in the table *Authors* in the *books* database
- The SQL statement is executed using *executeUpdate* method of the statement object
- The method is used to execute statements like **INSERT, UPDATE, DELETE** that do not return any results. It returns **number of rows** affected
- *Authors* is one of the tables in the *books.mdb* database. *Books, Quotations, Topics* are some of the other fields.  
The *Authors* table consists of four fields
- *Authord Id, first name, last name* and *notes*. In the above example, they have the values *'5', 'Walter', 'Lippman'* and *'Journalist'* respectively

## ► Step 4: Enquiring the Database

- The **Statement** object returns a **java.sql.ResultSet** object upon executing an SQL statement using **executeQuery** method

```
ResultSet rs = statement.executeQuery("SELECT * FROM Authors");
```

- The method returns all rows in *Authors* table as a *ResultSet*
- The **next()** method of **ResultSet** allows to move from one row to the next

```
while (rs.next()) {  
    // rs stands for a row in each iteration; print author details  
}
```

- ResultSet* contains several methods for extracting various fields in a row. The methods require column name or column index as an argument. The method used depends on type of the field. For example, author's first name can be obtained by

```
rs.getString( "FirstName" )
```

*FirstName* is the name of the field assigned while creating the database. The method returns author's first name which is a string

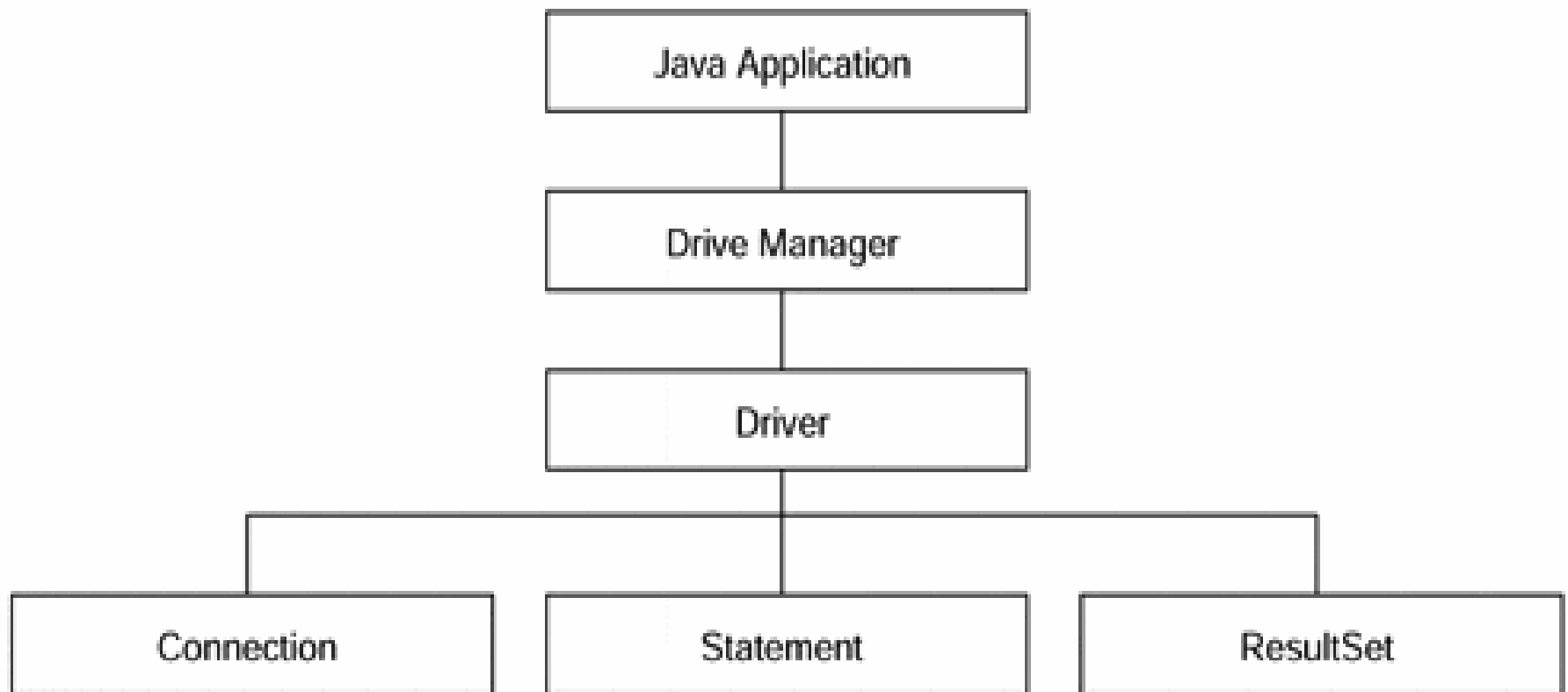
# ► Before compiling - Register a database as an ODBC Data Source

- The computer should have Microsoft Access installed
- Invoke *ODBC Data Source Administrator*:
  - In Windows *Control Panel*, double click “*ODBC Data Sources*”.
- The dialog is used to register *user data source name*. The tab *User DSN* must be selected
- Since a new data source is to be created, click *Add* in the dialog. *Create New Data Source* dialog appears.
- There are several drivers listed in the dialog including dBase, Oracle, Microsoft Access etc. *books* is a Microsoft Access database. So, select *Microsoft Access Driver* and click *Finish*.
- Another *dialog ODBC Microsoft Access Setup* appears.
- In the field *Data Source Name* enter a name by which the database is to be referred in JDBC program. In the example, the name *books* is used.
- This name should be associated with an actual database. Click *Select* button. This displays *Select Database dialog* which allows to select a database on the local file system or across the network.
- Click *OK* to dismiss the *Select Database dialog* and return to *ODBC Microsoft Access Setup*

# ► Registering as an ODBC Data Source

- Click **Advanced** button in **ODBC Microsoft Access Setup** dialog. **Set Advanced Options** dialog appears
- Enter authorisation information **Login name** and **Password**. In this example, the login name is *anonymous* and password is *guest*
- Click **OK** to dismiss the dialog
- Dismiss the **ODBC Microsoft Access Setup dialog** by clicking **OK**.
- Now **ODBC Data Source Administrator** dialog contains data source *books* with *Microsoft Access Driver* associated with it.
- Now the database can be accessed using JDBC-ODBC bridge driver

# ► JDBC key components



# ► The DriverManager Object.

- Once a driver is installed, you need to load it into your Java object by using the **DriverManager**. It provides a common interface to a JDBC driver object without having to delve into the internals of the database itself
- The driver is responsible for creating and implementing the **Connection**, **Statement**, and **ResultSet** objects for the specific database.
- **DriverManager** then is able to acquire those object implementations for itself. In so doing, applications that are written using the **DriverManager** are isolated from the implementation details of databases.

# ► Database Connection Interface.

- The **Connection** object is responsible for establishing the link between the **Database Management System** and the Java application.
- It also enables the programmer to select the proper driver for the required application.
- The **Connection.getConnection** method accepts a **URL** and enables the **JDBC** object to use **different drivers depending** on the situation, isolates applets from connection-related information, and gives the application a means by which to **specify the specific database to which it should connect**. The URL takes the form of **jdbc:<subprotocol>:<subname>**. The subprotocol is a kind of connectivity to the database



# ► Database Statement Object.

- A **Statement** encapsulates a **query** written in **Structured Query Language** and enables the **JDBC** object to compose a series of steps to look up information in a database.
- Using a **Connection**, the **Statement** can be forwarded to the database and obtain a **ResultSet**

## ► **ResultSet Access Control.**

- A **ResultSet** is a container for a series of rows and columns acquired from a **Statement** call. Using the **ResultSet's** iterator routines, the JDBC object can step through each row in the result set. Individual column fields can be retrieved using the get methods within the **ResultSet**. Columns may be specified by their field name or by their index.

# ► Basic Database operations

```
public static void main(String[] args) {
    String url = "jdbc:odbc:sach";
    String userName = "ltmang";
    String password = "ltmang";
    try {
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        Connection connection =
            DriverManager.getConnection(url,userName,password);
        Statement statement = connection.createStatement();
        String sql = "select * from sach";
        ResultSet rs = statement.executeQuery(sql);
        while (rs.next()) {
            System.out.println(rs.getString( "tens" ));
        }
    }
```

# ► Basic Database operations

```
sql = "insert into sach values('P6','Tu hoc Internet','Nha xuat ban  
lao dang')";
```

```
statement.executeUpdate(sql);
```

```
sql = "select * from sach";
```

```
rs = statement.executeQuery(sql);
```

```
while (rs.next()) {
```

```
    System.out.println(rs.getString( "tens" ));
```

```
}
```

```
rs.close();
```

```
statement.close();
```

```
connection.close();
```

```
}
```

```
catch (Exception e) {
```

```
    System.out.println( e.getMessage());
```

```
}
```

```
}
```

# ▶ **Java RMI** **(Remote Method Invocation)**

# ► Tài liệu tham khảo

- Advanced Java Networking 118
- Java Network Progr. & Distributed 255
- Core Java 2 : 405

# ► Overview

- **R**emote **M**ethod **I**nvocation (**RMI**) is a **distributed systems technology** that allows one Java Virtual Machine (JVM) to invoke object methods that will be run on another JVM located elsewhere on a network.
- **RMI** is a Java technology that allows one JVM to communicate with another JVM and have it execute an object method. Objects can invoke methods on other objects located remotely as easily as if they were on the local host machine
- Each RMI service is defined by an **interface**, which describes object methods that can be executed **remotely**. This interface must be shared by all developers who will write software for that service—it acts as a blueprint for applications that will use and provide implementations of the service.

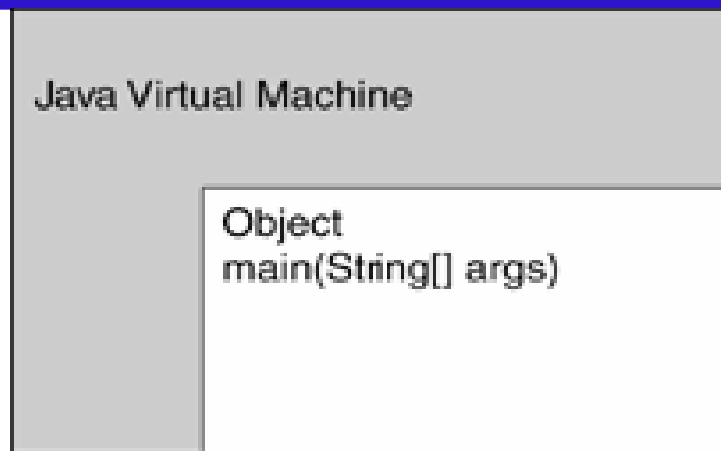
# ► Overview

09-2006

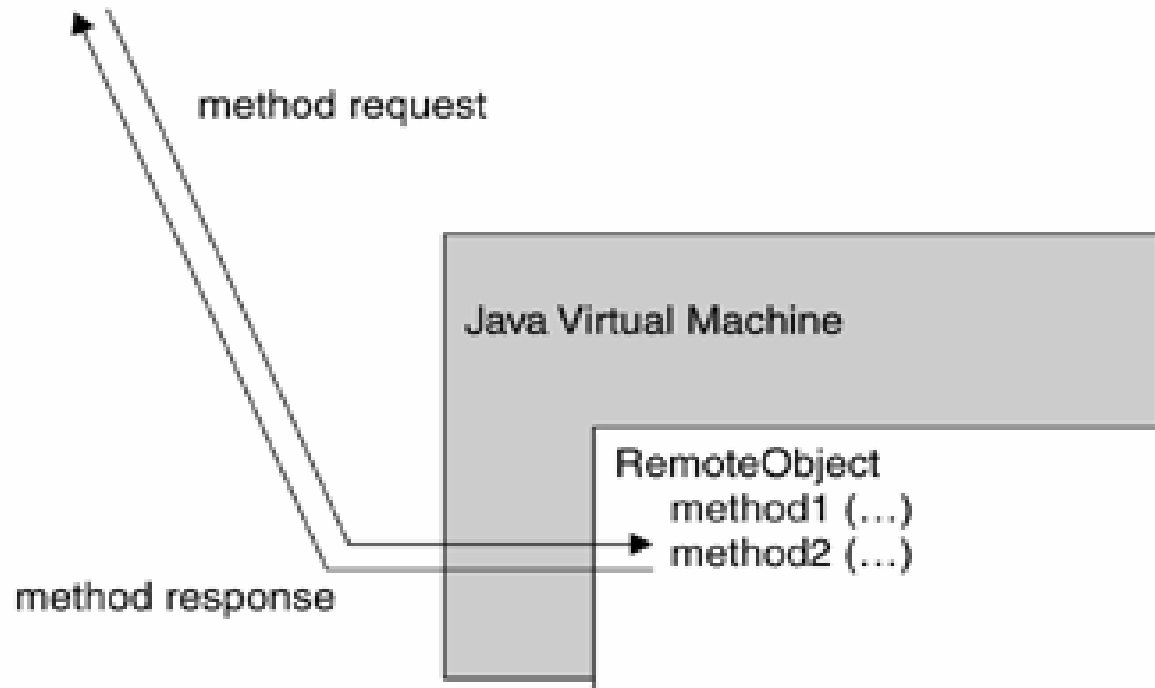
Khoa CNTT

4/37

PHẠM VĂN TÍNH

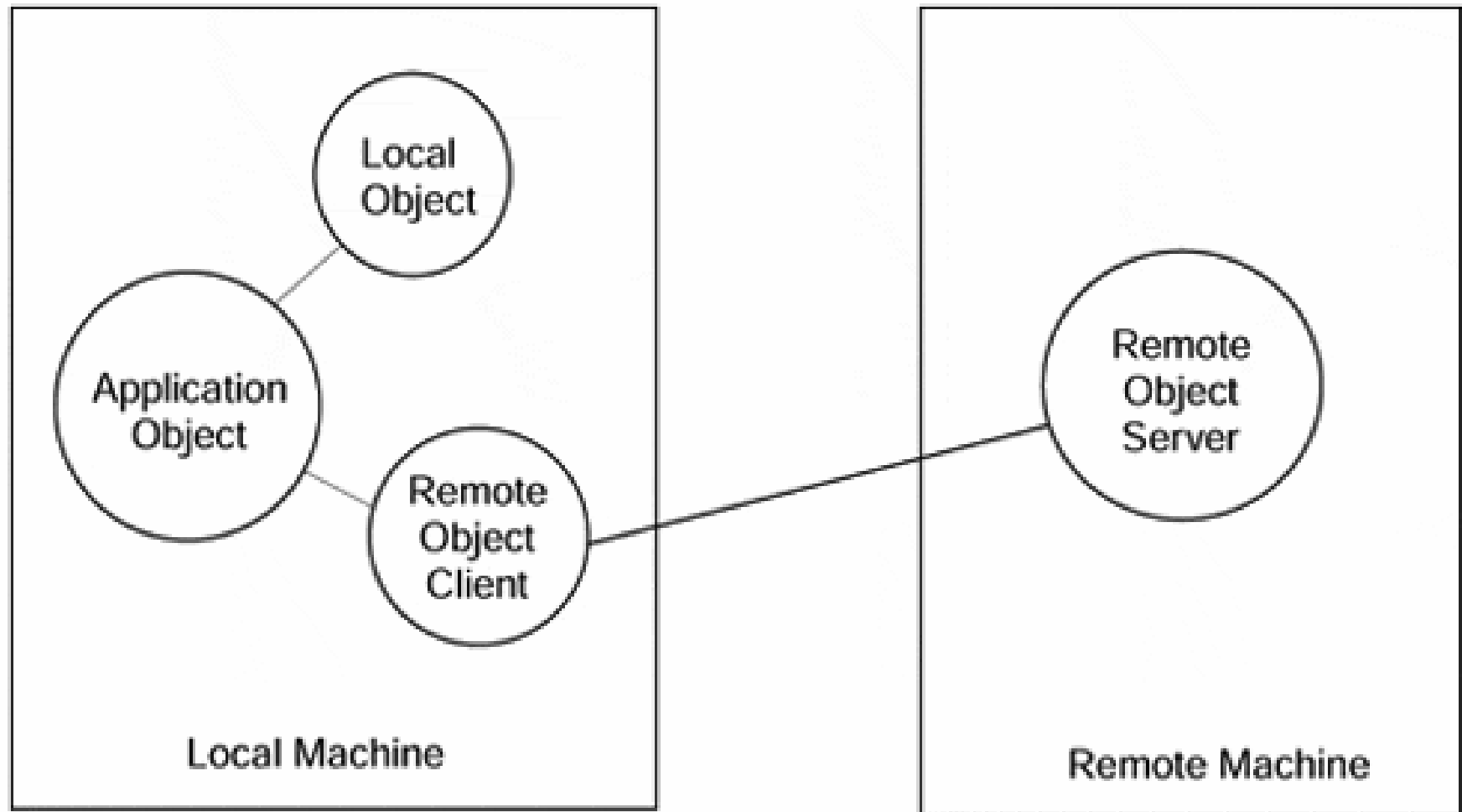


**Invocation of a method on a remote object, executing on a remote machine**



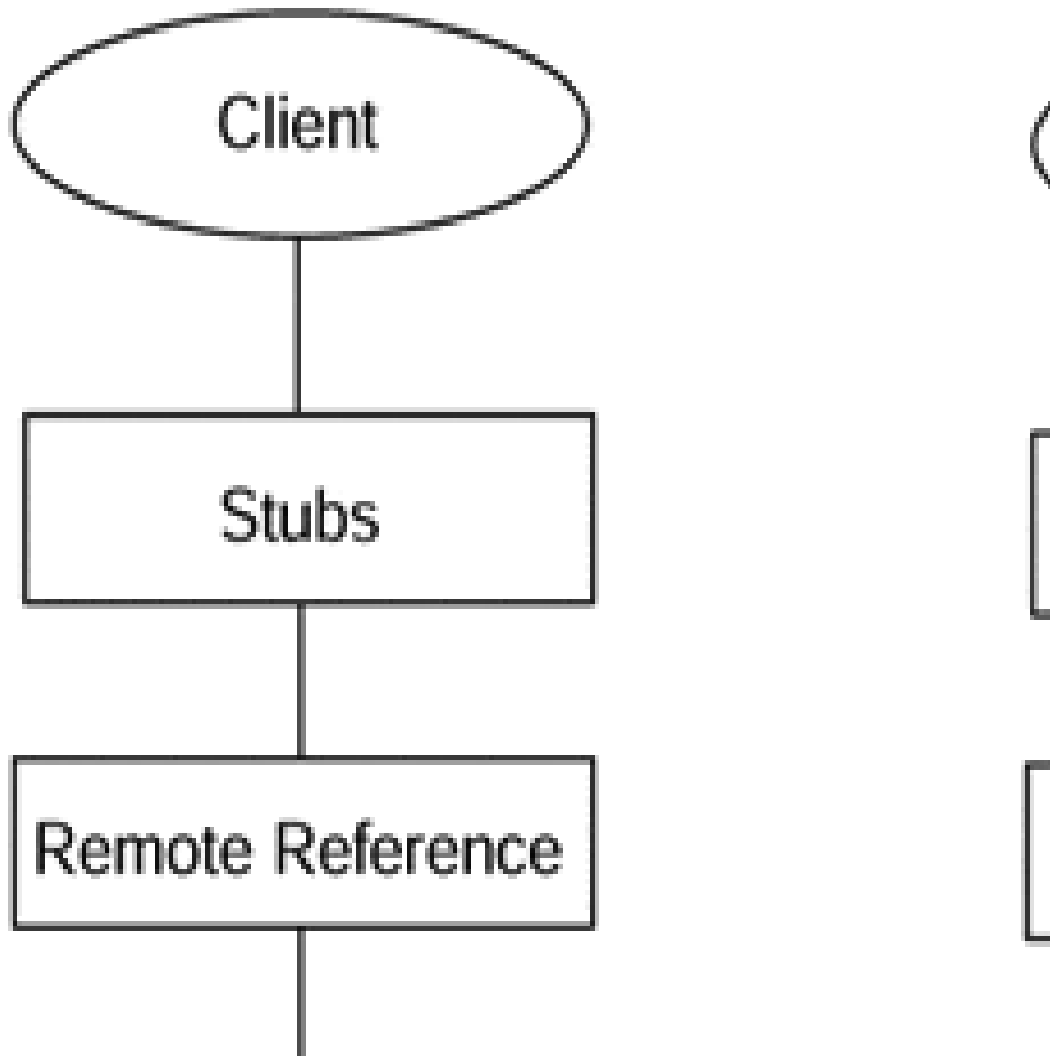


# ► Overview



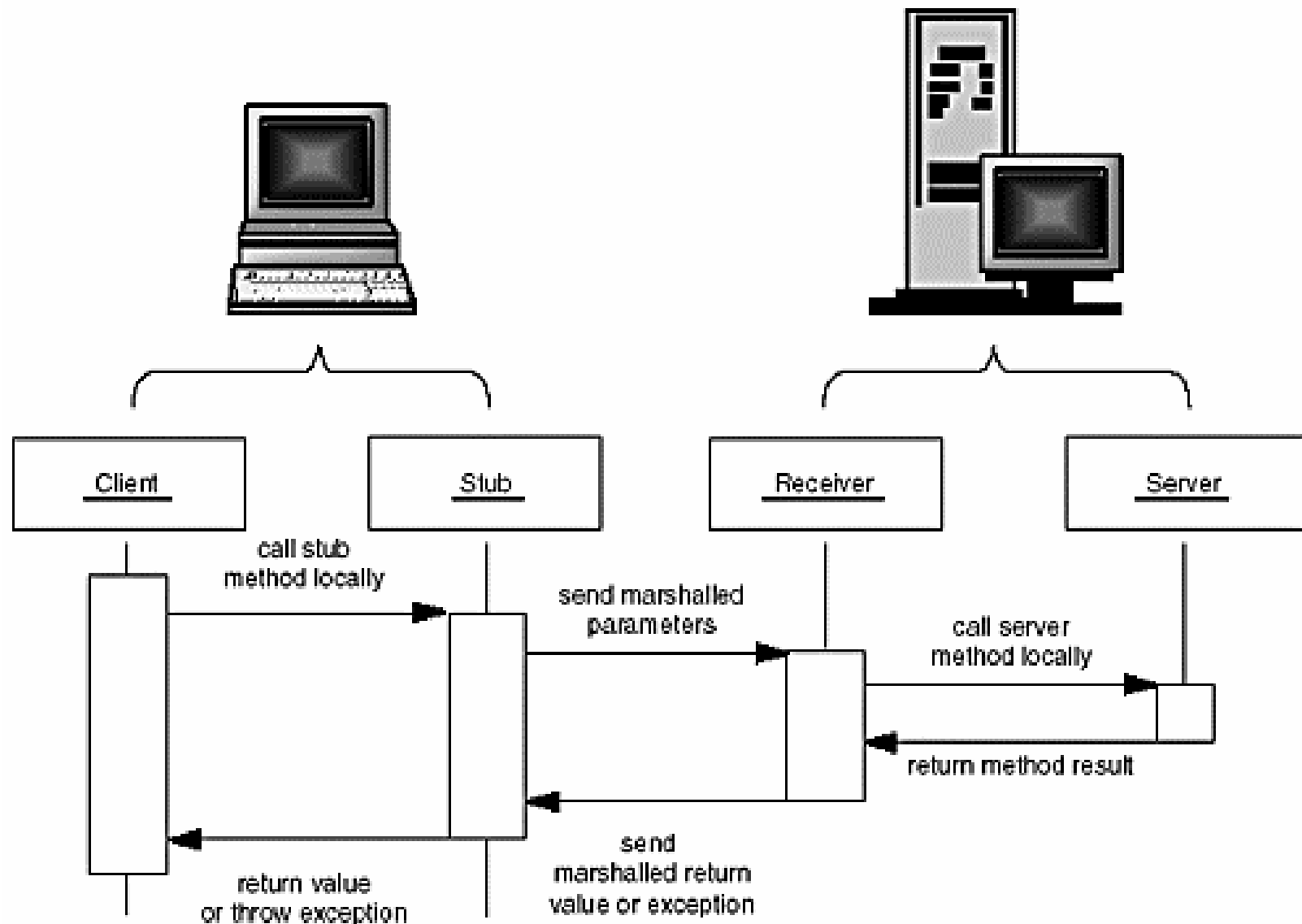
**Invocations on remote objects appear the same as invocations on local objects.**

# ► Overview



**Java RMI architecture.**

# ▶ Parameter marshalling



# ► Stubs and Parameter Marshalling

- When client code wants to invoke a remote method on a remote object, it actually calls an ordinary method of the Java programming language that is encapsulated in a surrogate object called a **stub**. The stub resides on the client machine, not on the server. The stub packages the parameters used in the remote method into a block of bytes.
- The process of encoding the parameters is called **parameter marshalling**. The purpose of parameter marshalling is to convert the parameters into a format suitable for transport from one virtual machine to another.

# ► Stubs and Parameter Marshalling

- The stub method on the client builds an information block that consists of:
  - An identifier of the remote object to be used;
  - A description of the method to be called;
  - The marshalled parameters.

The stub then sends this information to the server.

- On the server side, a receiver object performs the following actions for every remote method call:
  - It unmarshals the parameters.
  - It locates the object to be called.
  - It calls the desired method.
  - It captures and marshals the return value or exception of the call.
  - It sends a package consisting of the marshalled return data back to the stub on the client.

The client stub unmarshals the return value or exception from the server. This value becomes the return value of the stub call.

# ► RMI Applications

- **RMI applications** are often comprised of two separate programs: a **server** and a **client**.
- A typical **server** application **creates** some **remote objects**, **makes references** to them accessible, and **waits for clients** to invoke methods on these remote objects.
- A typical **client** application **gets** a **remote reference** to one or more remote objects in the server and then invokes methods on them. RMI provides the mechanism by which the server and the client communicate and pass information back and forth.
- Such an application is sometimes referred to as a ***distributed object application***.

# ► Creating Distributed Applications Using RMI

1. Design and implement the components of your distributed application.
2. Compile sources and generate stubs.
3. Make classes network accessible.
4. Start the application.

## Design and Implement the Application Components

- ***Defining the remote interfaces***: A remote interface specifies the methods that can be invoked remotely by a client.
- ***Implementing the remote objects***: Remote objects must implement one or more remote interfaces. The remote object class may include implementations of other interfaces (either local or remote) and other methods (which are available only locally).
- ***Implementing the clients***: Clients that use remote objects can be implemented at any time after the remote interfaces are defined, including after the remote objects have been deployed.

# ► Designing a Remote Interface

- Your client program needs to manipulate server objects, but it doesn't actually have copies of them. The objects themselves reside on the server. The client code must still know what it can do with those objects.
- Any system that uses RMI will use a service interface. The service interface defines the object methods that can be invoked remotely, and specifies parameters, return types, and exceptions that may be thrown.
- All RMI service interfaces extend the `java.rmi.Remote` interface, which assists in identifying methods that may be executed remotely. To define a new interface for an RMI system, you must declare a new interface extending the `Remote` interface and All the methods in those interfaces must also declare that they will throw a `RemoteException`.

```
// shared by client and server
interface Product extends Remote {
    String getDescription() throws
                                RemoteException;
}
```



# ► Designing a Remote Interface

```
package rmidemo;
```

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

```
public interface Product extends Remote{
```

```
    String getDescription() throws
```

```
        RemoteException;
```

```
}
```

# ► Implementing a Remote Interface

- In general the implementation class of a remote interface should at least
  - Declare the remote interfaces being implemented
  - Define the constructor for the remote object
  - Provide an implementation for each remote method in the remote interfaces
- The server needs to create and to install the remote objects. This setup procedure can be encapsulated in a main method in the remote object implementation class itself, or it can be included in another class entirely. The setup procedure should
  - Create and install a security manager
  - Create one or more instances of a remote object
  - Register at least one of the remote objects with the RMI remote object registry (or another naming service such as one that uses JNDI), for bootstrapping purposes

# ► Implementing a Remote Interface

```
import java.rmi.*;
import java.rmi.server.*;
public class ProductImpl extends UnicastRemoteObject
                                implements Product {
    private String name;
    public ProductImpl(String n) throws RemoteException {
        super();
        name = n;
    }
    public String getDescription() throws RemoteException {
        return "I am a " + name + ". Buy me!";
    }
}
```

# ► Naming conventions for RMI classes

- **Product** A remote interface
- **ProductImpl** A server class implementing that interface
- **ProductImpl\_Stub** A stub class that is automatically generated by the **rmic** program
- **ProductImpl\_Skel** A skeleton class that is automatically generated by the **rmic** program needed for SDK 1.1
- **ProductServer** A server program that creates server objects
- **ProductClient** A client program that calls remote methods

# ► Creating Stub and Skeleton Classes

09-2006

Khoa CNTT

17/37

PHẠM VĂN TÍNH

The screenshot shows the JBuilder 2005 IDE interface. The title bar reads 'JBuilder 2005 - D:/BaiGiang/LTMang1/Curriculum04/RMIDemo/src/rmidemo'. The menu bar includes File, Edit, Search, Refactor, View, Project, Run, Team, Enterprise, Tools, and Window. The toolbar contains icons for file operations and development tools. The 'Project' window on the left shows a tree view of the project structure: RMIDemo.jpx, <Project Source>, rmidemo (expanded), Product.java, ProductClient.java, **ProductImpl.java** (selected and circled in red), ProductServer.java, Additional Settings, Application, Executable JAR, and ProductImpl\_Stub.class. The 'Properties for ProductImpl.java' dialog is open, showing the 'Build' tab. Under 'RMI compiler settings', the checkbox 'Generate RMI stub/skeleton' is checked and circled in red. The 'Options' field for RMI settings contains '-v1.2'. Under 'JNI compiler settings', the checkbox 'Generate JNI header file' is unchecked. The 'Options' field for JNI settings is empty. At the bottom of the IDE, a code editor shows a snippet of Java code: `catch(Exception e)`.

# ► Creating an RMI Server

```
import java.rmi.*;
import java.rmi.server.*;
public class ProductServer {
    public static void main(String args[]) {
        try {
            System.out.println("Constructing server implementations...");
            // Load the service
            ProductImpl EOS350D = new ProductImpl("Canon EOS 350D");
            // Examine the service, to see where it is stored
            RemoteRef location = EOS350D.getRef();
            System.out.println (location.remoteToString());
            // Load the service
            ProductImpl nikon70D = new ProductImpl("Nikon 70D");
            // Examine the service, to see where it is stored
            location = nikon70D.getRef();
            System.out.println (location.remoteToString());
        }
    }
}
```

# ► Creating an RMI Server

```
System.out.println ("Binding server implementations to registry...");
```

```
// Registration format: rmi://registry_hostname :port /service
```

```
// Note the :port field is optional
```

```
// Register with service so that clients can find us
```

```
Naming.rebind("rmi://localhost/EOS350D", EOS350D);
```

```
Naming.rebind("rmi://localhost/nikon70D", nikon70D);
```

```
System.out.println ("Waiting for invocations from clients...");
```

```
}  
catch (RemoteException re) {  
    System.err.println ("Remote Error - " + re);  
}
```

```
catch (Exception e) {  
    System.out.println(e.getMessage());  
}
```

```
}
```

```
}
```

## ► Starting the server

- Our server program isn't quite ready to run, yet. Because it uses the bootstrap **RMI registry**, that **service must be available**. To start the RMI registry under **UNIX**, you execute the statement  
**rmiregistry &**
- Under **Windows**, call  
**start rmiregistry**
- at a DOS prompt or from the Run dialog box. (The start command is a Windows command that starts a program in a new window.)
- Now you are ready to start the server. Under **UNIX**, use the command:  
**java ProductServer &**
- Under **Windows**, use the command:  
**start java ProductServer**



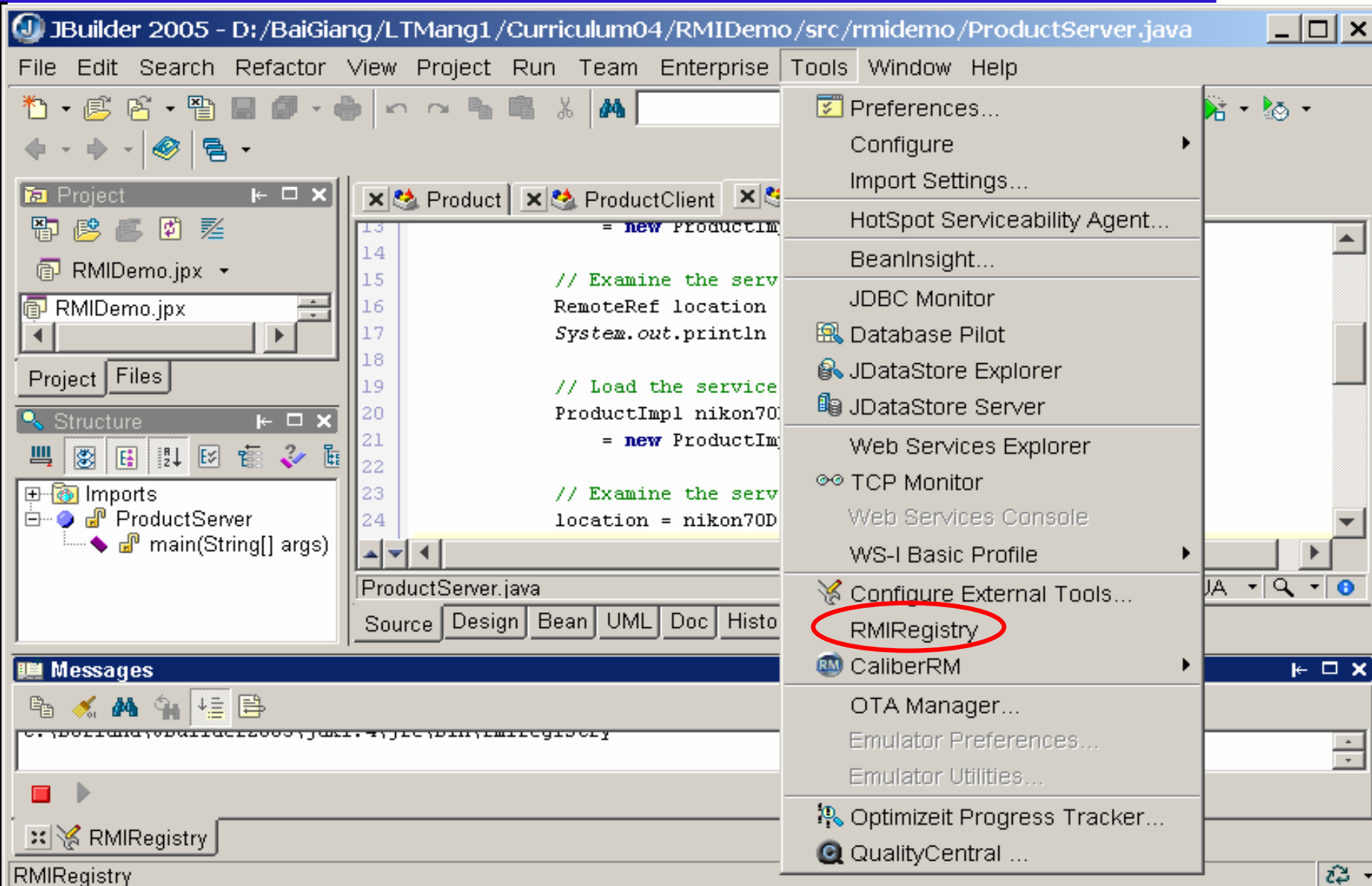
# ▶ Starting the server

09-2006

Khoa CNTT

21/37

PHẠM VĂN TÍNH



# ► Creating an RMI Client

```
public class ProductClient {
    public static void main(String[] args) {
        System.setProperty("java.security.policy", "client.policy");
        System.setSecurityManager(new RMISecurityManager());
        // Registration format: rmi://registry_hostname :port /service
        // Note the :port field is optional
        String url = "rmi://localhost/";
        try{
            // Lookup the service in the registry, and obtain a remote service
            Remote remoteService1 = Naming.lookup(url + "EOS350D");
            Remote remoteService2 = Naming.lookup(url + "nikon70D");
            // Cast to a Product interface
            Product c1 = (Product) remoteService1;
            Product c2 = (Product) remoteService2;
            System.out.println(c1.getDescription());
            System.out.println(c2.getDescription());
        }
        catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

# ▶ Running the client

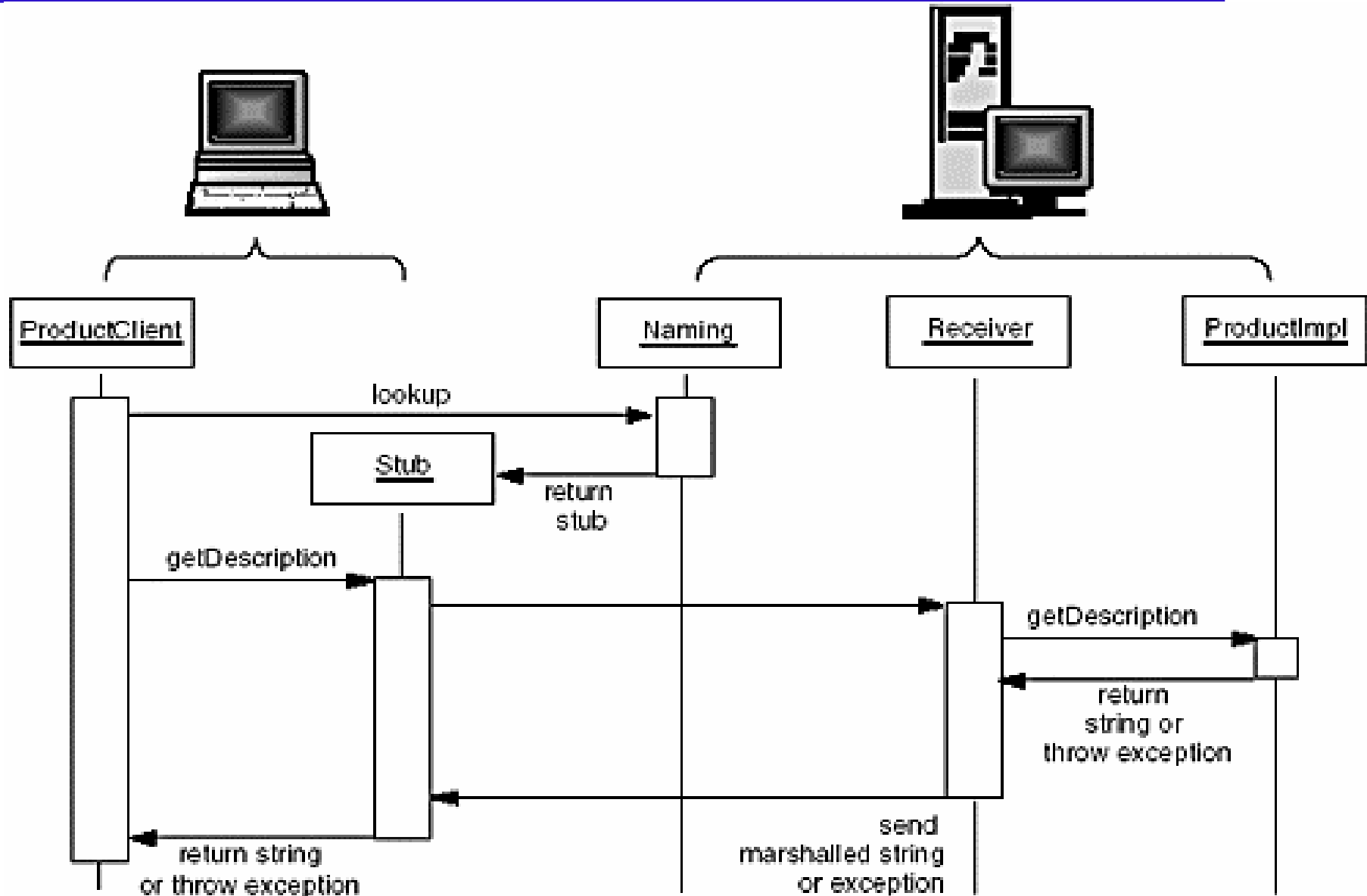
- By default, the `RMI Security Manager` restricts all code in the program from establishing network connections. But the program needs to make network connections
  - To reach the RMI registry;
  - To contact the server objects.
- To allow the client to connect to the RMI registry and the server object, you need to supply a *policy file*.. Here is a policy file that allows an application to make any network connection to a port with port number at least 1024. (The RMI port is 1099 by default, and the server objects also use ports 1024.)

```
grant {  
    permission java.net.SocketPermission  
        "*:1024-65535", "connect";  
};
```

- In the client program, we instruct the security manager to read the policy file, by setting the `java.security.policy` property to the file name.

```
System.setProperty("java.security.policy", "client.policy")
```

# ▶ Calling the remote getDescription method



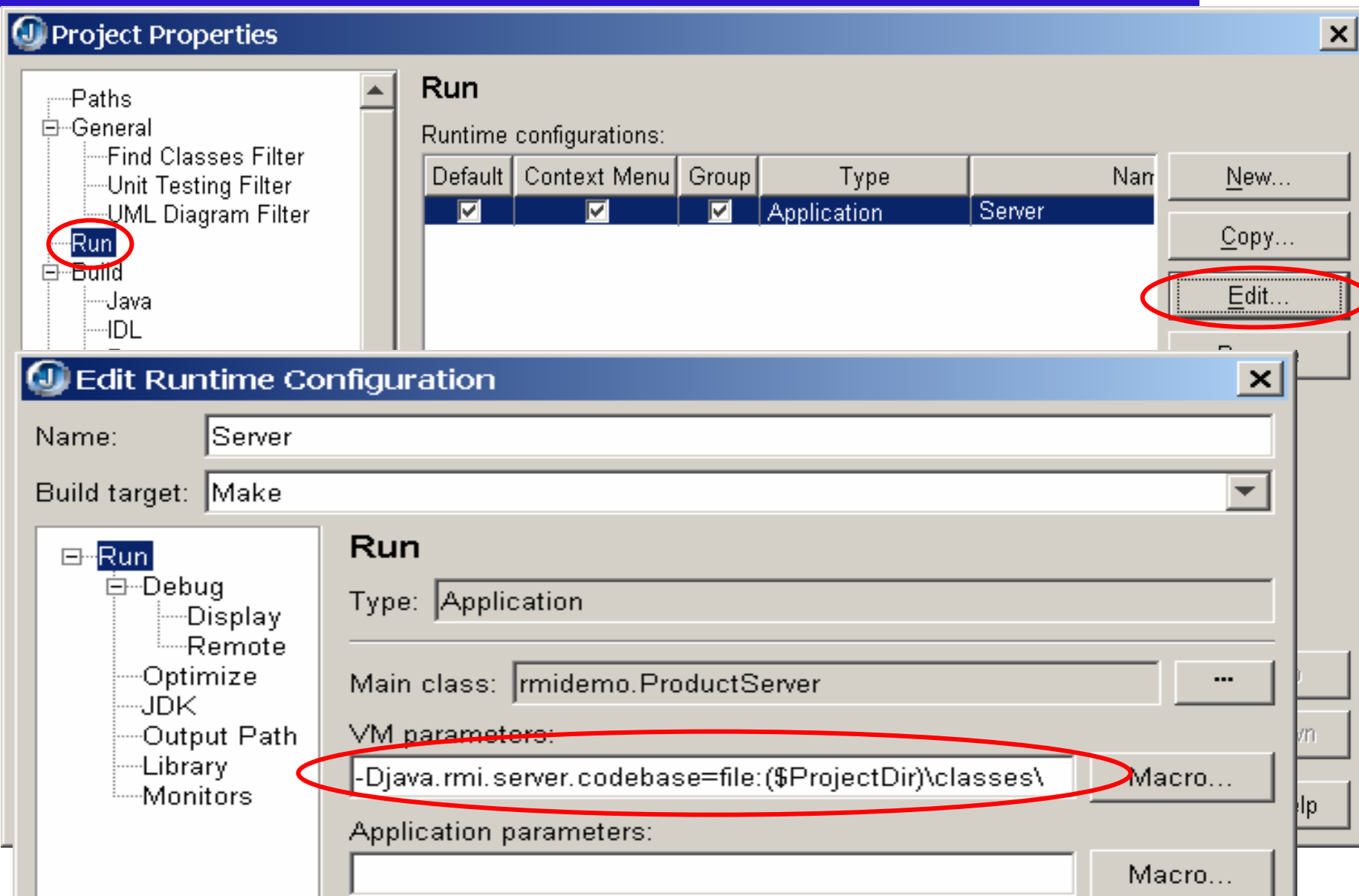
# ► **Creating RMI Program Step by Step**

## **Steps to run this sample:**

- 1. Compile the project** (Project | Rebuild project "SimpleRMI.jpj").
- 2. Start the RMI registry** (Tools | RMIRegistry).
- 3. Start the server class** (check VM `java.rmi.server.codebase` and `java.rmi.server.policy` settings, Project | Project properties...).
- 4. Run the client program** (right-click SimpleRMIClient.java, select Run).  
Or use the runtime configuration, SimpleRMIClient for Unix or Windows

# ▶ Start the server class

09-2006  
Khoa CNTT  
26/37  
PHẠM VĂN TÍNH



# ► 1. Create an interface

- The interface created for this example is **SimpleRMInterface.java**. It contains only one method; the method takes no arguments and returns an object of type `java.util.Date`. Note two things about this interface:
  1. It extends the **java.rmi.Remote** interface (all interfaces used in RMI must do this).
  2. The method throws a **java.rmi.RemoteException** (every method in a remote object's interface must specify this exception in its "throws" clause; this exception is a superclass of all RMI exceptions that can be thrown).

## ► 2. Create a class that implements the interface.

- In this example, the implementation is found in `SimpleRMImpl.java`. This class must extend `java.rmi.UnicastRemoteObject` and must implement the interface you created in step 1. In this example, the only method that needs to be implemented is `getDate()`, which returns the current date and time on the system. Note 2 things about the constructor:

1. The call to `super()`.

2. The call to `Naming.rebind(name, this)`. This call informs the RMI registry that this object is available with the name given in "String name".

- Other than that, this object simply implements all the methods declared in the interface



### ► 3. Create a server that creates an instance of the "impl" class.

- In this example, the server class is **SimpleRMIServer.java**. In this case, the server is pretty simple. It does 2 things:
  1. **Installs a new RMISecurityManager** (Note that RMI uses a different security manager from the security manager used for applets).
  2. **Creates an instance of the SimpleRMImpl class, and gives it the name "SimpleRMImpl instance"**. The SimpleRMImpl object takes care of registering the object with the RMI registry. After this code is run, the object will be available to remote clients as **"//<your server here>/ SimpleRMImpl instance"**. (and in fact, this is how the client connects to it in step 4).

## ► Steps 4 - 6

4. Create a client that connects to the server object using `Naming.lookup()`.  
In this example, the client class is `SimpleRMIClient.java`. The client first installs a new RMI Security Manager (see previous step), then uses the static method `Naming.lookup()` to get a reference to the remote object. Note that the client is **using the *interface* to hold the reference and make method calls**. You should make sure you've created your interface before you try to build the client.
5. Set the compile options on the RMI implementation file. In the IDE, Right-click the `SimpleRMImpl.java` file in the navigator pane and choose "Properties..." In the properties dialog, check "**Generate RMI stub/skeleton**", enter "**-v1.2**" in the Options field, and click OK.
6. Compile the project by selecting **Project | Rebuild project "SimpleRMI.jpj"** from the menu.

## ► 7. Start the RMI registry.

- Now you're setting up the environment so that you can run it. From within JBuilder, you can select **Tools | RMIRegistry** to start the RMI registry. The next time you look at the tools menu, you will notice there is a check next to this menu option. This is your clue that the registry is running. Selecting it again will close the RMI registry. Alternatively, you can start the RMI registry from the command-line. The name of the executable varies by platform, but typically begins with "**rmiregistry**" and resides in the **jre/bin** directory of your JDK installation.
- **Note: The RMI registry must be started before you can start your server.**

## ► 8. Start the RMI server program.

- It is necessary to grant an RMI server special security rights in order for it to listen for and accept client RMI requests over a network. Typically, these rights are specified in a java security policy file defined by a special property (**java.security.policy**) passed via a command-line argument to the VM of the server. Similarly, a property specifying the location of classes used by the server (**java.rmi.server.codebase**) is also passed to the server's VM via a command-line argument. Before running the server, select **Project | Project properties...** and select the **"Run" tab** of the dialog to inspect the parameters passed to the server's VM.
- Next, select **"Run | Run project"** from the menu to start the rmi server. When the message "SimpleRMImpl ready" appears, the server is ready to accept client requests. A message like **"connection refused"** indicates that either your **rmiregistry** is not running, or the **java.security.policy** parameter does not point to the proper location of the **SimpleRM.policy** file.

## ► 9. Run the RMI client program.

- It is necessary to grant an RMI client special security rights in order for it to connect to an RMI server over a network. Typically, these rights are specified in a java security policy file defined by a special property (**java.security.policy**) passed via a command-line argument to the VM of the client. Before running the client, select **Run | Configurations... | "Run SimpleRMI Client" | <Edit>** to inspect the parameters passed to the client's VM.
- Next, to run the client.
- **SimpleRMIClient** asks the **RMI** registry to lookup a reference to the rmi object named "**SimpleRMImpl** instance" created by **SimpleRMIServer**. After **SimpleRMIClient** then uses this reference to invoke the method declared in the **SimpleRMIInterface** interface just as if the object were a local object. It receives a **java.util.Date** object from the server (indicating the current time on the server) and displays its contents.

# ► RMI Applications for JDK 1.5

pregenerating a stub class for a remote object's class is **only required** if the remote object needs to support **pre-5.0 clients**. As of **the 5.0 release**, if a pregenerated stub class for a remote object's class cannot be loaded when the remote object is exported, **the remote object's stub class is generated dynamically**.

- Designing a Remote Interface
- Implementing a Remote Interface
- Creating a RMI Server
- Createing a RMI Client

# ► RMI Applications for JDK 1.5

## Creating a RMI Server

- Installs a new RMISecurityManager  
`System.setProperty("java.security.policy", "client.policy");`  
`System.setSecurityManager(new RMISecurityManager());`

Both the server and client applications use a security policy file that grants permissions only to files in the local class path (the current directory). The server application needs permission to accept connections, and both the server and client applications need permission to make connections. The permission `java.net.SocketPermission` is granted to the specified codebase URL, a "file:" URL relative to the current directory. This permission grants the ability to both accept connections from and make connections to any host on unprivileged ports (that is ports  $\geq 1024$ ).

- `grant codeBase "file:." {`  
    `permission java.net.SocketPermission "*:1024-",`  
    `"connect,accept";`  
`};`

# ► RMI Applications for JDK 1.5

## Creating a RMI Server

- Creates and exports a Registry instance on the local host that accepts requests on the specified port  
`Registry RMIReg =  
LocateRegistry.createRegistry(1099);`
- Creates a remote object  
`ProductImpl EOS350D = new  
ProductImpl("Canon EOS 350D");`
- Binds a remote reference to the specified name in this registry  
`RMIReg.bind("EOS350D", EOS350D);`



# ► RMI Applications for JDK 1.5

## Creating a RMI Client

- Installs a new RMI Security Manager  
`System.setProperty("java.security.policy", "client.policy");`  
`System.setSecurityManager(new RMISecurityManager());`
- Gets a reference to the remote object Registry on the specified host and port. If host is null, the local host is used.  
`Registry RMIReg =`  
`LocateRegistry.getRegistry("localhost", 1099);`
- Looks up the service in the registry, and obtains a remote service  
`Remote remoteService1 =`  
`RMIReg.lookup("EOS350D");`
- Cast to a Remote (Product) interface  
`Product c1 = (Product) remoteService1;`

# ► Scrollable and Updatable Result Sets

# ► Scrollable and Updatable Result Sets

- The most useful improvements in JDBC 2 are in the **ResultSet** class. As you have seen, the **next** method of the **ResultSet** class iterates over the rows in a result set.
- You usually want the user to be able to move both forward and backward in the result set. But in JDBC 1, there was no **previous** method. Programmers who wanted to implement backwards iteration had to manually cache the result set data. The **scrolling** result set in JDBC 2 lets you move forward and backward through a result set and jump to any position in the result set.
- Furthermore, once you display the contents of a result set to users, they may be tempted to edit it. If you supply an editable view to your users, you have to make sure that the user edits are posted back to the database. In JDBC 1, you had to program **UPDATE** statements. In JDBC 2, you can simply update the result set entries, and the database is automatically updated.

# ► Scrollable Result Sets (JDBC 2)

- To obtain scrolling result sets from your queries, you must obtain a different **Statement** object with the method  
**Statement stat = conn.createStatement(type, concurrency);**
- For a prepared statement, use the call  
**PreparedStatement stat = conn.prepareStatement(command, type, concurrency);**
- **ResultSet type** values:
  - TYPE\_FORWARD\_ONLY** : The result set is not scrollable.
  - TYPE\_SCROLL\_INSENSITIVE** : The result set is scrollable but not sensitive to database changes.
  - TYPE\_SCROLL\_SENSITIVE** : The result set is scrollable and sensitive to database changes.
- **ResultSet concurrency** values:
  - CONCUR\_READ\_ONLY** : The result set cannot be used to update the database.
  - CONCUR\_UPDATABLE** : The result set can be used to update the database.

## ► Scrollable Result Sets (JDBC 2)

- For example, if you simply want to be able to scroll through a result set but you don't want to edit its data, you use:
- **Statement stat =**  
**conn.createStatement(ResultSet.TYPE\_SCROLL\_INSENSITIVE, ResultSet.CONCUR\_READ\_ONLY);**
- All result sets that are returned by method calls  
**ResultSet rs = stat.executeQuery(query)**  
are now scrollable. A scrolling result set has a *cursor* that indicates the current position.
- Scrolling is very simple. You use
- **if (rs.previous()) . . .**
- to scroll backward. The method returns **true** if the cursor is positioned on an actual row; **false** if it now is positioned before the first row.

## ► Scrollable Result Sets (JDBC 2)

- You can move the cursor backward or forward by a number of rows with the command
- `rs.relative(n);`
- If `n` is positive, the cursor moves forward. If `n` is negative, it moves backwards. If `n` is zero, the call has no effect. If you attempt to move the cursor outside the current set of rows, then, the method returns `false` and the cursor does not move. The method returns `true` if the cursor landed on an actual row.
- Alternatively, you can set the cursor to a particular row number:
- `rs.absolute(n);`
- You get the current row number with the call
- `int n = rs.getRow();`
- The first row in the result set has number 1. If the return value is 0, the cursor is not currently on a row—it is either before the first or after the last row.

## ► Updatable Result Sets (JDBC 2)

- If you want to be able to edit result set data and have the changes automatically reflected in the database, you need to create an updatable result set.
- **Statement stat =**  
**conn.createStatement(ResultSet.TYPE\_SCROLL\_INSENSITIVE, ResultSet.CONCUR\_UPDATABLE);**
- Then, the result sets returned by a call to **executeQuery** are updatable.
- **NOTE: Not all queries return updatable result sets.** If your query is a join that involves multiple tables, the result may not be updatable.
- For example, suppose you want to raise the prices of some books, but you don't have a simple criterion for issuing an **UPDATE** command. Then, you can iterate through all books and update prices, based on arbitrary conditions.

## ► Updatable Result Sets (JDBC 2)

- ```
String query = "SELECT * FROM Books";  
ResultSet rs = stat.executeQuery(query);  
while (rs.next()) {  
    if (. . .){  
        double increase = . . .  
        double price = rs.getDouble("Price");  
        rs.updateDouble("Price", price + increase);  
        rs.updateRow();  
    }  
}
```
- There are **updateXxx** methods for all data types that correspond to SQL types, such as **updateDouble**, **updateString**, and so on. As with the **getXxx** methods, you specify the name or the number of the column. Then, you specify the new value for the field.



## ► Updatable Result Sets (JDBC 2)

- The **updateXxx** method only changes the row values, not the database. When you are done with the field updates in a row, you must call the **updateRow** method. That method sends all updates in the current row to the database.
- If you move the cursor to another row without calling **updateRow**, all updates are discarded from the row set and they are never communicated to the database. You can also call the **cancelRowUpdates** method to cancel the updates to the current row.
- If you want to add a new row to the database, you first use the **moveToInsertRow** method to move the cursor to a special position, called the *insert row*. You build up a new row in the insert row position by issuing **updateXxx** instructions. Finally, when you are done, call the **insertRow** method to deliver the new row to the database. When you are done inserting, call **moveToCurrentRow** to move the cursor back to the position before the call to **moveToInsertRow**.

## ► Updatable Result Sets (JDBC 2)

- `rs.moveToInsertRow();`
- `rs.updateString("Title", title);`
- `rs.updateString("ISBN", isbn);`
- `rs.updateDouble("Price", price);`
- `rs.insertRow();`
- `rs.moveToCurrentRow();`
- Finally, you can delete the row under the cursor.
- `rs.deleteRow();`
- The `deleteRow` method immediately removes the row from both the result set and the database.
- The `updateRow`, `insertRow`, and `deleteRow` methods of the `ResultSet` class give you the same power as executing **UPDATE**, **INSERT**, and **DELETE** SQL commands. However, programmers who are used to the Java programming language will find it more natural to manipulate the database contents through result sets than by constructing SQL statements.

# ▶ **javax.sql.Connection**

- **Statement** `createStatement(int type, int concurrency)`
- **PreparedStatement** `prepareStatement(String command, int type, int concurrency)`
- (JDBC 2) create a statement or prepared statement that yields result sets with the given type and concurrency.
- **Parameters:**
  - command** : the command to prepare
  - type** : one of the constants  
`TYPE_FORWARD_ONLY`,  
`TYPE_SCROLL_INSENSITIVE`, or  
`TYPE_SCROLL_SENSITIVE`
  - concurrency** : one of the constants  
`CONCUR_READ_ONLY` or  
`CONCUR_UPDATABLE`

# ► **java.sql.ResultSet**

- **int getType()**  
(JDBC 2) returns the type of this result set, one of **TYPE\_FORWARD\_ONLY**, **TYPE\_SCROLL\_INSENSITIVE**, or **TYPE\_SCROLL\_SENSITIVE**.
- **int getConcurrency()**  
(JDBC 2) returns the concurrency setting of this result set, one of **CONCUR\_READ\_ONLY** or **CONCUR\_UPDATABLE**.
- **boolean previous()**  
(JDBC 2) moves the cursor to the preceding row. Returns **true** if the cursor is positioned on a row.
- **int getRow()**  
(JDBC 2) gets the number of the current row. Rows are numbered starting with 1.
- **boolean absolute(int r)**  
(JDBC 2) moves the cursor to row **r**. Returns **true** if the cursor is positioned on a row.

# ► **java.sql.ResultSet**

- **boolean relative(int d)**  
(JDBC 2) moves the cursor by **d** rows. If **d** is negative, the cursor is moved backward. Returns **true** if the cursor is positioned on a row.
- **boolean first()**
- **boolean last()**  
(JDBC 2) move the cursor to the first or last row. Return **true** if the cursor is positioned on a row.
- **void beforeFirst()**
- **void afterLast()**  
(JDBC 2) move the cursor before the first or after the last row.
- **boolean isFirst()**
- **boolean isLast()**  
(JDBC 2) test if the cursor is at the first or last row.
- **boolean isBeforeFirst()**
- **boolean isAfterLast()**  
(JDBC 2) test if the cursor is before the first or after the last row.

# ► **java.sql.ResultSet**

- **void moveToInsertRow()**  
(JDBC 2) moves the cursor to the *insert row*. The insert row is a special row that is used for inserting new data with the **updateXxx** and **insertRow** methods.
- **void moveToCurrentRow()**  
(JDBC 2) moves the cursor back from the insert row to the row that it occupied when the **moveToInsertRow** method was called.
- **void insertRow()**  
(JDBC 2) inserts the contents of the insert row into the database and the result set.
- **void deleteRow()**  
(JDBC 2) deletes the current row from the database and the result set.
- **void updateXxx(int column, Xxx data)**
- **void updateXxx(String columnName, Xxx data)**  
(**Xxx** is a type such as **int**, **double**, **String**, **Date**, etc.) (JDBC 2) update a field in the current row of the result set.

# ▶ ***java.sql.ResultSet***

- **void updateRow()**  
(JDBC 2) sends the current row updates to the database.
- **void cancelRowUpdates()**  
(JDBC 2) cancels the current row updates.

# ► Metadata

- JDBC can give you additional *information* about the *structure of a database and its tables*. For example, you can get a *list of the tables* in a particular database or the *column names and types of a table*.
- if you design the tables, you know the tables and their structure. Structural information is, however, extremely useful for programmers who *write tools that work with any database*.
- To find out more about the database, you need to request an object of type **DatabaseMetaData** from the database connection.
- **DatabaseMetaData meta = conn.getMetaData();**
- Now you are ready to get some metadata. For example, the call
- **ResultSet rs = meta.getTables(null, null, null, new String[] { "TABLE" });**  
returns a result set that contains information about all tables in the database.



# ► **ResultSetMetaData**

- **ResultSetMetaData** reports information about a result set. Whenever you have a result set from a query, you can inquire about the **number of columns** and each **column's name, type, and field width**. We will make use of this information to make a label for each name and a text field of sufficient size for each value.

```
ResultSet rs = stat.executeQuery("SELECT * FROM " +  
tableName);
```

```
ResultSetMetaData meta = rs.getMetaData();  
for (int i = 1; i <= meta.getColumnCount(); i++) {  
    String columnName = meta.getColumnLabel(i);  
    int columnWidth = meta.getColumnDisplaySize(i);  
    ...  
}
```

## ► ***java.sql.Connection***

---

- DatabaseMetaData getMetaData()  
returns the metadata for the connection as a  
DatabaseMetaData object.

# ▶ **java.sql.DatabaseMetaData**

- **ResultSet getTables(String catalog, String schemaPattern, String tableNamePattern, String types[])**  
gets a description of all tables in a catalog that match the schema and table name patterns and the type criteria. The **catalog** and **schema** parameters can be "" to retrieve those tables without a catalog or schema, or **null** to return tables regardless of catalog or schema.
- The **types** array contains the names of the table types to include. Typical types are **TABLE**, **VIEW**, **SYSTEM TABLE**, **GLOBAL TEMPORARY**, **LOCAL TEMPORARY**, **ALIAS**, and **SYNONYM**. If **types** is **null**, then tables of all types are returned.
- The result set has five columns, all of which are of type String:

|          |                    |                                     |
|----------|--------------------|-------------------------------------|
| <b>1</b> | <b>TABLE_CAT</b>   | Table catalog (may be <b>null</b> ) |
| <b>2</b> | <b>TABLE_SCHEM</b> | Table schema (may be <b>null</b> )  |
| <b>3</b> | <b>TABLE_NAME</b>  | Table name                          |
| <b>4</b> | <b>TABLE_TYPE</b>  | Table type                          |
| <b>5</b> | <b>REMARKS</b>     | Comment on the table                |

## ▶ ***java.sql.DatabaseMetaData***

- `int getJDBCMajorVersion()`
- `int getJDBCMinorVersion()`
- (JDBC 3) Return the major and minor JDBC version numbers of the driver that established the database connection. For example, a JDBC 3.0 driver has major version number 3 and minor version number 0.
- `int getMaxStatements()`
- Returns the maximum number of concurrently open statements per database connection, or 0 if the number is unlimited or unknown.

# ▶ ***java.sql.ResultSet***

- `ResultSetMetaData getMetaData()`  
gives you the metadata associated with the current `ResultSet` columns.

## ► ***java.sql.ResultSetMetaData***

- **int getColumnCount()**  
returns the number of columns in the current **ResultSet** object.
- **int getColumnDisplaySize(int column)**  
tells you the maximum width of the column specified by the index parameter.
- **String getColumnLabel(int column)**  
gives you the suggested title for the column.
- **String getColumnName(int column)**  
gives the column name associated with the column index specified.

# ► Transactions

- The major reason for grouping commands into transactions is *database integrity*. For example, suppose we want to add a new book to our book database. Then, it is important that we simultaneously update the **Books**, **Authors**, and **BooksAuthors** table. If the update were to add new rows into the first two tables but not into the third, then the books and authors would not be properly matched up.
- If you group updates to a transaction, then the transaction either succeeds in its entirety and it can be *committed*, or it fails somewhere in the middle. In that case, you can carry out a *rollback* and the database automatically undoes the effect of all updates that occurred since the last committed transaction.
- By default, a database connection is in *autocommit* mode, and each SQL command is committed to the database as soon as it is executed. Once a command is committed, you cannot roll it back.

# ► Transactions

- To check the current autocommit mode setting, call the **getAutoCommit** method of the **Connection** class. You turn off autocommit mode with the command
- **conn.setAutoCommit(false);**
- Now you create a statement object in the normal way:
- **Statement stat = connection.createStatement();**
- Call **executeUpdate** any number of times:  
**stat.executeUpdate(command1);**  
**stat.executeUpdate(command2);**  
**stat.executeUpdate(command3);**  
**...**
- When all commands have been executed, call the **commit** method:
- **connection.commit();**
- However, if an error occurred, call **conn.rollback();**
- Then, all commands until the last commit are automatically reversed. You typically issue a rollback when your transaction was interrupted by a **SQLException**.



## ► Batch Updates (JDBC 2)

- Suppose a program needs to execute many **INSERT** statements to populate a database table. In JDBC 2, you can improve the performance of the program by using a **batch update**. In a batch update, a sequence of commands is collected and submitted as a batch.
- Use the **supportsBatchUpdates** method of the **DatabaseMetaData** class to find out if your database supports this feature.
- The commands in a batch can be actions such as **INSERT**, **UPDATE**, and **DELETE** as well as data definition commands such as **CREATE TABLE** and **DROP TABLE**.
- You cannot add **SELECT** commands to a batch since executing a **SELECT** statement returns a result set.
- To execute a batch, you first create a **Statement** object in the usual way:

```
Statement stat = conn.createStatement();
```

## ► Batch Updates (JDBC 2)

- Now, instead of calling **executeUpdate**, you call the **addBatch** method:

```
String command = "CREATE TABLE ..."  
stat.addBatch(command);
```

```
while (. . .){  
    command = "INSERT INTO ... VALUES (" + ... + ")";  
    stat.addBatch(command);  
}
```

- Finally, you submit the entire batch.  
**int[] counts = stat.executeBatch();**
- The call to **executeBatch** returns an array of the row counts for all submitted commands.
- **For proper error handling in batch mode, you want to treat the batch execution as a single transaction.** If a batch fails in the middle, you want to roll back to the state before the beginning of the batch.

## ► Batch Updates (JDBC 2)

- First, turn autocommit mode off, then collect the batch, execute it, commit it, and finally restore the original autocommit mode:

```
boolean autoCommit = conn.getAutoCommit();  
conn.setAutoCommit(false);  
Statement stat = conn.createStatement();  
...  
// keep calling stat.addBatch(. . .);  
...  
stat.executeBatch();  
conn.commit();  
conn.setAutoCommit(autoCommit);
```

# ► ***java.sql.Connection***

- **void setAutoCommit(boolean b)**  
sets the autocommit mode of this connection to **b**. If autocommit is true, all statements are committed as soon as their execution is completed.
- **boolean getAutoCommit()**  
gets the autocommit mode of this connection.
- **void commit()**  
commits all statements that were issued since the last commit.
- **void rollback()**  
undoes the effect of all statements that were issued since the last commit.

# ► ***java.sql.Statement***

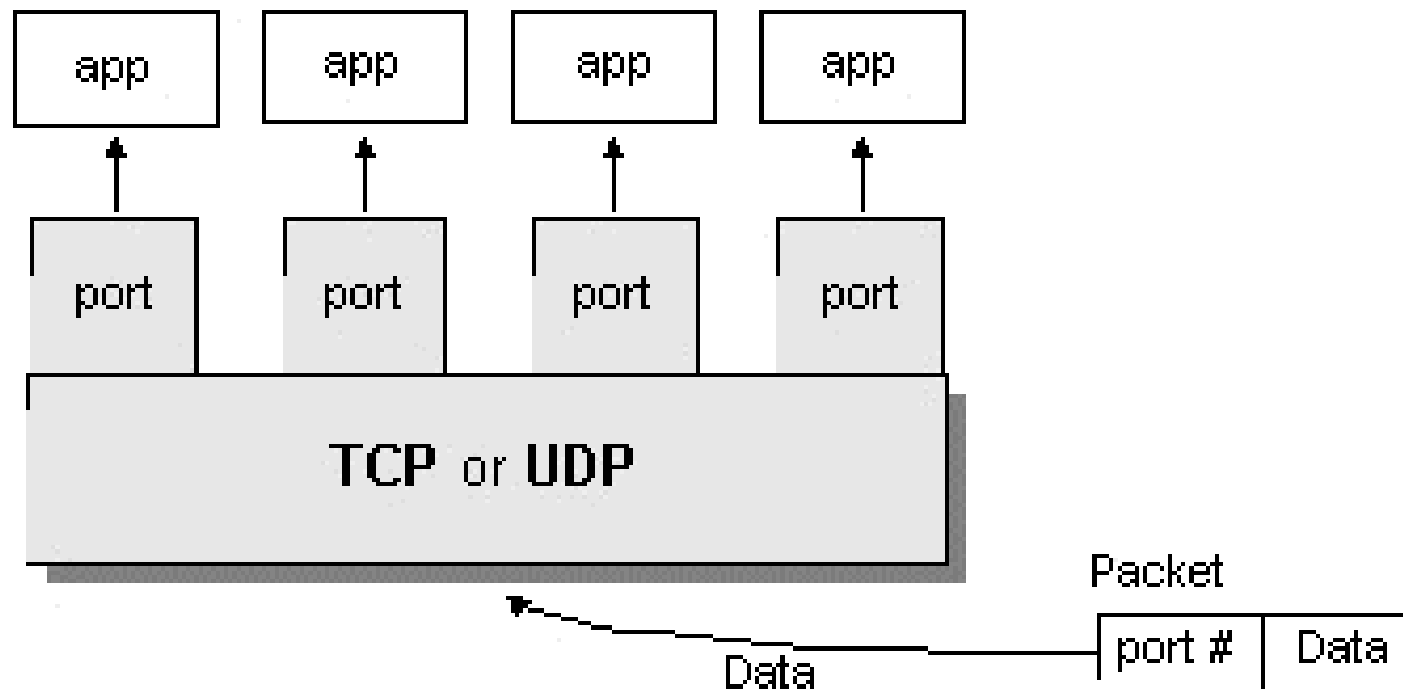
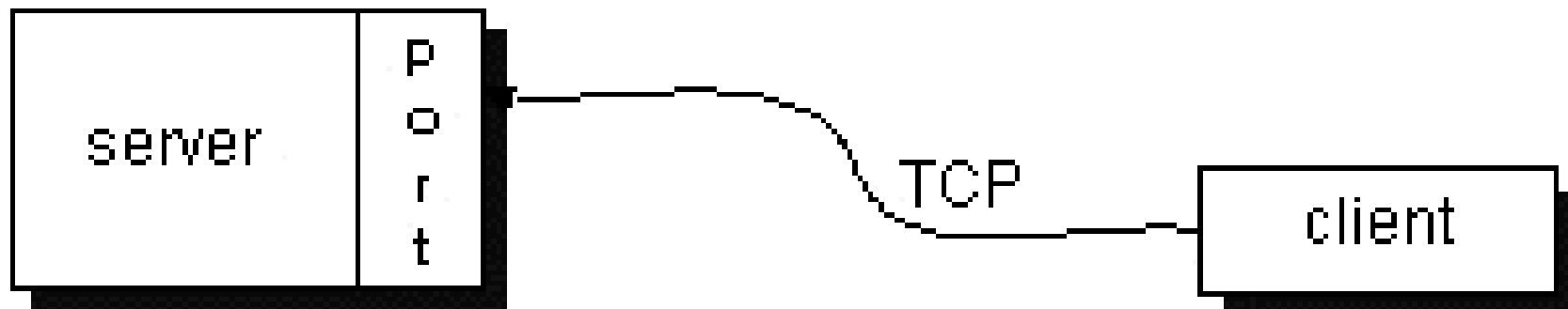
- **void addBatch(String command)**  
(JDBC 2) adds the command to the current batch of commands for this statement.
- **int[] executeBatch()**  
(JDBC 2) executes all commands in the current batch. Returns an array of row counts, containing an element for each command in the batch that denotes the number of rows affected by that command.
- ***java.sql.DatabaseMetaData***
- **boolean supportsBatchUpdates()**  
(JDBC 2) returns true if the driver supports batch updates.

# ▶ **SOCKET PROGRAMMING**

# ► Networking Basics

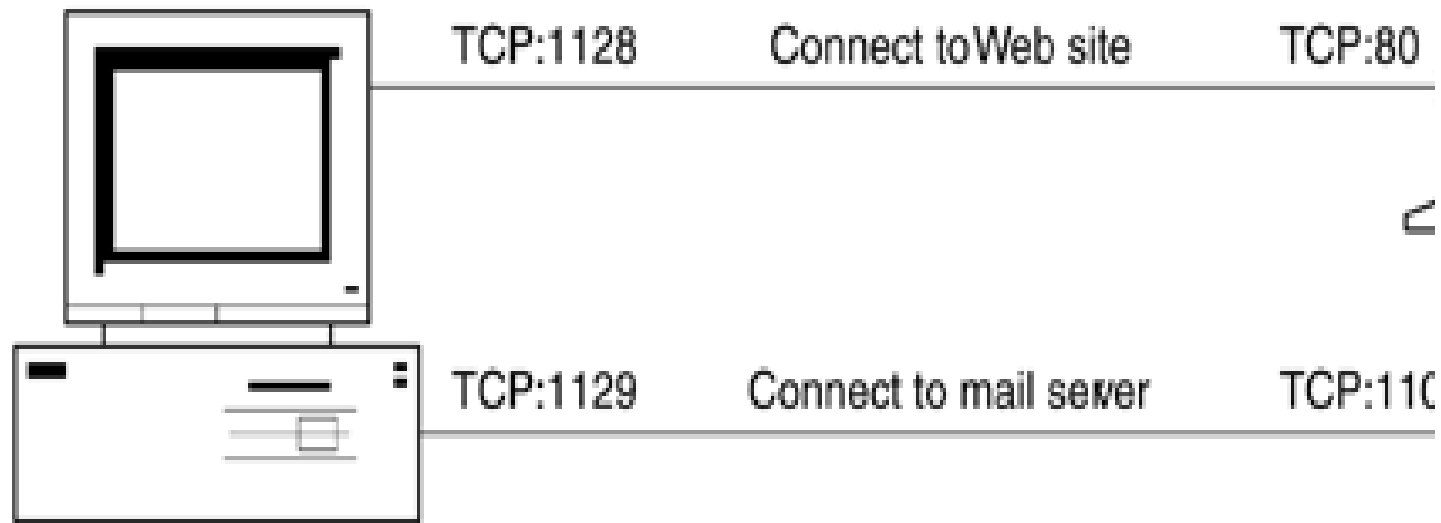
- Computers running on the Internet communicate to each other using either the Transmission Control Protocol (TCP) or the User Datagram Protocol (UDP)
- TCP
  - When two applications want to communicate to each other reliably, they establish a connection and send data back and forth over that connection
  - TCP provides a point-to-point channel for applications that require reliable communications. The Hypertext Transfer Protocol (HTTP), File Transfer Protocol (FTP), and Telnet are all examples of applications that require a reliable communication channel
- UDP
- The UDP protocol provides for communication that is not guaranteed between two applications on the network. UDP is not connection-based like TCP. Rather, it sends independent packets of data, called *datagrams*, from one application to another

# ► Understanding Ports





# ► Communication between Applications Using Ports

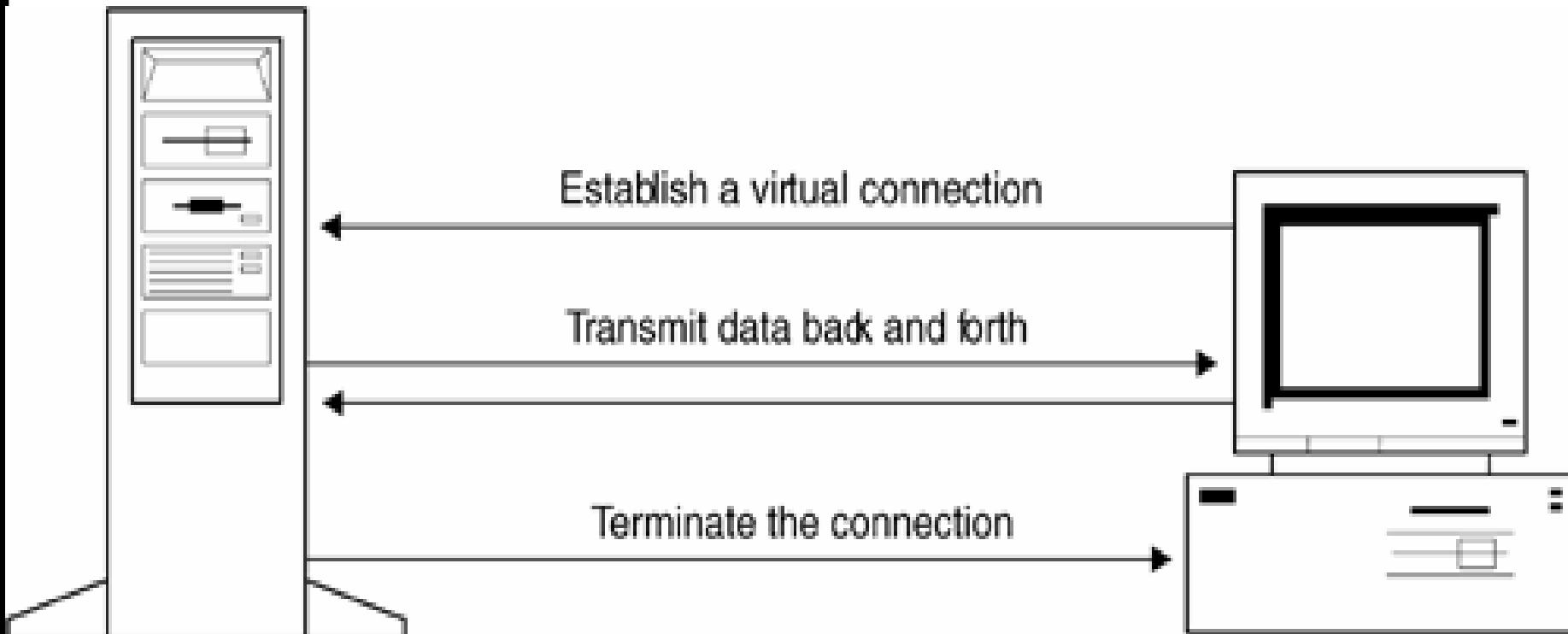


**Local applications** establishing a connection from other programs, allowing multiple TCP applications to run on the same machine.

## ► Some Well-Known Port Services

| Port | Protocol                      | RFC                   |
|------|-------------------------------|-----------------------|
| 13   | DayTime                       | RFC 867               |
| 7    | Echo                          | RFC 862               |
| 25   | SMTP (e-mail)                 | RFC 821 (SMTP)        |
|      |                               | RFC 1869 (Extnd SMTP) |
|      |                               | RFC 822 (Mail Format) |
|      |                               | RFC 1521 (MIME)       |
| 110  | Post Office Protocol          | RFC 1725              |
| 20   | File Transfer Protocol (data) | RFC 959               |
| 80   | Hypertext Transfer Protocol   | RFC 2616              |

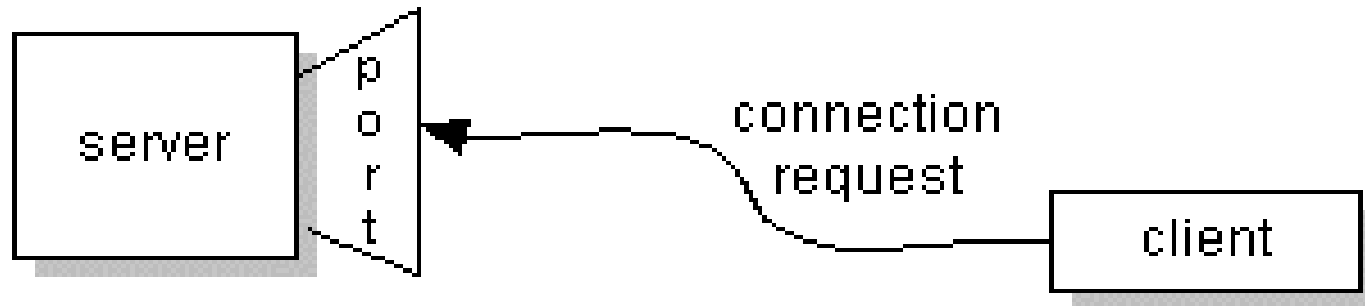
# ► Transmission Control Protocol



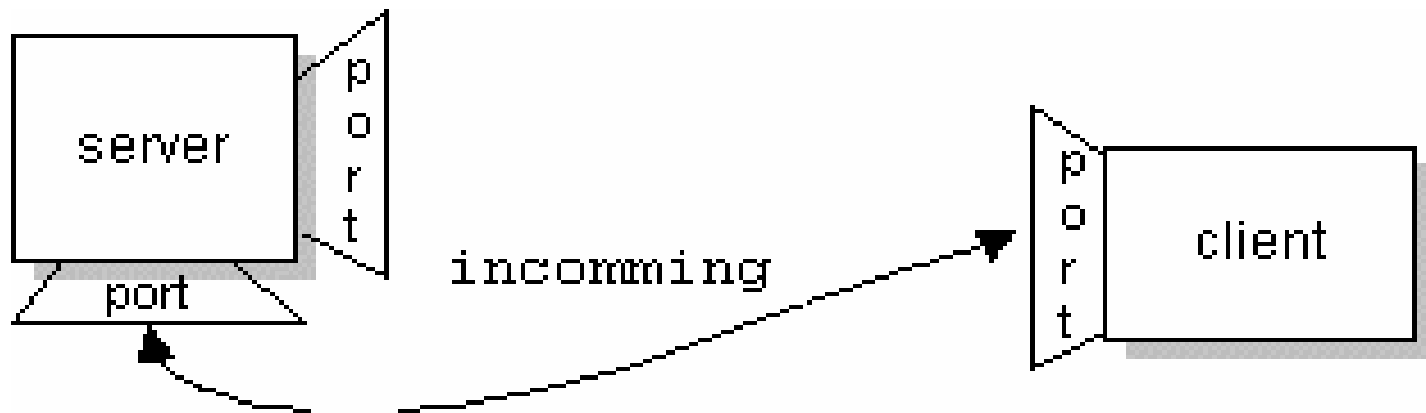
**TCP establishes a virtual connection to transmit data**

# ► TCP Programmning in Java

```
Socket clientSoc = new Socket(servername,port);
```



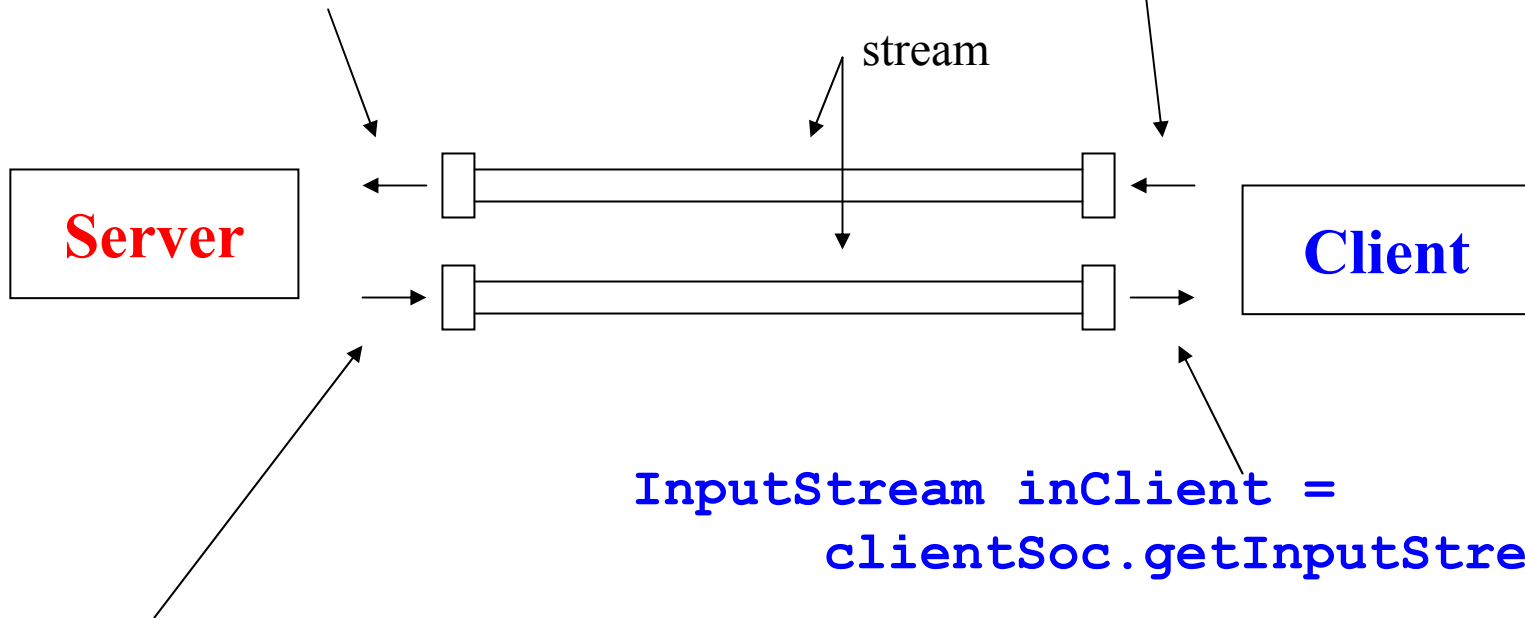
```
ServerSocket server = new ServerSocket(port);  
Socket serverSoc = server.accept();
```



# ► TCP Programming in Java

```
InputStream inServer =  
    serverSoc.getInputStream();
```

```
OutputStream outClient =  
    clientSoc.getOutputStream();
```



```
InputStream inClient =  
    clientSoc.getInputStream();
```

```
OutputStream outServer =  
    serverSoc.getOutputStream();
```

# ► Socket Basics

- A socket is a connection between two hosts. It can perform seven basic operations:
  - Connect to a remote machine
  - Send data
  - Receive data
  - Close a connection
  - Bind to a port
  - Listen for incoming data
  - Accept connections from remote machines on the bound port
- Java's Socket class, which is used by both **clients** and servers, has methods that correspond to the **first four of these operations**. The **last three operations** are needed only by **servers**, which wait for clients to connect to them. They are implemented by the **ServerSocket** class

# ► Program using client socket

1. The program creates a new socket with a **Socket( )** constructor.
2. The socket **attempts to connect to the remote host**.
3. Once the connection is established, the local and remote hosts **get input and output streams** from the socket and use those streams to send data to each other. This connection is **full-duplex**; both hosts can send and receive data simultaneously.
4. **When the transmission of data is complete, one or both sides close the connection**. Some protocols, such as HTTP 1.0, require the connection to be closed after each request is serviced. Others, such as FTP, allow multiple requests to be processed in a single connection.

# ► Socket Basics - Constructors

- **public Socket(String host, int port) throws UnknownHostException, IOException**
- This constructor creates a TCP socket to the specified port on the specified host and attempts to connect to the remote host. For example:

```
try {  
    Socket toOReilly = new  
        Socket("www.oreilly.com", 80);  
    // send and receive data...  
} catch (UnknownHostException e) {  
    System.err.println(e);  
} catch (IOException e) {  
    System.err.println(e);  
}
```



# ► LowPortScanner Program

```
import java.net.*;
import java.io.*;
public class LowPortScanner {
    public static void main(String[] args) {
        String host = "localhost";
        if (args.length > 0) host = args[0];
        for (int i = 1; i < 1024; i++) {
            try {
                System.out.print("Scanning on port : "+i + " ; ");
                Socket s = new Socket(host, i);
                System.out.println("There is a server on port " + i + " of "+
                                   host);
            }
            catch (UnknownHostException e) {
                System.out.println("The Server adress is unknown");
                break;
            }
            catch (IOException e) {
                System.out.println("The Server is not found");
            }
        }
    }
}
```

# ► Socket Basics - Constructor

## • 10.3.1.2 public Socket(InetAddress host, int port) throws IOException

Like the previous constructor, this constructor creates a TCP socket to the specified port on the specified host and tries to connect. It differs by using an InetAddress object to specify the host rather than a hostname. It throws an IOException if it can't connect, but does not throw an UnknownHostException; if the host is unknown, you will find out when you create the InetAddress object. For example:

```
try {  
    InetAddress OReilly=  
        InetAddress.getByName("www.oreilly.com");  
    Socket OReillySocket = new Socket(OReilly, 80);  
    // send and receive data...  
}  
catch (UnknownHostException e) {  
    System.err.println(e);  
}  
catch (IOException e) {  
    System.err.println(e);  
}
```

# ► Socket Basics - Constructor

- **public Socket(String serverAdd, int serverPort, InetAddress interface, int localPort) throws IOException**

This constructor creates a socket to the specified port on the specified host and tries to connect. It connects *to* the host and port specified in the first two arguments. It connects *from* the local network interface and port specified by the last two arguments. The network interface may be either physical (e.g., a different Ethernet card) or virtual (a multihomed host). If is passed for the localPort argument, Java chooses a random available port between 1024 and 65,535. For example, if I were running a program on *metallab.unc.edu* and wanted to make sure that my connection went over its 100 megabit-per-second (Mbps) fiber-optic interface (*fddisunsite.oit.unc.edu*) instead of the 10Mbps Ethernet interface (*helios.oit.unc.edu*), I would open a socket like this:

```
try {  
    InetAddress fddi =  
        InetAddress.getByName("fddisunsite.oit.unc.edu");  
    Socket OReillySocket = new Socket("www.oreilly.com", 80, fddi, 0);  
    // work with the sockets...  
}
```

# ► Socket Basics - Constructor

- **10.3.1.4 public Socket(InetAddress host, int port, InetAddress interface, int localPort) throws IOException**
- This constructor is identical to the previous one except that the host to connect to is passed as an InetAddress, not a String. It creates a TCP socket to the specified port on the specified host from the specified interface and local port, and tries to connect. If it fails, it throws an IOException. For example:

```
try {  
    InetAddress metalab =  
        InetAddress.getByName("metalab.unc.edu");  
    InetAddress oreilly =  
        InetAddress.getByName("www.oreilly.com");  
    Socket oreillySocket = new Socket(oreilly, 80, metalab, 0);  
}
```

# ► Socket Basics - Getting Information About a Socket

- **public InetAddress getInetAddress( )**

Given a Socket object, the `getInetAddress( )` method tells you which remote host the Socket is connected to or, if the connection is now closed, which host the Socket was connected to when it was connected. For example:

```
try {  
    Socket theSocket = new Socket("java.sun.com", 80);  
    InetAddress host = theSocket.getInetAddress( );  
    System.out.println("Connected to remote host " + host);  
}  
catch (UnknownHostException e) {  
    System.err.println(e);  
}  
catch (IOException e) {  
    System.err.println(e);  
}
```

# ► Socket Basics - Getting Information About a Socket

## • public int getPort( )

The getPort( ) method tells you which port the Socket is (or was or will be) connected to on the remote host. For example:

```
try {  
    Socket theSocket = new Socket("java.sun.com", 80);  
    int port = theSocket.getPort( );  
    System.out.println("Connected on remote port " + port);  
}
```

## • public int getLocalPort( )

There are two ends to a connection: the remote host and the local host. To find the port number for the local end of a connection, call getLocalPort( ). For example:

```
try {  
    Socket theSocket = new Socket("java.sun.com", 80, true);  
    int localPort = theSocket.getLocalPort( );  
    System.out.println("Connecting from local port " + localPort);  
}
```

# ► Socket Basics – SocketInfo Program

```
public class SocketInfo {
    public static void main(String[] args) {
        String[] hostNames = {"www.hcmuaf.edu.vn", "mail.hcmuaf.edu",
                                "testweb.hcmuaf.edu.vn"};

        for (int i = 0; i < hostNames.length; i++){
            try {
                Socket theSocket = new Socket(hostNames[i], 80);
                System.out.println("Connected to " + theSocket.getInetAddress( )
                                    + " on port " + theSocket.getPort( ) + " from port "
                                    + theSocket.getLocalPort( ) + " of " +
theSocket.getLocalAddress( ));
            }
            catch (UnknownHostException e) {
                System.err.println("I can't find " + hostNames[i]);
            }
            catch (SocketException e) {
                System.err.println("Could not connect to " + hostNames[i]);
            }
            catch (IOException e) {
                System.err.println(e);
            }
        }
    }
}
```

# ► Socket Basics - Getting Information About a Socket

- `public InputStream getInputStream( )` throws `IOException`
- The `getInputStream( )` method returns an input stream that can read data from the socket into a program. You usually chain this `InputStream` to a `filter stream` or `reader` that offers more functionality—`DataInputStream` or `InputStreamReader`, for example—before reading input. It's also extremely helpful to buffer the input by chaining it to a `BufferedInputStream` or a `BufferedReader` for performance reasons
- When reading data from the network, it's important to keep in mind that **not all protocols use ASCII** or even text.



# ► Socket Basics - Getting Information About a Socket

- `public OutputStream getOutputStream( )` throws `IOException`
- The `getOutputStream( )` method returns a raw `OutputStream` for writing data from your application to the other end of the socket. You usually chain this stream to a more convenient class like `DataOutputStream` or `OutputStreamWriter` before using it. For performance reasons, it's a good idea to buffer it as well.
- The following example uses `getOutputStream( )` and `getInputStream( )` to implement a simple echo client. The user types input on the command-line, which is then sent to the server. The server echoes it back

# ► Socket Basics - An Echo Client

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

```
public class EchoClient {
    public static final int ECHO_PORT = 7;
    public static void main(String[] args) {
        String hostname = "localhost";
```

```
        PrintWriter out = null;
```

```
        BufferedReader networkIn = null;
```

```
        try {
```

```
            Socket theSocket = new Socket(hostname, ECHO_PORT);
```

```
            networkIn = new BufferedReader(
```

```
                new InputStreamReader(theSocket.getInputStream()));
```

```
            BufferedReader userIn = new BufferedReader(
```

```
                new InputStreamReader(System.in));
```

```
            out = new PrintWriter(theSocket.getOutputStream());
```

```
            System.out.println("Connected to echo server");
```

```
            System.out.println(networkIn.readLine());
```

# ► Socket Basics - An Echo Client

```
while (true) {
    String theLine = userIn.readLine();
    out.println(theLine); out.flush();
    System.out.println(networkIn.readLine());
    if (theLine.equals("BYE")) break;
}
} // end try
catch (IOException e) {
    System.err.println(e);
}
finally {
    try {
        if (networkIn != null) networkIn.close();
        if (out != null) out.close();
    }
    catch (IOException e) {}
}
} // end main
}
```

# ► Socket Basics - Closing the Socket

- **public synchronized void close( ) throws IOException**
- When you're through with a socket, you should call its close( ) method to disconnect. Ideally, you put this in a finally block so that the socket is closed whether or not an exception is thrown. The syntax is straightforward:
- Socket connection = null;  
**try {**  
    **Socket connection = new Socket("www.oreilly.com", 13);**  
    // interact with the socket  
**} // end try**  
    catch (UnknownHostException e) {  
        System.err.println(e);  
    }  
    catch (IOException e) {  
        System.err.println(e);  
    }  
**finally {**  
    **if (connection != null) connection.close( );**  
**}**

# ► Socket Basics - Half-closed sockets

- When a client program sends a request to the server, the server needs to be able to determine when the end of the request occurs. For that reason, many Internet protocols (such as SMTP) are line-oriented. Other protocols contain a header that specifies the size of the request data. Otherwise, indicating the end of the request data is harder than writing data to a file. With a file, you'd just close the file at the end of the data. But if you close a socket, then you immediately disconnect from the server.
- The *half-close* overcomes this problem. You can close the output stream of a socket, thereby indicating to the server the end of the request data, but keep the input stream open so that you can read the response.
- Starting in Java 1.3, the `shutdownInput()` and `shutdownOutput()` methods let you close only half of the connection:
  - `public void shutdownInput()` throws `IOException`
  - `public void shutdownOutput()` throws `IOException`

# ► Socket Basics - Half-closed sockets

```
Socket connection = null;
try {
    connection = new Socket("www.oreilly.com", 80);
    BufferedReader reader = new BufferedReader( new
        InputStreamReader(socket.getInputStream()));

    Writer out = new OutputStreamWriter(connection.getOutputStream( ),
        "UTF-8");

    out.write("GET / HTTP 1.0\r\n\r\n");
    out.flush( );
    connection.shutdownOutput( );
    // now socket is half closed; read response data
    String line;
    while ((line = reader.readLine()) != null)
        ...
} catch (IOException e) {}
finally {
    try {
        if (connection != null) connection.close( );
    } catch (IOException e) {}
}
```

# ► Socket Basics - Socket timeouts

- `public synchronized void setSoTimeout(int milliseconds) throws SocketException`
- `public synchronized int getSoTimeout( ) throws SocketException`
- Normally when you try to read data from a socket, the `read( )` call blocks as long as necessary to get enough bytes. By setting `SO_TIMEOUT`, you ensure that the call will not block for more than a fixed number of milliseconds. When the timeout expires, an `InterruptedException` is thrown, and you should be prepared to catch it. However, the socket is still connected. Although this `read( )` call failed, you can try to read from the socket again. The next call may succeed.
- `Socket s = new Socket0();`  
`s.setSoTimeout(10000); // time out after 10 seconds`  
`s.connect(...);`
- Timeouts are given in milliseconds. Zero is interpreted as an infinite timeout, and is the default value.

# ► Socket Basics - Sockets for Servers

- The basic life cycle of a server is:
  - 1. A new **ServerSocket** is created on a particular port using a **ServerSocket()** constructor.
  - 2. The **ServerSocket** listens for incoming connection attempts on that port using its **accept()** method. **accept()** blocks until a client attempts to make a connection, at which point **accept()** returns a Socket object connecting the client and the server.
  - 3. Depending on the type of server, either the Socket's **getInputStream()** method, **getOutputStream()** method, or both are called to get input and output streams that communicate with the client.
  - 4. The server and the client interact according to an agreed-upon protocol until it is time to close the connection.
  - 5. The server, the client, or both close the connection.
  - 6. The server returns to step 2 and waits for the next connection.



# ► Socket Basics - Sockets for Servers

- **public ServerSocket(int port)** throws **IOException**, **BindException**
- This constructor creates a server socket on the port specified by the argument.
- For example, to create a server socket that would be used by an HTTP server on port 80, you would write:

```
try {  
    ServerSocket httpd = new ServerSocket(80) ;  
}  
catch (IOException e) {  
    System.err.println(e) ;  
}
```

- The constructor throws an **IOException** (specifically, a **BindException**) if the socket cannot be created and bound to the requested port. An **IOException** when creating a **ServerSocket** almost always means one of two things. Either another server socket is already using the requested port, or you're trying to connect to a port from 1 to 1023 on Unix without root (superuser) privileges.

# ► Socket Basics - LocalServerPortScanner

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

public class LocalServerPortScanner {
    public static void main(String[] args) {
        for (int port = 1; port <= 1024; port++) {
            try {
                // the next line will fail and drop into the catch block if
                // there is already a server running on the port
                ServerSocket server = new ServerSocket(port);
            }
            catch (IOException e) {
                System.out.println("There is a server on port " +
                    port + ".");
            } // end try
        } // end for
    }
}
```

# ► Socket Basics - Sockets for Servers

- **public Socket accept( ) throws IOException**

When server setup is done and you're ready to accept a connection, call the *ServerSocket's accept( )* method. This method "blocks": it stops the flow of execution and waits until a client connects. When a client does connect, the *accept( )* method returns a Socket object. You use the streams returned by this Socket's *getInputStream( )* and *getOutputStream( )* methods to communicate with the client. For example:

```
ServerSocket server = new ServerSocket(5776);
```

```
while (true) {
```

```
    Socket connection = server.accept( );
```

```
    PrintWriter out
```

```
    = new PrintWriter(connection.getOutputStream( ), TRUE);
```

```
    out.println("You've connected to this server. Bye-bye now.");
```

```
    connection.close( );
```

```
}
```

# ► ServerSocket - Socket Options

- The only socket option supported for server sockets is `SO_TIMEOUT`. `SO_TIMEOUT` is the amount of time, in milliseconds, that `accept ( )` waits for an incoming connection before throwing a `java.io.InterruptedIOException`. If `SO_TIMEOUT` is 0, then `accept ( )` will never time out. The default is to never time out.
- **`public void setSoTimeout(int timeout) throws SocketException`**
- The `setSoTimeout ( )` method sets the `SO_TIMEOUT` field for this server socket object. The countdown starts when `accept ( )` is invoked. When the timeout expires, `accept ( )` throws an `InterruptedIOException`. You should set this option before calling `accept ( )`; you cannot change the timeout value while `accept ( )` is waiting for a connection. The `timeout` argument must be greater than or equal to zero; if it isn't, the method throws an `IllegalArgumentException`. For example:

# ► ServerSocket - Socket Options

```
try {
    ServerSocket server = new ServerSocket(2048);
    // block for no more than 30 seconds
    server.setSoTimeout(30000);
    try {
        Socket s = server.accept();
        // handle the connection
        // ...
    }
    catch (InterruptedException e) {
        System.err.println("No connection within 30
                           seconds");
    }
    finally {
        server.close( );
    }
    catch (IOException e) {
        System.err.println("Unexpected IOException:" + e);
    }
}
```

# ► Implement EchoServer

```
import java.io.*;
import java.net.*;
public class EchoServer {
    public static final int ECHO_PORT = 7;
    public static void main(String[] args) {
        try {
            // establish server socket
            ServerSocket s = new ServerSocket(ECHO_PORT);

            // wait for client connection
            Socket incoming = s.accept();
            BufferedReader in = new BufferedReader (new
                InputStreamReader(incoming.getInputStream()));
            PrintWriter out = new PrintWriter
                (incoming.getOutputStream(), true /* autoFlush */);

            out.println("Welcome to ECHO SERVER! Enter BYE to
                        exit.");
        }
    }
}
```

# ► Implement EchoServer

```
// echo client input
boolean done = false;
while (!done) {
    String line = in.readLine();
    if (line == null) done = true;
    else {
        out.println("Echo: " + line);

        if (line.trim().equals("BYE"))
            done = true;
    }
}
incoming.close();
}
catch (Exception e) {
    e.printStackTrace();
}
}
```

# ► Implement ThreadedEchoServer

```
public class ThreadedEchoServer {
    public static final int ECHO_PORT = 7;
    public static void main(String[] args) {
        try {
            int i = 1;
            ServerSocket s = new ServerSocket(ECHO_PORT);
            for (; ; ) {
                Socket incoming = s.accept();
                System.out.println("Connection number:" + i);
                System.out.println("Local Port: " + incoming.getLocalPort() +
                    "Foreign Port : " + incoming.getPort());
                Thread t = new ThreadedEchoHandler(incoming, i);
                t.start();
                i++;
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```



# ► Implement ThreadedEchoServer

```
class ThreadedEchoHandler extends Thread {
private Socket incoming;
private int counter;
public ThreadedEchoHandler(Socket i, int c) {
    incoming = i;
    counter = c;
}
public void run() {
    try {
        BufferedReader in = new BufferedReader (new
  InputStreamReader(incoming.getInputStream()));
        PrintWriter out = new PrintWriter (incoming.getOutputStream(), true);
        out.println("Welcom to Threaded ECHO SERVER! Enter BYE to exit.");
        boolean done = false;
        while (!done) {
            String str = in.readLine();
            if (str == null) done = true;
            else {
                out.println("Echo (" + counter + "): " + str);
                if (str.trim().equals("BYE")) done = true;
            }
        }
        incoming.close();
    }
    catch (Exception e) {
        e.printStackTrace();
    }
}}
```

# ► Implement EchoClient

```
public class EchoClient {
    public static final int ECHO_PORT = 7;
    public static void main(String[] args) {
        String hostname = "localhost";

        PrintWriter out = null;
        BufferedReader networkIn = null;
        try {
            Socket theSocket = new Socket(hostname, ECHO_PORT);
            networkIn = new BufferedReader(
                new InputStreamReader(theSocket.getInputStream()));
            BufferedReader userIn = new BufferedReader(
                new InputStreamReader(System.in));
            out = new PrintWriter(theSocket.getOutputStream());
            System.out.println("Connected to echo server");
            System.out.println(networkIn.readLine());
        }
    }
}
```

# ► Implement EchoClient

```
while (true) {
    String theLine = userIn.readLine();
    out.println(theLine);
    out.flush();
    System.out.println(networkIn.readLine());
    if (theLine.equals("BYE")) break;
}
} // end try
catch (IOException e) {
    System.err.println(e);
}
finally {
    try {
        if (networkIn != null) networkIn.close();
        if (out != null) out.close();
    }
    catch (IOException e) {}
}
} // end main
}
```

## ► **InetAddress class**

- Usually, you don't have to worry too much about Internet addresses—the numerical host addresses that consist of four bytes such as 132.163.4.102. However, you can use the **InetAddress** class if you need to convert between host names and Internet addresses.
- The static **getByName** method returns an **InetAddress** object of a host.
- For example,
- **InetAddress address = InetAddress.getByName("time-A.timefreq.bldrdoc.gov");**
- returns an **InetAddress** object that encapsulates the sequence of four bytes 132.163.4.102. You can access the bytes with the **getAddress** method.
- **byte[] addressBytes = address.getAddress();**

## ► InetAddress class

- Some host names with a lot of traffic correspond to multiple Internet addresses, to
- facilitate load balancing. You can get all hosts with the **getAllByName** method.
- **InetAddress[] addresses = InetAddress.getAllByName(host);**
- Finally, you sometimes need the address of the local host. If you simply ask for the address of **localhost**, you always get the address 127.0.0.1, which isn't very useful. Instead, use the static **getLocalHost** method to get the address of your local host.
- **InetAddress address = InetAddress.getLocalHost();**

# ► InetAddress class

- **public String getHostName()**
  - Gets the host name for this IP address. If this InetAddress was created with a host name, this host name will be remembered and returned; otherwise, a reverse name lookup will be performed and the result will be returned based on the system configured name lookup service.
- **public String getCanonicalHostName()**
  - Gets the fully qualified domain name for this IP address. Best effort method, meaning we may not be able to return the FQDN depending on the underlying system configuration.
- **public byte[] getAddress()**
  - Returns the raw IP address of this InetAddress object. The result is in network byte order: the highest order byte of the address is in getAddress()[0].

## ► InetAddress class

- **public String getHostAddress()**
  - Returns the IP address string in textual presentation for example "132.163.4.102".
- **public String toString()**
  - Converts this IP address to a String. The string returned is of the form: hostname / literal IP address. If the host name is unresolved, no reverse name service lookup is performed. The hostname part will be represented by an empty string.
- **public static InetAddress getByName(String host)**  
throws UnknownHostException
  - Determines the IP address of a host, given the host's name. The host name can either be a machine name, such as "java.sun.com", or a textual representation of its IP address.

# ► InetAddress class

- **public static InetAddress[] getAllByName(String host)** throws UnknownHostException
  - Given the name of a host, returns an array of its IP addresses, based on the configured name service on the system. The host name can either be a machine name, such as "java.sun.com", or a textual representation of its IP address.
- **public static InetAddress getByAddress(byte[] addr)** throws UnknownHostException
  - Returns an InetAddress object given the raw IP address . The argument is in network byte order: the highest order byte of the address is in getAddress()[0].
- **public static InetAddress getLocalHost()** throws UnknownHostException
  - Returns the local host.



# ► NSLookup

```
public class NSLookup {
    public static void main(String[] args) {
        String hostName = "localhost";
        String hostNameIP = "127.0.0.1";
        InetAddress add;
        try{
            add = InetAddress.getByName(hostName);
            System.out.println("DNS host name: "+add.getCanonicalHostName());
            System.out.println("IP Address: "+add.getHostAddress());

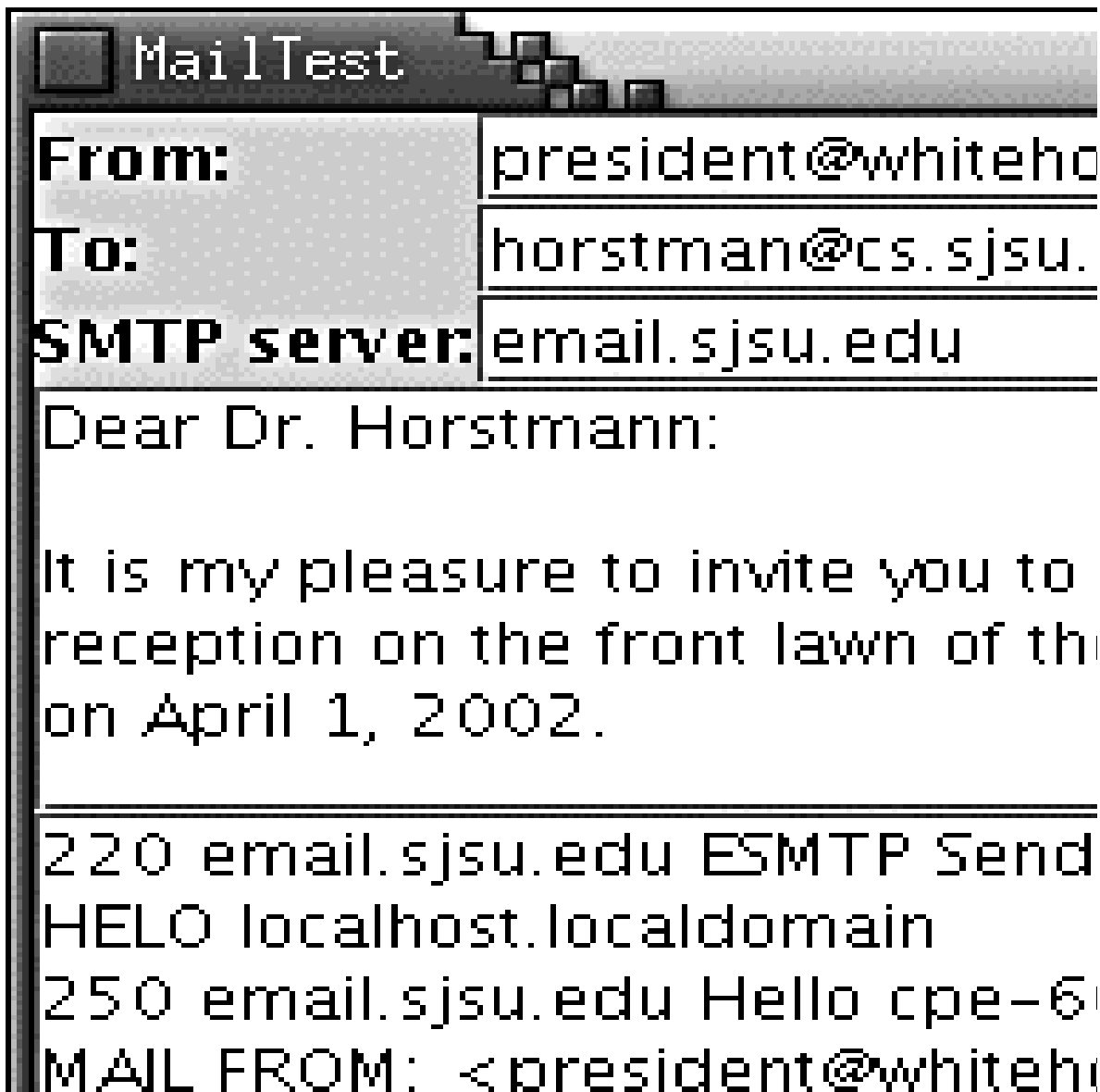
            add = InetAddress.getByName(hostNameIP);
            System.out.println("DNS host name: "+add.getCanonicalHostName());
            System.out.println("IP Address: "+add.getHostAddress());

            System.out.println("InetAddress toString: "+add);

            InetAddress[] addresses = InetAddress.getAllByName(hostName);
            for (int i = 0; i < addresses.length; i++) System.out.println(addresses[i]);

        } catch(UnknownHostException e){
            System.out.println("The Address not exist"); }
    }
}
```

## ► Sending E-Mail



# ► Sending E-Mail

- To send e-mail, you make a socket connection to port 25, the SMTP port. SMTP is the Simple Mail Transport Protocol that describes the format for e-mail messages.
- Open a socket to your host.  
**Socket s = new Socket("mail.yourserver.com", 25);**  
**PrintWriter out = new PrintWriter(s.getOutputStream());**
- Send the following information to the print stream:  
**HELO** *sending host # Domain name*  
**MAIL FROM:** *<sender email address>*  
**RCPT TO:** *<recipient email address>*  
**DATA**  
*mail message*  
*(any number of lines)*  
.  
**QUIT**

# ► Sending E-Mail

```
private BufferedReader in;  
private PrintWriter out;  
private JTextField from;  
private JTextField to;  
private JTextArea message;
```

```
.....  
public void sendMail() {  
try {
```

```
    Socket s = new Socket(smtpServer.getText(), 25);  
    out = new PrintWriter(s.getOutputStream());  
    in = new BufferedReader(new  
        InputStreamReader(s.getInputStream()));  
    String hostName =  
        InetAddress.getLocalHost().getHostName();  
    receive();  
    send("HELO " + hostName);  
    receive()
```

# ► Sending E-Mail

```
send("MAIL FROM: <" + from.getText() + ">");
receive();
send("RCPT TO: <" + to.getText() + ">");
receive();
send("DATA");
receive();
StringTokenizer tokenizer = new StringTokenizer(
    message.getText(), "\n");
while (tokenizer.hasMoreTokens())
    send(tokenizer.nextToken());
send(".");
receive();
s.close();
}
catch (IOException exception) {
    .....
}
}
```

## ► Sending E-Mail

```
public void send(String s) throws IOException {
```

```
    .....
```

```
    out.print(s);
```

```
    out.print("\r\n");
```

```
    out.flush();
```

```
}
```

```
public void receive() throws IOException {
```

```
    String line = in.readLine();
```

```
    if (line != null) {
```

```
        .....
```

```
    }
```

```
}
```

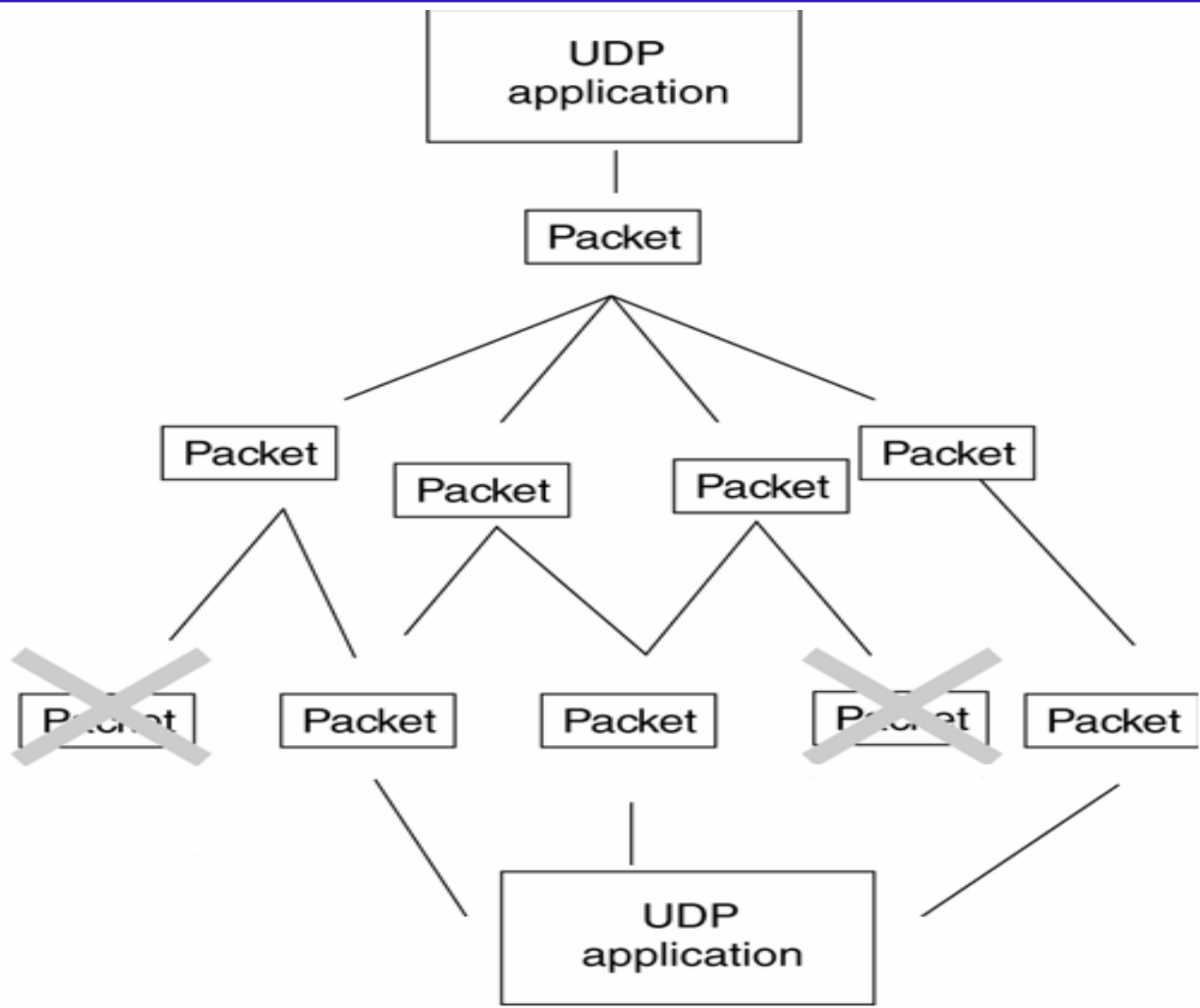
# ► Receive E-Mail – POP3 port 110

- // Login by sending USER and PASS commands
- **USER** username
- **PASS** password
- // Get mail count from server ....
- **STAT**  
message\_Number , meassage\_Size
- **RETR** meassage\_Number  
... message body
- 
- **QUIT**

# ► UDP Programming



# ► Overview



# ► Overview

- **UDP communication can be more efficient than guaranteed-delivery data streams.** If the amount of data is small and the data is sent frequently, it may make sense to avoid the overhead of guaranteed delivery.
- **Unlike TCP streams, which establish a connection, UDP causes fewer overheads.** If the amount of **data being sent is small** and the data is sent **infrequently**, the **overhead** of **establishing** a connection might not be worth it. UDP may be preferable in this case, particularly if data is being sent from a large number of machines to one central one, in which case the sum total of all these connections might cause significant overload.
- **Real-time applications that demand up-to-the-second or better performance may be candidates for UDP, as there are fewer delays due to the error checking and flow control of TCP.** UDP packets can be used to saturate available network bandwidth to deliver large amounts of data (such as streaming **video/audio**, or telemetry data for a multiplayer network game). In addition, if **some data is lost**, it can be replaced by the next set of packets with updated information, eliminating the need to resend old data that is now out of date.
- **UDP sockets can receive data from more than one host machine.** If several machines must be communicated with, then UDP may be more convenient than other mechanisms such as TCP.

# ► DatagramPacket Class

**Port**

**DataPacket(port)**

**...setPort(port)**

**Remote Port**

DatagramPacket

IP address (java.net.InetAddress)

Port address (int)

Packet data

```
byte[] = { .., .., .., .., .., .., .., ..,  
           .., .., .., .., .., .., .., ..,  
           .., .., .., .., .., .., .., .. }
```

# ► Creating a DatagramPacket

- Constructor requires the specification of a byte array, which will be used to store the UDP packet contents, and the length of the data packet.

There are two reasons to create a new DatagramPacket:

- **1.To send data to a remote machine using UDP**  
DatagramPacket(byte[] buffer, int length, InetAddress dest\_addr, int dest\_port).  
For example:  
InetAddress addr = InetAddress.getByName("192.168.0.1");  
DatagramPacket packet = new DatagramPacket ( new byte[128],128, addr, 2000);
- **2.To receive data sent by a remote machine using UDP**  
DatagramPacket(byte[] buffer, int length). For example:  
DatagramPacket packet = new DatagramPacket(new byte[256], 256);

## ► Using a DatagramPacket

- **InetAddress getAddress()**— returns the IP address from which a DatagramPacket was sent, or (if the packet is going to be sent to a remote machine), the destination IP address.
- **byte[] getData()**— returns the contents of the DatagramPacket, represented as an array of bytes.
- **int getLength()**— returns the length of the data stored in a DatagramPacket. This can be less than the actual size of the data buffer.
- **int getPort()**— returns the port number from which a DatagramPacket was sent, or (if the packet is going to be sent to a remote machine), the destination port number.

## ► Using a DatagramPacket

- **void setAddress(InetAddress addr)**— assigns a new destination address to a DatagramPacket.
- **void setData(byte[] buffer)**— assigns a new data buffer to the DatagramPacket. Remember to make the buffer long enough, to prevent data loss.
- **void setLength(int length)**— assigns a new length to the DatagramPacket. Remember that the length must be less than or equal to the maximum size of the data buffer, or an IllegalArgumentException will be thrown. When sending a smaller amount of data, you can adjust the length to fit—you do not need to resize the data buffer.
- **void setPort(int port)**— assigns a new destination port to a DatagramPacket.

# ► DatagramSocket Class

- The DatagramSocket class provides access to a UDP socket, which allows UDP packets to be sent and received. A DatagramPacket is used to represent a UDP packet, and must be created prior to receiving any packets. The same DatagramSocket can be used to receive packets as well as to send them. However, read operations are blocking, meaning that the application will continue to wait until a packet arrives.
- A DatagramSocket can be used to both send and receive packets. Each DatagramSocket binds to a port on the local machine, which is used for addressing packets. The application is a UDP server, it will usually choose a specific port number. If the DatagramSocket is intended to be a client, and doesn't need to bind to a specific port number, a blank constructor can be specified.

# ► Creating a DatagramSocket

- To create a client **DatagramSocket**, the following constructor is used:  
**DatagramSocket()** throws `java.net.SocketException`.
- To create a server **DatagramSocket**, the following constructor is used, which takes as a parameter the port to which the UDP service will be bound:  
**DatagramSocket(int port)** throws `java.net.SocketException`.
- Although rarely used, there is a third constructor for **DatagramSocket**. If a machine is known by several IP addresses, you can specify the IP address and port to which a UDP service should be bound. It takes as parameters the port to which the UDP service will be bound, as well as the **InetAddress** of the service. This constructor is:  
**DatagramSocket (int port, InetAddress addr)** throws `java.net.SocketException`.
- **Port is Local !!!**



# ► Using a DatagramSocket

- **void close()**— closes a socket, and unbinds it from the local port.
- **void connect(InetAddress remote\_addr int remote\_port)**— restricts access to the specified remote address and port. The designation is a misnomer, as UDP doesn't actually create a "connection" between one machine and another.
- **void disconnect()**— disconnects the DatagramSocket and removes any restrictions imposed on it by an earlier connect operation.
- **InetAddress getInetAddress()**— returns the remote address to which the socket is connected, or null if no such connection exists.
- **int getPort()**— returns the remote port to which the socket is connected, or -1 if no such connection exists.
- **InetAddress getLocalAddress()**— returns the local address to which the socket is bound.
- **int getLocalPort()**— returns the local port to which the socket is bound.
- **int getReceiveBufferSize()** throws java.net.SocketException— returns the maximum buffer size used for incoming UDP packets.

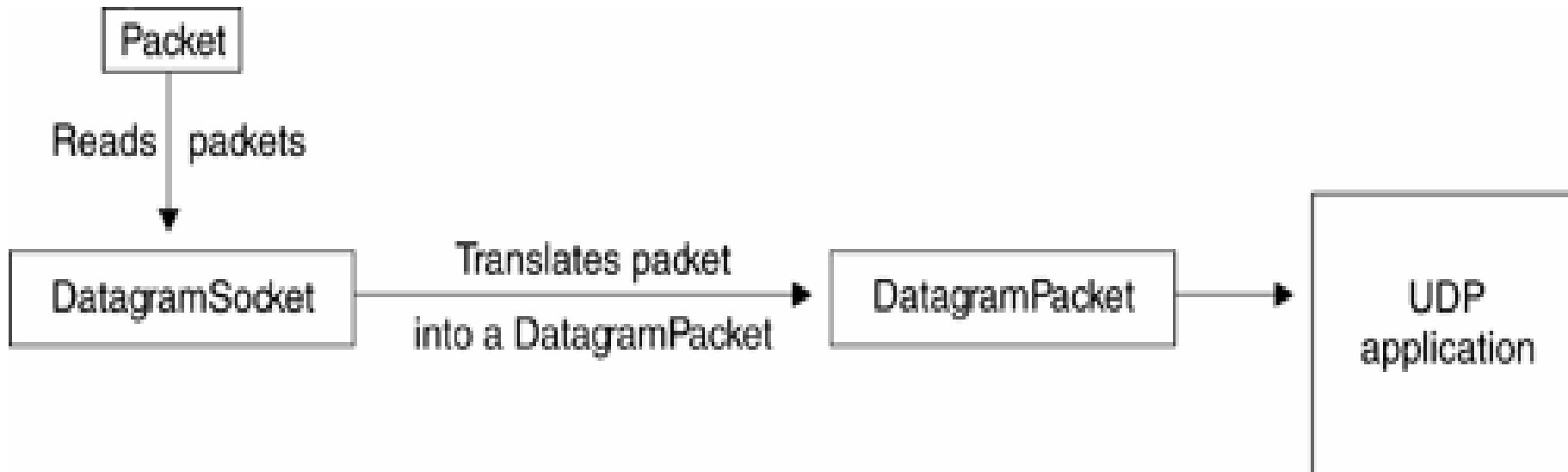
# ► Using a DatagramSocket

- **int getSendBufferSize()** throws `java.net.SocketException`— returns the maximum buffer size used for outgoing UDP packets.
- **int getSoTimeout()** throws `java.net.SocketException`— returns the value of the timeout socket option. By default, this value will be zero, indicating that blocking I/O will be used.
- **void receive(DatagramPacket packet)** throws `java.io.IOException`— reads a UDP packet and stores the contents in the specified packet. The address and port fields of the packet will be overwritten with the sender address and port fields, and the length field of the packet will contain the length of the original packet, which can be less than the size of the packet's byte-array. If a timeout value has been specified, a `java.io.InterruptedIOException` will be thrown if the time is exceeded.
- **void send(DatagramPacket packet)** throws `java.io.IOException`— sends a UDP packet, represented by the specified packet parameter.

## ► Using a DatagramSocket

- **void setReceiveBufferSize(int length)** throws `java.net.SocketException`— sets the maximum buffer size used for incoming UDP packets. Whether the specified length will be adhered to is dependent on the operating system.
- **void setSendBufferSize(int length)** throws `java.net.SocketException`— sets the maximum buffer size used for outgoing UDP packets. Whether the specified length will be adhered to is dependent on the operating system.
- **void setSoTimeout(int duration)** throws `java.net.SocketException`— sets the value of the timeout socket option. This value is the number of milliseconds a read operation will block before throwing a `java.io.InterruptedIOException`.

# ► Listening for UDP Packets



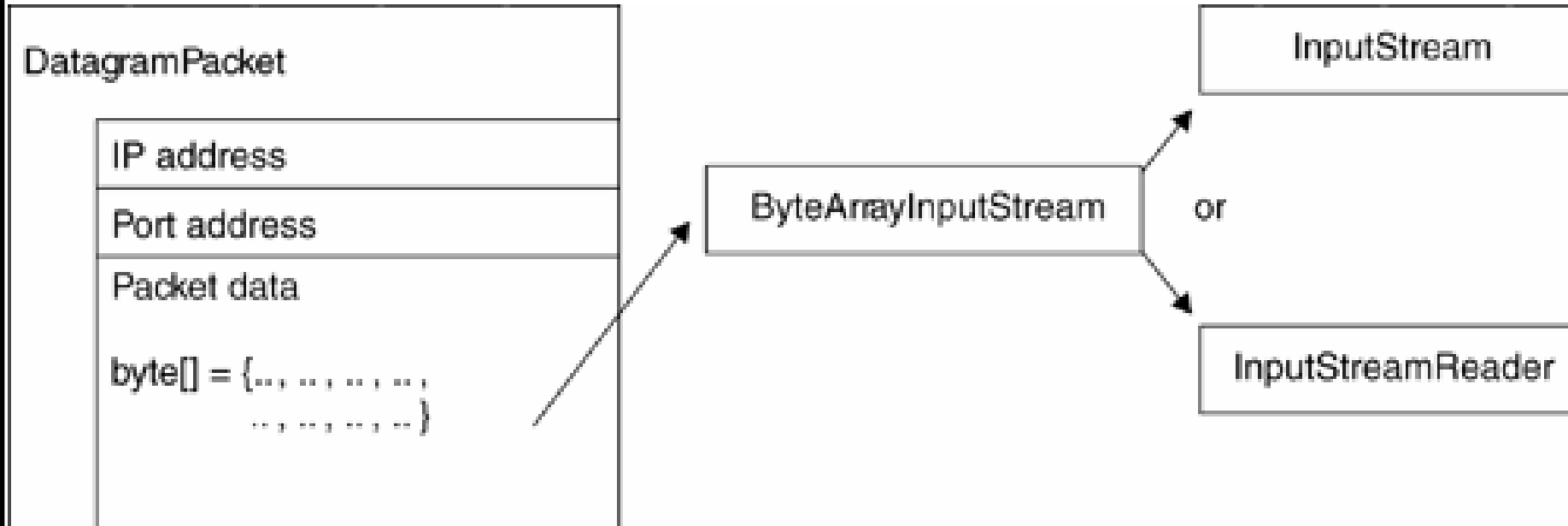
- UDP packets are received by a **DatagramSocket** and translated into a **DatagramPacket** object.
- When an application wishes to read UDP packets, it calls the **DatagramSocket.receive** method, which copies a UDP packet into the specified **DatagramPacket**. The contents of the **DatagramPacket** are processed, and the process is repeated as needed.

# ► Listening for UDP Packets

```
DatagramPacket packet = new DatagramPacket (new byte[256],  
  256);  
DatagramSocket socket = new DatagramSocket(2000);  
boolean finished = false;  
while (! finished ){  
    socket.receive (packet);  
    // process the packet  
}  
socket.close();
```

When processing the packet, **the application must work directly with an array of bytes**. If, however, your application is better suited to reading text, you can use classes from the Java I/O package to convert between a byte array and another type of stream or reader. By hooking a **ByteArrayInputStream** to the contents of a datagram and then to another type of **InputStream** or an **InputStreamReader**, you can access the contents of UDP packets relatively easily. Many developers prefer to use Java I/O streams to process data, using a **DataInputStream** or a **BufferedReader** to access the contents of byte arrays.

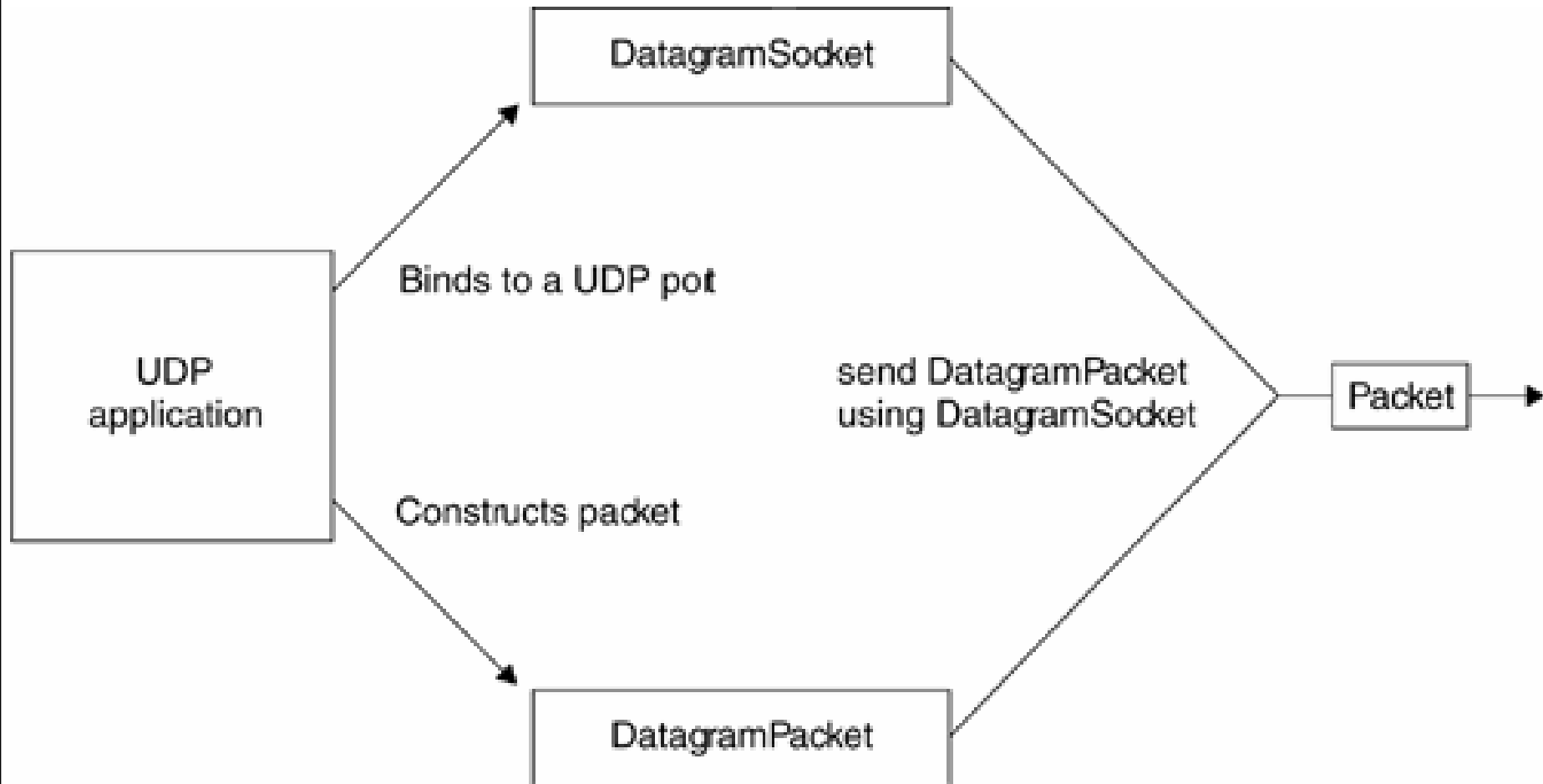
# ► Listening for UDP Packets



```
ByteArrayInputStream bin = new  
    ByteArrayInputStream(packet.getData() );  
DataInputStream din = new DataInputStream (bin);  
// Read the contents of the UDP packet
```

.....

# ▶ Sending UDP packets



# ► Sending UDP packets

```
DatagramSocket socket = new DatagramSocket();
DatagramPacket packet = new DatagramPacket (new
  byte[256],256);
packet.setAddress (InetAddress.getByName (someServer));
packet.setPort (2000);
boolean finished = false;
while !finished ){
    // Write data to packet buffer
    .....
    packet.setData(.....);
    packet.setLength(...);
    socket.send (packet);
    // Do something else, like read other packets, or check to
    // see if no more packets to send
    .....
}
socket.close();
```



# ► UDP PacketReceiveDemo - Server

- The application starts by binding to a specific port, 2000. Applications offering a service generally bind to a specific port. When acting as a receiver, your application should choose a specific port number, so that a sender can send UDP packets to this port. Next, the application prepares a **DatagramPacket** for storing UDP packets, and creates a new buffer for storing packet data.

// Create a datagram socket, bound to the specific port 2000

```
DatagramSocket socket = new DatagramSocket(2000);
```

```
System.out.println ("Bound to local port " +  
socket.getLocalPort());
```

// Create a datagram packet, containing a maximum buffer of 256 bytes

```
DatagramPacket packet = new DatagramPacket( new  
byte[256], 256);
```

# ► UDP PacketReceiveDemo - Server

- Now the application is ready to read a packet. The read operation is blocking, so until a packet arrives, the server will wait. When a packet is successfully delivered to the application, the addressing information for the packet is displayed so that it can be determined where it came from.  
// Receive a packet - remember by default this is a blocking  
// operation  
`socket.receive(packet);`  
// Display packet information  
`InetAddress remote_addr = packet.getAddress();`  
`System.out.println ("Sent by : " +`  
`remote_addr.getHostAddress() );`  
`System.out.println ("Send from: " + packet.getPort());`

# ► UDP PacketReceiveDemo - Server

- To provide easy access to the contents of the UDP packet, the application uses a **ByteArrayInputStream** a **DataInputStream** to read from the packet. Reading one character at a time, the program displays the contents of the packet and then finishes.

```
// Display packet contents, by reading from byte array
```

```
DataInputStream din = new DataInputStream(new  
    ByteArrayInputStream(packet.getData()));
```

```
// Display only up to the length of the original UDP packet
```

```
for (int i = 0; i < packet.getLength()/2; i++) {
```

```
    System.out.print(din.readChar());
```

```
}
```

```
socket.close();
```

```
}
```

# ► UDP PacketSendDemo - Client

- The application starts by binding a UDP socket to a local port, which will be used to send the data packet. Unlike the receiver demonstration, it doesn't matter which local port is being used. In fact, any free port is a candidate, and you may find that running the application several times will result in different port numbers. After binding to a port, the port number is displayed to demonstrate this.

```
// Create a datagram socket, bound to any available local port
DatagramSocket socket = new DatagramSocket();
System.out.println ("Bound to local port " +
socket.getLocalPort());
```

- Before sending any data, we need to create a **DatagramPacket**. First, a **ByteArrayOutputStream** is used to create a sequence of bytes. Once this is complete, the array of bytes is passed to the **DatagramPacket** constructor.

# ► UDP PacketSendDemo - Client

```
// Create a message to send using a UDP packet
    ByteArrayOutputStream bout = new ByteArrayOutputStream();
    DataOutputStream dout = new DataOutputStream(bout);
    dout.writeChars("Greetings!");
// Get the contents of our message as an array of bytes
    byte[] barray = bout.toByteArray();
// Create a datagram packet, containing our byte array
    DatagramPacket packet = new DatagramPacket(barray,
  barray.length);
```

- Now that the packet has some data, it needs to be correctly addressed. As with a postal message, if it lacks correct address information it cannot be delivered. We start by obtaining an **InetAddress** for the remote machine, and then display its IP address. This **InetAddress** is passed to the **setAddress** method of **DatagramPacket**, ensuring that it will arrive at the correct machine. However, we must go one step further and specify a port number. In this case, port 2000 is matched, as the receiver will be bound to that port.

# ► UDP PacketSendDemo - Client

```
System.out.println ("Looking up hostname " + hostname );
// Lookup the specified hostname, and get an InetAddress
InetAddress remote_addr =
    InetAddress.getByName(hostname);
System.out.println ("Hostname resolved as " +
    remote_addr.getHostAddress());
// Address packet to sender
packet.setAddress (remote_addr);
// Set port number to 2000
packet.setPort (2000);
// Send the packet - remember no guarantee of delivery
socket.send(packet);
```

# ► URL Connection

# ► URLs and URIs

01-2004

Khoa CNTT

2/10

PHẠM VĂN TÍNH

| URL      |    |          |            |      |                 |
|----------|----|----------|------------|------|-----------------|
| protocol | :: | hostname | ( : port ) | path | ( # reference ) |

- The **URL** and **URLConnection** classes encapsulate much of the complexity of retrieving information from a remote site. Here is how you specify a URL.
- **URL url = new URL(urlString);**
- The Java 2 platform supports both **HTTP** and **FTP** resources.
- If you simply want to fetch the contents of the resource, then you can use the **openStream** method of the **URL** class. This method yields an **InputStream** object. Using this stream object, you can **easily read the contents of the resource**.



# ► URLs and URIs

```
InputStream uin = url.openStream();
BufferedReader in
= new BufferedReader(new InputStreamReader(uin));
String line;
while ((line = in.readLine()) != null) {
    process line;
}
```

The java.net package makes a useful distinction between **URLs** (*uniform resource locators*) and **URIs** (*uniform resource identifiers*).

A **URI** is a purely syntactical construct that specifies the various parts of the string specifying a web resource. A **URL** is a special kind of **URI**, namely one with sufficient information to *locate* a resource. Other URIs, such as **mailto:cay@horstmann.com**

are not locators—there is no data to locate from this identifier. Such a URI is called a **URN** (uniform resource *name*).

# ► Using a **URLConnection** to Retrieve Information

- If you want additional information about a web resource, then you should use the **URLConnection** class, which gives you much more control than the basic **URL** class.
- 1. Call the **openConnection** method of the **URL** class to obtain the **URLConnection** object:  
**URLConnection connection = url.openConnection();**
- 2. Set any request properties, using the methods
  - **setDoInput**
  - **setDoOutput**
  - **setIfModifiedSince**
  - **setUseCaches**
  - **setAllowUserInteraction**
  - **setRequestProperty**
- 3. Connect to the remote resource by calling the **connect** method.
  - **connection.connect();**

# ► Using a URLConnection to Retrieve Information

4. After connecting to the server, you can query the header information. There are two methods, **getHeaderFieldKey** and **getHeaderField**, to enumerate all fields of the header. As of SDK 1.4, there is also a method **getHeaderFields** that gets a standard **Map** object containing the header fields. For your convenience, the following methods query standard fields.

- **getContentType**
- **getContentLength**
- **getContentEncoding**
- **getDate**
- **getExpiration**
- **getLastModified**

# ► Using a **URLConnection** to Retrieve Information

- Finally, you can access the resource data. Use the **getInputStream** method to obtain an input stream for reading the information. (This is the same input stream that the **openStream** method of the **URL** class returns.).
- Some programmers form the wrong mental image when using the **URLConnection** class and think that the **getInputStream** and **getOutputStream** methods are similar to those of the **Socket** class. But that isn't quite true. The **URLConnection** class does quite a bit of magic behind the scenes, in particular the handling of request and response headers.
- **URL ( String url\_str )** throws **java.net.MalformedURLException—creates** a **URL** object based on the string parameter. If the **URL** cannot be correctly parsed, a **MalformedURLException** will be thrown.

# ► URL Class

- **URL ( String protocol, String host, String file )** throws **java.net.MalformedURLException**— creates a **URL** object with the specified **protocol**, **host**, and **file path**.
- **URL ( String protocol, String host, int port, String file )** throws **java.net.MalformedURLException**— creates a **URL** object with the specified **protocol**, **host**, **port**, and **file path**.
- **String getProtocol()**— returns the protocol component of a **URL**.
- **String getHost()**— returns the hostname component of a **URL**.
- **String getPort()**— returns the port component of a **URL**. This is an optional component, and if not present a value of **-1** will be returned.
- **String getFile()**— returns the pathname component of a **URL**.

# ► URL Class

- **String getRef()**— returns the reference component of a URL. A null value will be returned if no reference was specified.
- **URLConnection.openConnection()**— returns a **URLConnection** object, which can be used to establish a connection to the remote resource. The name of this method can be deceiving, though, as no connection will be established until further methods of the **URLConnection** object are invoked.
- **InputStream openStream()** throws **java.io.IOException**— establishes a connection to the remote server where the resource is located, and provides an **InputStream** that can be used to read the resource's contents. This method provides a quick and easy way to retrieve the contents of a URL, without the added complexity of dealing with a **URLConnection** object.

## ► Retrieving a Resource with the URL Class

- There are two URL methods that can assist in retrieving the contents of a remote resource:
  1. **InputStream URL.openStream()**
  2. **URLConnection URL.openConnection();**
- For greater control over how the request is made, a **URLConnection** object created by invoking the **URLConnection()** method would be used. In many situations, however, a simpler way to retrieve the contents of a resource is called for. The **openStream()** method returns an **InputStream**, which makes reading a resource simple.

# ► Retrieving a Resource with the URL Class

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

public class FtpUrls {
    public static void main(String[] args) throws IOException {
        // URL url = new URL("ftp://pvtinh:Tinh7570@192.168.1.6/t.txt");
        URL url = new URL("http://192.168.1.2/cackhoa.html");
        InputStream uin = url.openStream();
        BufferedReader in = new BufferedReader(new
  InputStreamReader(uin));

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