Input/Output[Part - 1]

- **Input/Output** in java is stream based. A stream represents sequence of bytes or characters. Stream-based input/output has following advantages over conventional I/O.
  1. Abstraction
  2. Flexibility
  3. Performance
- *In Java, two types of input output streams are defined :-*
  - Byte oriented streams(*8-bit sequence*)
  - Character oriented streams(*16-bit sequence*)
- ***Commonly used byte oriented streams-***

InputStream

is an abstract class that is extended by all *byte oriented input* streams.

- **ByteArrayInputStream** is an input stream that is used to read bytes from a byte array.
- **BufferedInputStream** is an input stream used to read bytes from a buffer.
- **FileInputStream** is an input stream used to read bytes from a file.

InputStream class contains an abstract **read()** method that is defined by all its subclasses.

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

OutputStream

is an abstract class that is extended by all *byte oriented output* streams.

- **ByteArrayOutputStream** is an output stream that is use to write bytes to a byte array.
- **BufferedOutputStream** is an output stream use to write bytes to a buffer.
- **FileOutputStream** is an output stream used to write bytes to a file.
- **PrintStream** is an output stream use to write primitive types, characters & strings to another stream. This stream provides **print()** and **println()** methods.

OutputStream class contains an abstract **write()** method that is defined by all its subclasses. It is used to write a byte or block of bytes to **OutputStream**.

- *public void write(int byte) throws IOException;*
- *public void write(byte[ ]) throws IOException;*

General Signature of creating an Input/output stream –

public TypeInputStream(Object source);
public TypeOutputStream(Object sink);

Example -

- *Create an InputStream to read bytes from a byte array.*

```
byte a[ ] = {1, 2, 3, 4, 5};
ByteArrayInputStream b = new ByteArrayInputStream(a);
b.read();
```

- *Stream to read bytes from a file named "a.txt".*
  FileInputStream f = new FileInputStream(a);
  f.read();

- *Stream to read bytes from keyboard*
  BufferedInputStream b = new BufferedInputStream(System.in);
  b.read();

- ***Commonly used character oriented streams-***

Reader

is an abstract class that is extended by all *character oriented input* streams.

- **CharArrayReader** is an input stream use to read characters from a character array.
- **BufferedReader** is an input stream use to read characters and strings from a buffer.
- **FileReader** is an input stream use to read characters from a file.
- **InputStreamReader** is an input stream use to convert bytes to characters.

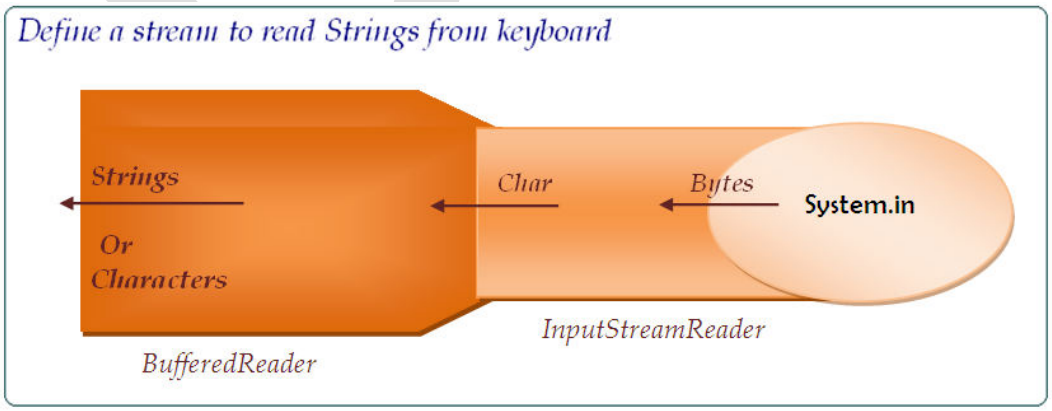
Writer

is an abstract class that is extended by all *character oriented output* streams.

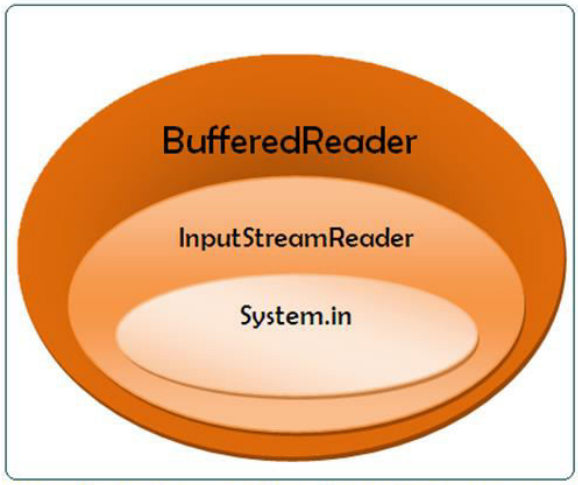
- **CharArrayWriter** is an output stream use to write characters to a character array.
- **BufferedWriter** is an output stream use to write characters to a buffer.
- **FileWriter** is an output stream use to write characters to a file.
- **OutputStreamWriter** is an output stream use to convert characters to bytes.
  - **PrintWriter** is the character oriented version of **PrintStream**.
  - etc.

Reader class provides an abstract **read()** method that is defined by all its subclasses. Apart from **read()** method **BufferedReader** class defines an additional method named **readLine()** method that reads a complete line of text from a stream & returns it as a string.

public String readLine() throws IOException;



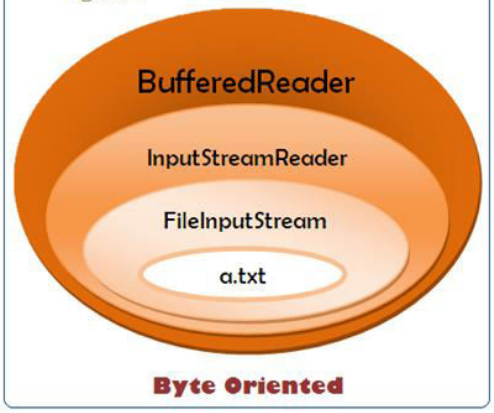
The same diagram can also be represented as shown below. Diagram shown below will be used everywhere further in the notes ahead.



BufferedReader b = new BufferedReader(new InputStreamReader(System.in));

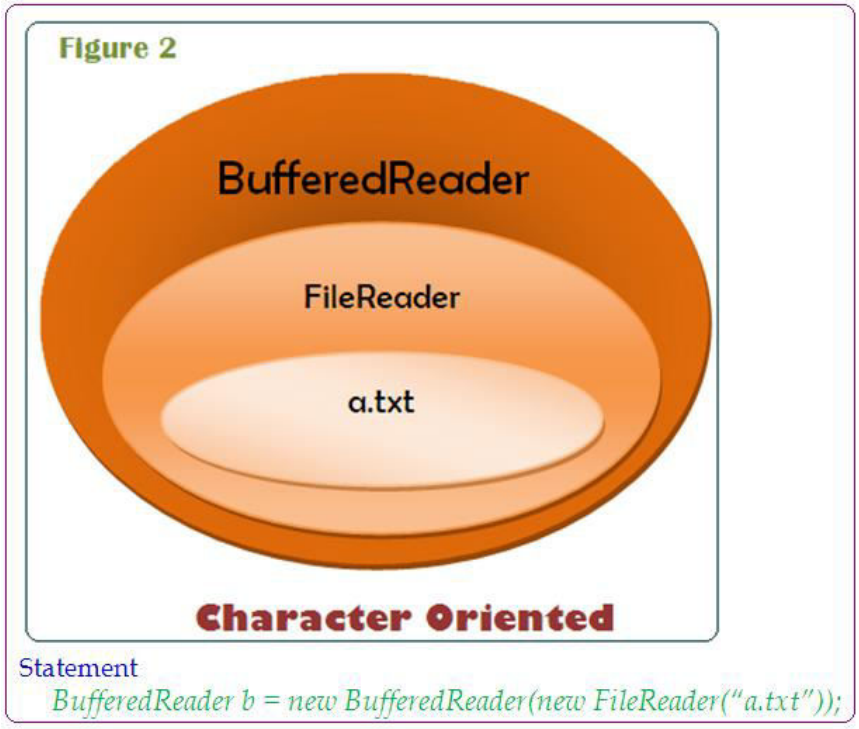
Define two input streams to read data from a file named "a.txt" line by line.

Figure 1



Statement

BufferedReader b = new BufferedReader(new InputStreamReader(new FileInputStream("a.txt")));

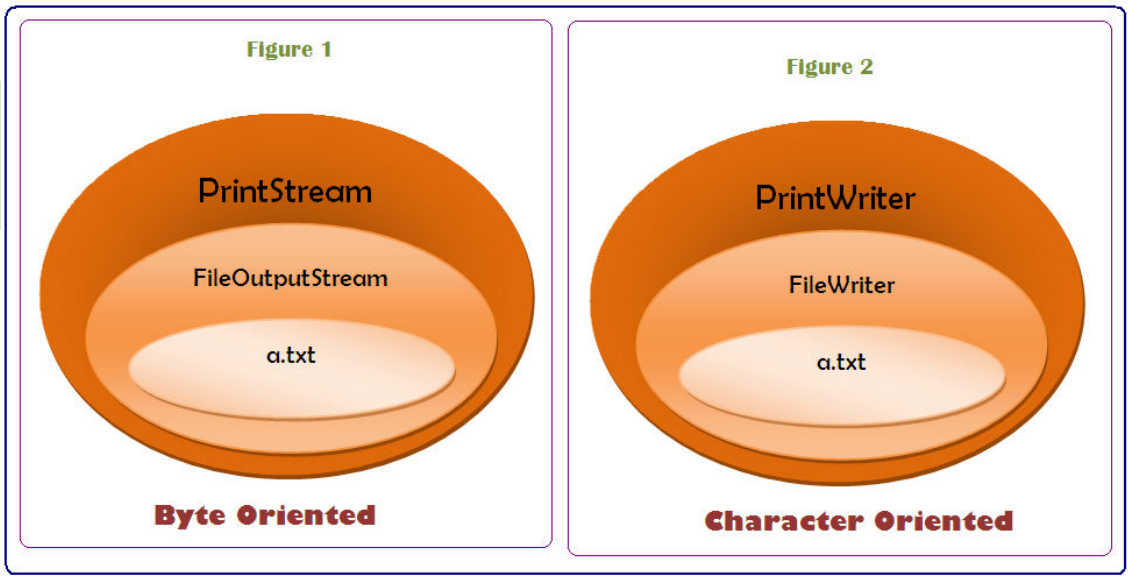


Statement

BufferedReader b = new BufferedReader(new FileReader("a.txt"));

♣

Define two output streams to write data to a file named "a.txt", line by line.



- Statements

NOTE : All these I/O streams are defined in java.io package.

- *Example1*

```
//to display the contents of a text file on console.
// name of file is given as command line argument.
// data is read from the file byte by byte.

import java.io.*;

class Display
{
public static void main(String[] args)
{
        try
        {
                FileInputStream f = new FileInputStream(args[0]);
                int ch;
                while(true)
                {
                        ch=f.read();
                        if(ch==1)
                                break;
                        System.out.println((char) ch);
                }
        }
        catch(Exception e)
        {
                System.out.println(e);
        }
}
}
```

- *Example2*

```
//to display the contents of a text file on console.
// name of file is given as command line argument.
// data is read from the file line by line.

import java.io.*;

class Display1
{
public static void main(String[] args)
{
        try
```

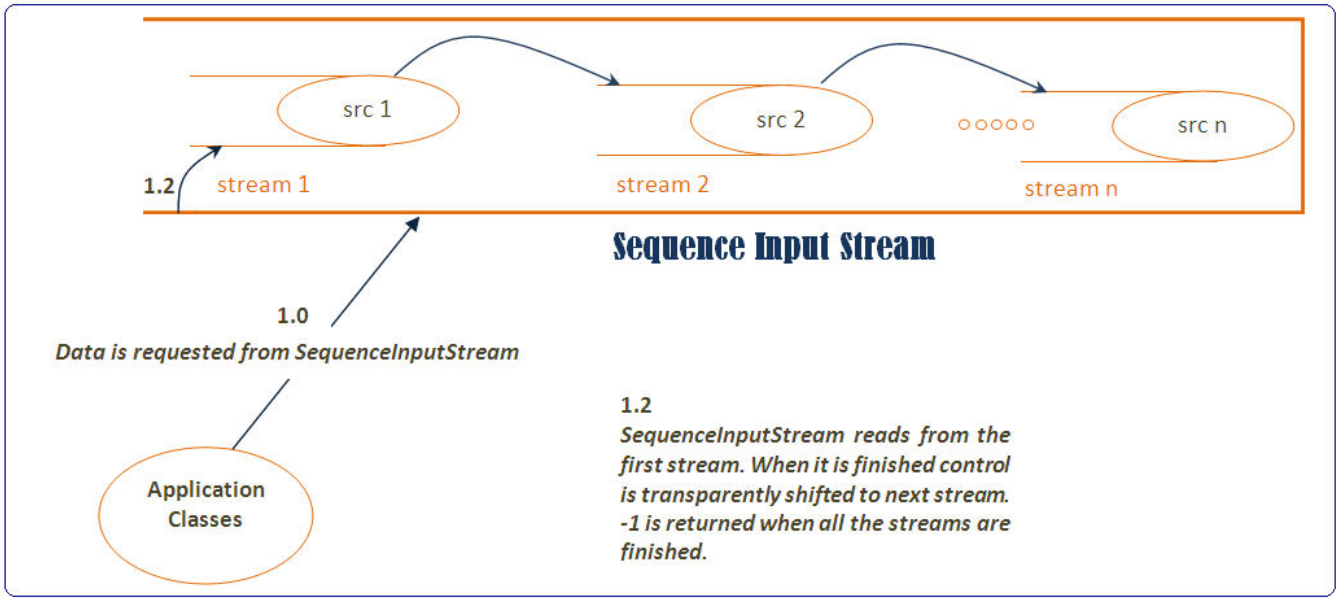
```
        {
                BufferedReader f = new BufferedReader(new InputStreamReader(new
FileInputStream(args[0])));
                while(true)
                {
                        String line=f.readLine();
                        if(line==null)
                                break;
                        System.out.println(line);
                }
        }
        catch(Exception e)
        {
                System.out.println(e);
        }
}
}
```

- *Example3*

```
        //to display the contents of a text file on console.
        // name of file is given as command line argument.
        // data is read from the file in one go.

import java.io.*;

class Display3
{
public static void main(String[] args)
{
        try
        {
                FileInputStream f = new FileInputStream(args[0]);
                byte a[] = new byte[f.available()];
                f.read(a);
                String s = new String(a);
                System.out.println(s);
        }
        catch(Exception e)
        {
                System.out.println(e);
        }
}
}
```

Date: 08.08.10

- **SequenceInputStream** is used to read data from multiple input streams in a sequence. This stream groups multiple input streams into a single logical input stream.



- **SequenceInputStream** object can be created using either of the following constructors.

  public SequenceInputStream (InputStream stream1, InputStream stream2);
  public SequenceInputStream (Enumeration streams);

PROGRAM

"Copier.java"

```
package IO;
import java.io.*;

// to copy contents of two files to a file.
// file names are provided as command line arguments.

class Copier
{
     public static void main(String[] arr)
     {
          try
          {
               SequenceInputStream in = new SequenceInputStream(new
                    FileInputStream(arr[0]), new FileInputStream(arr[1]));
               FileOutputStream out = new FileOutputStream(arr[2]);
```

```
                while(true)
                {
                        int ch = in.read();
                        if(ch==1)
                                break;
                        out.write(ch);
                }
                out.close();
                System.out.println("Successfully copied.");
        }
        catch(Exception e)
        {
                System.out.println(e);
        }
    }
}
```

"Creator.java"

```
package IO;
import java.io.*;

// to create a text file.
// file name is given as command line arguments.
// contents are read from keyboard line by line and saved to the file.

class Creator
{
    public static void main(String[] arr)
    {
        try
        {
            BufferedReader b = new BufferedReader(new
                InputStreamReader(System.in));

            PrintStream out = new PrintStream ( new
                FileOutputStream(arr[0]));

            System.out.println("Enter text, end to save.");
            while(true)
            {
                String s = b.readLine();
                if(s.equals("end"))
                    break;
                out.println(s);
            }
            out.close();
```

```
                System.out.println("Successfully created.");
            }
            catch(Exception e)
            {
                System.out.println(e);
            }
        }
}
```

----------X----------

PROGRAM

"MyCreator.java"

```
package IO;
import java.io.*;

class MyCreator
{
    public static void main(String[] arr)
    {
        try
        {
            BufferedReader b = new BufferedReader(new
                InputStreamReader(System.in));

            PrintStream myout = new PrintStream ( new
                FileOutputStream(arr[0]));

            System.out.println("Enter text, end to save.");
            PrintStream temp = System.out; // Reference of standard
output stream is used.
            System.setOut(myout); // Standard output is redirected to a
file.
            while(true)
            {
                String s = b.readLine();
                if(s.equals("end"))
                    break;
                System.out.println('s'); //data shall be written to the
file
            }
            myout.close();
            System.out.println(temp); //standard output is reset.
            System.out.println("Successfully created.");
        }
        catch(Exception e)
```

```
        {
            System.out.println(e);
        }
    }
}
```

- **Serialization** is the process of converting the state of an object into a sequence of bytes in such a manner that given this sequence of bytes object with the same state can be reconstructed at a later stage.
  - **ObjectOutputStream**
  - **ObjectInputStream**
  - These classes are used to serialize & deserialize the objects.
  - **writeObject()** method of ObjectOutputStream is used for serializing objects.

    public void writeObject(Object o) throws NotSerilizableException;

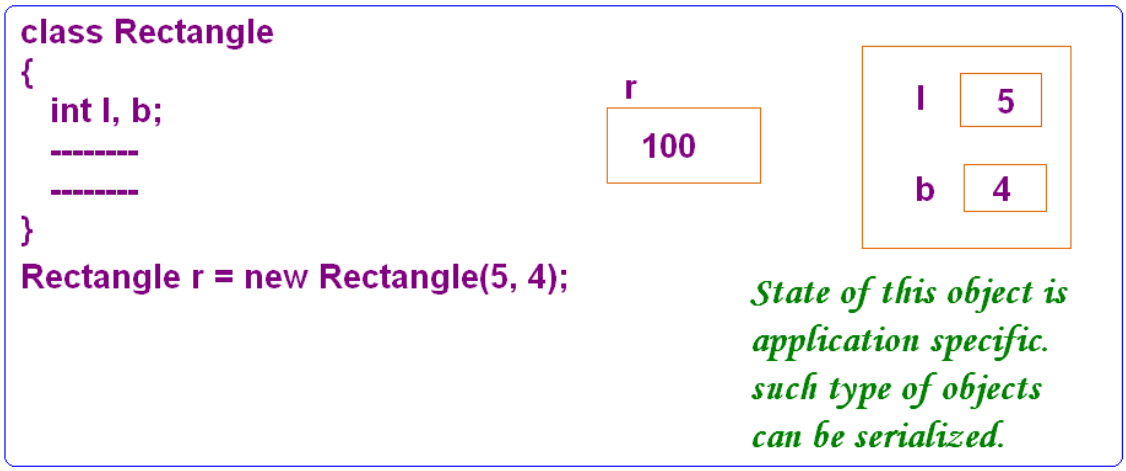
NOTE: *Facility of serialization is not provided to objects of all classes by default.*

- In order to serialize objects of a class, class must implement **java.io.Serializable** interface. This is a marker interface.
  - A marker interface is an interface that doesn't contain any methods.
  - Example

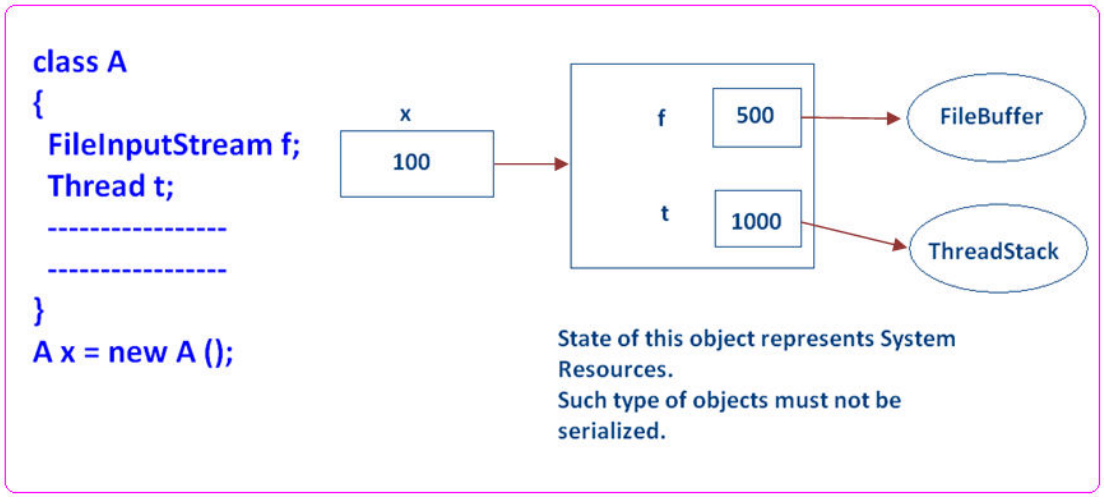
    ```
    interface Serializable
    { }
    ```
  - Marker interfaces are used to mark a classes as part of a group so that some additional service can be provided to the classes of this group.
- Facility of serializable is not provided by default to objects of all classes because state of all objects must not be serialized.
  - State of objects can be of two types-
    - Application Specific.
    - System/JRE Specific.

    Example

    1.



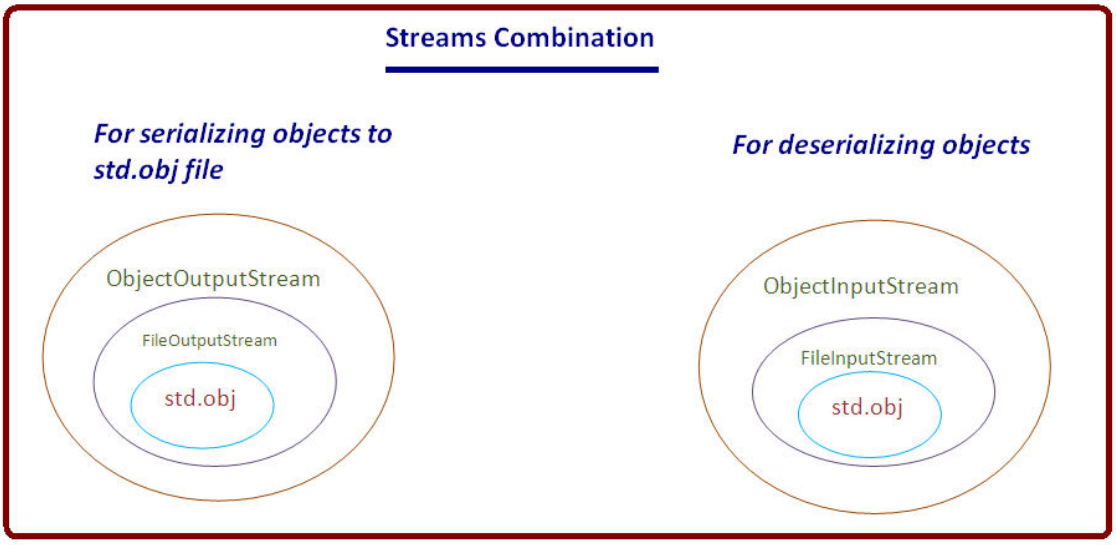
2.



Dated:14.08.10

```
package IO;

public class Student implements java.io.Serializable
{
      private String name, course;
      int fee;

      public Student(String n, String c, int f)
      {
            name = n;
            course = c;
            fee = f;
      }

      public void display()
      {
```

```
            System.out.println(name + "\t" + course + "\t" + fee);
      }
}
```



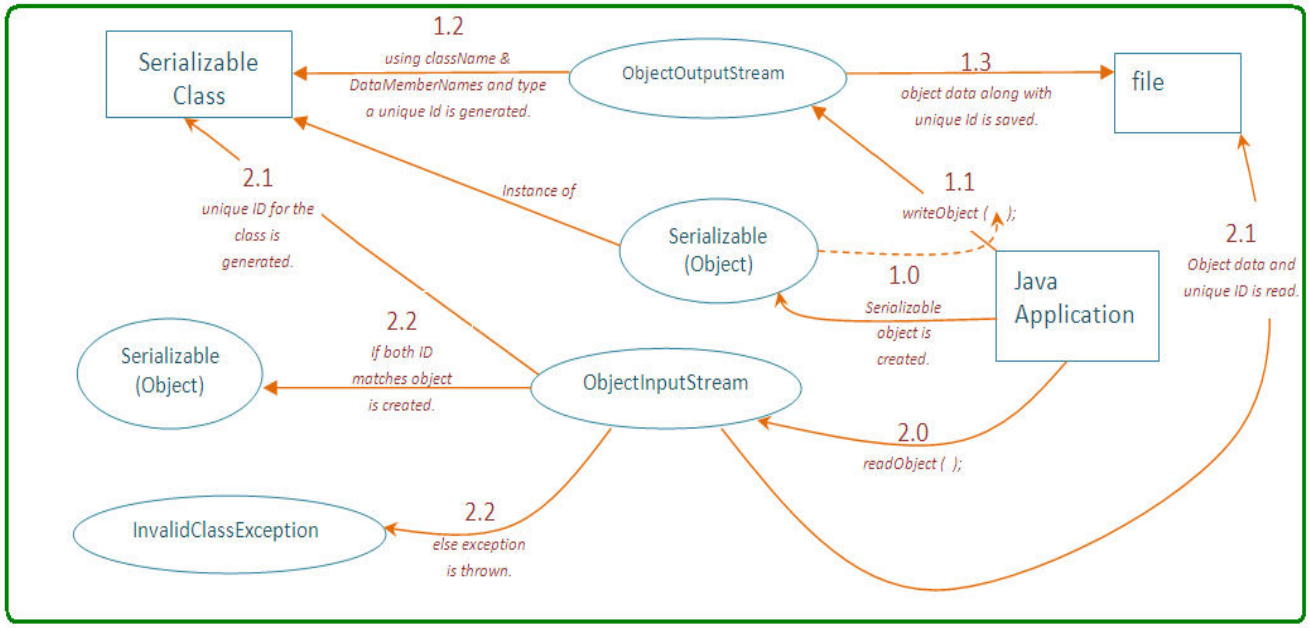
"ObjSaver.java"

```
package IO;
import java.io.*;

class ObjSaver
{
      public static void main(String[] args)
      {
            try
            {
                  Student s1 = new Student("Amar", "Java", 12000);
                  Student s2 = new Student("Ravi", ".Net", 8000);
                  System.out.println("Serializing following objects...");
                  s1.display();
                  s2.display();
                  ObjectOutputStream out = new ObjectOutputStream(new
FileOutputStream("std.obj"));
                  out.writeObject(s1);
                  out.writeObject(s2);
                  out.close();
                  System.out.println("Successfully serialized.");
            }
            catch(Exception e)
            {
                  System.out.println(e);
            }
      }
}
```

"ObjGetter.java"

```
package IO;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

class ObjGetter
{
     public static void main(String[] args)
     {
          try
          {
               System.out.println("Deserializing following objects...");
               ObjectInputStream in = new ObjectInputStream(new
FileInputStream("std.obj"));

               Student s1 = (Student) in.readObject();
               Student s2 = (Student) in.readObject();
               System.out.println("Following objects are serialized.");
               s1.display();
               s2.display();
          }
          catch(Exception e)
          {
               System.out.println(e);
          }
     }
}
```

- **transient** keyword is used to mark those data members of a **Serializable** class which are not to be Serialized.

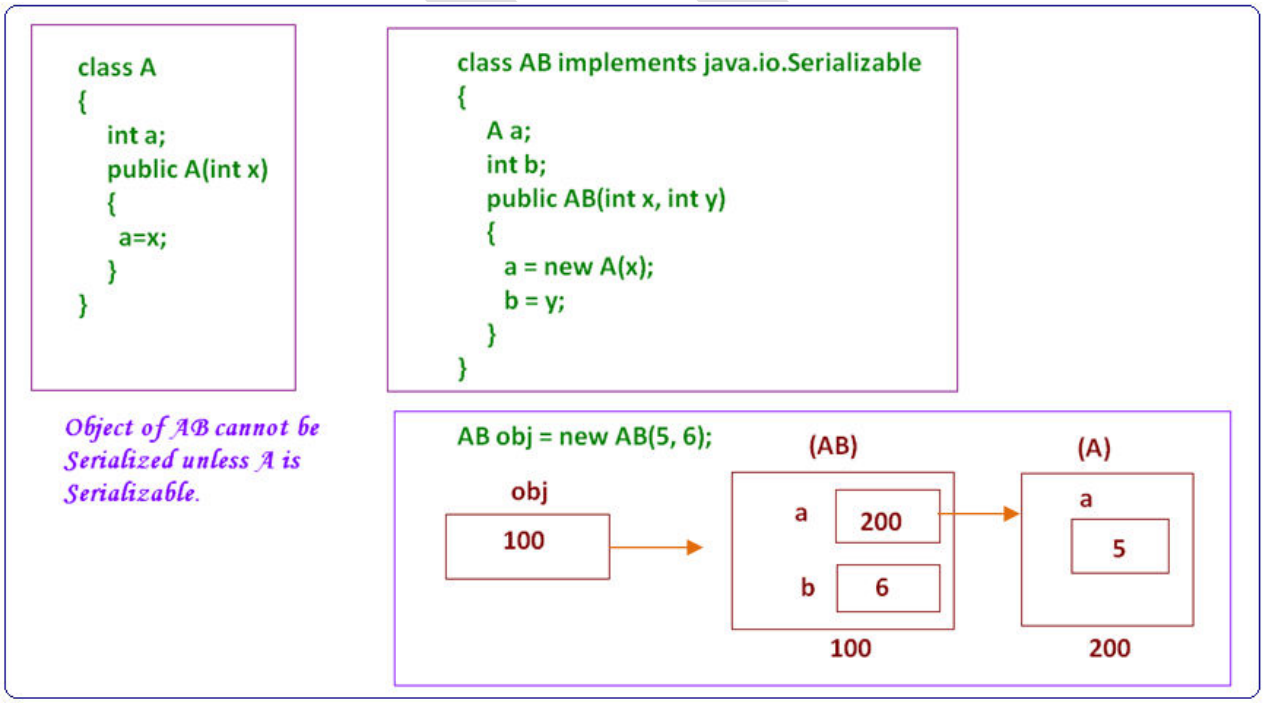


NOTE:

ObjectOutputStream & **ObjectInputStream** classes generates unique Id's for the class only if a class does not contain

static final long data member by the name **serialversionUID**.

- jdk provides a tool by the name **serialver** that generates a **serialversion ID** for a class.
- NOTE: Composite objects can only be serialized if all their constituent members are serializable.



- *Reading & writing primitive types*

  DataInputStream & **DataOutputStream** classes were used for reading & writing primitive types. These classes are deprecated in Jdk 1.5.

From Jdk 1.5 onwards **java.util.Scanner** class is used for reading primitive values from a stream.

- *Commonly used methods of Scanner class are:-*
  public int nextInt();
  public char nextChar();
  public byte nextByte();
  public float nextFloat();
  public String nextLine();
  etc.

Example 1

```
package IO;

import java.util.Scanner;

class Adder
{
	public static void main(String[] args)
	{
		Scanner s = new Scanner(System.in);
		System.out.println("Enter First Number....");
		int a = s.nextInt();
		System.out.println("Enter Second Number....");
		int b = s.nextInt();
		int c = a+b;
		System.out.println("Result is = " + c);
	}
}
```

```
1  package IO;
2
3  import java.util.Scanner;
4
5  class Adder
6  {
7      public static void main(String[] args)
8      {
9          Scanner s = new Scanner(System.in);
10         System.out.println("Enter First Number....");
11         int a = s.nextInt();
12         System.out.println("Enter Second Number....");
13         int b = s.nextInt();
14         int c = a+b;
15         System.out.println("Result is = " + c);
16     }
17 }
```

Problems | Javadoc | Declaration | Console

```
<terminated> Adder [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (Aug 19, 2010 1:03:42 PM)
Enter First Number....
12
Enter Second Number....
23
Result is = 35
```

Example 2

```
package IO;

import java.util.Scanner;
import static java.lang.System.*;

class Adder
{
     public static void main(String[] args)
     {
          Scanner s = new Scanner(in);
          out.println("Enter First Number....");
          int a = s.nextInt();
          out.println("Enter Second Number....");
          int b = s.nextInt();
          int c = a+b;
          out.println("Result is = " + c);
     }
}
```

```
1  package IO;
2
3  import java.util.Scanner;
4  import static java.lang.System.*;
5
6  class Adder
7  {
8      public static void main(String[] args)
9      {
10         Scanner s = new Scanner(in);
11         out.println("Enter First Number....");
12         int a = s.nextInt();
13         out.println("Enter Second Number....");
14         int b = s.nextInt();
15         int c = a+b;
16         out.println("Result is = " + c);
17     }
18 }
```

Problems | Javadoc | Declaration | Console

```
<terminated> Adder [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (Aug 19, 201
Enter First Number....
12
Enter Second Number....
26
Result is = 38
```

```
1  package IO;
2
3  import java.util.Scanner;
4  import static java.lang.System.*;
5
6  class Adder
7  {
8      public static void main(String[]
9      {
10         Scanner s = new Scanner(in);
11         out.println("Enter First Number....");
12         int a = s.nextInt();
13         out.println("Enter Second Number....");
14         int b = s.nextInt();
15         int c = a+b;
16         out.println("Result is = " + c);
17     }
18 }
```

See the changes here.....

See the changes here.....

See the changes here.....

See the changes here.....

See the changes here.....

- *Types of Application can be made in Java-*
  1. Console Application.

2. Windows Application.
3. Web Application.
4. Web Services.
5. Network Application.

- **static import** is the facility of using static members of a class without using their fully qualified name

Syntax-

import static pkgName.className.staticMemberName;

or

import static pkgName.className.*;

- **for-each loop** was introduced to traverse the elements of a collection(group of elements) in a convenient manner.

Syntax –

for(type identifier : collection)

identifier return Next value of the collection in each iteration.

Example –

```
package IO;
import java.util.*;
import static java.lang.System.*;

class ForEachDemo
{
     public static void main(String[] args)
     {
          int a[] = {1, 2, 3, 4, 5};
          out.println("Contents of array using conventional method....");
          for(int i=0; i<a.length; i++)
               out.println(a[i]);
          out.println("Contents of array using for each loop...");
          for(int x:a)
               out.println(x);
     }
}
```

Output –

```
Contents of array using conventional method....
1
2
3
4
5
Contents of array using for each loop...
1
2
3
```

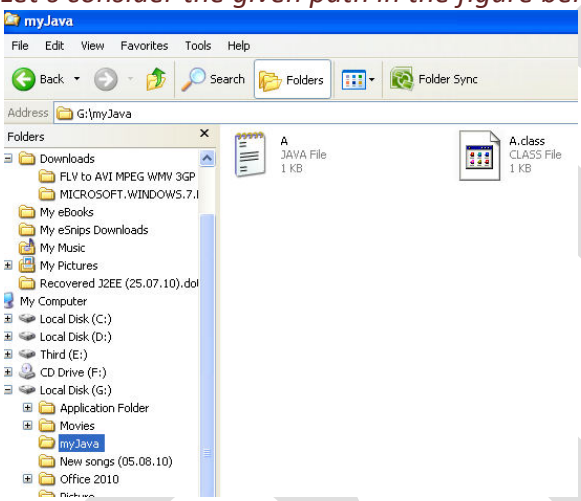
4
5

INPUT-OUTPUT [Part - 2]

- **java.io.File** provides an interface of the File System to a Java Application i.e. using this class a Java Application can find out File & Folders of a drive, can obtain or change their attributes, can create or remove file or folders etc.

- *File object can be created using either of the following constructors:-*
  public File (String path);
  public File (File path, String name);

  Let's consider the given path in the figure below :-



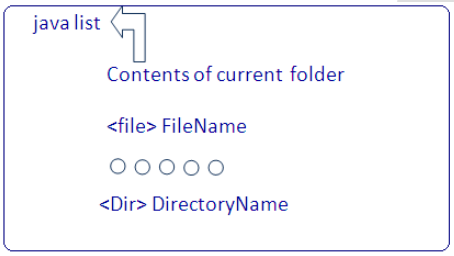
- File objects for A.java & A.class can be created as:-

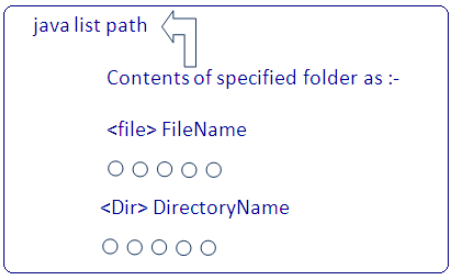
```
File f1  = new File("g:\\myJava\\A.java");
File f2  = new File("g:\\myJava\\A.class");
                or
File p  = new File("g:\\A.java");
File f1  = new File(p, "A.java");
File f2  = new File(p, "A.class");
```

- *Commonly used methods of File class:-*
  - **getName()** method returns the name of the File or Folder.
    public String getName();
  - **getPath()** method returns the path of the File or Folder.
    public String getPath();
  - **isDirectory()** method returns true if File object represents a folder.
    public boolean isDirectory();
  - **isFile()** method returns true if File object represents a file.
    public boolean isFile();

- **exists()** method returns true if File or Folder for the File object exists in the File system.
  public boolean exists();
- **isHidden()** returns true if File object represents a hidden File or Folder.
  public boolean isHidden();
- **isReadOnly()** returns true if File object represents a read-only File or Folder.
  public boolean isReadOnly();
- **list()** method returns the contents of a folder.
  public String[] list();
- **mkdir()** method is used to create a folder.
  public boolean mkdir();
- **renameTo()** method is used to change the name of the File or Folder.
  public boolean renameTo();
- etc.

- Define a class NamedList that contains main method which can be provided path of a folder as command line argument.
  This class displays the contents of the specified or current folder in the following format.

1


2


Dated: 21.08.10

Networking

- java.net package provides classes which facilitate development of networking applications in Java.
  - *Commonly used classes of this package are –*
    - InetAddress
    - Socket
    - ServerSocket
    - DatagramSocket
    - DatagramPacket

- InetAddress class provides object representation of IP Address of machines on a network. This class doesn't provide public constructors rather it provides factory methods for creating its objects.
  - Factory is a creational design pattern that is used to control the creation of objects. This design pattern is implemented with the help of factory classes. A factory class is a class that contains factory methods.
- A factory method is a method which creates objects.
  - *Types of Design Patterns*
    - Creational Design Pattern.
    - Structural Design Pattern.
    - Behaviour Design Pattern.

- What is Singleton?
  - When we create only one object of a class is called singleton.
    - Example

```
class A
{
 private static final obj;

 private A()
 {}

 public static A getA()
 {
  if(obj == null)
       obj =new A();
  return obj;
 }
}
```

~~A x = new A();~~ ×

A x = A.getA(); ✓

A y = A.getA(); ✓

- *Following factory methods are provided by* **InetAddress** *class-*
  - **getLocalHost()** method returns an **InetAddress** object which represents the IP Address of the current machine.
    public Static InetAddress getLocalHost() throws UnknownHostException;

    Example –
    InetAddress a = InetAddress.getLocalHost();

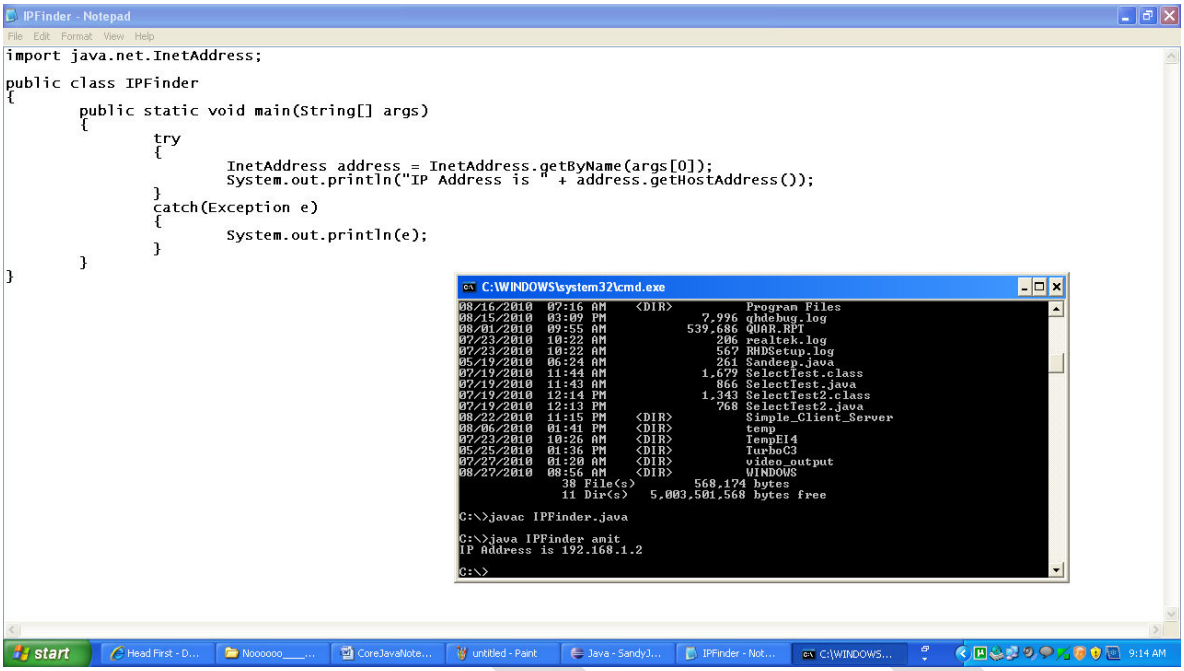
  - **getName()** method returns an **InetAddress** object which represents IP Address of the given machine.

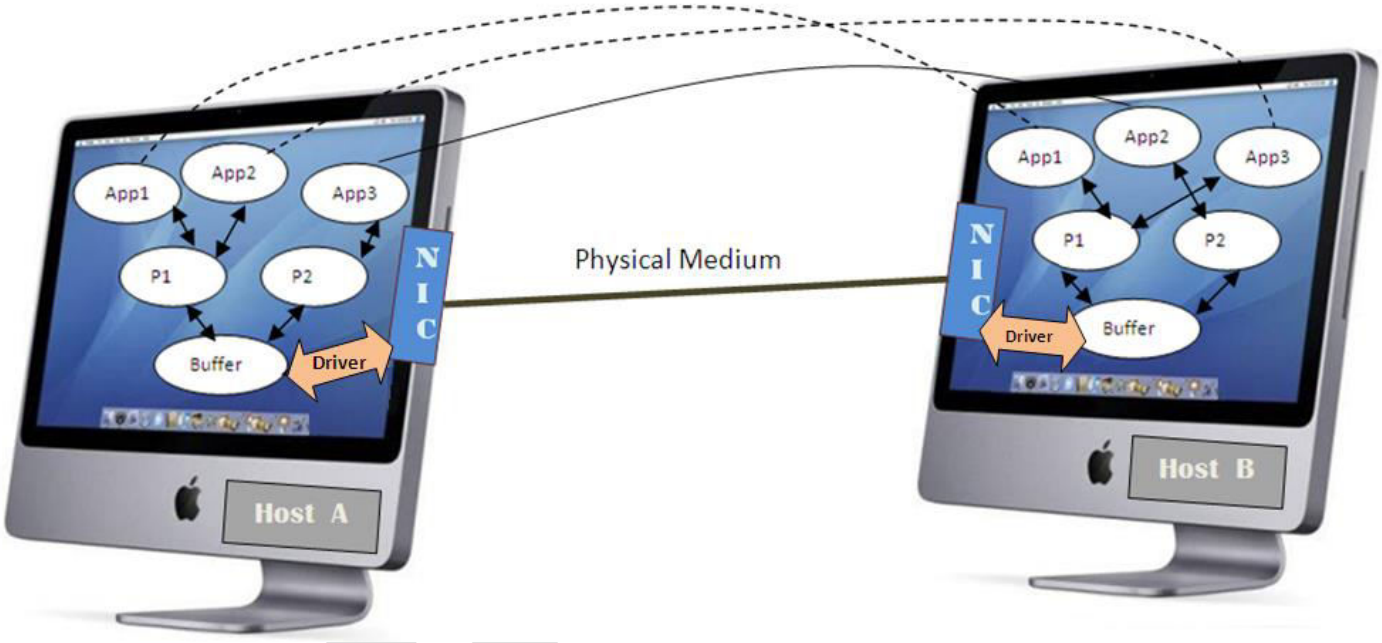
public static InetAddress getByName(String hostName) throws UnknownHostException;

- **getAllByName()** method returns an array of **InetAddress** objects. Each element of the array represents an IP Address of the given host.
  public static InetAddress[] getAllByName(String hostName) throws UnknownHostException;
- getHostName() method returns name of the machine.
  public String getHostName();
- **getHostAddress()** method returns the IP Address as a String.
  public String getHostAddress();
- etc.

```
//Program to find out IP Address of a host on a network.
//hostname is provided as command line argument.

package Networking;

import java.net.InetAddress;

public class IPFinder
{
	public static void main(String[] args)
	{
		try
		{
			InetAddress address = InetAddress.getByName(args[0]);
			System.out.println("IP Address is " + address.getHostAddress());

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

Output -





- **Socket** is a logical end point of a connection.
  - From an application programmer's point of view, a socket is a process that is used by the Application Programmer to send or retrieve data over the network.
  - This process handles protocol specific details on behalf of Application Developer.
  - To facilitate multiplexing of different logical connections over a single physical medium concept of port was introduced.
  - A port is a numbered socket.

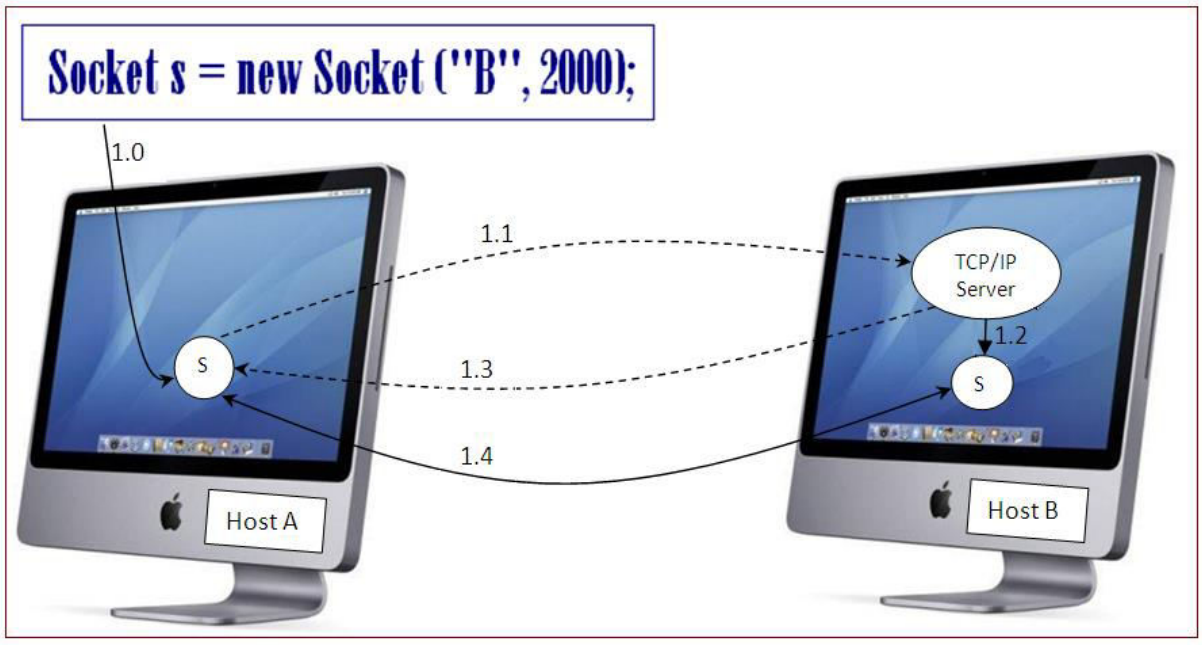
- Port Number 0 to 1024 are reserved for standard protocols such as TCP/IP, Http, Ftp, SMTP etc.

- **java.net.Socket** class represents a TCP/IP socket.

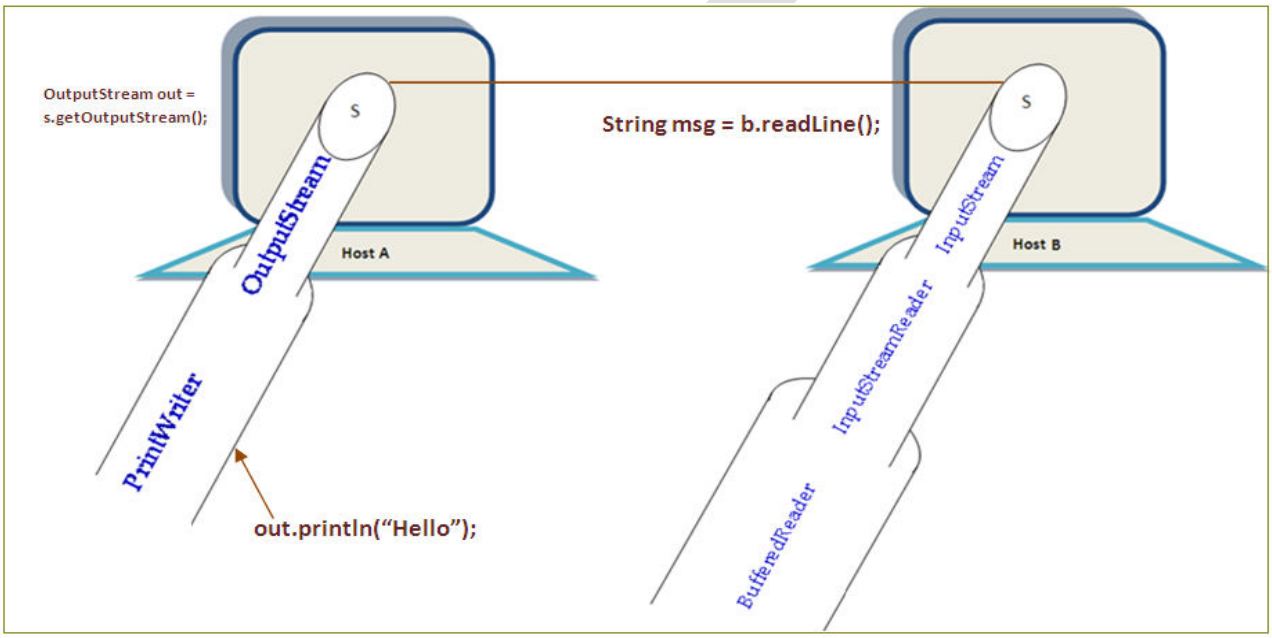
  A **Socket** *object can be created using either of the following constructors:-*

  public Socket (String hostname, int port) throws UnknownHostException, IOException;
  public Socket (InetAddress ipAddress, int port) throws UnknownHostException, IOException;

  - *Methods-*
    - **getInputStream()** method returns an **InputStream** to read data from a **Socket**.
      public InputStream getInputStream();
    - **getOutputStream()** method returns an **OutputStream** to write data to a **Socket**.
      public OutputStream getOututStream();
    - **close()** method is used to close the **Socket**.
      public void close();



1.0 Socket object is created.
1.1 From the constructor of Socket, a connection request is sent to the TCP/IP Server running on specified host for a connection on given port.
1.2 If TCP/IP Server is configured for the requested port, connection is completed by creating server-side Socket.
1.3 Acknowledgement of connection is sent.
1.4 Communication is initiated.



- **java.net.ServerSocket** class represents the functionality of a TCP/IP server.
- A TCP/IP server is responsible for receiving & completing TCP/IP connection requests.
- A **ServerSocket** object can be created using either of the following constructor.
  public ServerSocket(int port);
  public ServerSocket(int port, int maxQueueLength);

  Methods-
  - **accept()** method is used to instruct TCP/IP server to start listening connection request. This method blocks the TCP/IP server until a connection request is received.
    public Socket accept();
  - **close()** is used to close the TCP/IP server.
    public void close();

Dated : 22.08.10

Program (**Server.java)**

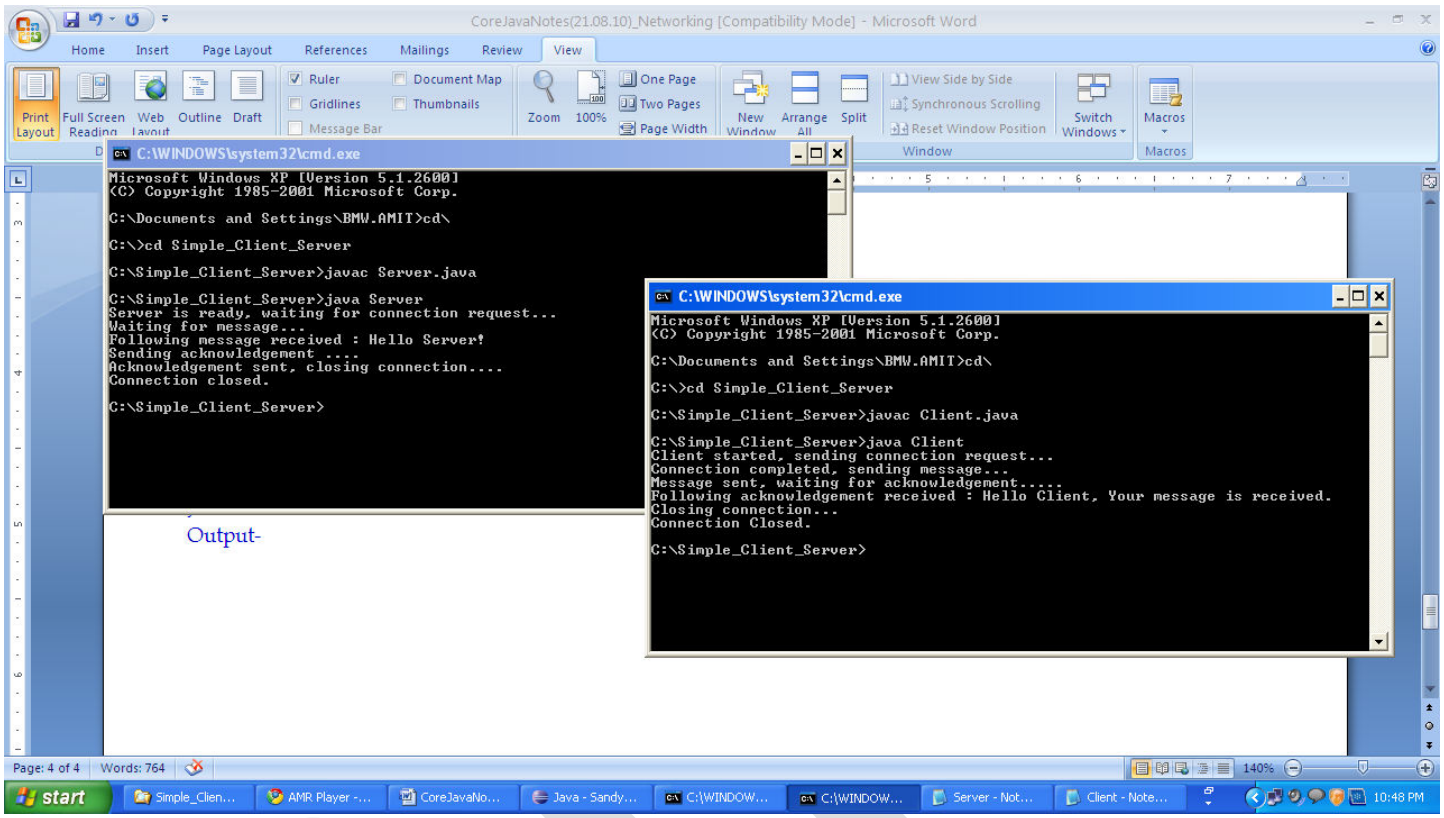
```
import java.net.*;
import java.io.*;

class Server
{
	public static void main(String[] args)
	{
		try
		{
			ServerSocket server = new ServerSocket(2000);
			System.out.println("Server is ready, waiting for connection request...");
			Socket s = server.accept();
			System.out.println("Waiting for message...");
			BufferedReader b = new BufferedReader(new
InputStreamReader(s.getInputStream()));
			String msg = b.readLine();
			Thread.sleep(1000);
			System.out.println("Following message received : " + msg);
			System.out.println("Sending acknowledgement ....");
			Thread.sleep(2000);
			PrintWriter out = new PrintWriter(s.getOutputStream());
			out.println("Hello Client, Your message is received.");
			out.flush();
			System.out.println("Acknowledgement sent, closing connection....");
			Thread.sleep(5000);
			System.out.println("Connection closed.");
			s.close();
		}
		catch(Exception e)
		{
			System.out.println(e);
		}
	}
}
```

Program **(Client.java)**

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

class Client
{
	public static void main(String[] args)
	{
		try
		{
			System.out.println("Client started, sending connection request...");
			Thread.sleep(2000);
			Socket s = new Socket("localhost", 2000);
			Thread.sleep(1000);
			System.out.println("Connection completed, sending message...");
			PrintWriter out = new PrintWriter(s.getOutputStream());
			Thread.sleep(2000);
			out.println("Hello Server!");
			out.flush();
			System.out.println("Message sent, waiting for acknowledgement.....");
			BufferedReader b = new BufferedReader(new
InputStreamReader(s.getInputStream()));
			String msg = b.readLine();
			Thread.sleep(1000);
			System.out.println("Following acknowledgement received : " + msg);
			System.out.println("Closing connection...");
			Thread.sleep(5000);
			System.out.println("Connection Closed.");
			s.close();
		}
		catch(Exception e)
		{
			System.out.println(e);
		}
	}
```

}

Output-



- **DatagramSocket** class provides the facility of sending & receiving UDP packets.
  public DatagramSocket (int port);
  Methods-
  - **send()** method is used to send UDP packets.
    public void send(DatagramPacket packet);
  - **receive()** method is used to receive UDP packets.
    public void receive(DatagramPacket packet);
  - **close()** method is used to close **DatagramSocket**.
    public void close();

- **DatagramPacket** class provides the object representation of UDP packets.
  - *Constructors are –*
    public DatagramPacket (byte[] data, int size, InetAddress hostAddress, int port); // used to send data
    public DatagramPacket (byte[] data, int size); // used to receive data
  - *Methods are –*

public byte[] getData();
public InetAddress getHost();
public int getPort();
etc.

Program (**UdpSender.java**)

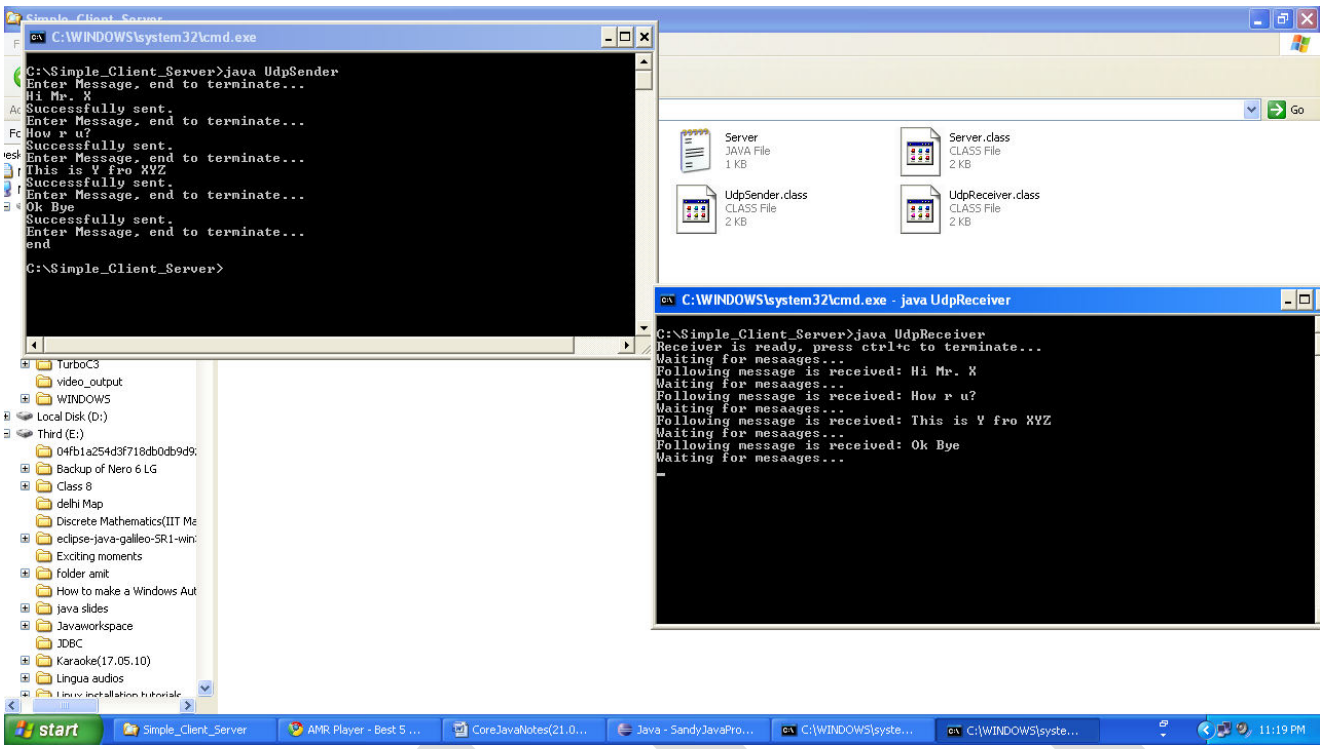
```
import java.net.*;
import java.util.Scanner;
import java.io.*;

class UdpSender
{
	public static void main(String[] args)
	{
		try
		{
			DatagramSocket sender = new DatagramSocket(3000);
			Scanner in = new Scanner(System.in);
			while(true)
			{
				System.out.println("Enter Message, end to terminate...");
				String msg = in.nextLine();
				if(msg.equals("end"))
					break;
				DatagramPacket packet = new DatagramPacket(msg.getBytes(),
msg.length(), InetAddress.getLocalHost(), 4000);
				sender.send(packet);
				System.out.println("Successfully sent.");
			}
			sender.close();
		}
		catch(Exception e)
		{
			System.out.println(e);
		}
	}
```

```
}
```

Program **(UdpReceiver.java)**

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

class UdpReceiver
{
	public static void main(String[] args)
	{
		try
		{
			DatagramSocket receiver = new DatagramSocket(4000);
			System.out.println("Receiver is ready, press ctrl+c to terminate...");
			while(true)
			{
				System.out.println("Waiting for mesaages...");
				DatagramPacket packet = new DatagramPacket(new byte[100], 100);
				receiver.receive(packet);
				String msg = new String(packet.getData());
				System.out.println("Following message is received: " + msg.trim());
			}
		}
		catch(Exception e)
		{
			System.out.println(e);
		}
	}
}
```

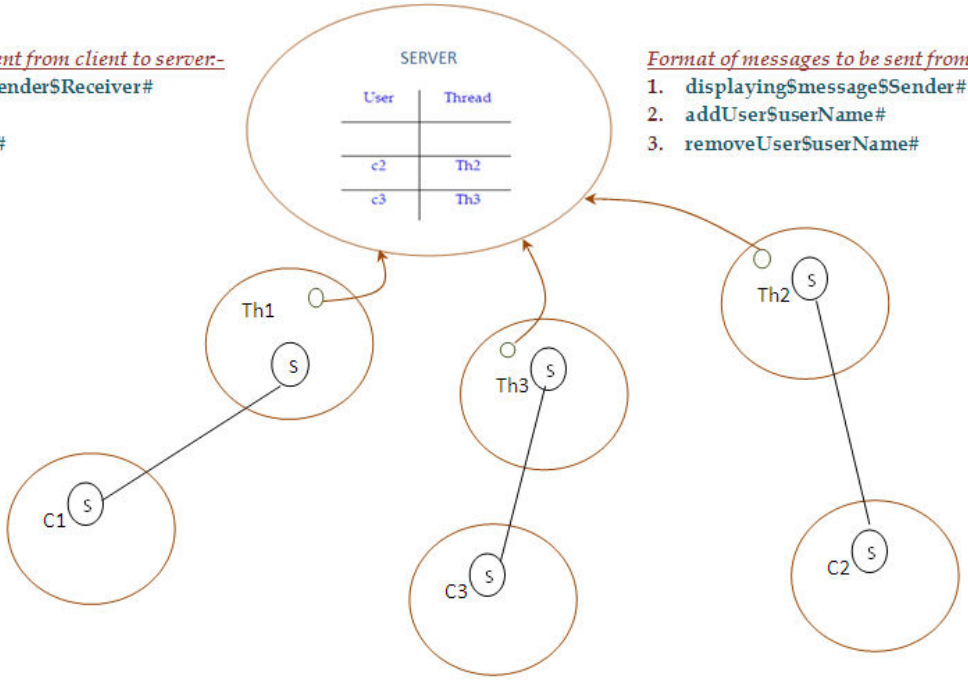
Output-



-

Format of messages to be sent from client to server:-

1. delivering$message$Sender$Receiver#
2. Connect$userName#
3. disconnect$userName#



Format of messages to be sent from server to client:-

1. displaying$message$Sender#
2. addUser$userName#
3. removeUser$userName#