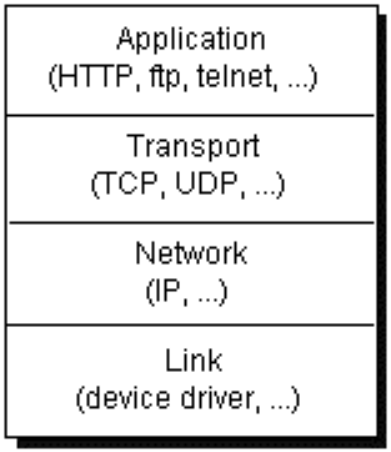
The Java Series

Network and WWW Programming
Servlets

Networking Basics

IP Based networking:



- Systems can talk to the network using a series of standards (stack of protocols).
- Any system (O.S.) contain network modules (drivers, etc.) which know how to access the network using those protocols
  Applications interact with the higher level protocols (TCP and UDP in the case of IP -Internet-).
- O.S. drivers isolate programmers from details.
- Network programmers worry on sending/receiving data, not on how this is done.

Networking Basics

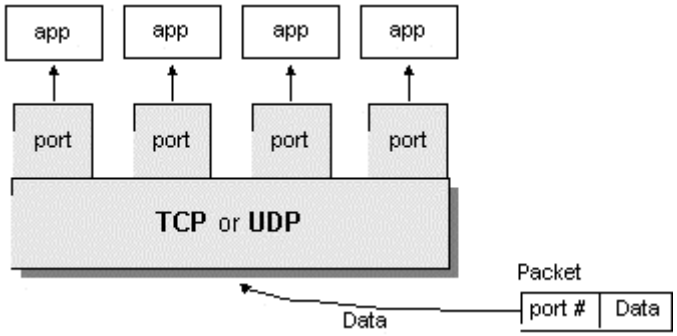
In IP there are two protocols available for programmers:

- TCP: Connection oriented, reliable protocol, providing a reliable data flow between computers.
- UDP: Connectionless protocol which sends independent packages (called datagrams) with no guarantees upon arrival.

When making a program using the network you choose what protocol to use and follow its principles.

Networking Basics: Ports

The same computer usually has a **single physical network connection** but may have **more than one application** using the network.



Network drivers must assure that each message or package sent to the same computer arrive to the application it is intended to.

A **port** is a unique identifier for an application using the network.

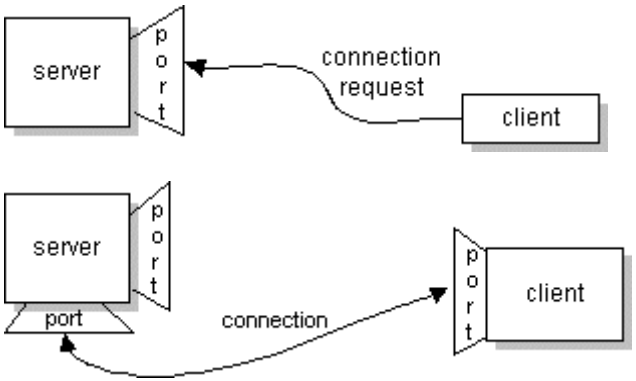
When sending data to a machine one must specify the machine address **and** the port.

This way the machine knows to what application the data must be handled.

Network Basics: Sockets

A socket is one endpoint of a two-way communication link between two programs running on the network. A socket is bound to a port number so that the TCP layer can identify the application that data is destined to be sent

Client/Server TCP:



1. Server listens on a socket bound to a specific port.
2. Client requests connection through that port number.
3. Server creates another socket (bound to a different port number) for communication and continues listening on the first one.

Networking in Java

As always, Java provides a set of classes with which we can implement all these mechanisms from an Object Oriented point of view.

TCP Networking is based on Streams.

UDP Networking is very simple.

There is just a few classes:

- Socket
- ServerSocket
- DatagramSocket
- DatagramPacket

Networking classes in java.net package.

Using TCP in Java (Sockets)

Sending/receiving data on TCP from a client:

- Create a socket.
- Obtain Input/Output stream from socket.
- Build stream chain as desired.
- Write/Read from stream chain.
- Close stream and socket.

Sending/receiving data on TCP from a server:

- Create socket.
- Wait for connections.
- Upon connection create a new socket.
- Obtain Input/Output streams from socket.
- Build stream chain.
- Write/Read from stream chain and close everything.

Scenario 1: A Simple TCP Client

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

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

      try {
            echoSocket = new Socket(args[0], 4444);
            out = new PrintWriter(
                          echoSocket.getOutputStream(), true);
            in = new BufferedReader(
                      new InputStreamReader(
                          echoSocket.getInputStream()));
      } … catch exceptions …

      System.out.println("Server welcome is: "+in.readLine());

      out.println("Hello how are you");
      System.out.println("Server responded: "+in.readLine());

      out.println("Some other stuff");
      System.out.println("Server now responded: "+in.readLine());

      out.close(); in.close(); echoSocket.close();
} }
```

Create socket (machine, port)

Build stream chains for I&O

Write/read from top of stream chains

Scenario 1: A Simple TCP Server

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

        ServerSocket serverSocket = null;
        try {    serverSocket = new ServerSocket(4444);
        } catch (IOException e) { … exception handling code … }

      while (true) {
        Socket clientSocket = null;
        try {    clientSocket = serverSocket.accept();
        } catch (IOException e) { … exception handling code … }

        PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);
        BufferedReader in = new BufferedReader(
                                new InputStreamReader(
                                clientSocket.getInputStream()));
        String inputLine, outputLine;
        out.println("Welcome to the echo server.");
        while ((inputLine = in.readLine()) != null) {
             if (inputLine.equals("Bye.")) break;
             Date now = new Date(System.currentTimeMillis());
             out.println(now+": "+inputLine);
        }
        out.close(); in.close(); clientSocket.close();
      }
    }
}
```

Create server socket (machine)

Accept waits and returns a new socket

Use new socket to build stream chains

Write back whatever is read

TCP Networking

Server process (accept + socket duplication) is different from client process (just send + receive)

Once the connection is established data is sent through the socket with specifying the destination.

For this reason, in Java, there are Server specific classes (ServerSocket) and client specific classes (Socket).

Using UDP in Java (Datagrams)

Same mechanism for client AND server.

To RECEIVE data:

- Create a DatagramSocket with port.
- Create a DatagramPacket.
- Receive the data through Socket into the Packet.

To SEND data:

- Create DatagramSocket with address and port.
- Create DatagramPacket.
- Fill Packet with Data.
- Send Packet through Socket.

Scenario 2: A Simple UDP Client

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

    // get a datagram socket
    DatagramSocket socket = new DatagramSocket();

    // build packet and send request
    byte[] buf = new byte[256];
    String hello = "Hello how re you?";
    buf = hello.getBytes();
    InetAddress address = InetAddress.getByName(args[0]);
    DatagramPacket packet = new DatagramPacket(buf, buf.length, address, 4445);
    socket.send(packet);

    // get response
    byte[] rbuf = new byte[256];
    packet = new DatagramPacket(rbuf, rbuf.length);
    socket.receive(packet);

    // display response
    String received = new String(packet.getData());
    System.out.println("From server : " + received);
    socket.close();
  }
}
```

Create a Datagram Socket

Build data,creates packet including destination address and send it

Build a packet to hold incoming data

Receive packet through Datagram Socket

Do things with received data

Scenario 2: A Simple UDP Server

```
public class DataServer {
  public static void main(String args[]) {
     while (true) {  try {
        DatagramSocket socket = new DatagramSocket(4445);
        byte[] inbuf = new byte[256];

        // receive request
        DatagramPacket packet = new DatagramPacket(inbuf, inbuf.length);
        System.out.println("Waiting to receive packet");
        socket.receive(packet);

        // figure out response
        byte[] outbuf = new byte[256];
        Date now = new Date(System.currentTimeMillis());
        String outs = new String(inbuf);
        outs = now.toString() + " " + outs;
        System.out.println("Replying to "+packet.getAddress());
        outbuf = outs.getBytes();

        // send the response to the client at "address" and "port"
        InetAddress address = packet.getAddress();
        int port = packet.getPort();
        packet = new DatagramPacket(outbuf, outbuf.length, address, port);
        socket.send(packet);
        socket.close();
     } catch (Exception e) { e.printStackTrace(); }
}}
```

Creates a Datagram Socket

Create packet to hold data and receive data

Retrieve address from incoming packet

Fill in data, create outgoing data and send it

UDP Networking

Client and Server implementations are symmetric. The differentiation is given by the functionality they implement not by the code.

socket.receive() to wait for something.

socket.send() to send something.

There is no concept of a continuous connection, we have to include the destination address and port in any packet sent.

HTTP Interaction

HTTP is a TCP based protocol.

We could use the raw Socket class to access directly to port 80 and follow the HTTP specification to retrieve pages, post forms, etc.

But Java provides classes to perform all this interaction in a transparent way.

We can:

- Connect directly to a URL, creating a stream
- Read directly from a URL (stream)
- Write directly to a URL (stream), such as to imitate the data posted to a form.

Scenario 3: Reading from a URL

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

class FetchURL {

   public static void main (String[] args) {

      try {
         URL cernHome = new URL ("http://www.cern.ch");
         URLConnection cernHomeConnection = cernHome.openConnection();

         BufferedReader in = new BufferedReader( new InputStreamReader (
                   cernHomeConnection.getInputStream()));
         String inputLine;

         while ( (inputLine=in.readLine())!=null) {
              System.out.println(inputLine);
         }
         in.close();
      } catch (MalformedURLException e) {
         System.out.println("Malformed URL Exception "+e);
      } catch (IOException e ) {
         System.out.println("IO Exception "+e);
      }
   }
}
```

Create a connection from a URL

Obtain a Stream from the connection

Read from the stream

Scenario 3: Writing to a URL

```
<HTML><HEAD><TITLE>Backwards test</TITLE></HEAD>

<BODY>

<FORM METHOD="POST" ACTION="http://java.sun.com/cgi-bin/backwards">
Please, enter any string:
<br><INPUT TYPE="text" NAME="string" SIZE=58>
<INPUT TYPE="submit" Value="Submit">
<INPUT TYPE="reset" Value="Clear">

</FORM>
</hr>
</BODY>
</HTML>
```

This is a sample form, with an action and one variable (filled in by the user)
The browser just sends to the www server what the user fills in

The following Java application invokes directly the action on the www server, providing the value for the **`string`** variable.

Scenario 3: Writing to a URL

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

        if (args.length != 1) {
            System.err.println("Usage:  java Reverse string_to_reverse");
            System.exit(1);
        }

        String stringToReverse = URLEncoder.encode(args[0]);

        URL url = new URL("http://java.sun.com/cgi-bin/backwards");
        URLConnection connection = url.openConnection();
        connection.setDoOutput(true);

        PrintWriter out = new PrintWriter(connection.getOutputStream());
        out.println("string=" + stringToReverse);
        out.close();

        BufferedReader in = new BufferedReader(
                                new InputStreamReader(
                                connection.getInputStream()));
        String inputLine;

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

        in.close();
    }
}
```

String has to be encoded conforming HTTP

Create a URL connection (set write)

Write variables to it

Read from it

Java Virtual Machines

The JVM is the central mechanism of the JAVA idea.

JVMs can be included in any piece of software.

Anything including a JVM can understand and execute Java bytecodes.

The JDK contains a JVM => we can compile and run java code with the JDK.

Browsers contain JVMs => we can run java code from the browser.

If you include the JVM in an OS s kernel it will be able to run java applications out of the box.

JVMs and software

SW including a JVM must **access** the class files:

- jdk s java interpreter will explore the CLASSPATH variable to look for class files.
- an OS including a JVM may explore predetermined library locations.
- can t a browser get files from through http, ftp,...? well, then it can also get class files from the net

SW including a JVM must give **windowing capabilities** to GUI Java programs.

- jdk s interpreter can open Frames on the desktop.
- browsers allow a java program to use part of their display normally used to render HTML.

JVMs and Browsers

So browsers can be told to:

- Download remote class files.
- Execute those class files in their JVM.
- Allow for a certain space for GUI java programs.

Consequences are important:

- Since the JVM uses the space dedicated for HTML rendering, we need an HTML TAG to specify what java program to run and where in the screen.
- You are going to be executing foreign code locally, so browser s JVM contain restrictions to limit access into your local resources.

Applets

A Java application which can be run from a browser is called an **APPLET** (note the attractive name).

The process a browser follows:

- Upon a request it downloads an HTML page
- If the HTML page contains an <APPLET> tag:
- Initializes its JVM (Starts Java) if not initialized.
- Download the class file pointed by <APPLET>
- Creates an instance of that class.
- Executes the init() and start() methods of that instance.
- Further classes used by that classes are looked for in the same WWW location as the one the class came from.

Writing Applets

1. Write a class derived from the **`Applet`** class.
2. Use the **`init`** method to insert other components (an Applet is also a Container).
3. Use the **`paint`** method to draw things around.
4. Compile your class.
5. Create an HTML file with an APPLET tag pointing to your class.
6. Place BOTH the HTML and .class files visible through a URL in the same directory

Use JDK s appletviewer to test them

Scenario 4: Your first applet

Inherits from Applet class

```
import java.applet.Applet;
import java.awt.Graphics;
import java.awt.Button;

public class SampleApplet extends Applet {

     public void init () {
          add (new Button("Click me"));
     }

     public void paint(Graphics g) {
           //Draw a Rectangle around the applet's display area.
           g.drawRect(0, 0, getSize().width - 1, getSize().height - 1);

           //Draw the current string inside the rectangle.
           g.drawString("Hi buddy", 5, 15);
      }
}
```

Init method to add components

Paint method to draw things around

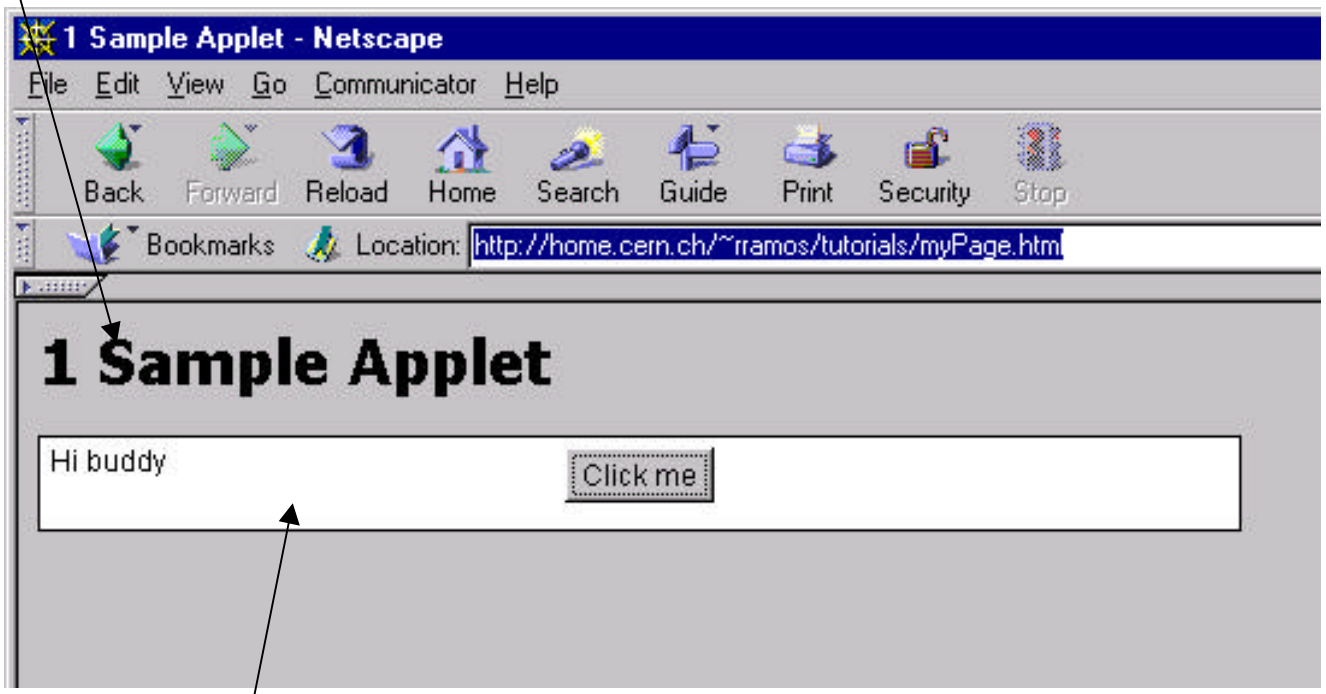
Scenario 4: the APPLET tag

```
<html><title>1 Sample Applet</title>
<body>
<h1>
1 Sample Applet
</h1>

    <applet code="SampleApplet.class" width=500 height=40>
    </applet>

</body>
</html>
```

APPLET tag referencing our class and defining the space in the browser window for it

Scenario 4: the result

Rendered HTML



Space for the applet

The Applet lifecycle

There is a defined lifecycle for an applet with these phases:

- Initialization: The browser loads (or reloads) the applet.
- Startup: After (re)loading an applet or when the user comes back to the page containing the applet.
- Running: User interacts with the applet and its HTML page
- Stopping: The user leaves the page containing the applet.
- Destruction: The applet is unloaded

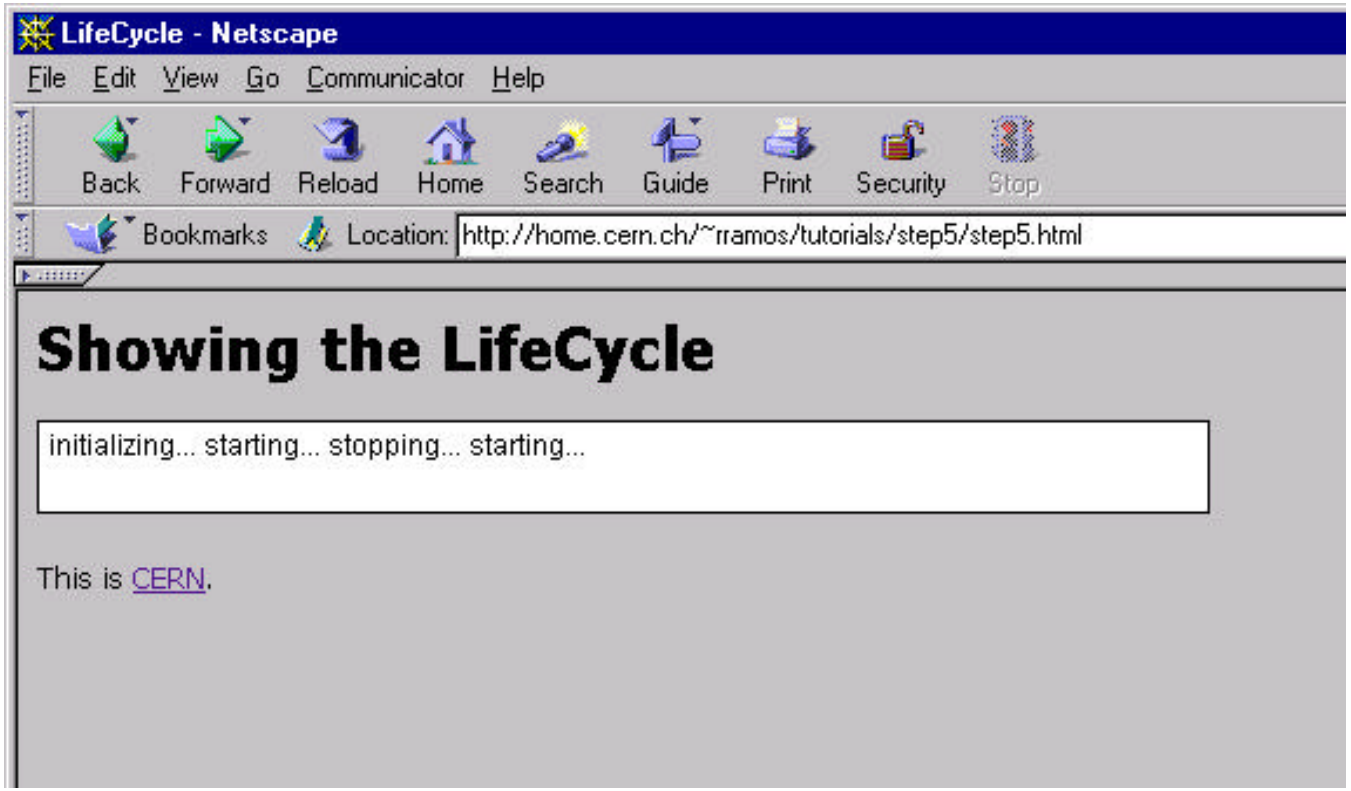
For each one of these phases we have a chance to do something. The browser invokes the init(), start(), stop() and destroy() methods of our applet.

Scenario 5: lifecycle

```
public class LifeCycle
     extends Applet {

    String s;

    public void init() {
        s = new String ("");
        addItem("initializing... ");
    }

    public void start() {
        addItem("starting... ");
    }

    public void stop() {
        addItem("stopping... ");
    }

    public void destroy() {
        addItem("preparing for unloading...");
    }

    void addItem(String newWord) {
            System.out.println(newWord);
            s = s + newWord;
            repaint();
        }

        public void paint(Graphics g) {
            //Draw a Rectangle
            g.drawRect(0, 0,
                       getSize().width - 1,
                       getSize().height - 1);

            //Draw the current string
            g.drawString(s, 5, 15);
        }
    }
```

Just define these methods and they will be invoked by the browser at the appropriate time

Scenario 5


LifeCycle - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Guide Print Security Stop
Bookmarks Location: http://home.cern.ch/~rramos/tutorials/step5/step5.html
Showing the LifeCycle
initializing... starting... stopping... starting...
This is CERN.

LifeCycle

You can use these methods to include your initialization code, clean up code, etc

For instance, some applets load images and in the meantime show some progress indicator: this is done in the init() method.

The init() method is similar to a constructor but the reason for it is:

- The constructor is invoked by the OO engine right after object instantiation.
- init() is invoked by the browser when it is fully ready to hold an applet (the window space has been initialized, etc.)

The lifecycle methods

Remember, your applet will remain in your browser s memory until it s destroyed.

Typically, use the init() method to build your GUI, add components, set event listeners, load images, initial data, etc.

Typically, use the start() method to start periodic refresh of the data you are showing (animation), reset the GUI to the state you what it to show when arriving to its page, etc..

Typically, use the stop() method to stop the periodic refresh of the data you are showing, clean up temporary variables, etc.

Using more class files

If your code uses other class files the browser will automatically download them from the same location it downloaded the Applet.

You just have to make all class files available through the same base URL.

In the next scenario open the java console and press 9 (this sets the debug level to show all messages).

Scenario 6: More classes

```
imports…

public class SampleApplet extends Applet implements ActionListener {
     Counter myCounter;

     public void init () {
          myCounter = new Counter();
          Button b = new Button ("Click me");
          b.addActionListener(this);
          add (b);
     }

     public void actionPerformed (ActionEvent e) {
          myCounter.increment();
          repaint();
     }

     public void paint(Graphics g) {
           g.drawRect(0, 0, getSize().width - 1, getSize().height - 1);

           Integer i = new Integer(myCounter.getValue());
           g.drawString(i.toString(), 5, 15);
      }
}
```

Note that I define the applet instance itself (this) to be the listener of the button

Here we are using the counter class to keep track of how many times the button was clicked

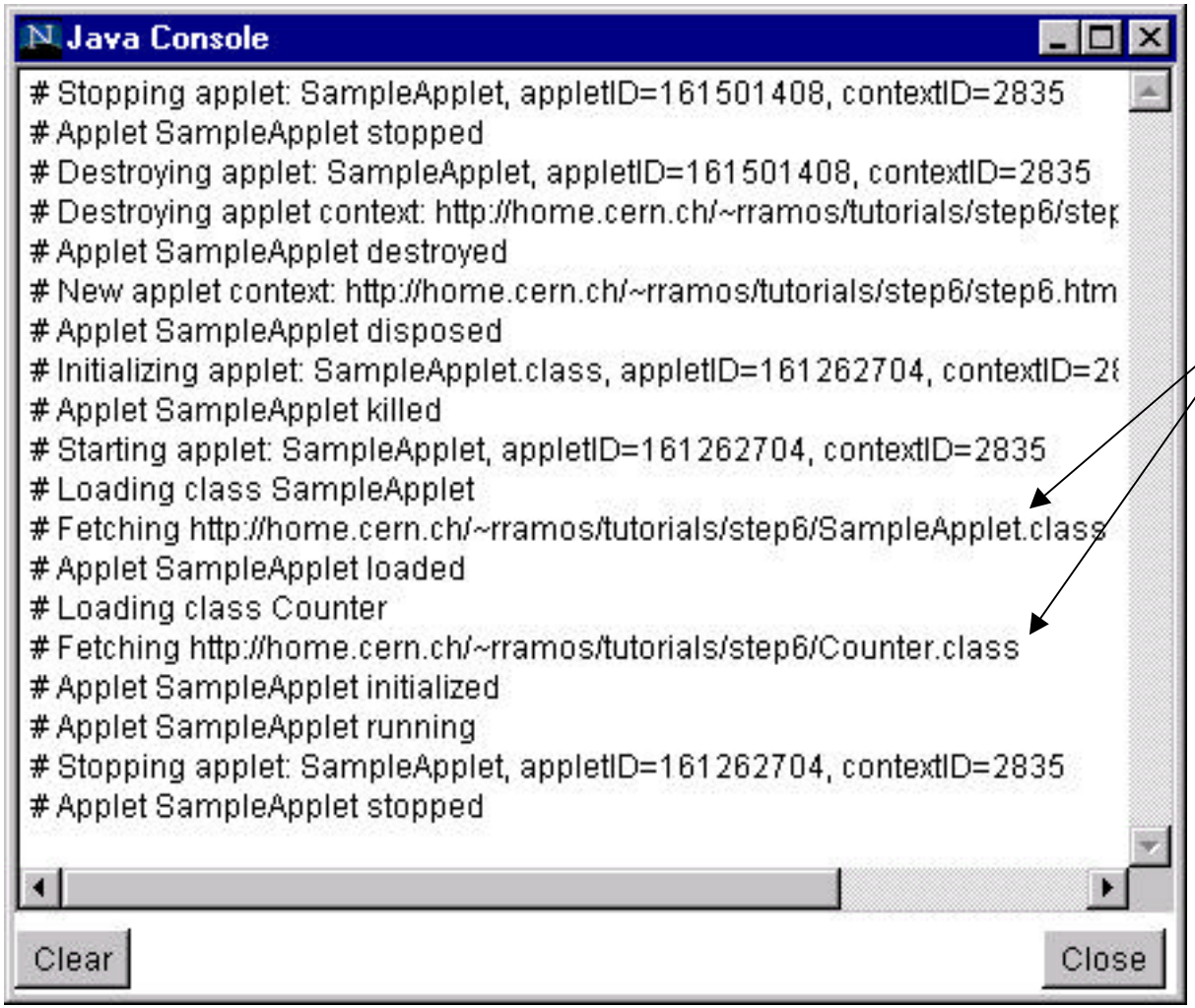
Scenario 6: The Counter class

```
public class Counter {
        int value = 0;
        public int increment() {
                value ++;
                return value;
        }
        public int getValue() {
                return value;
        }
}
```

Both class files (Counter.class and SampleApplet.class) have to be in the same directory as the html page.

Every class is retrieved through a separate HTTP connection

Scenario 6: Java console



See how the browser has automatically constructed a URL for each class and fetched them separately

JAR Files

If your applet uses many classes, loading them separately through HTTP is very unefficient.

Instead you can put them all together in a JAR file and make your HTML page point to your JAR file.

With JAR files only one HTTP connection is needed to retrieve your classes

JAR files are gzipped so save bandwidth.

JAR files are also used to store other information (certificates for your software, etc.)

Scenario 6bis

```
[rsplus]> jar cvf SampleApplet.jar *.class
adding: Counter.class (in=393) (out=268) (deflated 31%)
adding: SampleApplet.class (in=1293) (out=743) (deflated 42%)
```

AND THE HTML FILE

```
<html><title>More classes</title><body>
<h1>More classes</h1>
Click the button to increment the counter<P>

    <applet code="SampleApplet.class"
            ARCHIVE="Everything.jar" width=500 height=40>
    </applet>

</body>
</html>
```

We tell the browser that we have stored the classes in Everything.jar.

Applets in Swing

You can also write applets with Swing, but you d have to:

- Derive your applet class from JApplet
- Add components to your class through its contentPane:

Extend JApplet

```
public class SampleApplet extends JApplet {

     public void init () {
          getContentPane().add (new JButton("Click me"));
     }
}
```

Add components through its content pane

But...

To run your Swing applets users MUST have a browser containing a JVM version 1.2

Few browsers support jdk 1.2 which includes Swing.

In general Sun is interested in browsers including their latest version in their browsers.

But this is a great effort and Sun cannot enforce it.

So it has created the idea of the **Java Plug-In**

It s a piece of software distributed by Sun to be installed within a browser.

Your HTML pages have to invoke the Java Plugin of your browser and not the browser s JVM.

Scenario 7: HTML Page

```
<html><title>More classes</title>
<body>
<h1>
More classes
</h1>
Click the button to increment the counter<P>

<EMBED type="application/x-java-applet;version=1.2" width="500"
   height=”100" align="baseline" code="SampleApplet.class"
    pluginspage="http://java.sun.com/products/plugin/1.2/
                 plugin-install.html">
 <NOEMBED>
    No JDK 1.2 support for APPLET!!
 </NOEMBED>
 </EMBED>

</body>
</html>
```

Embeed your class so that the browser knows it has to invoke the java plugin and not its JVM.

Java Plugin

Provides independence from browser s development.

It is to a totally clean solution, requires the user to install the plugin AND the developer to change his HTML pages.

It is not available for all platforms.

Since the JAVA 2 platform is THE ONE, browsers will sooner or later include it.

The plugin is a temporary solution meanwhile.

JVM and WWW Servers

What happens if we include a JVM within a WWW server.

Think about what a WWW server does

Waits for URL requests:

- If the request is a page, just gives it back
- If the request is a form submission, retrieves the user data and executes a CGI script.

Typically a CGI script is PERL, ASP, etc

If the WWW server contains a JVM it can also EXECUTE JAVA CODE TO SERVE A CGI REQUEST

SERVLETS

A SERVLET is a piece of Java code which serves client requests.

All the generic logistics (waiting for user connection, spawning a thread to process the request, etc..) are implemented in the SERVLET class.

HTTP specific logistics are implemented in the HttpServlet class.

WWW Servers containing a JVM honour the SERVLET mechanism

The JSDK

All the servlet classes and development tools are included within Sun s Java Servlets Development Kit (jsdk). Download it from:
`http://java.sun.com/products/servlet/download.html`

Contains:

- The javax.servlet package
- Simple WWW server to test your servlets
- Examples and documentation

The javax.servlet package is needed only for development. It is part of the JVM within WWW servers.

Creating an HTTP SERVLET

Create a web form with some fields. Place it behind a URL

Derive a new class from the HttpServlet class included within javax.servlet.http:

- override the doGet, or doPost methods
- use the request object to obtain user data
- use the response object to write back to the user browser.

Include jdsk s servlet.jar in your CLASSPATH

Compile and place you class in the place where your web server if configured to run servlets from.

Point your WWW form action to your servlet

Scenario 8: The www form

The form ACTION points my servletclass

```
<HTML><HEAD><TITLE>Test Form</TITLE></HEAD>
<BODY>

<FORM METHOD=POST ACTION=/examples/servlet/step8>
First name: <INPUT TYPE=TEXT SIZE=20 NAME=firstname><BR>
Last name:  <INPUT TYPE=TEST SIZE=20 NAME=lastname>
<INPUT TYPE=submit VALUE="Click to Submit to Servlet">
</FORM>

</BODY>
```

Scenario 8: the servlet class

```
import java.io.*;    import javax.servlet.*;  import javax.servlet.http.*;

public class step8 extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException
    {
        response.setContentType("text/html");

        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<body>");
        out.println("<head>");

        out.println("<title>Java Series: test Servlet </title>");
        out.println("</head>");
        out.println("<body bgcolor=\"white\">");

        out.println("<h3> Java Series: test Servlet</h3>");
        String firstName = request.getParameter("firstname");
        String lastName = request.getParameter("lastname");
        out.println(" First Name = " + firstName + "<br>");
        out.println(" Last Name = " + lastName);

        out.println("</body>");
        out.println("</html>");
    }
}
```

We extend the HttpServlet class

Override the doGet method

Use the response to write HTML back

Use the request to get user data

Summary

We have

- Created TCP client/server
- Created UDP client/server
- Read/written through HTTP
- Created applets
- Controlled their lifecycle
- Created JAR files
- Played around with Swing applets and the java plugin.
- Created servlets