CHP-31
Java Networking

The term *network programming* refers to writing programs that execute across multiple devices (computers), in which the devices are all connected to each other using a network.
The java.net package of the J2SE APIs contains a collection of classes and interfaces that provide the low-level communication details, allowing you to write programs that focus on solving the problem at hand.
The java.net package provides support for the two common network protocols:
 TCP: TCP stands for Transmission Control Protocol, which allows for reliable communication between two applications. TCP is typically used over the Internet Protocol, which is referred to as TCP/IP.
 UDP: UDP stands for User Datagram Protocol, a connection-less protocol that allows for packets of data to be transmitted between applications.
This tutorial gives good understanding on the following two subjects:
 Socket Programming: This is most widely used concept in Networking and it has been explained in very detail.
 URL Processing: This would be covered separately. Click here to learn about URL Processing in Java language.

Url Processing

URL stands for Uniform Resource Locator and represents a resource on the World Wide Web, such as a Web page or FTP directory.
This section shows you how to write Java programs that communicate with a URL. A URL can be broken down into parts, as follows:

```
protocol://host:port/path?query#ref
```

Examples of protocols include HTTP, HTTPS, FTP, and File. The path is also referred to as the filename, and the host is also called the authority.
The following is a URL to a Web page whose protocol is HTTP:

```
http://www.amrood.com/index.htm?language=en#j2se
```

Notice that this URL does not specify a port, in which case the default port for the protocol is used. With HTTP, the default port is 80.

CHAPTER

URL Class Methods:

The **java.net.URL** class represents a URL and has complete set of methods to manipulate URL in Java.
The URL class has several constructors for creating URLs, including the following:

SN Methods with Description

1 **public URL(String protocol, String host, int port, String file) throws MalformedURLException.**
Creates a URL by putting together the given parts.

2 **public URL(String protocol, String host, String file) throws MalformedURLException**

Identical to the previous constructor, except that the default port for the given protocol is used.
3 **public URL(String url) throws MalformedURLException**
Creates a URL from the given String
4 **public URL(URL context, String url) throws MalformedURLException**
Creates a URL by parsing the together the URL and String arguments
The URL class contains many methods for accessing the various parts of the URL being represented.
Some of the methods in the URL class include the following:
SN Methods with Description
1 **public String getPath()**
Returns the path of the URL.
2 **public String getQuery()**
Returns the query part of the URL.
3 **public String getAuthority()**
Returns the authority of the URL.
4 **public int getPort()**
Returns the port of the URL.
5 **public int getDefaultPort()**
Returns the default port for the protocol of the URL.
6 **public String getProtocol()**
Returns the protocol of the URL.
7 **public String getHost()**
Returns the host of the URL.
8 **public String getHost()**
Returns the host of the URL.
9 **public String getFile()**
Returns the filename of the URL.
10 **public String getRef()**
Returns the reference part of the URL.
11 **public URLConnection openConnection() throws IOException**
Opens a connection to the URL, allowing a client to communicate with the resource.

Example:

The following URLDemo program demonstrates the various parts of a URL. A URL is entered on the command line, and the URLDemo program outputs each part of the given URL.

```
// File Name : URLDemo.java
import java.net.*;
import java.io.*;
public class URLDemo
{
public static void main(String[] args)
{
try
{
URL url =new URL(args[0]);
System.out.println("URL is "+ url.toString());
System.out.println("protocol is "+ url.getProtocol());
System.out.println("authority is "+ url.getAuthority());
System.out.println("file name is "+ url.getFile());
System.out.println("host is "+ url.getHost());
System.out.println("path is "+ url.getPath());
System.out.println("port is "+ url.getPort());
System.out.println("default port is "+ url.getDefaultPort());
System.out.println("query is "+ url.getQuery());
System.out.println("ref is "+ url.getRef());
}catch(IOException e)
{
e.printStackTrace();
}
}
}
```

A sample run of the thid program would produce the following result:

```
$ java URLDemo http://www.amrood.com/index.htm?language=en#j2se
URL is http://www.amrood.com/index.htm?language=en#j2se
protocol is http
authority is www.amrood.com
file name is/index.htm?language=en
host is www.amrood.com
path is/index.htm
port is-1
default port is80
query is language=en
refis j2se
```

URLConnections Class Methods:

The openConnection() method returns a **java.net.URLConnection**, an abstract class whose subclasses represent the various types of URL connections.
For example:

- If you connect to a URL whose protocol is HTTP, the openConnection() method returns an HttpURLConnection object.
- If you connect to a URL that represents a JAR file, the openConnection() method returns a JarURLConnection object.

- etc...

The URLConnection class has many methods for setting or determining information about the connection, including the following:

SN Methods with Description

1 **Object getContent()**
Retrieves the contents of this URL connection.

2 **Object getContent(Class[] classes)**
Retrieves the contents of this URL connection.

3 **String getContentEncoding()**
Returns the value of the content-encoding header field.

4 **int getContentLength()**
Returns the value of the content-length header field.

5 **String getContentType()**
Returns the value of the content-type header field.

6 **int getLastModified()**
Returns the value of the last-modified header field.

7 **long getExpiration()**
Returns the value of the expires header field.

8 **long getIfModifiedSince()**
Returns the value of this object's ifModifiedSince field.

9
public void setDoInput(boolean input)
Passes in true to denote that the connection will be used for input. The default value is true because clients typically read from a URLConnection.

10
public void setDoOutput(boolean output)
Passes in true to denote that the connection will be used for output. The default value is false because many types of URLs do not support being written to.

11 **public InputStream getInputStream() throws IOException**
Returns the input stream of the URL connection for reading from the resource.

12 **public OutputStream getOutputStream() throws IOException**
Returns the output stream of the URL connection for writing to the resource

13 **public URL getURL()**
Returns the URL that this URLConnection object is connected to

Example:

The following URLConnectionDemo program connects to a URL entered from the command line.
If the URL represents an HTTP resource, the connection is cast to HttpURLConnection, and the data in the resource is read one line at a time.

```
// File Name : URLConnDemo.java
import java.net.*;
```

```
import java.io.*;
public class URLConnDemo
```

```
{
public static void main(String[] args)
{
try
{
URL url =new URL(args[0]);
URLConnection urlConnection = url.openConnection();
HttpURLConnection connection =null;
if(urlConnection instanceof HttpURLConnection)
{
connection =(HttpURLConnection) urlConnection;
}
else
{
System.out.println("Please enter an HTTP URL.");
return;
}
BufferedReader in=new BufferedReader(
new InputStreamReader(connection.getInputStream()));
String urlString ="";
String current;
while((current =in.readLine())!=null)
{
urlString += current;
}
System.out.println(urlString);
}catch(IOException e)
{
e.printStackTrace();
}
}
}
```
A sample run of the thid program would produce the following result:
```
$ java URLConnDemo http://www.amrood.com
.....a complete HTML content of home page of amrood.com.....
```
Socket Programming:
Sockets provide the communication mechanism between two computers using TCP. A client program creates a socket on its end of the communication and attempts to connect that socket to a server.
When the connection is made, the server creates a socket object on its end of the communication. The client and server can now communicate by writing to and reading from the socket.
The java.net.Socket class represents a socket, and the java.net.ServerSocket class provides a mechanism for the server program to listen for clients and establish connections with them.
The following steps occur when establishing a TCP connection between two computers using sockets:
- The server instantiates a ServerSocket object, denoting which port number communication is to occur on.
- The server invokes the accept() method of the ServerSocket class. This method waits until a client connects to the server on the given port.

TUTORIALS POINT Simply Easy Learning
- After the server is waiting, a client instantiates a Socket object, specifying the server name and port number to connect to.
- The constructor of the Socket class attempts to connect the client to the specified server and port number. If communication is established, the client now has a Socket object capable of communicating with the server.
- On the server side, the accept() method returns a reference to a new socket on the server that is connected to the client's socket.

After the connections are established, communication can occur using I/O streams. Each socket has both an OutputStream and an InputStream. The client's OutputStream is connected to the server's InputStream, and the client's InputStream is connected to the server's OutputStream.
TCP is a twoway communication protocol, so data can be sent across both streams at the same time. There are

following usefull classes providing complete set of methods to implement sockets.

ServerSocket Class Methods:

The **java.net.ServerSocket** class is used by server applications to obtain a port and listen for client requests
The ServerSocket class has four constructors:

SN Methods with Description

1
public ServerSocket(int port) throws IOException
Attempts to create a server socket bound to the specified port. An exception occurs if the port is already bound by another application.

2
public ServerSocket(int port, int backlog) throws IOException
Similar to the previous constructor, the backlog parameter specifies how many incoming clients to store in a wait queue.

3
public ServerSocket(int port, int backlog, InetAddress address) throws IOException
Similar to the previous constructor, the InetAddress parameter specifies the local IP address to bind to. The InetAddress is used for servers that may have multiple IP addresses, allowing the server to specify which of its IP addresses to accept client requests on

4
public ServerSocket() throws IOException
Creates an unbound server socket. When using this constructor, use the bind() method when you are ready to bind the server socket

If the ServerSocket constructor does not throw an exception, it means that your application has successfully bound to the specified port and is ready for client requests.
Here are some of the common methods of the ServerSocket class:

SN Methods with Description

1
public int getLocalPort()
Returns the port that the server socket is listening on. This method is useful if you passed in 0 as the port number in a constructor and let the server find a port for you.

2
public Socket accept() throws IOException
Waits for an incoming client. This method blocks until either a client connects to the server on the specified port or the socket times out, assuming that the time-out value has been set using the setSoTimeout() method. Otherwise, this method blocks indefinitely.

3 **public void setSoTimeout(int timeout)**
Sets the time-out value for how long the server socket waits for a client during the accept().

4
public void bind(SocketAddress host, int backlog)
Binds the socket to the specified server and port in the SocketAddress object. Use this method if you instantiated the ServerSocket using the no-argument constructor.

When the ServerSocket invokes accept(), the method does not return until a client connects. After a client does connect, the ServerSocket creates a new Socket on an unspecified port and returns a reference to this new Socket. A TCP connection now exists between the client and server, and communication can begin.

Socket Class Methods:

The **java.net.Socket** class represents the socket that both the client and server use to communicate with each other. The client obtains a Socket object by instantiating one, whereas the server obtains a Socket object from the return value of the accept() method.
The Socket class has five constructors that a client uses to connect to a server:

SN Methods with Description

1
public Socket(String host, int port) throws UnknownHostException, IOException.
This method attempts to connect to the specified server at the specified port. If this constructor does not throw an exception, the connection is successful and the client is connected to the server.

2
public Socket(InetAddress host, int port) throws IOException
This method is identical to the previous constructor, except that the host is denoted by an

InetAddress object.
3
public Socket(String host, int port, InetAddress localAddress, int localPort) throws IOException.
Connects to the specified host and port, creating a socket on the local host at the specified address and port.
4
public Socket(InetAddress host, int port, InetAddress localAddress, int localPort) throws IOException.
This method is identical to the previous constructor, except that the host is denoted by an InetAddress object instead of a String
5 **public Socket()**
Creates an unconnected socket. Use the connect() method to connect this socket to a server.
When the Socket constructor returns, it does not simply instantiate a Socket object but it actually attempts to connect to the specified server and port.
Some methods of interest in the Socket class are listed here. Notice that both the client and server have a Socket object, so these methods can be invoked by both the client and server.

SN Methods with Description

1
public void connect(SocketAddress host, int timeout) throws IOException
This method connects the socket to the specified host. This method is needed only when you instantiated the Socket using the no-argument constructor.
2 **public InetAddress getInetAddress()**
This method returns the address of the other computer that this socket is connected to.
3 **public int getPort()**
Returns the port the socket is bound to on the remote machine.
4 **public int getLocalPort()**
Returns the port the socket is bound to on the local machine.

5 **public SocketAddress getRemoteSocketAddress()**
Returns the address of the remote socket.
6
public InputStream getInputStream() throws IOException
Returns the input stream of the socket. The input stream is connected to the output stream of the remote socket.
7
public OutputStream getOutputStream() throws IOException
Returns the output stream of the socket. The output stream is connected to the input stream of the remote socket
8
public void close() throws IOException
Closes the socket, which makes this Socket object no longer capable of connecting again to any server

InetAddress Class Methods:

This class represents an Internet Protocol (IP) address. Here are following useful methods, which you would need while doing socket programming:

SN Methods with Description

1 **static InetAddress getByAddress(byte[] addr)**
Returns an InetAddress object given the raw IP address .
2 **static InetAddress getByAddress(String host, byte[] addr)**
Create an InetAddress based on the provided host name and IP address.
3 **static InetAddress getByName(String host)**
Determines the IP address of a host, given the host's name.
4 **String getHostAddress()**
Returns the IP address string in textual presentation.
5 **String getHostName()**
Gets the host name for this IP address.
6 **static InetAddress InetAddress getLocalHost()**
Returns the local host.
7 **String toString()**

Converts this IP address to a String.

Socket Client Example:

The following GreetingClient is a client program that connects to a server by using a socket and sends a greeting, and then waits for a response.

```
// File Name GreetingClient.java
import java.net.*;
import java.io.*;
public class GreetingClient
{
public static void main(String[] args)
{
String serverName = args[0];
int port =Integer.parseInt(args[1]);
try
```



```
{
System.out.println("Connecting to "+ serverName+" on port "+ port);
Socket client =new Socket(serverName, port);
System.out.println("Just connected to "+ client.getRemoteSocketAddress());
OutputStream outToServer = client.getOutputStream();
DataOutputStream out=new DataOutputStream(outToServer);
out.writeUTF("Hello from "+ client.getLocalSocketAddress());
InputStream inFromServer = client.getInputStream();
DataInputStreamin=newDataInputStream(inFromServer);
System.out.println("Server says "+in.readUTF());
client.close();
}catch(IOException e)
{
e.printStackTrace();
}
}
}
```

Socket Server Example:

The following GreetingServer program is an example of a server application that uses the Socket class to listen for clients on a port number specified by a command-line argument:

```
// File Name GreetingServer.java
import java.net.*;
import java.io.*;
public class GreetingServer extends Thread
{
private ServerSocket serverSocket;
public GreetingServer(int port)throws IOException
{
serverSocket =new ServerSocket(port);
serverSocket.setSoTimeout(10000);
}
public void run()
{
while(true)
{
try
{
System.out.println("Waiting for client on port "+
serverSocket.getLocalPort()+"...");
Socket server = serverSocket.accept();
System.out.println("Just connected to "
+ server.getRemoteSocketAddress());
DataInputStream in=new DataInputStream(server.getInputStream());
System.out.println(in.readUTF());
```

```
DataOutputStream out=new DataOutputStream(server.getOutputStream());
out.writeUTF("Thank you for connecting to "+
server.getLocalSocketAddress()+"\nGoodbye!");
server.close();
}catch(SocketTimeoutException s)
{
System.out.println("Socket timed out!");
```



```
break;
}catch(IOException e)
{
e.printStackTrace();
break;
}
}
}
public static void main(String[] args)
{
int port =Integer.parseInt(args[0]);
try
{
Thread t =new GreetingServer(port);
t.start();
}catch(IOException e)
{
e.printStackTrace();
}
}
}
```

Compile client and server and then start server as follows:

```
$ java GreetingServer6066
Waitingfor client on port 6066...
```

Check client program as follows:

```
$ java GreetingClient localhost 6066
Connecting to localhost on port 6066
Just connected to localhost/127.0.0.1:6066
Server says Thank you for connecting to /127.0.0.1:6066
Goodbye!
```

CHP-32
Java Sending E-mail

To send an e-mail using your Java Application is simple enough but to start with you should have **JavaMail API** and **Java Activation Framework (JAF)** installed on your machine.

- You can download latest version of JavaMail (Version 1.2) from Java's standard website.
- You can download latest version of JAF (Version 1.1.1) from Java's standard website.

Download and unzip these files, in the newly created top level directories you will find a number of jar files for both the applications. You need to add **mail.jar** and **activation.jar** files in your CLASSPATH.

Send a Simple E---mail:

Here is an example to send a simple e-mail from your machine. Here it is assumed that your **localhost** is connected to the internet and capable enough to send an e-mail.

```
// File Name SendEmail.java
import java.util.*;
import javax.mail.*;
import javax.mail.internet.*;
import javax.activation.*;
public class SendEmail
{
public static void main(String[] args)
{
// Recipient's email ID needs to be mentioned.
String to ="abcd@gmail.com";
// Sender's email ID needs to be mentioned
Stringfrom="web@gmail.com";
// Assuming you are sending email from localhost
String host ="localhost";
// Get system properties
Properties properties =System.getProperties();
// Setup mail server
properties.setProperty("mail.smtp.host", host);
// Get the default Session object.
```

CHAPTER


```
Session session =Session.getDefaultInstance(properties);
try{
// Create a default MimeMessage object.
MimeMessage message =new MimeMessage(session);
// Set From: header field of the header.
message.setFrom(new InternetAddress(from));
// Set To: header field of the header.
message.addRecipient(Message.RecipientType.TO,new InternetAddress(to));
```

```
// Set Subject: header field
message.setSubject("This is the Subject Line!");
// Now set the actual message
message.setText("This is actual message");
// Send message
Transport.send(message);
System.out.println("Sent message successfully....");
}catch(MessagingException mex){
mex.printStackTrace();
}
}
}
```

Compile and run this program to send a simple e-mail:

```
$ java SendEmail
Sent message successfully....
```

If you want to send an e-mail to multiple recipients, then following methods would be used to specify multiple e-mail IDs:

```
void addRecipients(Message.RecipientType type,
Address[] addresses)
throwsMessagingException
```

Here is the description of the parameters:

- **type:** This would be set to TO, CC or BCC. Here CC represents Carbon Copy and BCC represents Black Carbon Copy. Example *Message.RecipientType.TO*
- **addresses:** This is the array of e-mail ID. You would need to use InternetAddress() method while specifying email IDs

Send an HTML E---mail:

Here is an example to send an HTML e-mail from your machine. Here, it is assumed that your **localhost**is connected to the internet and capable enough to send an e-mail.

This example is very similar to previous one, except here we are using setContent() method to set content, whose second argument is "text/html" to specify that the HTML content is included in the message.

Using this example, you can send as big as HTML content you like.

```
// File Name SendHTMLEmail.java
import java.util.*;
import javax.mail.*;
```



```
import javax.mail.internet.*;
import javax.activation.*;
public class SendHTMLEmail
{
public static void main(String[] args)
{
// Recipient's email ID needs to be mentioned.
String to ="abcd@gmail.com";
// Sender's email ID needs to be mentioned
Stringfrom="web@gmail.com";
// Assuming you are sending email from localhost
String host ="localhost";
// Get system properties
Properties properties =System.getProperties();
// Setup mail server
properties.setProperty("mail.smtp.host", host);
// Get the default Session object.
Session session =Session.getDefaultInstance(properties);
try{
// Create a default MimeMessage object.
MimeMessage message =new MimeMessage(session);
// Set From: header field of the header.
message.setFrom(new InternetAddress(from));
// Set To: header field of the header.
message.addRecipient(Message.RecipientType.TO,
newInternetAddress(to));
```

```
// Set Subject: header field
message.setSubject("This is the Subject Line!");
// Send the actual HTML message, as big as you like
message.setContent("<h1>This is actual message</h1>",
"text/html");
// Send message
Transport.send(message);
System.out.println("Sent message successfully....");
}catch(MessagingException mex){
mex.printStackTrace();
}
}
}
```

Compile and run this program to send an HTML e-mail:

```
$ java SendHTMLEmail
Sent message successfully....
```


Send Attachment in E---mail:

Here is an example to send an e-mail with attachment from your machine. Here, it is assumed that your**localhost** is connected to the internet and capable enough to send an e-mail.

```
// File Name SendFileEmail.java
import java.util.*;
import javax.mail.*;
import javax.mail.internet.*;
import javax.activation.*;
public class SendFileEmail
{
public static void main(String[] args)
{
// Recipient's email ID needs to be mentioned.
String to ="abcd@gmail.com";
// Sender's email ID needs to be mentioned
Stringfrom="web@gmail.com";
// Assuming you are sending email from localhost
String host ="localhost";
// Get system properties
Properties properties =System.getProperties();
// Setup mail server
properties.setProperty("mail.smtp.host", host);
// Get the default Session object.
Session session =Session.getDefaultInstance(properties);
try{
// Create a default MimeMessage object.
MimeMessage message =new MimeMessage(session);
// Set From: header field of the header.
message.setFrom(new InternetAddress(from));
// Set To: header field of the header.
message.addRecipient(Message.RecipientType.TO,
new InternetAddress(to));
// Set Subject: header field
message.setSubject("This is the Subject Line!");
// Create the message part
BodyPart messageBodyPart =new MimeBodyPart();
// Fill the message
messageBodyPart.setText("This is message body");
// Create a multipar message
Multipart multipart =new MimeMultipart();
// Set text message part
multipart.addBodyPart(messageBodyPart);
```



```
// Part two is attachment
```

```
messageBodyPart =new MimeBodyPart();
String filename ="file.txt";
DataSource source =new FileDataSource(filename);
messageBodyPart.setDataHandler(new DataHandler(source));
messageBodyPart.setFileName(filename);
multipart.addBodyPart(messageBodyPart);
// Send the complete message parts
message.setContent(multipart );
// Send message
Transport.send(message);
System.out.println("Sent message successfully....");
}catch(MessagingException mex){
mex.printStackTrace();
}
}
}
```

Compile and run this program to send an HTML e-mail:

```
$ java SendFileEmail
Sent message successfully....
```

User Authentication Part:

If it is required to provide user ID and Password to the e-mail server for authentication purpose, then you can set these properties as follows:

```
props.setProperty("mail.user","myuser");
props.setProperty("mail.password","mypwd");
```

Rest of the e-mail sending mechanism would remain as explained above.