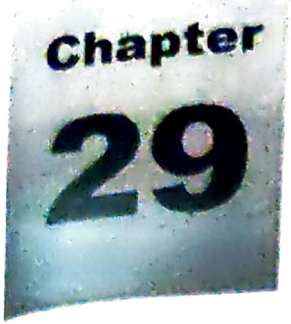


SECTION VII: NETWORKING AND I/O

Sockets And Network

In today's computing world, there are many expensive resources such as Laser printers, Fax machines and so on that needs to be shared. Business data is another expensive resource that needs to be shared. To facilitate this, robust networks have come into existence. Networks allow expensive resources to be shared.

Any programming environment must provide standard techniques to permit its applications to communicate across a network. Java is a programming environment and it provides simple yet robust techniques that permit Java applications to communicate across networks and share valuable resources.

Java communication is built on standard Client/Server architecture. A Server must have a port number on which some software is a listener/talker. What this really means is that all Servers have software listening on a port determined by the hardware architecture of the computer.

This software is constantly listening for Client requests. When the software hears a Client's request it becomes a talker and replies to the Client appropriately. Imagine multiple Servers on a network. For this system to work properly, each listener software [mounted on the Server] must clearly and unambiguously recognize that a Client is making a specific request to it. Additionally, when the listener becomes a talker it must clearly and unambiguously speak back only to the Client that spoke to it in the first place.

This necessitates that both the Clients and the Servers on the network must be uniquely identifiable. This is where TCP/IP comes into the picture. TCP/IP is an acronym for Transmission Control Protocol / Internet Protocol. Using TCP/IP each computer on the network whether a Server or a Client can be uniquely identified using a number system that has the pattern XXX.XXX.XXX.XXX [i.e. a series of three numbers, followed by a period, followed by a second series of three numbers and so on]. This is often called the unique [IP] address of a computer on a network. Using this kind of numbering system, all Servers and Clients on a network can be uniquely identified.

Once all the Servers and Clients on a network are uniquely identified all that remains is that this identity number is passed to the listener/talker software resident on the Servers and the Clients. Immediately such listener/talker software will be able to uniquely identify the computer on which it is mounted as well as every other computer on the network. Now communication can be effectively setup between any of the Server/Client pairs. One half of network programming has fallen into place.

All that remains is to have a set of standards that will determine **how** two computers will talk to each other on the network. If all computers spoke a common language on the network then any programming environment could provide an Application Programmers Interface [API] that would allow programmers to write applications that communicated effectively across a network and shared valuable resources.

Java provides a **Socket based API** that shields a programmer from a lot of low level code writing while making it pretty simple to create network based applications. Conceptually, creating network based applications requires a Java programmer to understand and use socket based programming.

Socket Overview

A **network** socket is a lot like an electric socket. Various plugs around the network have a standard way of delivering their payload. Anything that understands standard networking protocol can **plug-in** to the socket and communicate. When speaking about network sockets, TCP/IP packets and IP addresses immediately come to mind. Internet Protocol [IP] is a low-level routing protocol that breaks data into small packets and sends them to an address across a network. This does not guarantee the delivery of the packets to the destination.

Transmission Control Protocol [TCP] is a higher-level protocol that manages to string together these packets, sorting and retransmitting them as necessary, to **robustly** and reliably transmit data to an address across a network. A third protocol, User Datagram Protocol [UDP], does a similar job as TCP and can be used directly to support fast, [though unreliable] transport of packets from source to destination.

Client / Server Networking

The term Client/Server is often mentioned in the context of networking. It seems complicated when read about, but it is actually quite simple. A Server is an entity that has some **resource that can be shared**. A Client is simply another entity that **wants to gain access** to those **resources**. In a networking environment, a **Socket** on the server allows a client to plug-in and access a server's resources. **Server Sockets** allow a computer to single handedly serve different clients different kinds of information.

This feat is managed by the introduction of a **port**, which is a **numbered socket** on a particular machine. A server process is said to **listen** at a port until a client connects to it. A server is allowed to accept multiple clients connected to the same port number, although each session is unique. To manage multiple client connections, the server's listening process **must be multi-threaded**.

Proxy Servers

A **proxy server** is a software that runs on Server that speaks the language of Clients. This software **hides the actual Server** from the Clients that communicate with it. This is required when Clients have restrictions on which Servers they can connect to. Thus, a client would connect to a proxy server, which did not have such restrictions and the proxy server would in turn service the client while communicating back to back with a Server on which the resources required by the Client reside.

A proxy server has the additional ability to filter Client requests **or** cache the results of those requests for future use. A caching proxy HTTP server can help reduce the bandwidth demands on a local network's connection to the Internet. When a popular website is being hit by hundreds of users, a proxy server can get the contents of the web server's popular pages once and save these in its cache, thus saving expensive Internet data transfers while providing fast access to the same pages to its clients.

Internet Addressing

The Internet is really the world's largest network and it is growing bigger every day. Every computer on the Internet has a unique IP address.

An Internet address is a number that uniquely identifies each computer on the Internet just like any other network. There are 32 bits in an IP address and they are often referred to as a sequence of four numbers between **0** and **255** separated by **periods** [.] [*remember the XXX.XXX.XXX.XXX*].

The first few bits define which class of network lettered as A, B, C, D or E. Most Internet users are on a class C network. There are over two million networks in class C. The first byte of a class C network is between 192 and 224, with the last byte actually identifying an individual computer among the 256 allowed on a single class C network.

Domain Naming Service [DNS]

The Internet would not be a very friendly place to navigate if everyone had to refer to their addresses as a series of numbers separated by a period. For example: It is hard to imagine seeing **http://192.9.9.1/** at the bottom of an advertisement. Thankfully, a clearing house exists for a parallel hierarchy of **names** bound to each of these unique numbers. It is called the Domain Naming Service [DNS]. Just as the four numbers of an IP address describe a network hierarchy from left to right, the **name** of an Internet address, called its [**domain name**], describes a machine's location in a name space, from right to left.

For example: **www.starwave.com** is in the **COM domain** which is reserved for U.S. based commercial sites, is called **starwave** [*after the company name*] and **www** is the name of the specific computer that is Starwave's physical Web server. The **www** corresponds to the rightmost number in the underlying [and equivalent] IP address.

Using UDP Connection

To make the concepts of Network programming fall into place a simple example of network programming using the UDP protocol will help. In this example there are two .class files created. One will run on a Client and the other will run on a Server. The Client and Server should be connected via an Ethernet network.

The example is really a simple one, the Server broadcasts a message at one second intervals to a computer on which the Client application is running. The Client listens on a specific port and receives the Server's broadcast. Then displays what the Server has sent on its VDU. For this exchange to take place the Server application will have to know the **Name of the Client computer**.

The Client application will have to know the:

- Port number on which to listen for the Server's broadcasts

- Create a socket and link to the Server
- Accept the packets sent from the Server
- Process the packets sent from the server
- Display the contents of the packets sent from the Server on the Client's VDU

In this case, since the physical existence of a network is not known, both the Client .class file and the Server .class file will be run on the same computer. Hence, the Server application takes the IP address of the Client as **localhost** [i.e. 127.0.0.1], which internationally points to the local computer. What this means is the Client and Server applications both run on the same computer but in **different Root Console**.

REMINDER

If a physical network actually exists then all that needs to be done is to **replace** the keyword **localhost** with the **name** of the computer on which the Client application [Client.class] will be run. Each computer on a network is **named** by the network administrator so as to allow users to reference them when required.

Physically copy the Client application [i.e. Client.class] onto the computer whose name matches the one mentioned in the Server's application [i.e. TimeServer.class]. Start a Root Console on the Server and run the Server's .class file in it. Once the Server .class file is up and running successfully the broadcaster is running. Then start a root console on the Client and run the Client's .class file in it. The Server's .class file broadcasts its message to the Client on the network, on port 1313. The Client's .class file is listening on port 1313 and therefore picks up [responds to] the Server's broadcast.

Once this is achieved the basics of Network programming have fallen into place.

Client/Server Network Example

The following example is the Server application's code base. The Server's .class file broadcasts a message. The transmission consists of the text string "**The Server is started.....**", which is written to standard output [i.e. the VDU] on the Client.

Code spec for **TimeServer.java**:

```
1 import java.net.*;
2 import java.time.Instant;
3
4 public class TimeServer {
5    DatagramSocket datagramSocket;
6    DatagramPacket datagramPacket;
7    InetAddress address;
```

```
8
9     public static void main(String args []) throws Exception {
10       TimeServer timeServer = new TimeServer();
11       timeServer.process();
12    }
13
14    public TimeServer() throws Exception {
15       address = InetAddress.getByName("localhost");
16       datagramSocket = new DatagramSocket();
17    }
18
19    public void process() throws Exception    {
20       byte[] message;
21       System.out.println("UDP Server started\n.\n.\n.\n.\n.");
22       System.out.println("\nExtracting System Time\n.\n.\n.\n.\n.");
23
24       while(true) {
25          Instant currentDateTime = Instant.now();
26          String getDate = currentDateTime.toString();
27
28          String str = "Server Time: " + getDate;
29          message = str.getBytes();
30          datagramPacket = new DatagramPacket(message, message.length,
            address, 1313);
31          datagramSocket.send(datagramPacket);
32          System.out.println("\nServer Time sent sucessfully to the client.");
33          Thread.sleep(10000);
34       }
35    }
36 }
```

Explanation:

DatagramSocket is a class, which represents a socket for sending and receiving datagram packets. A datagram socket is the sending or receiving point for a packet delivery service. Each packet sent or received on a datagram socket is individually addressed and routed.

UDP broadcasts sends are always enabled on a DatagramSocket. In order to receive broadcast packets a DatagramSocket should be bound to the wildcard address. In some implementations, broadcast packets may also be received when a DatagramSocket is bound to a more specific address.

DatagramPacket is a class, which represents a datagram packet. Datagram packets are used to implement a connectionless packet delivery service. Each message is routed from one machine to another, based solely on information contained within that packet. Packet delivery is not guaranteed.

InetAddress is a class, which represents an Internet Protocol [IP] address. An IP address is either a 32-bit or 128-bit unsigned number used by IP, a lower-level protocol on which protocols like UDP and TCP are built.

An instance of an InetAddress consists of an IP address and possibly its corresponding host name [depending on whether it is constructed with a host name or whether it has already done reverse host name resolution].

An object of **TimeSever** is created, which calls its process().

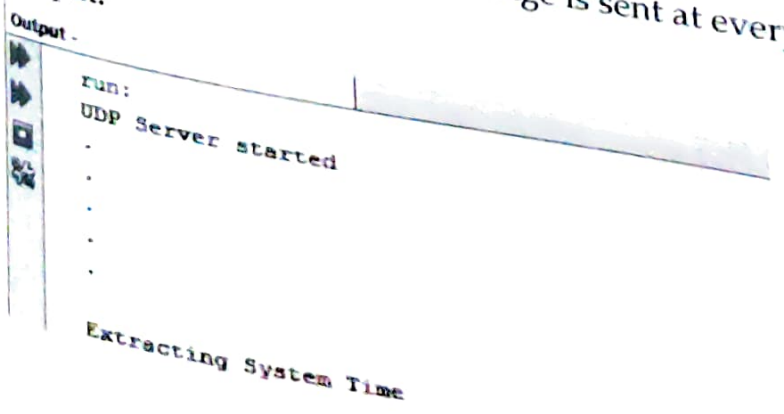
TimeSever() constructor is constructed, which throws an exception:

- The name of the IP address is retrieved by **getByName()**. getByName() determines the IP address of a host if the host's name is given. In the above code spec, the IP address of the localhost host is retrieved
- A new object of DatagramSocket is created

process() is constructed, which throws exception:

- An array of type byte is created, which is printed out via println()
- A While loop is constructed to send the message **Server Time sent successfully to the client** is displayed as long as the application is running. Here, first a variable of type Instant [explained in *Chapter 42: Date And Time API*] is declared, which retrieves the current date. Instant represents a particular instant in the time line and can be used to create legacy java.util.Date objects
- A variable of type String is created, which stores the date that is retrieved earlier in String format. This string format is then converted into bytes according to the platform's default character encoding and stores the result into a byte array using getBytes()
- A new **DatagramPacket** is constructed for sending packets of length byte's length to the specified port number on the specified host. Following parameters are passed to the DatagramPacket constructor:
  - The packet data
  - The packet length
  - The destination address
  - The destination port number
- send() of DatagramSocket sends the datagram packet from the datagram socket
- Using **Thread.sleep()** the message is sent at every 10000 seconds

Output:



```
.
.
.
.
.

Server Time sent sucessfully to the client.
```

Now let's create the client application. The Client program is even simpler than that of the Server. A buffer is created to hold all the characters received. A socket is created that listens at a specific port and a **DatagramPacket** object is created to collect data being broadcasted from the Server after which the content of the Server's broadcast is displayed on the Client's VDU. The code is written using an ASCII editor and saved in a file called Client.java.

Code spec for Client.java:

```
1  import java.net.*;
2
3  public class Client {
4     DatagramSocket datagramSocket;
5     DatagramPacket datagramPacket;
6
7     public static void main(String args []) {
8        Client client = new Client();
9        client.run();
10    }
11
12    public void run() {
13       byte message[] = new byte[64];
14       String strMsg;
15
16       try {
17          datagramSocket = new  DatagramSocket(1313);
18          datagramPacket = new  DatagramPacket(message, message.length);
19
20          while (true) {
21             datagramSocket.receive(datagramPacket);
22             strMsg = new String(datagramPacket.getData());
23             System.out.println("\n\nMessage received from the server: " +
               datagramPacket.getAddress() +  "\n  " + strMsg);
24          }
25       }
26       catch (Exception e) {
27          System.out.println("Exception raised: " + e.getMessage());
28       }
29    }
30 }
```

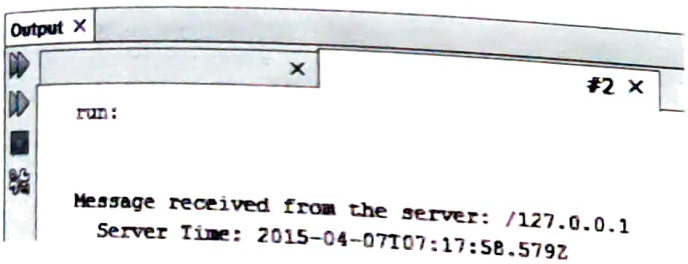
Explanation:

An object of Client is created, which calls its run().

run() is constructed:

- An array of type byte is created, which is printed out via println()
- A variable of type String is created
- A new **Datagram Socket** is constructed which is passed the port number as the argument
- A new **DatagramPacket** is constructed for sending packets of length byte's length, which is passed the packet data and the packet length as the argument
- A **while** loop is generated, wherein DatagramSocket.**receive()** retrieves the datagram packet and The String variable holds the data received by DatagramPacket
- Finally the IP address of the server and the date and time when the server started DatagramSocket is displayed

Output:



REMINDER

The output shown of Client.java assumes that both the Client and Server applications are being run on the same computer in different Root Console.

When the above code spec of TimeServer and Client was run, it was run in the same machine i.e. the server and the client was the same machine.

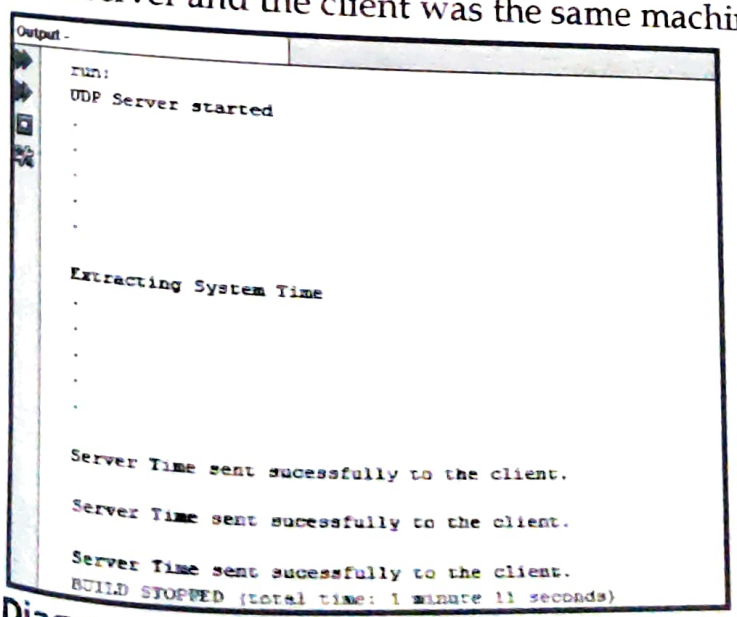


Diagram 29.1.1: The output displayed in the server machine

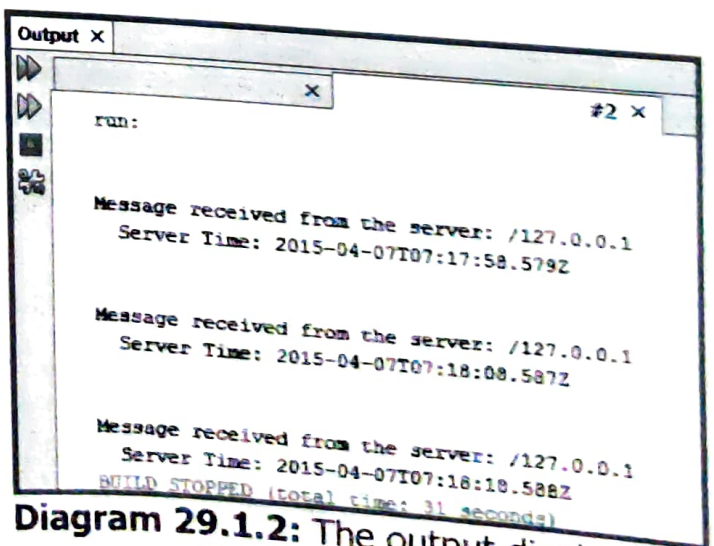


Diagram 29.1.2: The output displayed in the client machine

Since sleep() is used for the interval of sending messages after every 10000 seconds, the output of the server and the client will be as shown in diagram 29.1.1 and 29.1.2.

If the Client and Server applications are being run on two different machines, then change the following TimeServer.java code spec:

```
public TimeServer() throws Exception {
   address = InetAddress.getByName("localhost");
   datagramSocket = new DatagramSocket();
}
```

to [modifications marked in bold]:

```
public TimeServer() throws Exception {
   address = InetAddress.getByName("orchid");
   datagramSocket = new DatagramSocket();
}
```

In the above code spec the IP address of the client machine is changed in TimeServer.java i.e. the IP address of the machine from where the client application will be run is changed. Next compile the Client application on the client machine and the Server application on the server machine. Run both the applications and notice the change in the IP address of the Server machine in the Client machine.

Using TCP Connection

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.

java.net.Socket represents a socket and **java.net.ServerSocket** provides a mechanism for the server program to listen for clients and establish connections with them.

The following is what happens while 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 ServerSocket.**accept()**, which waits until a client connects to the server on the given port
- After the server is waiting, a client instantiates a **Socket** object, specifying the server name and port number to connect to

- The constructor of Socket 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, accept() 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 two way communication protocol, so data can be sent across both the streams at the same time.

Example

ServerData.java is a server application that uses the Socket class to listen for clients on a specified port number.

Code spec: [ServerData.java]

```
1  import java.io.DataInputStream;
2  import java.io.DataOutputStream;
3  import java.io.IOException;
4  import java.net.ServerSocket;
5  import java.net.Socket;
6  import java.net.SocketTimeoutException;
7
8  public class ServerData extends Thread {
9     private final ServerSocket serverSocket;
10
11    public ServerData(int port) throws IOException {
12       serverSocket = new ServerSocket(port);
13       serverSocket.setSoTimeout(10000);
14    }
15
16    @Override
17    public void run() {
18       while(true) {
19          try {
20             System.out.println("Waiting for client on port " +
21             serverSocket.getLocalPort() + "...");
22             try (Socket socket = serverSocket.accept()) {
                  System.out.println("Connected to Client on: " +
23                socket.getRemoteSocketAddress());
                  DataInputStream dataInputStreamServer = new
24                DataInputStream(socket.getInputStream());
25                System.out.println(dataInputStreamServer.readUTF());
                  DataOutputStream dataOutputStreamServer = new
```

```
                DataOutputStream(socket.getOutputStream());
26              dataOutputStreamServer.writeUTF("Thank you for connecting to
                Server: " + socket.getLocalSocketAddress());
27            }
28         }catch(SocketTimeoutException s) {
29            System.out.println("Socket timed out!");
30            break;
31         }catch(IOException e) {
32            break;
33         }
34      }
35   }
36
37   public static void main(String args[]) {
38      int port = Integer.parseInt("6066");
39      try {
40         Thread t = new ServerData(port);
41         t.start();
42      } catch(IOException e) {
43      }
44   }
45 }
```

Explanation:

java.io.DataInputStream allows an application read primitive Java data types from an underlying input stream in a machine-independent way. **java.io.DataOutputStream** allows an application write primitive Java data types to an output stream in a portable way.

java.net.ServerSocket implements server sockets. A server socket waits for requests to come in over the network. It performs some operation based on that request and then possibly returns a result to the requester. **java.net.Socket** implements client sockets [also known as **sockets**]. A socket is an endpoint for communication between two machines.

A variable of type ServerSocket is declared, which is then used to obtain a port and listen for client requests.

ServerSocket() constructor is constructed, which accepts a port. This ServerSocket() constructor attempts to create a server socket bound to the specified port. This constructor throws an IOException, in case an exception occurs, if the port is already bound to another application. ServerSocket.**setSoTimed()** sets the time-out value for how long the server socket waits for a client during accept(). If **ServerSocket()** constructor does not throw an exception, it means that the application is successfully bound to the specified port and is ready for client requests.

Thread.run() is executed by the thread after a call is made to start(). run():

- ServerSocket.getLocalPort() returns the port that the server socket is listening on
- Socket is declared, which 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 accept()
- ServerSocket.accept() waits for the an incoming client. accept() 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 **setSoTimeout()**. Otherwise, this method blocks indefinitely. So it is a good practice to use this in **try-with-resources** way

  When ServerSocket.accept() is invoked, it does not return until a client connects. After a client connects, 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.getRemoteSocketAddress()** returns the address of the endpoint this socket is connected to i.e. the address of the client
- **DataInputStream** is declared, which allows an application read primitive Java data types from an underlying input stream in a machine-independent way. An application uses a data output stream to write data that can later be read by a data input stream
- DataInputStream stores an input stream value returned from the client application using **Socket.getInputStream()**
- **DataInputStream.readUTF()** reads from the input stream from the client application a representation in Unicode character string encoded in modified UTF-8 format. This string of characters is then returned as a String data type
- **DataOutputStream** is declared, which allows an application write primitive Java data types to an output stream in a portable way. An application can then use a data input stream to read the data back in
- DataOutputStream stores an output stream value to be send to the client application using **Socket.getOutputStream()**
- **DataOutputStream.writeUTF()** writes a string to the underlying output stream using modified UTF-8 encoding in a machine independent manner

While the socket is read or accepted and a time out occurs, then that exception is caught in the catch block using the **SocketTimeoutException**. **java.net.SocketTimeoutException** signals that a timeout has occurred and the application stops running.

When the Server application is executed, **Port** number **6066** is passed for the server to start running. Thread is declared, which stores the port number of the server application. **Thread.start()** informs the JVM to begin execution of the declared thread, which in return calls run() of the server application.

Code spec: [ClientData.java]

```
1  import java.io.DataInputStream;
2  import java.io.DataOutputStream;
3  import java.io.IOException;
4  import java.io.InputStream;
5  import java.io.OutputStream;
6  import java.net.Socket;
7
8  public class ClientData {
9     public static void main(String [] args) {
10       String hostName = "localhost";
11       int port = Integer.parseInt("6066");
12       try {
13          System.out.println("Connecting to " + hostName + " on port " + port);
14          try (Socket socket = new Socket(hostName, port)) {
15             System.out.println("Just connected to: " +
                socket.getRemoteSocketAddress());
16             OutputStream outputStream = socket.getOutputStream();
17             DataOutputStream dataOutputStreamClient = new
                DataOutputStream(outputStream);
18             dataOutputStreamClient.writeUTF("Client says: Hello! from " +
                socket.getLocalSocketAddress());
19             InputStream inputStream = socket.getInputStream();
20             DataInputStream dataInputStreamClient = new
                DataInputStream(inputStream);
21             System.out.println("Server says: " +
                dataInputStreamClient.readUTF());
22          }
23       } catch(IOException e) {
24       }
25    }
26 }
```

Explanation:

This is the client application. Two variables are declared, one holds the name of the host and the other holds the port number.

A Socket is declared using the constructor **Socket(Hostname, Port)**, which attempts to connect to the specified server [hostname] at the specified port [port number]. If this constructor does not throw an exception, the connection is successful and the client is connected to the server.

Socket.getLocalSocketAddress() returns the address of the endpoint the socket is bound to i.e. the address of the client application.

Running the application:

First execute the Server application i.e. ServerData.java and the console window appears as shown in diagram 29.2.1.

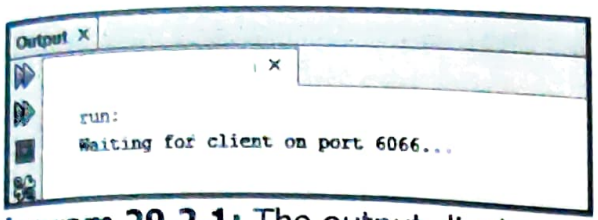


Diagram 29.2.1: The output displayed in the server machine

Then execute the Client application i.e. ClientData.java and the console window appears as shown in diagram 29.2.2. When the client application is run and messages are shared on the server side, then the console window of the server application appears as shown in diagram 29.2.3.

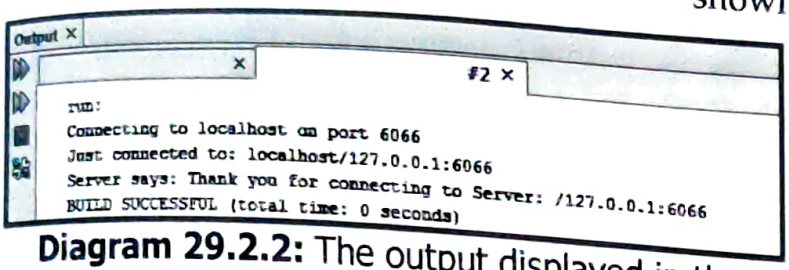


Diagram 29.2.2: The output displayed in the client machine

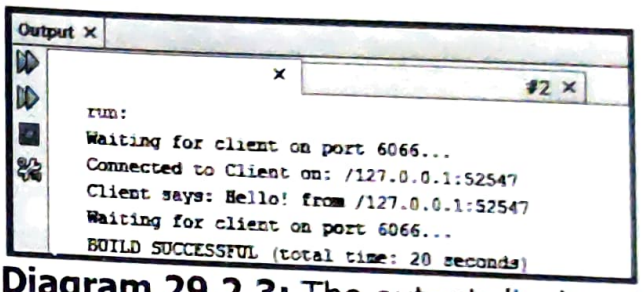


Diagram 29.2.3: The output displayed in the server machine

Working With URL

URL stands for **U**niform **R**esource **L**ocator, which represents a resource or an address on the World Wide Web such as a Web page or FTP directory. URLs are provided to the web browser so that it can locate files on the Internet in the same way that an address is provided on letters so that the post office can locate the correspondents.

Java programs that interact with the Internet also may use URLs to find the resources on the Internet they wish to access. Java uses **java.net.URL** to represent a URL address.

A URL has two main components:

- **Protocol identifier:** The protocol identifier indicates the name of the protocol to be used to fetch the resource

  For the URL: http://www.sharanamshah.com

  The protocol identifier is **http**

  The example uses the HyperText Transfer Protocol [HTTP], which is typically used to serve up hypertext documents. HTTP is just one of many different protocols used to access different types of resources on the net. Other protocols include File Transfer Protocol [FTP], Gopher, File and News.

- **Resource name:** The resource name is the complete address to the resource

  For the URL: http://www.sharanamshah.com

  The resource name is **www.sharanamshah.com**

The format of the resource name depends entirely on the protocol used, but for many protocols including HTTP, the resource name contains one or more of the following components:

- **Host Name:** The name of the machine on which the resource lives
- **File Name:** The pathname to the file on the machine
- **Port Number:** The port number to which to connect [optional]
- **Reference:** A reference to a named anchor within a resource that usually identifies a specific location within a file [optional]

For many Protocols, the host name and the file name are required, while the port number and reference are optional. For example: the resource name for an HTTP URL must specify a server on the network [Host Name] and the path to the document on that machine [File Name]; it also can specify a port number and a reference.

The protocol identifier and the resource name are separated by a colon [:] and two forward slashes [//].

Creating A URL

URL object is created from a String that represents the form of an URL address, which is readable to common man.

Creating Absolute URL

The following code spec uses a String containing the text to create a URL object:

```
URL url = new URL("http://www.sharanamshah.com/");
```

The URL object created above represents an **absolute URL**.

An absolute URL contains all of the information necessary to reach the resource in question.

Creating Relative URL

A relative URL contains only enough information to reach the resource relative to or in the context of another URL. Relative URL specifications are often used within HTML files.

Suppose an HTML file called **books.html** is written. Within this page, are links to other pages, java.html and database.html, which are on the same machine and in the same directory as index.html.

The links to java.html and database.html from index.html could be specified just as filenames in the following way:

```
<a href="java.html">Java Books</a>
<a href="database.html">Database Books</a>
```

These URL addresses are relative URLs i.e. the URLs are specified relative to the file in which they are contained i.e. index.html. So the two URLs looks like the following for the website www.sharanamshah.com:

```
http://www.sharanamshah.com/books/java.html
http://www.sharanamshah.com/books/database.html
```

The following code spec creates URL objects for the above pages relative to their common base URL http://www.sharanamshah.com:

```
URL url = new URL("http://www.sharanamshah.com/books/");
URL url1 = new URL(url, "java.html");
URL url2 = new URL(url, "database.html");
```

The code spec uses the URL() constructor, which allows creating a URL object from another URL object i.e. the base and a relative URL specification.

Syntax: [Constructor]

```
URL(URL BaseURL, String RelativeURL)
```

URL() Constructor is passed two parameters:

- The first parameter is a URL object that specifies the base of the new URL
- The second parameter is a String that specifies the rest of the resource name relative to the base

If the first parameter is null, then this constructor treats the second parameter like an absolute URL specification. Conversely, if the second parameter is an absolute URL specification, then the constructor ignores the first parameter.

URL() constructor is also useful for creating URL objects for named anchors, also called as references within a file. If the java.html has a named anchor called jee7 referencing to a book name of the file, then use the second parameter of URL() constructor to create a URL object for it:

```
URL url1book = new URL(url1,"#jee7");
```

Parsing A URL

java.net.URL provides several methods, which allows querying URL objects. The following can be retrieved from a URL using the accessor methods of URL class:

- Protocol
- Authority
- Host name
- Port number

The links to java.html and database.html from index.html could be specified just as filenames in the following way:

```
<a href="java.html">Java Books</a>
<a href="database.html">Database Books</a>
```

These URL addresses are relative URLs i.e. the URLs are specified relative to the file in which they are contained i.e. index.html. So the two URLs looks like the following for the website www.sharanamshah.com:

```
http://www.sharanamshah.com/books/java.html
http://www.sharanamshah.com/books/database.html
```

The following code spec creates URL objects for the above pages relative to their common base URL http://www.sharanamshah.com:

```
URL url = new URL("http://www.sharanamshah.com/books/");
URL url1 = new URL(url, "java.html");
URL url2 = new URL(url, "database.html");
```

The code spec uses the URL() constructor, which allows creating a URL object from another URL object i.e. the base and a relative URL specification.

Syntax: [Constructor]

URL(URL BaseURL, String RelativeURL)

URL() Constructor is passed two parameters:

- The first parameter is a URL object that specifies the base of the new URL
- The second parameter is a String that specifies the rest of the resource name relative to the base

If the first parameter is null, then this constructor treats the second parameter like an absolute URL specification. Conversely, if the second parameter is an absolute URL specification, then the constructor ignores the first parameter.

URL() constructor is also useful for creating URL objects for named anchors, also called as **references** within a file. If the java.html has a named anchor called jee7 referencing to a book name of the file, then use the second parameter of URL() constructor to create a URL object for it:

```
URL url1book = new URL(url1,"#jee7");
```

Parsing A URL

java.net.URL provides several methods, which allows querying URL objects. The following can be retrieved from a URL using the accessor methods of URL class:

- Protocol
- Authority
- Host name
- Port number

- Path
- Query
- Filename
- Reference

Example

The following example creates a URL from a string specification and then uses the URL object's methods to parse the URL.

Solution: [ParsingURL.java]

```
1  import java.io.IOException;
2  import java.net.URL;
3
4  public class ParsingURL {
5     public static void main(String[] args) {
6        try {
7           URL url = new
            URL("http://www.sharanamshah.com/index.html?book=java#jee7");
8           System.out.println("The URL is: " + url.toString());
9           System.out.println("The Protocol is: " + url.getProtocol());
10          System.out.println("The Authority is: " + url.getAuthority());
11          System.out.println("The File name is: " + url.getFile());
12          System.out.println("The Hostname is: " + url.getHost());
13          System.out.println("The Port is: " + url.getPort());
14          System.out.println("The Query is: " + url.getQuery());
15          System.out.println("The Anchor (Reference) is: " + url.getRef());
16       }catch(IOException e) {
17          System.out.println("Error message: " + e.getMessage());
18       }
19    }
20 }
```

Explanation:

A new URL object is created from a string specification of a URL:

```
URL url = new
URL("http://www.sharanamshah.com/index.html?book=java#jee7");
System.out.println("The URL is: " + url.toString());
```

toString() constructs a string representation of the URL.

The following accessor methods are used to retrieve information about the URL regardless of the constructor that used to create the URL object:

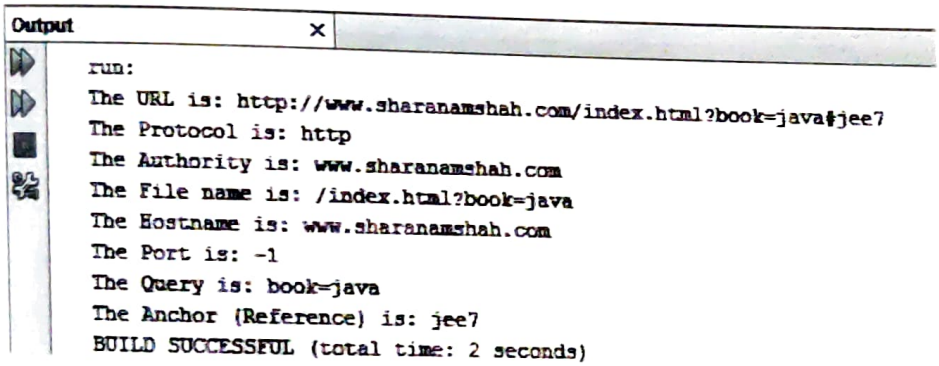
- **getProtocol()** returns the protocol identifier component of the URL
- **getAuthority()** returns the authority component of the URL
- **getFile()** returns the file name component of the URL
- **getHost()** returns the host name component of the URL

- getPort() returns the port number component of the URL. getPort() returns an integer that is the port number and if port number is not specified, then getPort() returns -1
- getQuery() returns the query component of the URL
- getRef() returns the reference component of the URL

REMINDER

Not all URL addresses contain all these components. **java.net.URL** provides these methods because HTTP URLs do not contain these components and are perhaps the most commonly used URLs. **java.net.URL** is somewhat **HTTP-centric**.

Output:



Reading A Web Page Directly From URL

After creating an URL object, a call to **URL.openStream()** is made to retrieve stream from which the contents of the URL can be read. openStream() returns a java.io.InputStream object, so reading from a URL is as easy as reading from an input stream.

Example

The following example uses openStream() to retrieve an input stream on the URL http://www.sharanamshah.com/. It opens **BufferedReader** on the input stream and reads from BufferedReader thereby reading from the URL. Everything read is copied to the standard output stream i.e. the console window.

Solution: [URLReading.java]

```
1 import java.io.BufferedReader;
2 import java.io.IOException;
3 import java.io.InputStreamReader;
4 import java.net.MalformedURLException;
5 import java.net.URL;
6
```

```
7  public class URLReading {
8     public static void main(String[] args) throws IOException {
9        try {
10          URL url = new URL("http://www.sharanamshah.com/");
11          try (BufferedReader in = new BufferedReader(new
            InputStreamReader(url.openStream()))) {
12             String inputLine;
13             while ((inputLine = in.readLine()) != null)
14                System.out.println(inputLine);
15          }
16       } catch (MalformedURLException ex) {
17          System.out.println("Error: " + ex.getMessage());
18       }
19    }
20 }
```

Explanation:

BufferedReader object is created, which reads text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays and lines.

```
try (BufferedReader in = new BufferedReader(new
InputStreamReader(url.openStream()))) {
```

An **InputStreamReader** is passed as the parameter in BufferedReader() constructor, which is a bridge from byte streams to character streams i.e. 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.

URL.openStream() is passed as parameter to InputStreamReader() constructor, which opens a connection to the given URL and returns an InputStream for reading from that connection

All this is coded in **try-with-resource** method.

A while loop is generated, whereby **BufferedReader.readLine()** reads a line of text, till it is terminated by any one of the following:

- Line feed [\n]
- A carriage return [\r]
- A carriage return followed immediately by a linefeed

```
while ((inputLine = in.readLine()) != null)
```

When URLReading.java is run, just scroll by in the console window and see the HTML commands and textual content from the HTML file located at http://www.sharanamshah.com/.

HINT

If the application hangs or an exception stack tree is shown, then just set the proxy host so that the application can find the server.

Connecting To A URL

Now the next step is to retrieve a **URLConnection** object or one of its protocol specific subclasses i.e. **java.net.HttpURLConnection**, once a URL object is created via URL.openConnection(). **URLConnection** is used to setup parameters and general request properties that may be needed before connecting.

Connection to the remote object represented by the URL is only initiated when the **URLConnection.connect()** is called. When this is done, it initializes a communication link between the application and the URL over the network.

Example

The following code spec opens a connection to the website www.sharanamshah.com.

Solution: [ConnectingURL.java]

```
1  import java.io.BufferedReader;
2  import java.io.IOException;
3  import java.io.InputStreamReader;
4  import java.net.MalformedURLException;
5  import java.net.URL;
6  import java.net.URLConnection;
7
8  public class ConnectingURL {
9      public static void main(String[] args) {
10         try {
11             URL url = new URL("http://www.sharanamshah.com");
12             URLConnection urlConn = url.openConnection();
13             try (BufferedReader in = new BufferedReader(new
               InputStreamReader(urlConn.getInputStream()))) {
14                 String inputLine;
15                 while ((inputLine = in.readLine()) != null)
16                     System.out.println(inputLine);
17             }
18         } catch (MalformedURLException  ex) {
19             System.out.println(ex.getMessage());
20         } catch (IOException ex) {
21             System.out.println(ex.getMessage());
22         }
23     }
24 }
```

Explanation:

A URLConnection object is created by calling URL.openConnection().

```
URLConnection urlConn = url.openConnection();
```

openConnection() opens a connection to the URL, allowing a client to communicate with the resource. Rather than getting an input stream directly from the URL, this example explicitly retrieves an URLConnection object and gets an input stream from the connection.

REMINDER

A new URLConnection object is created every time by calling openConnection() of the protocol handler for this URL.

The connection is opened implicitly by calling **URLConnection.getInputStream():**

```
try (BufferedReader in = new BufferedReader(new
InputStreamReader(urlConn.getInputStream()))) {
```

Just like the earlier example of URLReading.java, this example also creates a BufferedReader on the input stream and reads from it. The output of this example is identical to the output from the earlier example of URLReading.java, which opens a stream directly from the URL.

However, reading from URLConnection instead of reading directly from a URL might be more useful. The reason being that the URLConnection object can be used for other tasks such as writing to the URL at the same time.

HTTP Access

Java provides a general purpose, lightweight HTTP client API to access resources via the HTTP or HTTPS protocol.

java.net.URL can be used to define a pointer to a web resource while **java.net.HttpURLConnection** can be used to access a web resource. HttpURLConnection is **http specific URLConnection**. It works for HTTP protocol only.

HttpURLConnection allows creating an InputStream.

By the help of HttpURLConnection, information of any HTTP URL such as header information, status code, response code and so on can be retrieved.

Example

The following example retrieves the website www.sharanamshah.com. The output of this example is identical to the output from the earlier example of ConnectingURL.java, which uses URLConnection.

Solution: [ConnectingHttpURL.java]

```
1  import java.io.BufferedReader;
2  import java.io.IOException;
3  import java.io.InputStreamReader;
4  import java.net.HttpURLConnection;
5  import java.net.MalformedURLException;
6  import java.net.URL;
7
8  public class ConnectingHttpURL {
9      public static void main(String[] args) {
10         try {
11             URL url = new URL("http://www.sharanamshah.com");
12             HttpURLConnection conn = (HttpURLConnection) url.openConnection();
13             try (BufferedReader in = new BufferedReader(new
               InputStreamReader(conn.getInputStream()))) {
14                 String inputLine;
15                 while ((inputLine = in.readLine()) != null)
16                     System.out.println(inputLine);
17             }
18         } catch (MalformedURLException  ex) {
19             System.out.println(ex.getMessage());
20         } catch (IOException ex) {
21             System.out.println(ex.getMessage());
22         }
23     }
24 }
```

Explanation:

URL.openConnection() is used to open the connection to the URL:

HttpURLConnection conn = (HttpURLConnection) url.openConnection();

openConnection() returns the object of URLConnection.

HttpURLConnection type is typecasted while opening the connection to the URL.

The output of this example is same to the output from the earlier examples of URLReading.java and ConnectingURL.java.