Network Programming

Prepared by: Aakash Raj Shakya

Network Programming

1. 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.
2. 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.

Networking Basics

The **java.net** package provides support for the two common network protocols:

TCP:- TCP stands for **T**ransmission **C**ontrol **P**rotocol, 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 **U**ser **D**atagram **P**rotocol, a connection-less protocol that allows for packets of data to be transmitted between applications.

Networking Classes in the JDK

Through the classes in java.net, Java programs can use TCP or UDP to communicate over the Internet.

The **URL**, **URLConnection**, **Socket**, and **ServerSocket** classes all use TCP to communicate over the network.

The **DatagramPacket**, **DatagramSocket**, and **MulticastSocket** classes are for use with UDP.

Working with URLS

The Java URL class represents an URL.

URL is an acronym for **U**niform **R**esource **L**ocator. It points to a resource on the World Wide Web. For example:

Working with URLs contd..

A URL contains many information such as:

Protocol: In this case, https is the protocol.

Server name or IP Address: In this case, www.javatpoint.com is the server name.

Port Number: It is an optional attribute. If we write http://www.javatpoint.com:80/sonoojaiswal/ , 80 is the port number. If port number is not mentioned in the URL, it returns -1.

File Name or directory name: For example “https://www.mysite.com/index.php”, here index.php is the file name

Creating a URL

The easiest way to create a URL object is from a String that represents the human-readable form of the URL address.

This is typically the form that another person will use for a URL. In your Java program, you can use a String containing this text to create a URL object:

URL myURL = **new URL**("http://example.com/");

Creating a URL Relative to Another

For an example if you know multiple pages of a website, then you can also create multiple URL objects such as:

```
URL myURL = new URL("http://example.com/pages/");

URL page1URL = new URL(myURL, "page1.html");

URL page2URL = new URL(myURL, "page2.html");
```

URL example:

```
import java.net.*;

public class NetworkExample {
   public static void main(String[] args) {
      try {
         //Setting a URL
         URL url = new URL("https://www.example.com/");
         URL firstPageURL = new URL(url, "first-page");
         URL secondPageURL = new URL(url, "second-page");

      } catch (MalformedURLException e) {
         System.out.println("URL exception occurred. " + e.getMessage());
      }
   }
}
```

Parsing a URL

The URL class provides several methods that let you query URL objects. You can get the protocol, authority, host name, port number, path, query, filename, and reference from a URL using these accessor methods:

1. **getProtocol():** Returns the protocol identifier component of the URL.
2. **getAuthority():** Returns the authority component of the URL.
3. **getHost():** Returns the host name component of the URL.
4. **getPort():** Returns the port number component of the URL. The getPort method returns an integer that is the port number. If the port is not set, getPort returns -1.
5. **getPath():** Returns the path component of this URL.
6. **getQuery():** Returns the query component of this URL.
7. **getFile():** Returns the filename component of the URL. The getFile method returns the same as getPath, plus the concatenation of the value of getQuery, if any.

Parsing a URL contd..

8. **getRef():** Returns the reference component of the URL.

Example:

```
import java.net.*;

public class NetworkExample {
  public static void main(String[] args) {
    try {
      //Setting a URL
      URL url = new URL("https://www.example.com:8080/");
      URL firstPageURL = new URL(url, "first-page");
      URL secondPageURL = new URL(url, "second-page");
      URL queryComponentURL = new URL(url, "?username=aakash&password=test");
      URL urlWithReference = new URL(url, "#contact-us");
      //Parsing a URL
      System.out.println("Protocol: " + url.getProtocol());
      System.out.println("Authority: " + url.getAuthority());
      System.out.println("Host: " + url.getHost());
      System.out.println("Port: " + url.getPort());
      System.out.println("Path component: " + firstPageURL.getPath());
      System.out.println("Query component: " + queryComponentURL.getQuery());
      System.out.println("File name component: " + secondPageURL.getFile());
      System.out.println("Reference Component: " + urlWithReference.getRef());
    } catch (MalformedURLException e) {
      System.out.println("URL exception occurred. " + e.getMessage());
    }
  }
}
```

Output:

Protocol: https
Authority: www.example.com:8080
Host: www.example.com
Port: 8080
Path component: /first-page
Query component: username=aakash&password=test
File name component: /second-page
Reference Component: contact-us

Reading Directly from URLS

After you've successfully created a URL, you can call the URL's openStream() method to get a stream from which you can read the contents of the URL.

The **openStream()** method returns a **java.io.InputStream** object, so reading from a URL is as easy as reading from an input stream.

Reading URL example:

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

public class UrlContentReader {
   public static void main(String[] args) {
      try {
         URL url = new URL("https://www.codezeronepal.com/");
         BufferedReader reader = new BufferedReader(new InputStreamReader(url.openStream()));
         String inputLine;
         while ((inputLine = reader.readLine()) != null) {
            System.out.println(inputLine);
         }
         reader.close();
      } catch (MalformedURLException e) {
         System.out.println("URL exception occurred. " + e.getMessage());
      } catch (IOException ioEx) {
         System.out.println("IO Exception occurred with message: "+ ioEx.getMessage());
      }
   }
}
```

InetAddress class

Java InetAddress class represents an IP address.

The **java.net.InetAddress** class provides methods to get the IP of any host name for example www.google.com, www.facebook.com, etc.

Example of InetAddress Class:

```
import java.net.*;

public class InetAddressExample {
  public static void main(String[] args) {
    try {
      InetAddress ip  = InetAddress.getByName("www.facebook.com");
      System.out.println("Host Name: " + ip.getHostName());
      System.out.println("IP Address: " + ip.getHostAddress());
    } catch (UnknownHostException e) {
      System.out.println("Unknown host exception: " + e.getMessage());
    }
  }
}
```

Output:
Host Name: www.facebook.com
IP Address: 157.240.13.35

Sockets

1. A socket is one end-point of a two-way communication link between two programs running on the network.
2. Socket classes are used to represent the connection between a client program and a server program.
3. The **java.net** package provides two classes **Socket** and **ServerSocket** that implement the client side of the connection and the server side of the connection, respectively.
4. A socket is a combination of both IP address and a port number.

Transmission Control Protocol(TCP) Socket

Through the classes in java.net, Java programs can use TCP or UDP to communicate over the Internet.

The **URL**, **URLConnection**, **Socket**, and **ServerSocket** classes all use TCP to communicate over the network.

TCP socket contd..

The following steps occur when establishing a TCP connection between two computers using sockets:

1. The server instantiates a ServerSocket object, denoting which port number communication is to occur on.
2. The server invokes the accept() method of the ServerSocket class. This method waits until a client connects to the server on the given port.
3. After the server is waiting, a client instantiates a Socket object, specifying the server name and the port number to connect to.

TCP Socket contd..

4. The constructor of the Socket class attempts to connect the client to the specified server and the port number. If communication is established, the client now has a Socket object capable of communicating with the server.
5. 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.

Summarizing the client-server application

1. **Create** a **Socket** or **ServerSocket** object and open it over a specified port and IP address or hostname.
2. **Instantiate InputStreamReader**, **BufferedReader** and **PrintStream** objects to stream data to and from each socket.
3. **Read** from and **Write** to the stream between the sockets using their agreed-upon protocol.
4. **Close** the input and output streams
5. **Close** the **Socket** or **ServerSocket** objects.

Server Side Application:

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

public class Server {
   public static void main(String[] args) {
      try {
         ServerSocket serverSocket = new ServerSocket(6666);
         Socket socket = serverSocket.accept(); //establishes connection

         DataInputStream inputStream = new DataInputStream(socket.getInputStream());
         String str = inputStream.readUTF();

         System.out.println("message= " + str);
         serverSocket.close();
      } catch (Exception e) {
         System.out.println("Error Message: " + e.getMessage());
      }
   }
}
```

Client side Application:

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

public class Client {
   public static void main(String[] args) throws IOException {
      try {
         Socket clientSocket = new Socket("localhost", 6666);

         DataOutputStream outputStream = new DataOutputStream(clientSocket.getOutputStream());
         outputStream.writeUTF("Hello Server");
         outputStream.flush();
         outputStream.close();

         clientSocket.close();
      } catch (Exception e) {
         System.out.println("Error Message: " + e.getMessage());
      }
   }
}
```

UDP Sockets

DatagramSockets are Java's mechanism for network communication via UDP instead of TCP.

DatagramSockets can be used to both send and receive packets over the Internet.

One of the examples where UDP is preferred over TCP is the live coverage of TV channels.

Example of Sending DatagramPacket by DatagramSocket

```
import java.net.*;

public class Sender {
  public static void main(String[] args) throws Exception {
    DatagramSocket socket = new DatagramSocket();
    try {
      String str = "Welcome java";
      InetAddress ip = InetAddress.getByName("127.0.0.1");
      DatagramPacket packet = new DatagramPacket(str.getBytes(), str.length(), ip, 3000);
      socket.send(packet);
    } finally {
      socket.close();
    }
  }
}
```

Example of Receiving DatagramPacket by DatagramSocket

```
import java.net.*;

public class Receiver {
  public static void main(String[] args) throws Exception {
    DatagramSocket socket = new DatagramSocket(3000);
    try {
      byte[] buf = new byte[1024];
      DatagramPacket packet = new DatagramPacket(buf, 1024);
      socket.receive(packet);
      String str = new String(packet.getData(), 0, packet.getLength());
      System.out.println(str);
    } finally {
      socket.close();
    }
  }
}
```

Half Close

When a client program sends a request to the server, the server needs to be able to determine when the end of the request occurs. For that reason, many Internet protocols (such as SMTP) are line oriented. Other protocols contain a header that specifies the size of the request data. Otherwise, indicating the end of the request data is harder than writing data to a file. With a file, you'd just close the file at the end of the data. However, if you close a socket, then you immediately disconnect from the server.

The half-close overcomes this problem. You can close the output stream of a socket, thereby indicating to the server the end of the request data, but keep the input stream open so that you can read the response.
We can use **socket.shutdownOutput();** or **socket.shutdownInput();** to achieve half close.

Interruptible Sockets

When you connect to a socket, the current thread blocks until the connection has been established or a timeout has elapsed. Similarly, when you read or write data through a socket, the current thread blocks until the operation is successful or has timed out.

In interactive applications, you would like to give users an option to simply cancel a socket connection that does not appear to produce results. However, if a thread blocks on an unresponsive socket, you cannot unblock it by calling interrupt. To interrupt a socket operation, you use a **SocketChannel**, a feature of the **java.nio** package. Open the SocketChannel like this:

SocketChannel channel = SocketChannel.open(new InetSocketAddress(host, port));

A channel does not have associated streams. Instead, it has read and write methods that make use of Buffer objects.These methods are declared in interfaces **ReadableByteChannel** and **WritableByteChannel**.

Interruptible Sockets contd..

If you don't want to deal with buffers, you can use the Scanner class to read from a SocketChannel because Scanner has a constructor with a ReadableByteChannel parameter:

Scanner in = new Scanner(channel);

To turn a channel into an output stream, use the static Channels.newOutputStream method.

OutputStream outStream = Channels.newOutputStream(channel);

That's all you need to do. **Whenever a thread is interrupted during an open, read, or write operation, the operation does not block but is terminated with an exception.**

Send Mail example:

```
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;

public class SendMail {
  public static void main(String[] args) {
     final String username = "test@gmail.com";
     final String password = "password";
     Properties props = new Properties();
     props.put("mail.smtp.auth", "true");
     props.put("mail.smtp.starttls.enable", "true");
     props.put("mail.smtp.host", "smtp.gmail.com");
     props.put("mail.smtp.port", "587");
     Session session = Session.getInstance(props,
           new Authenticator() {
              protected PasswordAuthentication getPasswordAuthentication() {
                 return new PasswordAuthentication(username, password);
              }
           });
     try {
        Message message = new MimeMessage(session);
        message.setFrom(new InternetAddress("test@gmail.com"));
        message.setRecipients(Message.RecipientType.TO,
              InternetAddress.parse("aakash_shakya@hotmail.com"));
        message.setSubject("Testing Subject");
        message.setText("Hello Aakash," Congrats you succeeded sending mail\n through Java.mail API.");
              Transport.send(message);
        System.out.println("Your email has been sent successfully");
     } catch (MessagingException e) {
        throw new RuntimeException(e);
     }
  }
}
```