

Miscellaneous Applications

Contents

Using the <code>URLConnection</code> class.....	2
Task	5
UDP and multicasting.....	6
Multicast Server	7
Multicast Client	10
Task	12
Using the <i>NetworkInterface</i> class.....	13
Task	13

Using the URLConnection class

- A simple way of accessing a server is to use the *URLConnection* class.
- This class represents a connection between an application and a URL instance.
 - A URL instance represents a resource on the Internet.
- Create an application to read HTML page of a website

```
public class URLConnect {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter a url: ");
        String url = scanner.nextLine();
        accessWebsite(url);
    }

    private static void accessWebsite(String url) {
        BufferedReader br = null;
        try {
            URL my_url = new URL(url);
            URLConnection urlConnection = my_url.openConnection();

            InputStreamReader isr = new
InputStreamReader(urlConnection.getInputStream());
            br = new BufferedReader(isr);

            String line;
            while ((line = br.readLine()) != null) {
                System.out.println(line);
            }
        } catch (IOException exception) {
            System.out.println("\n Cannot access URL");
            System.exit(1);
        } finally {
            System.out.println("\n Closing the connection");
            try {
                br.close();
                System.exit(1);
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

- Create a class and the *main* method and a static function *accessWebsite* with *String* parameter.

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

    }
}

private static void accessWebsite(String url){
}
```

- In the main method, read *URL* string from user and call *accessServer* mehtod.

```
public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    System.out.print("Enter a url: ");
    String url = scanner.nextLine();
    accessWebsite(url);
}
```

- In *accessServer*, set up *try – catch – finally* block.
- Create a *BufferedReader* variable to read incoming data.

```
private static void accessWebsite(String url){
    BufferedReader br = null;
    try {

    } catch () {

    }finally {

    }
}
```

- Inside *try* block, define an object of class *URL* that represents a pointer to a resource on the World Wide Web.
- Define an object of the abstract class *URLConnection*, that will handle the communication link between the application and a URL resource.
- Open the connection to the website using *openConnection* method in the *URL* class.

```
try {
    URL my_url = new URL(url);
    URLConnection urlConnection =
my_url.openConnection();
```

- Define an object of *InputStreamReader*, which is a bridge that reads a stream of bytes and decodes them into characters.
 - This gets the input stream from *URLConnection* object.
- Initialize the *BufferedReader* object with *InputStreamReader* as a parameter to the constructor.

```
InputStreamReader isr = new InputStreamReader(
urlConnection.getInputStream());
br = new BufferedReader(isr);
```

- In the *try* block, create a *while* loop to read the data line by line and print the data to the console.

```
String line;
while ((line = br.readLine()) != null) {
    System.out.println(line);
}
```

- In the *catch* block, catch the exception and print appropriate message.

```
catch (IOException exception) {  
    System.out.println("Cannot access URL");  
    System.exit(1);  
}
```

- In the *finally* block, close the connection

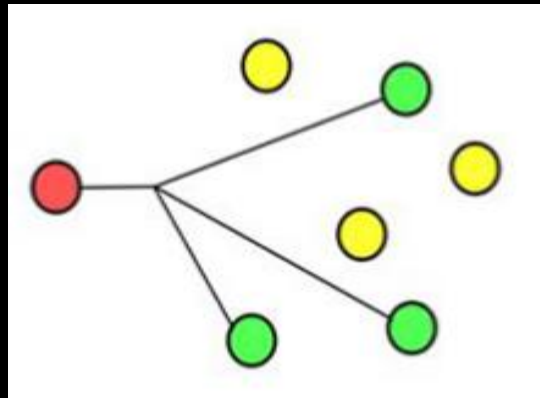
```
finally {  
    System.out.println("Closing connection");  
    try {  
        br.close();  
        System.exit(1);  
    } catch (IOException e) {  
        e.printStackTrace();  
    }  
}
```

Task

- Rewrite the same program above but print the output to an HTML file instead of displaying it on the console.

UDP and multicasting

- Multicasting is a technique for sending messages to a group of receivers on a periodic basis.
 - Multicasting is a type of one-to-many and many-to-many communication as it allows sender or senders to send data packets to multiple receivers at once across LANs or WANs.
 - It uses a UDP server and one or more UDP clients.



- Multicasting will send an identical message to every member of a group.
- A group is identified by a multicast address.
 - A multicast address must use the following IP address range:
224.0.0.0 through 239.255.255.255.
- Clients must join the group before they can receive any multicast messages.

Address Class	RANGE	Default Subnet Mask
A	1.0.0.0 to 126.255.255.255	255.0.0.0
B	128.0.0.0 to 191.255.255.255	255.255.0.0
C	192.0.0.0 to 223.255.255.255	255.255.255.0
D	224.0.0.0 to 239.255.255.255	Reserved for Multicasting
E	240.0.0.0 to 254.255.255.255	Experimental

Note: Class A addresses 127.0.0.0 to 127.255.255.255 cannot be used and is reserved for loopback testing.

Multicast Server

- We will create a simple time server. The server will send a date and time string to clients every second.

```
public class MulticastServer {
    public static void main(String[] args) {
        System.out.println("Multicast Time Server");
        DatagramSocket serverSocket = null;

        try {
            serverSocket = new DatagramSocket();
            while (true) {
                String dateText = new Date().toString();
                byte[] buffer = new byte[256];
                buffer = dateText.getBytes();
                InetAddress group =
InetAddress.getByName("224.0.0.0");
                DatagramPacket packet;
                packet = new DatagramPacket(buffer,
buffer.length,
                    group, 8888);
                serverSocket.send(packet);
                System.out.println("Time sent: " + dateText);
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException ex) {
                    System.out.println(ex.getMessage());
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
            System.exit(1);
        }
    }
}
```

- Create *MulticastServer* class with main method
- Inside *main* create *DatagramSocket* object.
- Inside *main*, set up *try – catch* block.
- Inside *try*, initialize *serverSocket* object.

```
public static void main(String[] args) {  
    System.out.println("Multicast Time Server");  
    DatagramSocket serverSocket = null;  
  
    try {  
        serverSocket = new DatagramSocket();  
    } catch (IOException e) {  
    }  
}
```

- Inside *try*, create an infinite *while* loop.
- Inside the loop create an array of bytes to hold the current date and time.

```
while (true) {  
    String dateText = new Date().toString();  
    byte[] buffer = new byte[256];  
    buffer = dateText.getBytes();  
}
```


- Inside loop, create an *InetAddress* object to represent the multicast group.
- Inside loop, create a *DatagramPacket* to hold the array of bytes, the multicast IP of the group and port number.
- Inside loop, cast (send) the message to the group using *send* method.

```
InetAddress group =
InetAddress.getByName("224.0.0.0");
DatagramPacket packet = new
DatagramPacket(buffer, buffer.length, group,
8888);
serverSocket.send(packet);
System.out.println("Time sent: " + dateText);
```

- To make the server send the message every 1 second, we pause the server for 1000 millisecond using *Thread.sleep()* method.

```
try {
    Thread.sleep(1000);
} catch (InterruptedException ex) {
    System.out.println(ex.getMessage());
    System.exit(1);
}
```

- Note that this server only sends messages. It never receives from client.
- Catch the exception.

```
catch (IOException e) {
    e.printStackTrace();
    System.exit(1);
}
```

Multicast Client

```
public class MulticastClient {
    public static void main(String args[]) {
        System.out.println("Multicast Time Client");
        try (MulticastSocket socket = new
MulticastSocket(8888)) {
            InetAddress group =
                InetAddress.getByName("224.0.0.0");
            socket.joinGroup(group);
            System.out.println("Multicast Group Joined");
            byte[] buffer = new byte[256];
            DatagramPacket packet =
                new DatagramPacket(buffer,
buffer.length);
            for (int i = 0; i < 5; i++) {
                socket.receive(packet);
                String received = new
String(packet.getData());
                System.out.println(received.trim());
            }
            socket.leaveGroup(group);
        } catch (IOException ex) {
            ex.printStackTrace();
        }
        System.out.println("Multicast Time Client
Terminated");
    }
}
```

- Create *MulticastClient* class with *main* function.
- Inside *main*, create a *try with resources – catch* block.
 - Remember that *try with resources* has the form *try() { } catch { }*. It ensures that each resource is closed at the end of the statement.

```
public class MulticastClient {
    public static void main(String args[]) {
        System.out.println("Multicast Time
Client");
        try ( ) {

            } catch ( ) {

            }
        System.out.println("Multicast Time
Client Terminated");
    }
}
```

- Inside () of the *try*, we define a *MulticastSocket* object, which is a UDP socket used for sending and receiving multicast packets and joining groups.
- Inside *try* block, create an *InetAddress* object to access the group at IP 224.0.0.0. Then, use the socket to join the group.

```
try (MulticastSocket socket = new
MulticastSocket(8888)) {
    InetAddress group =
        InetAddress.getByName("224.0.0.0");
    socket.joinGroup(group);
    System.out.println("Multicast Group
Joined");
} catch ( ) {

}
```

- Inside *try*, create a byte array to receive the incoming datagram packets.

```
byte[] buffer = new byte[256];
DatagramPacket packet =
    new DatagramPacket(buffer, buffer.length);
```

- We want the client to receive 5 packets, so we create a *for* loop to receive the packets, extract the data from the packet, and print it to the console.
- After the *for* loop, we leave the group using *socket.leaveGroup()*.

```
for (int i = 0; i < 5; i++) {
    socket.receive(packet);
    String received = new String(packet.getData());
    System.out.println(received.trim());
}
socket.leaveGroup(group);
```

- Catch the exception.

```
catch (IOException ex) {
    ex.printStackTrace();
    System.exit(1);
}
System.out.println("Multicast Time Client
Terminated");
```

- Now run the server then run the client.
- What happens if we run the client first?

Task

- What are unicast and broadcast? What are the differences between unicast, multicast, and broadcast?

Using the *NetworkInterface* class

- The *NetworkInterface* class provides a means of accessing the devices that act as nodes on a network.
 - This class also provides a means to get low-level device addresses.
 - This class represents a Network Interface made up of a name, and a list of IP addresses assigned to this interface.
- A network interface is the point of connection between a computer and a network (NIC).

```
public class NetInterface {
    public static void main(String[] args) {
        try {
            Enumeration<NetworkInterface> interfaceEnum =
                NetworkInterface.getNetworkInterfaces();
            System.out.printf("Name Display name\n");
            for (NetworkInterface element :
                Collections.list(interfaceEnum)) {
                System.out.printf("%-8s %-32s\n",
                    element.getName(),
                    element.getDisplayName());
            }
        } catch (SocketException ex) {
            ex.printStackTrace();
        }
    }
}
```

- The *Enumeration* interface defines the methods by which you can enumerate (obtain one at a time) the elements in a collection of objects.
- The *Collections.list(interfaceEnum)* line returns an array list of the elements in the *interfaceEnum*.

Task

- What are java Generics and wildcards?
- What is the difference between array list and enums?