PRAKTIKUM SW2

Networking

NETWORKING

Introduction

Sockets

ServerSockets

URL and URLConnection

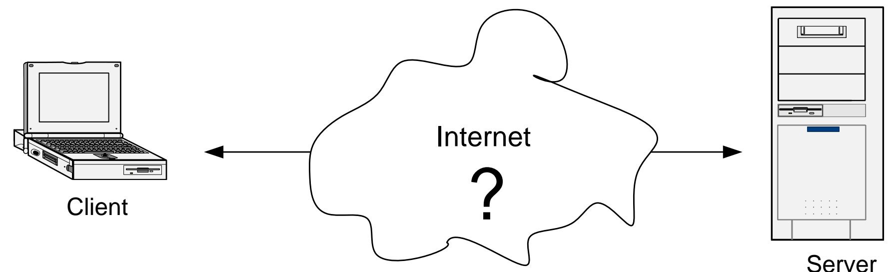
Summary

NETWORK PROGRAMMING

Writing programs where different components are executed on different programming nodes and which communicate over a network

Challenges of network programming

- Finding other computing nodes
- Setting up a connection
- Agreement on communication
- Exchanging messages
- Exchanging data



2 Models:

- Socket-Streaming (this chapter)
- RMI - Remote objects (chapter on Remoting)

Principles of Socket Streaming

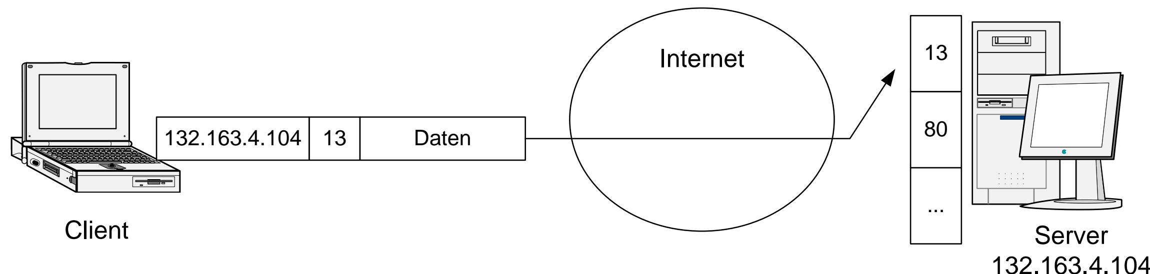
IP Addresses and Ports

Client und Server

- Client sends a request to server
- Server handles request and
- sends back response

Exchanging of messages and data between client and server by byte streams (see Streaming)

➔ Messages and data have to be interpreted: therefore a protocol is needed,
e.g., HTTP, SMTP and others



IP-Addressierung

4 byte IP address

divided in network node id and host id

Address types: A, B, C

Type	Netzwerk ID	Host ID	Description
A	7	24	for big providers
B	14	16	for median providers: support 65534 different hosts in network node
C	21	8	for smaller providers: support 65534 different hosts in network node

Domain names: symbolic Name for IP address

Ports

Ports for identifying services at a host computer

integer in the range of 0 .. 65535

Some ports for standardized services (well known ports: 0-1024):

Name	Port	Transport	Beschreibung
<nil>	0	-	reserved
echo	7	tcp/udp	server returns request as is
discard	9	tcp/udp	ignores requests from server
daytime	13	tcp/udp	returns ASCII string with date and time
ftp	21	tcp	seding and receiving files
telnet	23	tcp	interactive sessions
smtp	25	tcp	sending emails
whois	43	tcp	simple name service
finger	79	tcp	provides user information
www	80	tcp	Web server
pop3	110	tcp	receiving emails
rmi	1099	tcp	remote method invocation

Sockets

Sockets are interfaces for network communication

- stream-based
- channel-based (see Chapter NIO)

Stream-based with same protocol as for file streaming

- Setting up connection by IP address and port
- Creating Input- and OutputStream
- Reading and writing through streams
- Closing connection

Distinction between Sockets and Server Sockets

- Sockets (class **`Socket`**) for reading and writing (at client as well as server)
- Server sockets (class **`ServerSocket`**) for accepting client requests

Important Classes from java.net

`InetAddress`, `InetSocketAddress`

- Represent IP Address and IP address and port

`Socket`

- Socket for communication

`ServerSocket`

- for accepting requests

`URL`, `URLConnection`

- Allow direct access of Web ressource

Class `InetAddress` represents IP address

```
public class InetAddress extends Object implements Serializable
```

- static method for creating **`InetAddress`** object

```
static InetAddress[] getAllByName(String host)
static InetAddress   getByAddress(byte[] addr)
static InetAddress   getByAddress(String host, byte[] addr)
static InetAddress   getByName(String host)
static InetAddress   getLocalHost()
```

- Access to 4 Byte IP-Adresse and symbolic name

```
String getCanonicalHostName()
String getHostName()
String getHostAddress()
byte[] getAddress()
```

```
Canonical: oberon.ssw.uni-linz.ac.at
Name: oberon.ssw.uni-linz.ac.at
Host address: 140.78.145.1
Address: {140, 78, 145, 1}
```

Class InetSocketAddress

Class `InetSocketAddress` represents an IP address plus port

```
public class InetSocketAddress extends SocketAddress
```

- Constructors for creating **`InetAddress`** objects

```
InetSocketAddress(InetAddress addr, int port)
InetSocketAddress(String hostname, int port)
InetSocketAddress(int port)
```

- Access to host address and port

```
InetAddress getAddress()
String getHostName()
int getPort()
```

NETWORKING

Introduction

Sockets

ServerSockets

URL and URLConnection

Summary

Class Socket

Class `Socket` represents socket connection

```
public class Socket
```

- Standard constructor

```
public Socket()
```

NO connection established

- Constructor with address and port

```
public Socket(String address, int port)
                throws UnknownHostException, IOException
public Socket(InetAddress address, int port)
                throws IOException
```

connection established

- Connect method (when using standard constructor)

```
public void connect(SocketAddress endpoint) throws IOException
public void connect(SocketAddress endpoint, int timeout)
                       throws IOException
```

- Accessing **`InputStream`** and **`OutputStream`** for this socket

```
public InputStream getInputStream()

public OutputStream getOutputStream()
```

Example Client Socket: Accessing WebServer (1/2)

Example shows a client request to a Web server

- server address provides as command line argument (args[0])
- creating socket for provides server addreas and port 80

```
C:\> java GetPage "www.ssw.uni-linz.ac.at",  "/Teaching/Lectures/PSW2/2012W/index.html"
```

```java
public class GetPage {
  public static void main(String[] args) {
    Socket sock = null;
    try {
      InetAddress adr = InetAddress.getByName(args[0]);
      sock = new Socket(adr, 80);
```

- an InputStream and OutputStream are obtained from the socket
- the HTTP GET requrest is created for getting the resource whose name provided as second command line argument (args[1])

```java
      OutputStream out = sock.getOutputStream();
      BufferedReader in =
          new BufferedReader(new InputStreamReader(sock.getInputStream()));
      String s = "GET " + args[1] + " HTTP/1.0" + "\r\n\r\n";
      out.write(s.getBytes());
      out.flush();
```

DO NOT FORGET: only with flush() the data in the output buffer is sent out!!

Example Client Socket: Accessing WebServer (2/2)

Response from server is read line by line and put out on the console

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

null when no more lines are sent by the server, i.e., server has closed connection

A set of exections can occur; catch clauses for cathich exceptions

Finally close sockets (InputStream and OuputStream will be closed automatically)

```
    } catch (IOException e) {
      System.err.println(e.toString());
    } finally {
      try {
        sock.close();
      } catch (Exception e) { }
    }
  }
}
```

Closing resources in finally block

TRY-WITH-RESOURCES ANWEISUNG

try with resources automatically closes resources
Resources have to implement `AutoClosable`

```
try ( declare and create resources here ) {
    use resources
}
```

```
public interface AutoCloseable {
  void close() throws Exception;
}
```

Resources are closed automatically at end of block

Example: GetPage

```
public class GetPage2 {
    public static void main(String[] args) {
        try (
               Socket sock =  new Socket(args[0], 80);
               OutputStream out = sock.getOutputStream();
               BufferedReader in = new BufferedReader(
                                        new InputStreamReader(
                                            sock.getInputStream()))
            ) {
            // send GET command
            String s = "GET " + args[1] + " HTTP/1.0" + "\r\n\r\n";
            ...
        } catch (IOException e) {
            System.err.println(e.toString());
        }
    }
}
```

Example: Sending EMail (1/6)

Sending emails is done based on SMTP protokol on port 25

SMTP protokoll has the following format (example, see http://tools.ietf.org/html/rfc2821 for spezifikation)

```
HELO Sendender Host
MAIL FROM: <E-Mail Adresse des Absenders>
RCPT TO: <E-Mail Adresse des Empfängers>
DATA
From: <E-Mail Adresse des Absenders>
Subject: <Betreff>
To: <E-Mail Adresse des Empfängers>
Date: <Datum>

E-Mail Nachricht in mehreren Zeilen
.
QUIT
```

Server responds with status message (see http://www.greenend.org.uk/rjk/2000/05/21/smtp-replies.html):

- **1XX:** Mailserver hat die Anforderung akzeptiert, ist aber selbst noch nicht tätig geworden. Eine Bestätigungsmeldung ist erforderlich.
- **2XX:** Mailserver hat die Anforderung erfolgreich ohne Fehler ausgeführt.
- **3XX:** Mailserver hat die Anforderung verstanden, benötigt aber zur Verarbeitung weitere Informationen.
- **4XX:** Mailserver hat einen temporären Fehler festgestellt. Wenn die Anforderung ohne jegliche Änderung wiederholt wird, kann die Verarbeitung möglicherweise abgeschlossen werden.
- **5XX:** Mailserver hat einen fatalen Fehler festgestellt. Ihre Anforderung kann nicht verarbeitet werden.

EXAMPLE: SENDING EMAIL (2/6)

Client messages	Server responses	Explanation
	220 mail.example.com SMTP Foo Mailserver	Server response to connection
HELO client.exmpl.org		Client login
	250 Hello client.example.org, nice to meet you	Server acknowledges login.
MAIL FROM:<bar@exmpl.org>		Client sends sender address
	250 Sender OK	
RCPT TO:<foo@exmpl.com>		Client sends receiver address
	250 Recipient OK	
DATA		Client begins sending data
	354 End data with <CR><LF>.<CR><LF>	
From: <bar@exmpl.org><CR><LF> To: <foo@exmpl.com><CR><LF> Subject: Testmail<CR><LF> Date: Thu, 26 Oct 2006<CR><LF> 13:10:50 +0200<CR><LF> Testmail<CR><LF> .<CR><LF>		Client sends data line by line (ending lines with <CR><LF>) Clients signals end of data by single point in line.
	250 Message accepted for delivery	
QUIT		Client logs out
	221 See you later	

EXAMPLE: SENDING EMAIL (3/6)

Establishing connection by

- opening socket to email host
- creating **`PrintWriter`** and **`BufferedReader`** from socket

```
Socket mailSocket = new Socket("email.uni-linz.ac.at", 25);
PrintWriter out = new PrintWriter(mailSocket.getOutputStream());
BufferedReader in = new BufferedReader(
                  new InputStreamReader(mailSocket.getInputStream()));
```

Sending messages line by line

- each line has to end with **`"\r\n"`**
- after each command and with end of data the client has to read for the response from server

DATA section consists of

- first header with
  - sender (`From: …`)
  - receiver (`To: …`)
  - data (`Date: …`)
  - subject (`Subject: …`)
- multiple lines for email data
- at end of email data single point . in line (`".\r\n"`)

EXAMPLE: SENDING EMAIL (4/6)

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

        try (
                Socket mailSocket = new Socket("email.uni-linz.ac.at", 25);
                PrintWriter out = new PrintWriter(mailSocket.getOutputStream());
                BufferedReader in = new BufferedReader(
                                        new InputStreamReader(mailSocket.getInputStream()));

                ) {
          String ret = receive(in);
          String hostName = InetAddress.getLocalHost().getHostName();
          send(out, "HELO " + hostName);
          ret = receive(in);
          send(out, "MAIL FROM: <max.mustermann@jku.at>");
          ret = receive(in);
          send(out, "RCPT TO: <herbert.praehofer@jku.at>");
          ret = receive(in);
          send(out, "DATA");
          ret = receive(in);
          send(out, "From: max.mustermann@jku.at");
          send(out, "To: franz.mayr@jku.at");
          send(out, "Date: " + DateFormat.getDateTimeInstance().format(new Date()));
          send(out, "Subject: 2 zeilige Mail");
          send(out, "");
          send(out, "Dies ist eine Mail");
          send(out, "mit zwei Zeilen.");
          send(out, ".");
          ret = receive(in);
          send(out, "Quit");
          ret = receive(in);
        }
        catch (IOException e) {
            e.printStackTrace();
        }
    }
```

EXAMPLE: SENDING EMAIL (5/6)

```
    /** Receives a line from BufferedReader in
     * @param in
     * @return
     * @throws IOException
     */
     private static String receive(BufferedReader in) throws IOException {
          StringBuilder b = new StringBuilder();
          do  {
               b.append(in.readLine());
               System.out.println("smtp (receiving):" + b);
          } while (b.charAt(3) == '-');
          return b.toString();
     }


    /** Sends a line to PrintWriter out
     * @param out
     * @param str
     * @throws IOException
     */
    private static void send(PrintWriter out, String str) throws IOException {
        out.print(str);
        System.out.println("smtp (sending): " + str);
        out.print("\r\n");
        out.flush();
    }

}
```

EXAMPLE: SENDING EMAIL (6/6)

Output log

```
smtp (receiving):220 email.uni-linz.ac.at ESMTP
smtp (sending): HELO t410
smtp (receiving):250 email.uni-linz.ac.at
smtp (sending): MAIL FROM: <max.mustermann@jku.at>
smtp (receiving):250 2.1.0 Ok
smtp (sending): RCPT TO: <herbert.praehofer@jku.at>
smtp (receiving):250 2.1.5 Ok
smtp (sending): DATA
smtp (receiving):354 End data with <CR><LF>.<CR><LF>
smtp (sending): From: max.mustermann@jku.at
smtp (sending): To: franz.mayr@jku.at
smtp (sending): Date: 19.11.2012 20:05:37
smtp (sending): Subject: 2 zeilige Mail
smtp (sending):
smtp (sending): Dies ist eine Mail
smtp (sending): mit zwei Zeilen.
smtp (sending): .
smtp (receiving):250 2.0.0 Ok: queued as DC06D1E8CB1
smtp (sending): Quit
smtp (receiving):221 2.0.0 Bye
```

TIMEOUTS

Read operations are blocking ➔ Timeout required

- setting timeout for socket

```
try {
    socket.setSoTimeout(10000); // 10000 ms = 10 sec
    ...
```

- catching exception when timeout occurs

```
    ...
} catch (InterruptedIOException e) { ... }
```

Connection does block without timeout

```
try {
    socket = new Socket(domain, port);
...
```

blocks forever!

➔ Alternatively use **connect** with timeout

```
try {
    socket = new Socket();
    socket.connect(new InetSocketAddress(domain, port), TIMEOUT);
     ...
} catch (SocketTimeoutException e) { ...
} catch (IOException e) { ...
}
```

blocks until TIMEOUT!

NETWORKING

Introduction

Sockets

ServerSockets

URL and URLConnection

Summary

Klasse ServerSocket

Class `ServerSocket` for accepting client requests

```
public class ServerSocket
```

- Constructor for creating ServerSocket with respective port number
  ➔ ServerSocket then runs on local machine and can listen to requests for this port

```
    public ServerSocket(int port) throws IOException
```

- Method for accepting and creating client connection requests

```
    public Socket accept()
```

  - this method blocks until a request arrives from a client
  - then a connection is established and a socket for communication with the client is returned

Connecting Server – Client (1/2)

Server

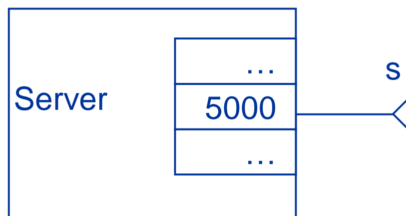
- Create **ServerSocket** on port (e.g., port 5000)
- Wait for client (blocks)

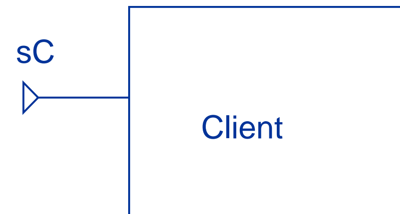
```
ServerSocket s = null;
Socket sS = null;
try {
  s = new ServerSocket(5000);
  sS = s.accept();
  …
```

Client

- creates **Socket** with host address and port

```
Socket sC = null;
try {
  sC = new Socket(host, 5000)
```





VERBINDUNGSAUFBAU SERVER – CLIENT (2/2)

- accept connection request

```
sS = s.accept();
```

- establish connection

```
sC = new Socket(host, 5000)
```

- create Input- und OutputStreams for communication
- close socket

```
  OutputStream out =
          sS. getOutputStream();
  InputStream in =
          sS.getInputStream();
  in.read()
  out.write(b);
  …
} catch (…) {…
} finally {
  sS.close();
}
```

- create Input- und OutputStreams for communication
- close socket

```
  OutputStream out =
          sC. getOutputStream();
  InputStream in =
          sC.getInputStream();
  out.write(b);
  in.read()
  …
} catch (…) {…
} finally {
  sC.close();
}
```

EXAMPLE SERVERSOCKET: SIMPLEECHOSERVER (1/2)

Example shows the creating of a simple EchoServer

- ServerSocket is created for port 7 (standard for echo)
- then listens for client requests

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

public class SimpleEchoServer {
  public static void main(String[] args) {
    ServerSocket echod = null;
    Socket socket = null;
    try {
      System.out.println("Warte auf Verbindung auf Port 7...");
      echod = new ServerSocket(7);
      socket = echod.accept();
```

- **accept** returns Socket from which **Input-** and **OuputStreams** are created

```
      System.out.println("Verbindung hergestellt");
      InputStream in = socket.getInputStream();
      OutputStream out = socket.getOutputStream();
```

Example ServerSocket: SimpleEchoServer (2/2)

- **`InputStream in`** und **`OutputStream out`** are used for reading data and sending data back; reading and writing is done byte by byte

```
      int c;
      while ((c = in.read()) != -1) {
        out.write((char) c);
        System.out.print((char) c);
      }
```

-1, for end of data
(client has closed connection)

- Closing client and server socket

```
      System.out.println("Closing connection");
    } catch (IOException e) {
      System.err.println(e.toString());
    } finally {
      try { socket.close(); } catch (Exception ioex) { }
      try { echod.close(); } catch (Exception ioex) { }
    }
  }
}
```

Example : EchoServer, multiple Clients (1/3)

In above server example, server was able to handle only one client

Now server accepts clients in a loop and handles clients in separate thread

```
public class EchoServer {
    private static volatile boolean terminate = false;

    public static void main(String[] args) {
        int cnt = 0;
        try (
                ServerSocket echod = new ServerSocket(7);
            ) {
            echod.setSoTimeout(1000);
            System.out.println("Warte auf Verbindungen auf Port 7...");
            while (! terminate) {
                try {
                    Socket socket = echod.accept();
                    new Thread(new EchoRunnable(++cnt, socket)).start();
                } catch (InterruptedIOException e) { }
            }
        } catch (IOException e) { }
    }

    public static void terminate() {
        terminate = true;
    }
```

Timeout required for termination as accept cannot be interrupted !

Example : EchoServer, multiple Clients (2/3)

In `run` method of `EchoRunnable` data are read and sent back

```
    private static class EchoRunnable implements Runnable {
        private int id;
        private Socket socket;

        public EchoRunnable(int id, Socket socket) {
            this.id = id;
            this.socket = socket;
        }

        public void run() {
            try (    InputStream in = socket.getInputStream();
                     OutputStream out = socket.getOutputStream();
                ) {
                Out.println("Runner " + id + " started ");
                int c;
                while ((c = in.read()) != -1) {
                    out.write((char) c);
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
            Out.println("Runner " + id + " terminated ");
        } finally {
            try { socket.close(); } catch (Exception e) {}
        }
    }
}
```

EXAMPLE : ECHOSERVER, MULTIPLE CLIENTS (3/3)

Closing output before input

```
public class EchoClient {
  public static void main(String[] args) {
    try (
        Socket clientSocket = new Socket("localhost", 7);
        Writer writer = new OutputStreamWriter(clientSocket.getOutputStream());
        Reader reader = new InputStreamReader(clientSocket.getInputStream());
      ) {
      Out.print("Eingabe: ");
      char c = In.read();
      while (c != '.') {
          writer.write(new char[] {ch});
          writer.flush();
          ch = In.read();
      }
      clientSocket.shutdownOutput();
      Out.print("Echo:     ");
      int c;
      while ((c = reader.read()) != -1) {
          Out.print((char) c);
      }
      Out.println();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Reading from console and sending data to server

Closing of ouput but input still open! Required that server sees end of data .

Reading of echo from server.

NETWORKING

Introduction

Sockets

ServerSockets

URL and URLConnection

Summary

URLs

Class `URL` for representing a remote resource

- `URL` created with string representing address of remove resource

```
public final class URL implements java.io.Serializable

  public URL(String spec) throws MalformedURLException
  public URL(String protocol, String host, int port, String file)
              throws MalformedURLException
  public URL(URL context, String spec) throws MalformedURLException
```

- Supports creating InputStream for retrieving remote resource

```
   public final InputStream openStream() throws IOException
```

- Getting content for remote resource

```
   public Object getContent() throws IOException
```

- or creating **`URLConnection`** (see next)

```
   public URLConnection openConnection() throws IOException
```

Example URL: Reading Remote Resource

Example reads remote resource and reads content

```
C:\> java SaveURL "http://www.ssw.uni-inz.ac.at/Teaching/Lectures/PSW2/2012W/index.html"
```

```java
public class SaveURL {

    public static void main(String[] args) {
        try {
            URL url = new URL(args[0]);
            try (
                    OutputStream out = new FileOutputStream(args[1]);
                    InputStream in = url.openStream();
                ) {
                int len;
                byte[] b = new byte[100];
                while ((len = in.read(b)) != -1) {
                    out.write(b, 0, len);
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        } catch (MalformedURLException e) {
            e.printStackTrace();
        }
    }
}
```

InputStream from URL!

URLConnection

URLConnection provides better access control

Example:

- create **URLConnection** by **openConnection** from URL

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

- setting properties for request

```
conn.setConnectTimeout(TIMEOUT);
conn.setDoInput(true);
```

Allows control of requests!

- connecting

```
conn.connect();
```

- reading header information

```
String contentType = conn.getContentType();
Map<String, List<String>> headers = conn.getHeaderFields();
```

- reading data

```
InputStream in = conn.getInputStream();
...
```

NETWORKING

Introduction

Sockets

ServerSockets

URL and URLConnection

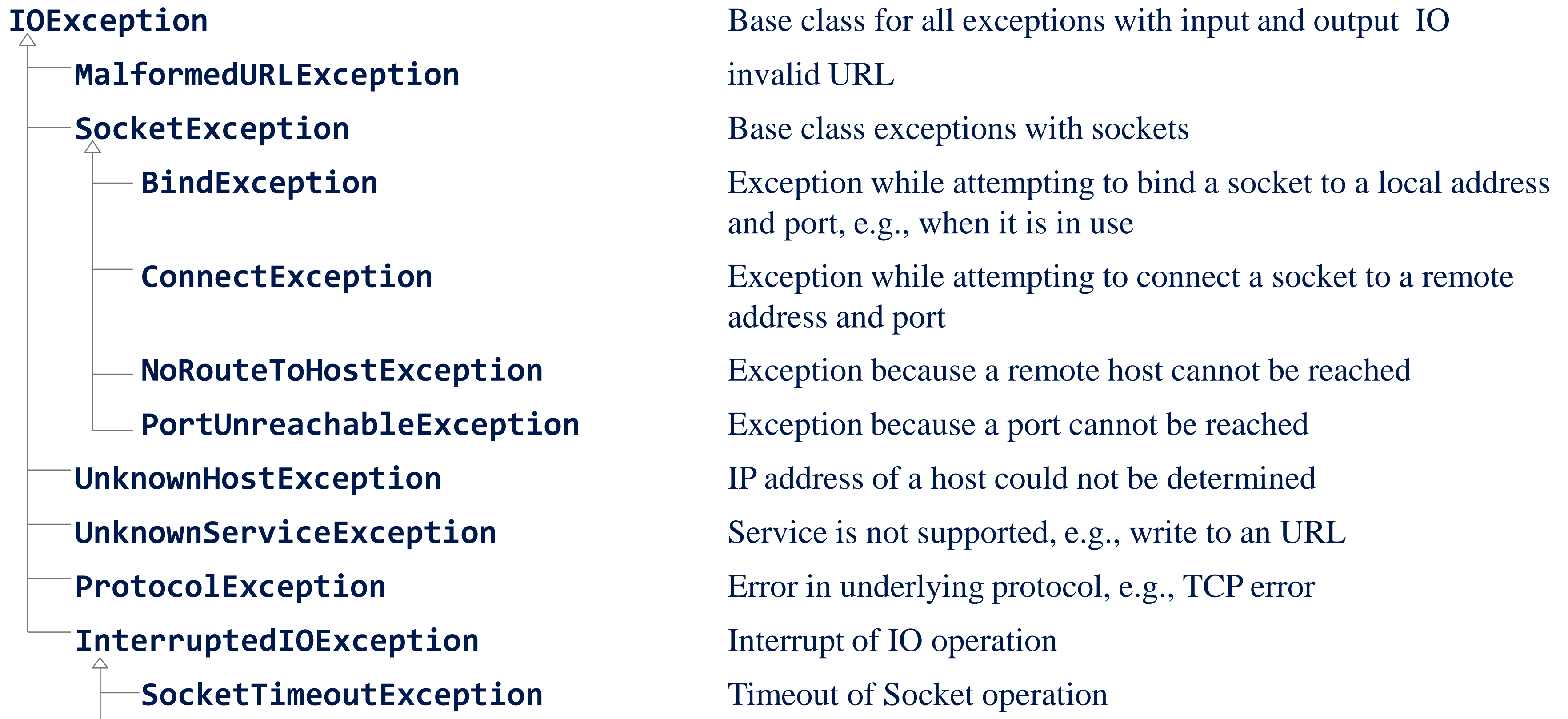
Summary

Exceptions

Network connection is principally unreliable

Thus operations throw a set of exceptions,

Exception hierarchy:



LITERATURE

Horstmann, Cornell, Core Java 2, Band2 Expertenwissen, Markt und Technik, 2002: Kapitel 3

Krüger, Handbuch der Java-Programmierung, 3. Auflage, Addison-Wesley, 2003, http://www.javabuch.de: Kapitel 45