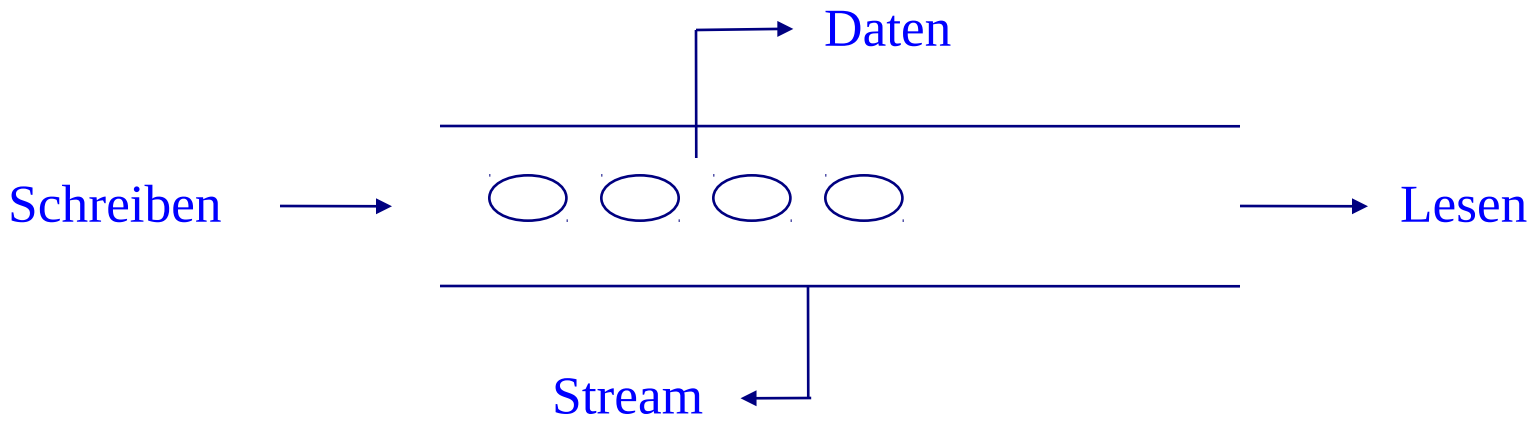
Ein- und Ausgabe in der Programmiersprache Java

Ein- und Ausgabe in Java

- Ein- und Ausgabe werden mit sogenannten Streams realisiert



- Streams sind immer unidirektional
- Das Paket java.io stellt verschiedene Klassen zur Verfügung die auf dem Streamkonzept basieren

Die abstrakte Klasse InputStream

- Die Klasse InputStream beschreibt allgemeine Methoden zum Lesen von Daten
- Methoden der Klasse InputStream
  - read() liest ein Byte vom InputStream
  - available() gibt die Anzahl an Bytes zurück, die gelesen werden können
  - close() schließt den InputStream
  - read(byte[]) liest soviele Bytes aus dem InputStream, wie ein Feld Elemente hat
- Methode read() ist abstrakt, und muss von allen Subklassen implementiert werden
- Standardeingabe von Java ist ein Beispiel eines InputStreams
  - Klassenvariable *in* der Klasse *System* enthält eine Referenz auf eine Implementierung der Klasse InputStream

Standardeingabe

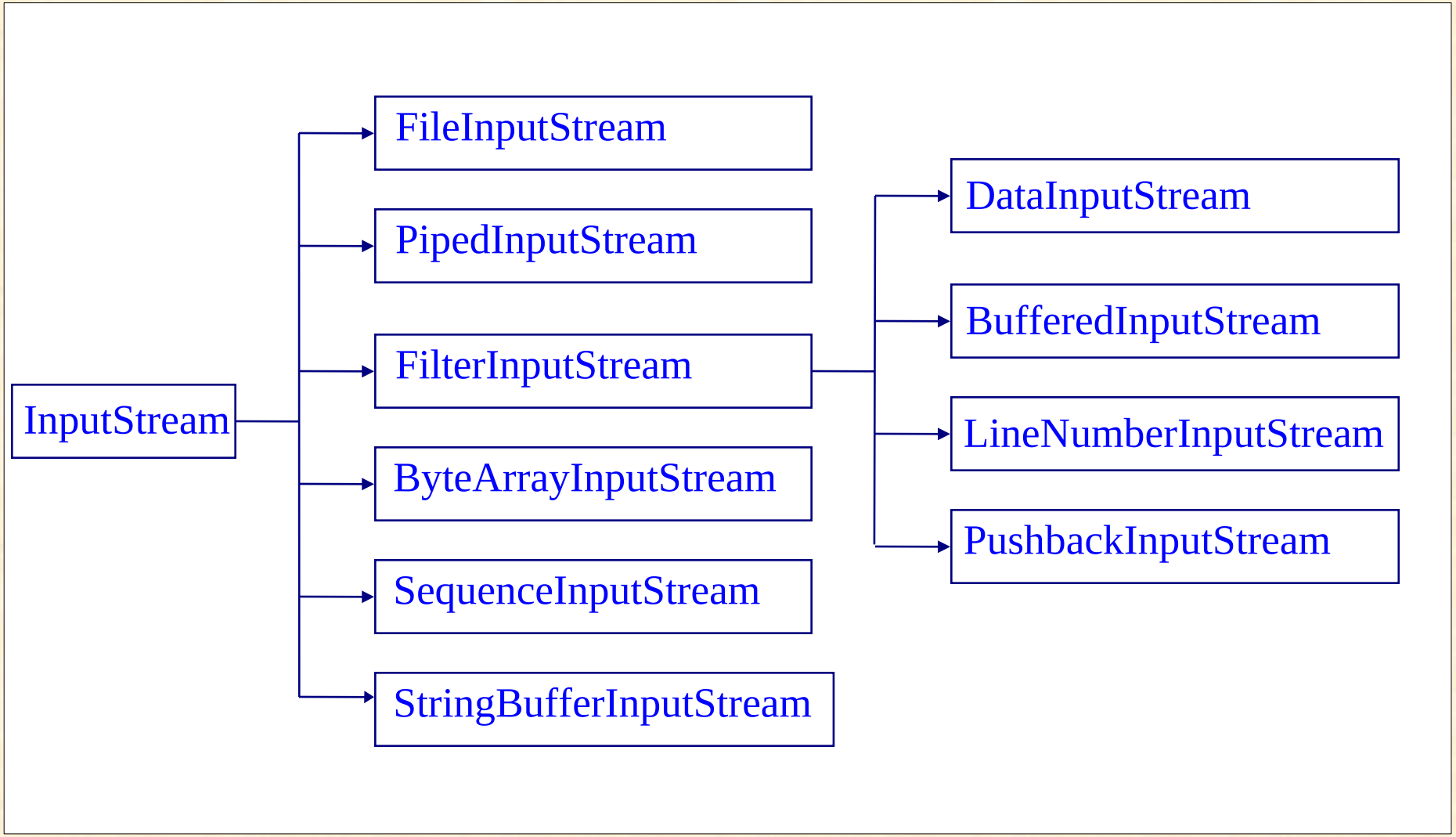
- read() liefert einen Integerwert zurück, obwohl es nur ein Byte an Informationen enthält

```
try {
    int val = System.in.read();
        . . .
} catch ( IOException e ) { }
```

- überladene read()-Methode zum Füllen einer Reihung

```
byte [] bity = new byte [1024];
int got = System.in.read( bity );
```

Unterklassen der Klasse InputStream



Die Klasse DataInputStream

- Mit Hilfe eines DataInputStreams können Werte einfacher Datentypen binär gelesen werden
  - readBoolean();
  - readByte();
  - readChar();
  - readDouble();
  - readFloat();
  - readInt();
  - readLine();
  - readLong();
  - readShort();
- Konstruktor erwartet einen InputStream als Parameter

```
public DataInputStream(InputStream is);
```

Die Klasse BufferedInputStream

- BufferedInputStream ermöglicht Daten zu lesen, ohne bei jeden Aufruf physikalisch auf das Gerät zuzugreifen

- Daten werden blockweise in einen Puffer gelesen, auf dem bei jeder Leseaktion zugegriffen wird

```
BufferedInputStream bis = new BufferedInputStream(mInputStream, 4096);
                                    . . .
bis.read();
```

Beispiel

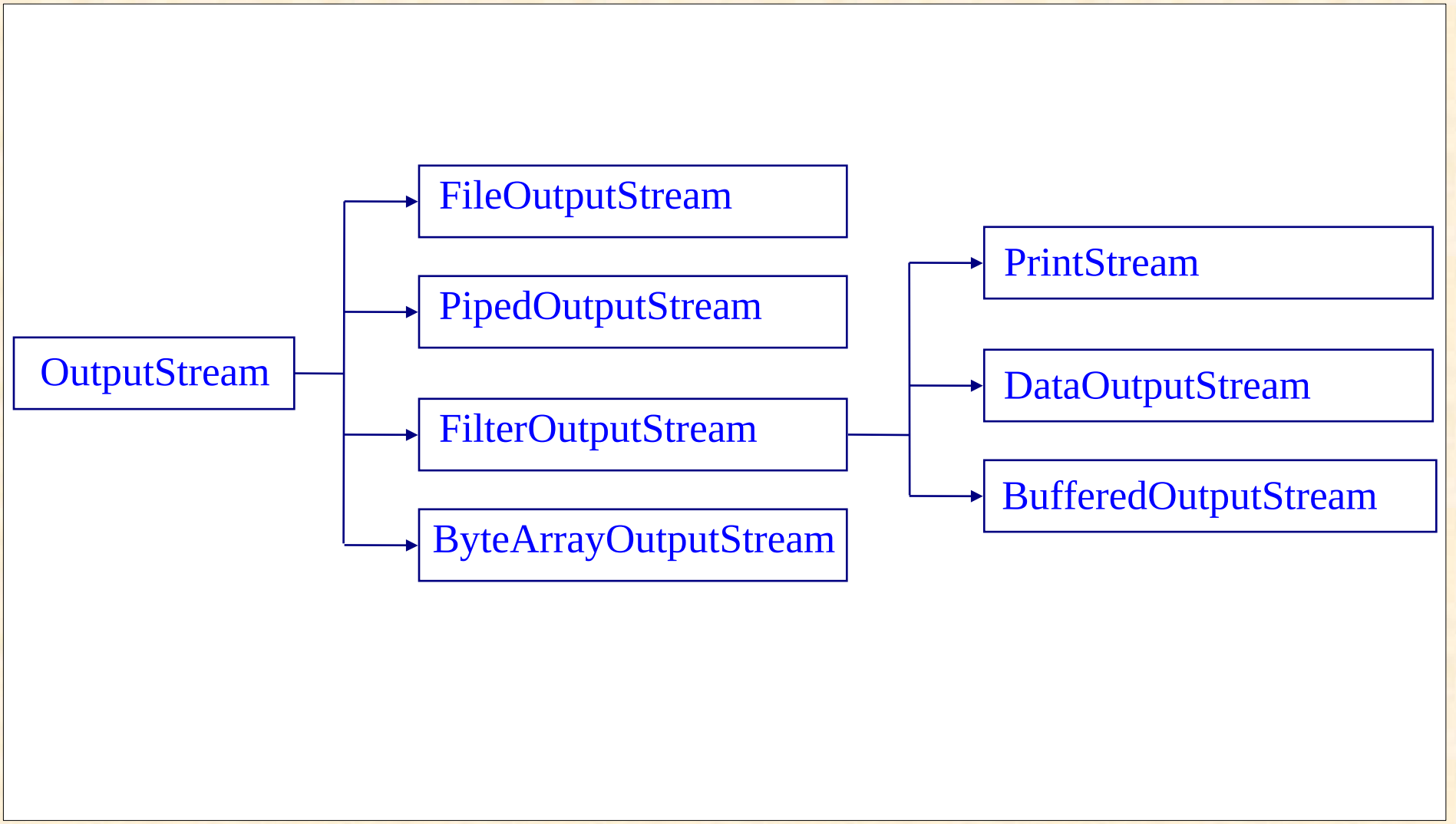
```
import java.io.*;

public class FileIO{
      public static void main(String [] args){
         int value;
         try {
            DataInputStream in = new DataInputStream(
                                       new BufferedInputStream(
                                             new FileInputStream("intData")));
            value = in.readInt();
            System.out.println(value);
         } catch( IOException e){ System.err.println(e); }
     }
}
```

Die abstrakte Klasse OutputStream

- Die Klasse OutputStream beschreibt allgemeine Methoden zum Schreiben von Daten
- Methoden der Klasse:
  - write(int) schreibt ein Byte auf den Stream
  - write(byte[]) schreibt ein Feld von Bytes auf den Stream
  - flush() schreibt gepufferte Daten, falls Puffer genutzt werden
  - close() schließt den OutputStream
- Die Methode write() ist abstrakt und muss deshalb von allen Subklassen implementiert werden
- Standardausgabe von Java ist ein Beispiel eines OutputStreams
  - Klassenvariable *out* der Klasse *System* enthält eine Referenz auf eine Implementierung der Klasse OutputStream

Unterklassen der Klasse OutputStream



Die Klasse PrintStream

- enthält print() bzw. println()-Methode, die Argumente in eine Zeichenkette umwandelt und dann auf dem Stream ausgibt
- Standard- und Standardfehlerausgabe von Java sind Beispiele eines PrintStreams
  - Klassenvariablen *out* und *err* der Klasse System enthalten eine Referenz auf ein Objekt der Klasse PrintStream

```
System.out.print("Hello world . . . \n");
System.out.println("Hello world . . .");
System.out.println("The answer is " + 17);
```

Die Klasse BufferedOutputStream

- Daten werden in einen Puffer geschrieben, dessen Inhalt nur dann in den Stream geschrieben wird, wenn er voll ist

```
BufferedOutputStream bos =
                new BufferedOutputStream(myOutputStream 4096);
                                . . .
                bis.write();
```

- BufferedOutputStream ermöglicht Daten zu schreiben, ohne bei jeden Aufruf physikalisch auf das Gerät zuzugreifen

- explizites Leeren des Puffers mit flush()

Die Klasse DataOutputStream

- Mit Hilfe eines DataOutputStreams können Werte einfacher Datentypen binär geschrieben werden
  - writeBoolean();
  - writeByte();
  - writeChar();
  - writeDouble();
  - writeFloat();
  - writeInt();
  - writeLine();
  - writeLong();
  - writeShort();
- Konstruktor erwartet einen OutputStream als Parameter

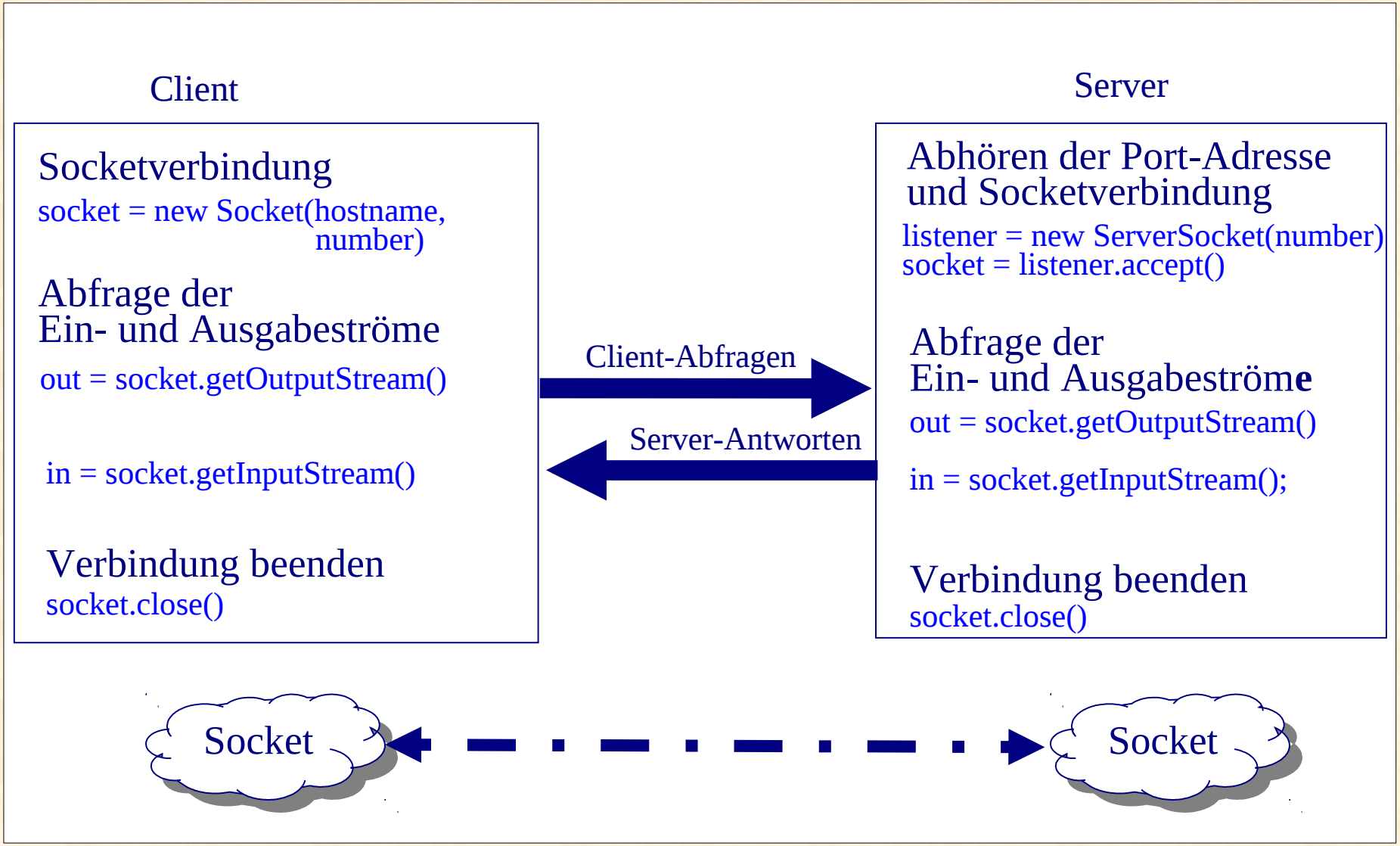
```
public DataOutputStream(OutputStream is);
```

Socket-Programmierung in Java

Sockets und Programmierung

- Sockets beschreiben abstrakte Schnittstellen zwischen Anwendung und Transport- und Netzwerkschicht
  - Datagram-Sockets
  - Stream-Sockets
- Stream-Sockets bieten sicheren und verbindungsorientierten Informationsaustausch
  - basieren auf Transmission Control Protocol (TCP)
  - Kommunikation zwischen Client und Server erfolgt über bidirektionale Datenströme
  - Server: wartet auf Verbindungen von Clients
  - Client: initiiert Verbindung mit Server
- Port-Adressen zur Unterscheidung von Server-Dienstleistungen

Stream-Sockets: Konzeptionelle Arbeitsweise



Stream-Sockets: Port-Adressen

- ganzzahlige 16 Bit-Nummern, die teilweise von der *Internet Assigned Numbers Authority (IANA)* reserviert sind

1 – 1023	wohl bekannte Port-Adressen für Standard-Anwendungen, z.B. 21 für FTP, 80 für Web-Server, etc.
1024 – 49151	registrierte Port-Adressen, welche von der IANA zwar gelistet aber nicht kontrolliert werden z.B. 6000 bis 6063 für XWindow-Server
49152 – 65535	dynamische und private Port-Adressen

Stream-Sockets: Initiieren einer Client-Verbindung

- Anlegen eines Sockets mit Zieladresse

  public Socket(String host, int port)

- nach Verbindung – Abfragen der Datenströme

  socket.getInputStream() – Empfangsdaten
  socket.getOutputStream() – Sendedaten

- Beenden der Verbindung

  socket.close()

Warten auf Verbindungen (Server)

- Anlegen eines Server-Sockets mit Port-Adresse

  public ServerSocket(int port)

- Warten auf Verbindung

  Socket socket = serverSocket.accept()

- nach Verbindung – Datenströme nutzen

  socket.getInputStream() – Empfangsdaten
  socket.getOutputStream() – Sendedaten

- Beenden der Verbindung

  socket.close()

Beispiel: Client

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

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

        try {
            Socket server = new Socket("iaxp16.inf.uni-jena.de", 1234);
            InputStream in = server.getInputStream();
            OutputStream out = server.getOutputStream();

            //  write a byte
            out.write(42);
            //  write a newline or carriage return delimited string
            PrintWriter pout = new PrintWriter(out, true);
            pout.println("Hello!");
```

Kommunikation mit eigenem Rechner
new Socket("localhost", 1234)

Beispiel: Client (Fortsetzung)

```
            ...
            // read a byte
            Byte back = (byte) in.read();

            // send a serialized Java object
            ObjectOutputStream oout = new ObjectOutputStream(out);
            oout.writeObject(new java.util.Date()); oout.flush();

            server.close();

        } catch(UnknownHostException e) {
            System.out.println("Can´t find host.");
        } catch (IOException e) {
            System.out.println("Error connecting to host.");  }
    }
}
```

Beispiel: Server

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

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

        try {

            ServerSocket listener = new ServerSocket(1234);

            while(true) {

                Socket client = listener.accept();    // wait for connection
                InputStream in = client.getInputStream();
                OutputStream out = client.getOutputStream();

                //  read a byte
                byte someByte = (byte) in.read();
```

Beispiel: Server (Fortsetzung)

```
                    ...
                    //  read a newline or carriage return delimited string
                    BufferedReader bin = new BufferedReader(new
                                                    InputStreamReader(in));
                    String someString = bin.readLine();
                    // write a byte
                    out.write(43);
                    // read a serialized Java object
                    ObjectInputStream oin = new ObjectInputStream(in);
                    Date date = (Date) oin.readObject();
                    client.close();
                }
                listener.close();
            } catch (Exception e) {}
        }
}
```

Beispiel: Multiplikationsserver (Server)

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

public class MulServer {
   public static void main(String[] args)throws IOException {
      ServerSocket server = new ServerSocket(3141);
      while (true) {
         Socket client = null;
         try{
            client = server.accept();
            DataInputStream  in  = new DataInputStream(client.getInputStream());
            DataOutputStream out = new DataOutputStream(client.getOutputStream());
            int factor1 = in.readInt();       // 1. Operanden lesen
            int factor2 = in.readInt();       // 2. Operanden lesen
            int result = factor1 * factor2;
            out.writeInt(result);             // Ergebnis dem Client senden
         } catch ( IOException e ) {}      // Fehler bei Ein- und Ausgabe
         finally {
            if ( client != null ) try { client.close(); } catch ( IOException e ) { }
         }
      } } }
```

Beispiel: Multiplikationsserver (Client)

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

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

      Socket server = null;
      try {
            server = new Socket("localhost",  3141);
            DataInputStream  in  = new DataInputStream(server.getInputStream());
            DataOutputStream out = new DataOutputStream(server.getOutputStream());
            out.writeInt(4);                    // sende 1. Operanden
            out.writeInt(10000);                // sende 2. Operanden
            int result = in.readInt();          // lese das Ergebnis
            System.out.println( result );
      } catch ( UnknownHostException e ) {}     // Verbindungsfehler
        catch ( IOException e ) {}              // Fehler bei Ein-und Ausgabe
        finally {
            if ( server != null )
               try { server.close(); } catch ( IOException e ) { }
      }
   }
}
```

Besser: Verwendung von Threads

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

public class MulServerThreadBased {

    public static void main(String[] args) throws IOException {

        ServerSocket server = new ServerSocket(3141);

        while ( true )  { // einzelner Thread bearbeitet eine aufgebaute Verbindung

            MulServerThread mulThread = new MulServerThread(server.accept());
            mulThread.start();

        }

    }

}

...
```

Besser: Verwendung von Threads (Fortsetzung)

```
...

class MulServerThread extends Thread {

    Socket client;
    MulServerThread(Socket client) { this.client = client; }

    public void run(){ // Bearbeitung einer aufgebauten Verbindung
       try {
          DataInputStream  in  = new DataInputStream(client.getInputStream());
          DataOutputStream out = new DataOutputStream(client.getOutputStream());

          int factor1 = in.readInt();
          int factor2 = in.readInt();
          int result = factor1 * factor2;
          out.writeInt(result);
       } catch ( IOException e ) {} // Fehler bei Ein- und Ausgabe
         finally { if ( client != null ) try { client.close(); } catch ( IOException e ) { } }
    }
}
```