

Streams und Files

Datenströme

Byteweises Lesen und Schreiben

Zeichenweises Lesen und Schreiben

Dateien

Serialisierung



- **Datenströme**

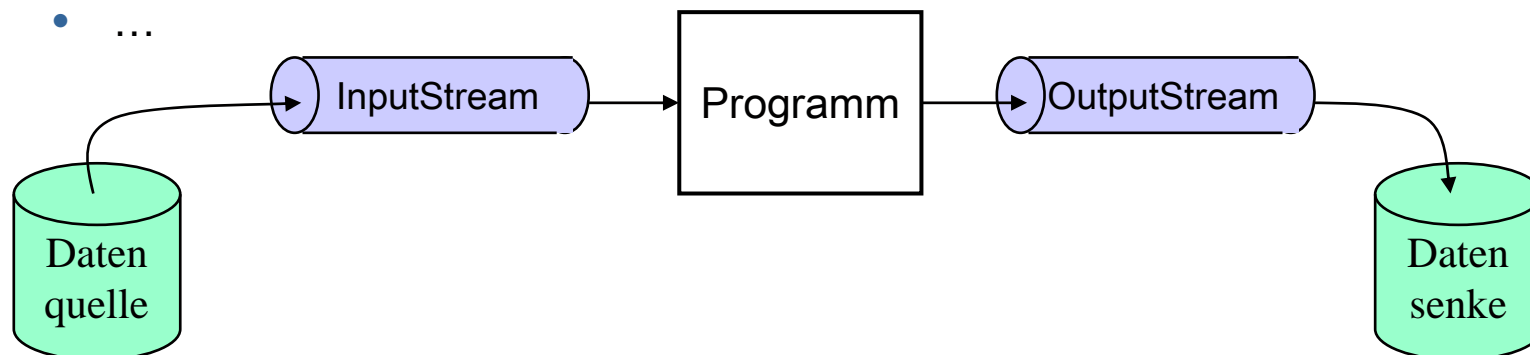
- Byteweises Lesen und Schreiben
- Zeichenweises Lesen und Schreiben

- Dateien
- Serialisierung



Streaming

- Ein/Ausgabe und Fluss von Daten
- Daten werden nicht auf einmal übertragen, sondern fließen in einem kontinuierlichen Prozess
- Datenquellen – Input – Output – Datensenken
 - Eingabestrom verbindet Datenquelle mit Programm
 - Ausgabestrom verbindet Programm mit Datensenke
- Streams abstrahieren von konkreten Quellen und Senken
 - Terminal- oder Konsolenfenster
 - Dateien
 - Client- und Serverprogramme
 - ...



Streaming in Java

- Paket `java.io`
 - Byteorientierte Ein- und Ausgabe
 - Zeichenorientierte Ein- und Ausgabe
- Paket `java.nio`
 - Puffer und Kanäle

- 2 Arten von Streams
 - Byte-Streams: byte-weises Lesen und Schreiben
 - `InputStream`- und `OutputStream`-Klassen
 - Char-Streams: zeichenweises Lesen und Schreiben (Unicode)
 - `Reader`- und `Writer`-Klassen



- Datenströme
 - **Byteweises Lesen und Schreiben**
 - Zeichenweises Lesen und Schreiben
- Dateien
- Serialisierung



Basisklasse InputStream

- abstrakte Basisimplementierung, von der konkrete InputStreams abgeleitet werden
- abstrakte Methode `read()`, die ein Zeichen liest
- konkrete Implementierungen aller anderen Methoden
- Methoden arbeiten synchron, d.h. blockieren, solange bis Operation abgeschlossen
- Methoden werfen `IOExceptions`, wenn etwas schief geht

```
public abstract class InputStream {  
    public abstract int read() throws IOException  
    public int read(byte b[]) throws IOException  
    public int read(byte b[], int off, int len)  
        throws IOException  
    public long skip(long n) throws IOException  
    public int available() throws IOException  
    public void close() throws IOException  
    public synchronized void mark(int readLimit)  
    public synchronized void reset() throws IOException  
    public boolean markSupported()  
}
```



Basisklasse OutputStream

- abstrakte Basisimplementierung, von der konkrete OutputStreams abgeleitet werden
- abstrakte Methode `write()`, die ein einzelnes Zeichen schreibt
- konkrete Implementierungen aller anderen Methoden
- Methoden arbeiten synchron, d.h. blockieren, solange bis Operation abgeschlossen
- Methoden werfen `IOExceptions`, wenn etwas schief geht

```
public abstract class OutputStream {  
    public abstract void write(int b) throws IOException;  
    public void write(byte b[]) throws IOException  
    public void write(byte b[], int off, int len)  
                throws IOException  
    public void flush() throws IOException  
    public void close() throws IOException  
}
```

```
public class IOException extends Exception {  
    public IOException()  
    public IOException(String s)  
}
```



Realisierung von konkreten Stream-Klassen

- Ableiten von `InputStream` bzw. `OutputStream`
- Überschreiben der abstrakten Methoden `read()` bzw. `write()`

Z.B.: `FileInputStream`: Lesen von Dateien

```
public class FileInputStream extends InputStream {  
    public FileInputStream(String name) throws FileNotFoundException  
    public FileInputStream(File file) throws FileNotFoundException  
    public native int read() throws IOException  
    ...  
}
```

native implementiert!

Z.B.: `FileOutputStream`: Schreiben auf Dateien

```
public class FileOutputStream extends OutputStream {  
    public FileOutputStream(String name) throws FileNotFoundException  
    public FileOutputStream(String name, boolean append)  
        throws FileNotFoundException  
    public FileOutputStream(File file)  
        throws FileNotFoundException  
    public native void write(int b) throws IOException  
    ...  
}
```

native implementiert!



Beispiel: Kopieren einer Datei

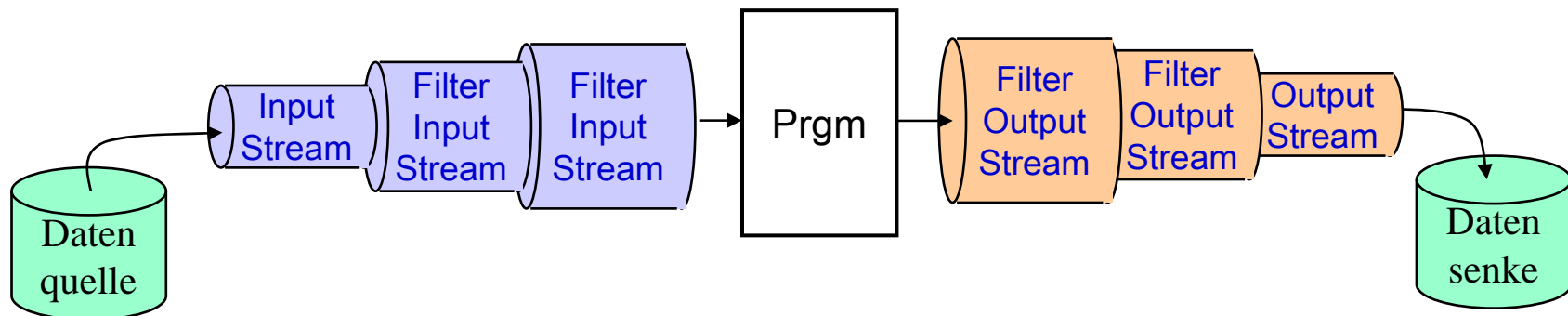
```
static void copyFile(String i file, String o file) {
    InputStream in = null;
    OutputStream out = null;
    try {
        in = new FileInputStream(i file);
        out = new FileOutputStream(o file);
        int b = in.read();
        while (b >= 0) {
            out.write(b);
            b = in.read();
        }
    } catch (FileNotFoundException e) {
        System.out.println("File does not exist");
    } catch (IOException e) {
        System.out.println("Error in reading or writing file");
    } finally {
        if (in != null) {
            try { in.close(); } catch (IOException ioe) {}
        }
        if (out != null) {
            try { out.close(); } catch (IOException ioe) {}
        }
    }
}
```

Man beachte das Abfangen
der Exceptions !!



Filter-Streams: Überlagern von Streams

- FilterStreams sind Streams, die auf einem anderen Stream aufbauen
- Ketten von Streams bilden, um
 - Basisstreams mit weiterer Funktionalität anzureichern
 - Höhere Schnittstellen zur Verfügung zu stellen



```
InputStream fin = new FileInputStream("input.txt");  
BufferedInputStream bin = new BufferedInputStream(fin);  
DataInputStream din = new DataInputStream(bin);
```

```
OutputStream fout = new FileOutputStream("output.txt");  
BufferedOutputStream bout = new BufferedOutputStream(fout);  
DataOutputStream dout = new DataOutputStream(bout);
```



FilterStreams: BufferedInputStream - BufferedOutputStream

- Gepufferte Eingabe: BufferedInputStream
 - Effizienz durch Vorauslesen

```
File iFile = new File("inputfile.txt");  
InputStream in = new BufferedInputStream(  
    new FileInputStream(iFile));
```

- Gepufferte Ausgabe: BufferedOutputStream
 - Effizienz durch verzögertes Schreiben
 - Tatsächliche Ausgabe erst wenn Puffer voll
 - Methode flush erzwingt Ausgabe des Puffers

```
File oFile = new File("outputfile.txt");  
OutputStream out = new BufferedOutputStream(  
    new FileOutputStream(oFile));
```



FilterStreams: DataInputStream - DataOutputStream

- Ausgabe primitiver Datentypen

```
public class DataOutputStream
    extends FilterOutputStream {
    void writeBoolean(boolean v);
    void writeByte(int v);
    void writeBytes(String s);
    void writeChar(int v);
    void writeChars(String s);
    void writeDouble(double v);
    void writeFloat(float v);
    void writeInt(int v);
    void writeLong(long v);
    void writeShort(int v);
    void writeUTF(String str);
    ...
}
```

```
public class DataInputStream
    extends FilterInputStream {
    boolean readBoolean();
    byte readByte();
    char readChar();
    double readDouble();
    float readFloat();
    int readInt();
    long readLong();
    short readShort();
    int readUnsignedByte();
    int readUnsignedShort();
    String readUTF();
    ...
}
```

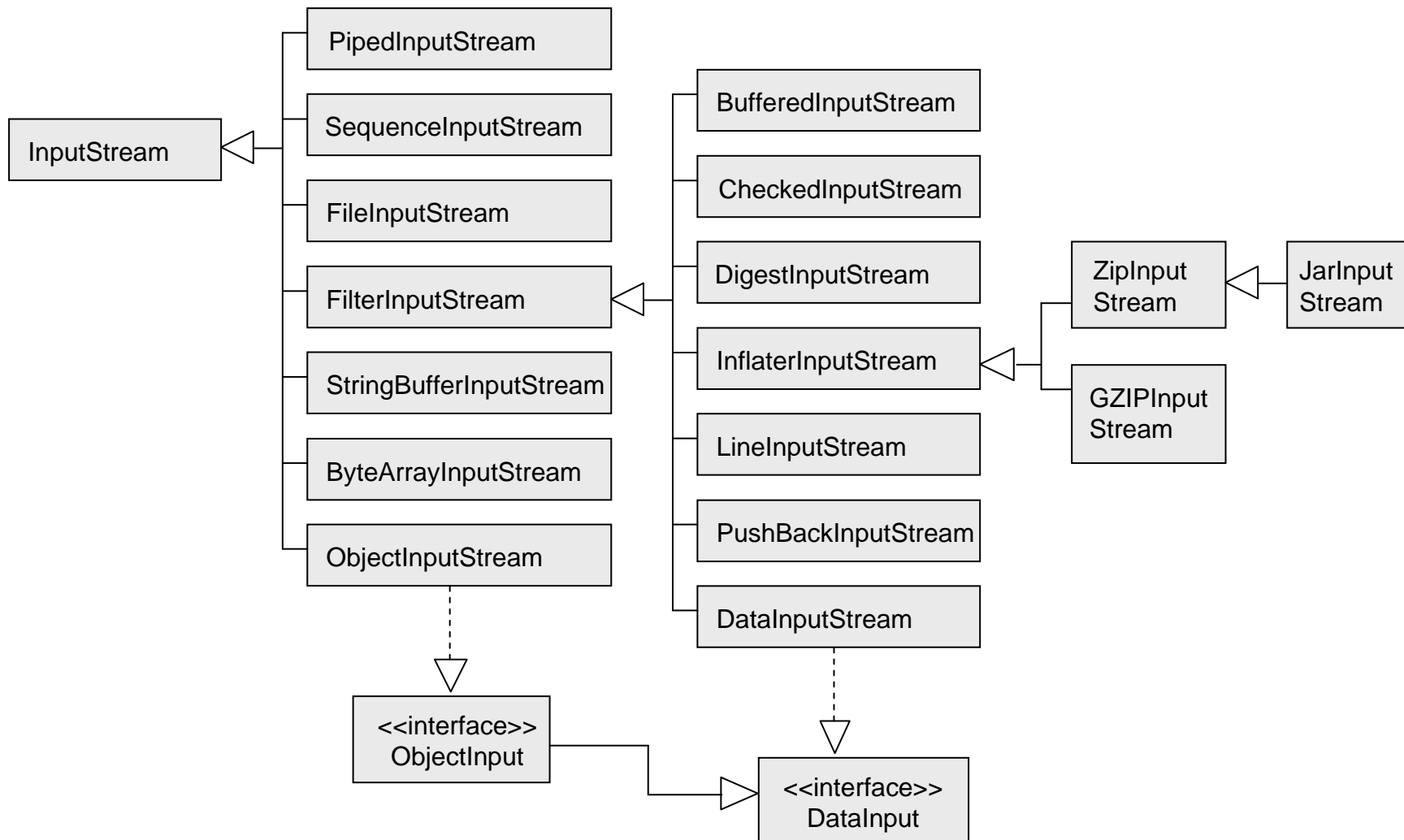
```
File fout = new File("output.data");
DataOutputStream out = new DataOutputStream(
    new FileOutputStream(fout));
out.writeDouble(Math.PI);
out.close();
```

output:

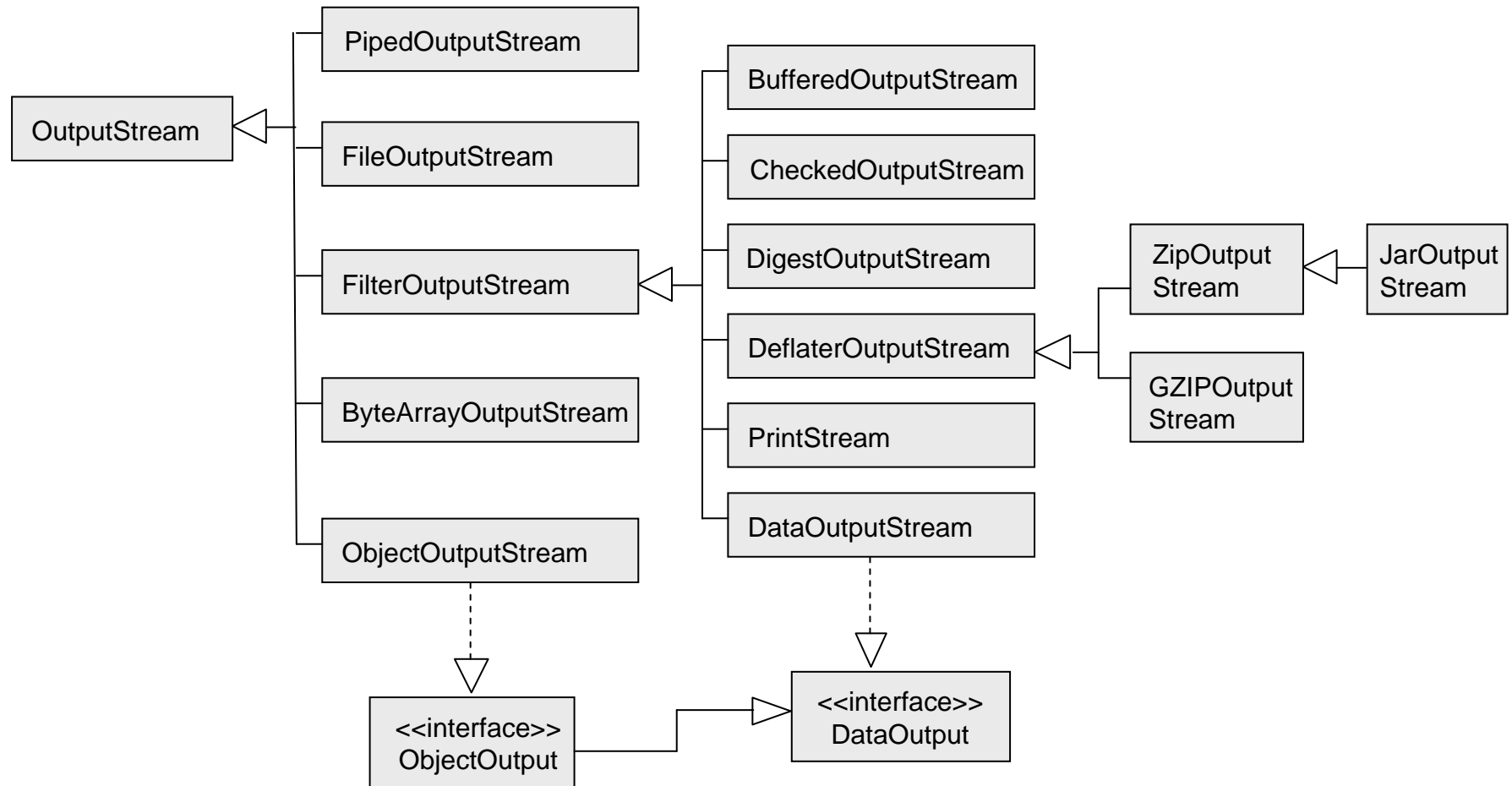
```
0x40 0x09 0x21 0xfb ; @. ! û
0x54 0x44 0x2d 0x18 ; TD-.
```



InputStream Klassenhierarchie



OutputStream Klassenhierarchie



Diverse Streamklassen

```
public class PipedOutputStream
    extends OutputStream {
    void connect(PipedInputStream snk);
    ...
}
```

```
public class PipedInputStream
    extends InputStream {
    void connect(PipedOutputStream src);
    ...
}
```

```
public class PushbackInputStream
    extends InputStream {
    int read();
    void unread(int b);
    ...
}
```

```
public class PrintStream
    extends FilterOutputStream {
    public void print(boolean b)
    public void print(int b)
    ...
    public void println(boolean b)
    ...
}
```

- Realisierung von Pipes

- unterstützt Zurückschreiben (unread)

- formatierte Ausgabe von Daten
- mittels print und println-Methoden



Standardstreams

- Streams in der Klasse System

```
static InputStream in; // the standard input stream
static PrintStream out; // the standard output stream
static PrintStream err; // the standard error stream
```

- Filtern der Standardstreams

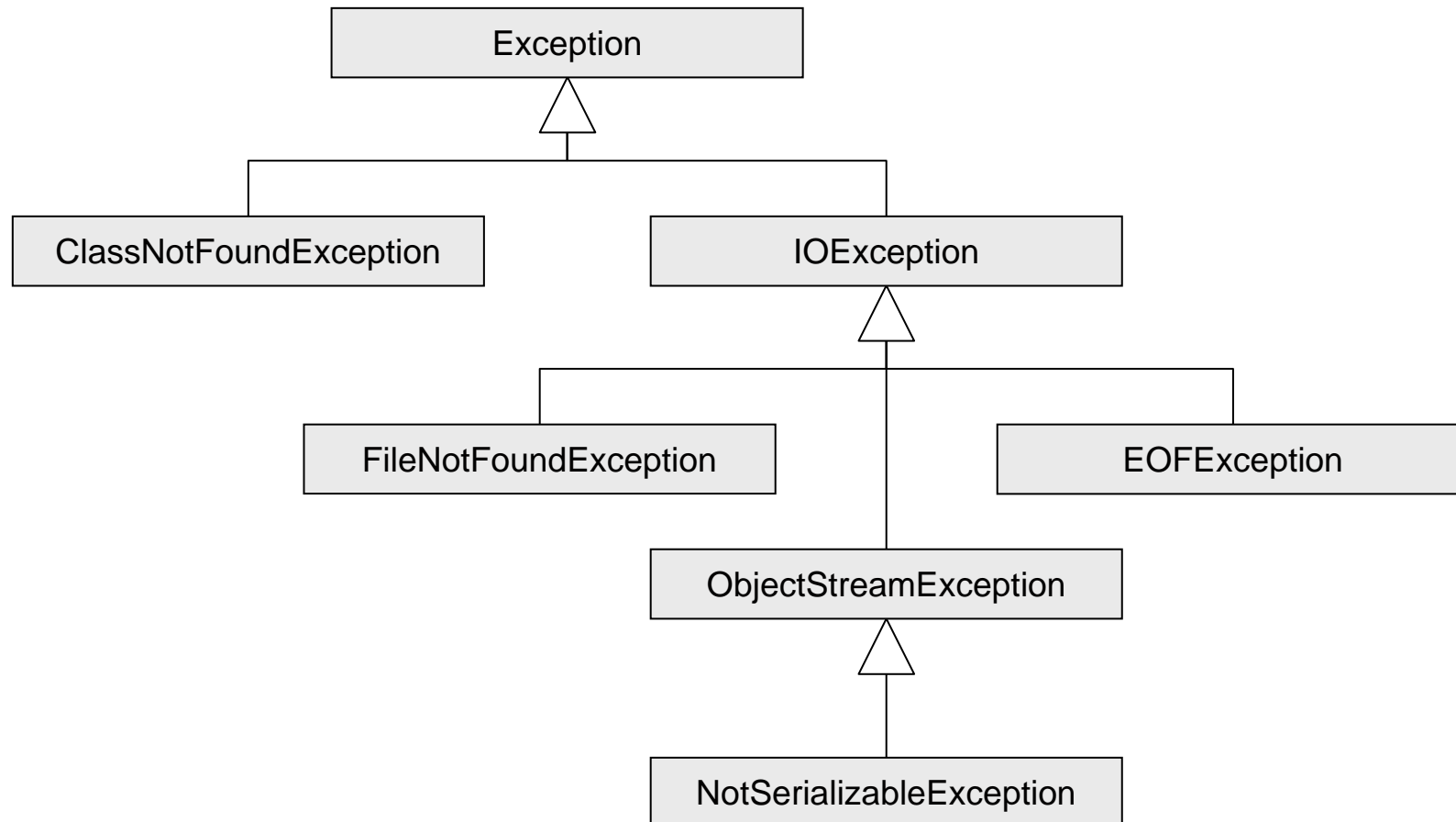
```
BufferedReader input =
    new BufferedReader(
        new InputStreamReader(System.in));
String line = input.readLine();
```

- Umleiten der Standardstreams

```
java Foo < input.txt > output.txt 2> error.txt
                        oder
java Foo > output.txt 2>&1
                        oder
java Foo >> output.txt
```



Hierarchie der Ausnahmen



- Datenströme
 - Byteweises Lesen und Schreiben
 - **Zeichenweises Lesen und Schreiben**
- Dateien
- Serialisierung



Zeichenorientierte Ein- und Ausgabe

- Ein- und Ausgabe von 16-Bit-Unicode-Zeichen über Readers und Writers
- Basisklassen
 - Reader (abstrakte Basisklasse analog zu InputStream)
 - Writer (abstrakte Basisklasse analog zu OutputStream)
- Dateien sind byteorientiert
 - Abbildung von Zeichen auf Bytes erforderlich
 - ByteToCharConverter und CharToByteConverter
 - FileReader und FileWriter verwenden Standardcodierung
 - Andere Codierungen mittels InputStreamReader und OutputStreamWriter

```
Writer out = new OutputStreamWriter(  
    new FileOutputStream("fname"),  
    "UTF8"           // encoding parameter  
);
```



Basisklasse Reader

- abstrakte Basisimplementierung, von der konkrete Reader abgeleitet werden
- abstrakte Methode

```
read(char[] cbuf, int off, int len)
```

welche einen char-Array schreibt

- anderen Methoden bauen auf dieser auf

```
public abstract class Reader {  
    public int read() throws IOException  
    public int read(char[] cbuf) throws IOException  
    public abstract int read(char[] cbuf, int off, int len)  
        throws IOException  
    public long skip(long n) throws IOException  
    public boolean ready() throws IOException  
    public synchronized void mark(int readLimit)  
    public synchronized void reset() throws IOException  
    public boolean markSupported()  
    public abstract void close() throws IOException  
}
```



Basisklasse Writer

- abstrakte Basisimplementierung, von der konkrete Writer abgeleitet werden
- abstrakte Methode

```
write(char[] cbuf, int off, int len)
```

welche einen char-Array füllt

- anderen Methoden bauen auf dieser auf

```
public abstract class Writer {  
    public void write(int c) throws IOException  
    public void write(char[] cbuf) throws IOException  
    public abstract void write(char[] cbuf, int off, int len)  
        throws IOException  
    public void write(String str) throws IOException  
    public void write(String str, int off, int len)  
        throws IOException  
    public abstract void flush() throws IOException  
    public abstract void close() throws IOException  
}
```



Beispiel: FilterReader - FilterWriter

- Filtern der Standardstreams

```
String line;
BufferedReader input = new BufferedReader(
    new InputStreamReader(System.in));

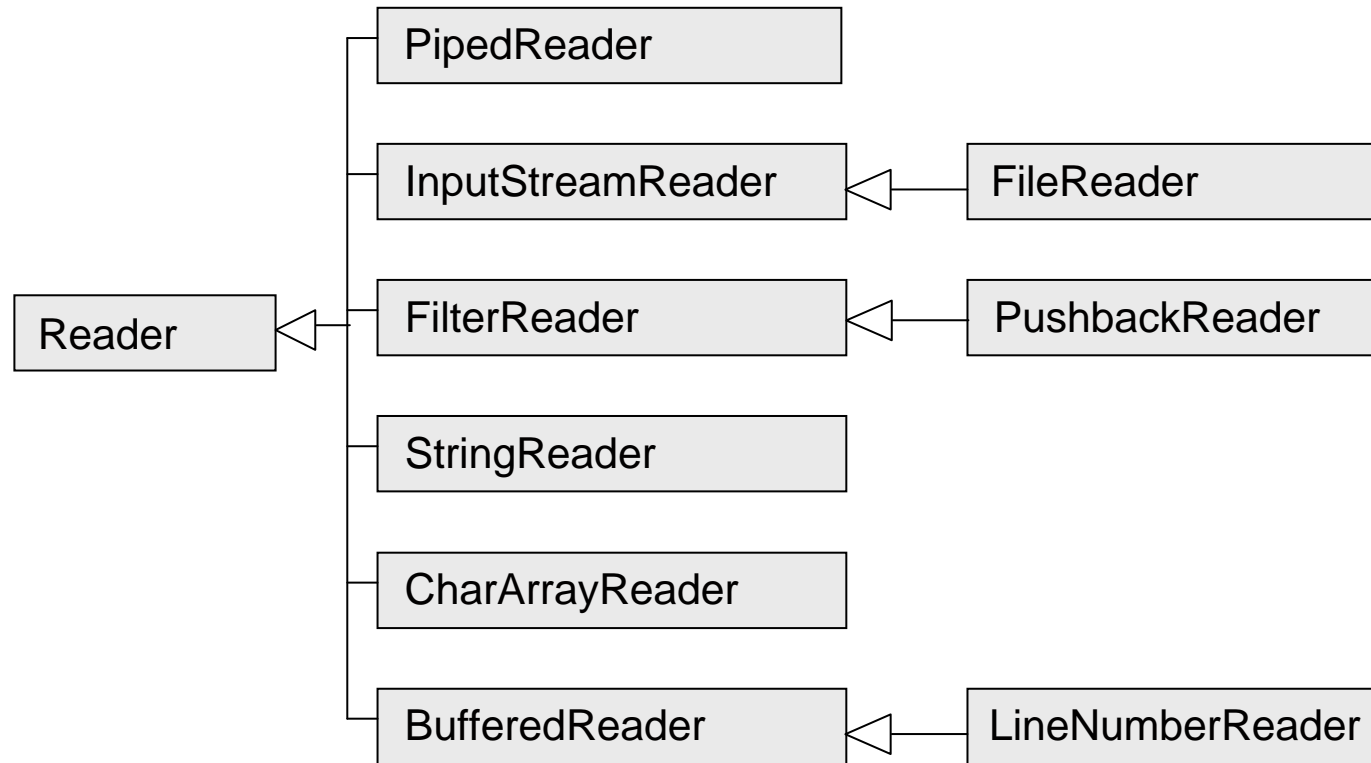
try {
    line = input.readLine();
} catch (IOException ioe) {
    ioe.printStackTrace();
}
```

```
BufferedWriter output = new BufferedWriter(
    new OutputStreamWriter(System.out));

try {
    output.write(line);
    output.newLine();
} catch (IOException ioe) {
    ioe.printStackTrace();
}
```



Reader Klassenhierarchie



Diverse Reader

```
public class InputStreamReader extends Reader {  
    public InputStreamReader(InputStream in)  
        ...  
}
```

- Brücke zu `InputStream`

```
public class LineNumberReader extends Reader {  
    int getLineNumber();  
    ...  
}
```

- Zugriff auf Zeilennummern

```
public class BufferedReader extends Reader {  
    public String readLine() throws IOException  
        ...  
}
```

- gepufferte Eingabe
- unterstützt zeilenweises Lesen (`readLine`)

```
public class PushbackReader extends Reader {  
    public void unread(char[] cbuf,  
        int off, int len) throws IOException  
        ...  
}
```

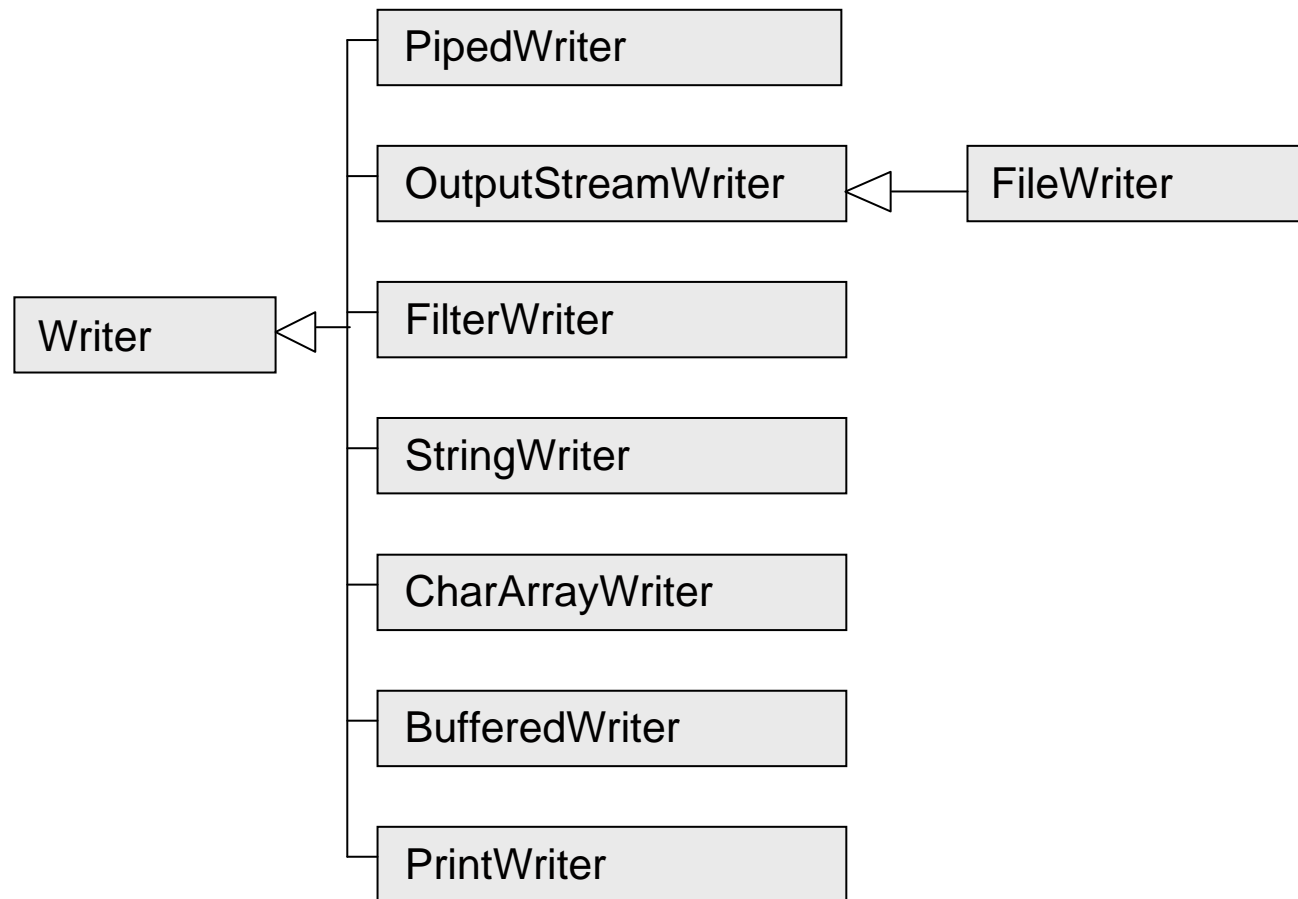
- unterstützt Zurückschreiben (`unread`)

```
public class PipedReader extends Reader {  
    public void connect(PipedWriter src)  
        throws IOException  
        ...  
}
```

- Realisierung von Pipes
- in Kombination mit `PipedWriter`



Writer Klassenhierarchie



Diverse Writer

```
public class OutputStreamWriter extends Writer {
    public OutputStreamWriter (OutputStream out)
    ...
}
```

- Brücke zu OutputStream

```
public class PrintWriter extends Writer {
    public void print(boolean b)
    public void print(int b)
    ...
    public void println(boolean b)
    ...
}
```

- formatierte Ausgabe von Daten als Text
- mittels `print` und `println`-Methoden

```
public class BufferedWriter extends Writer {
    public void newLine() throws IOException
    public void flush() throws IOException
    ...
}
```

- gepufferte Ausgabe
- unterstützt neue Zeile (`newLine`)

```
public class PipedWriter extends Writer {
    public void connect(PipedReader sink)
        throws IOException
    ...
}
```

- Realisierung von Pipes
- in Kombination mit `PipedReader`



- Datenströme
 - Byteweises Lesen und Schreiben
 - Zeichenweises Lesen und Schreiben
- **Dateien**
- Serialisierung



File

- Abstrakte Repräsentation einer Datei oder eines Verzeichnisses
 - Konstruktor greift noch nicht auf Dateisystem zu
- Definiert plattformspezifische Trennzeichen
 - separator und pathSeparator
- Erzeugen von Dateiströmen
 - Mittels Dateinamen oder File-Objekt

```
public class File {  
    boolean canRead();  
    boolean canWrite();  
    boolean delete();  
    boolean exists();  
    String getName();  
    boolean isDirectory();  
    boolean isFile();  
    long lastModified();  
    long length();  
    String[] list();  
    File[] listFiles();  
    ...  
}
```

```
static long totalSize(File dir) {  
    if (!dir.isDirectory()) {  
        return -1;  
    }  
    long size = 0;  
    File[] files = dir.listFiles();  
    for (int i = 0; i < files.length; i++) {  
        size += files[i].length();  
    }  
    return size;  
}
```



- Datenströme
 - Byteweises Lesen und Schreiben
 - Zeichenweises Lesen und Schreiben
- Dateien
- **Serialisierung**



Serialisierung von Objekten

- Speichern des Zustands von Objekten
 - Plattformunabhängige Transformation in Bytes
 - Objektpersistenz, Kommunikation von Programmen
- Serialisierung
 - Klasse muss `Serializable` implementieren
 - Keine Serialisierung statischer oder transienter Felder
- Transitive Hülle
 - Referenzen auf andere Objekte werden ebenfalls serialisiert

```
public class ObjectOutputStream {  
    void writeBoolean(boolean data);  
    void writeByte(int data);  
    ...  
    void writeObject(Object obj);  
    ...  
}
```

```
ObjectOutputStream out =  
    new ObjectOutputStream(  
        new FileOutputStream("data"));  
  
Person p = ...;  
out.writeObject(p);
```



Serializabl e

- Nur Objekte die Serializabl e implementieren, können serialisiert werden
- Damit werden alle Felder, die serialisierbar sind, automatisch mit dem Objekt serialisiert
 - Felder mit Basisdatentypen
 - Felder mit Typen, die serialisierbar sind
- Kennzeichnet man Felder mit `transient`, werden diese nicht serialisiert !!
 - ➔ `transient`: Verhinderung der Serialisierung

```
public interface Serializabl e {  
}
```

```
public class Person implements Serializabl e {  
  
    private String name;  
    private int age;  
    private List<Person> children;  
  
    private transient List<PropertyChangeListener> listeners;  
    ...  
}
```



Serialisierbare / Nicht-serialisierbare Datentypen

Serialisierbar:

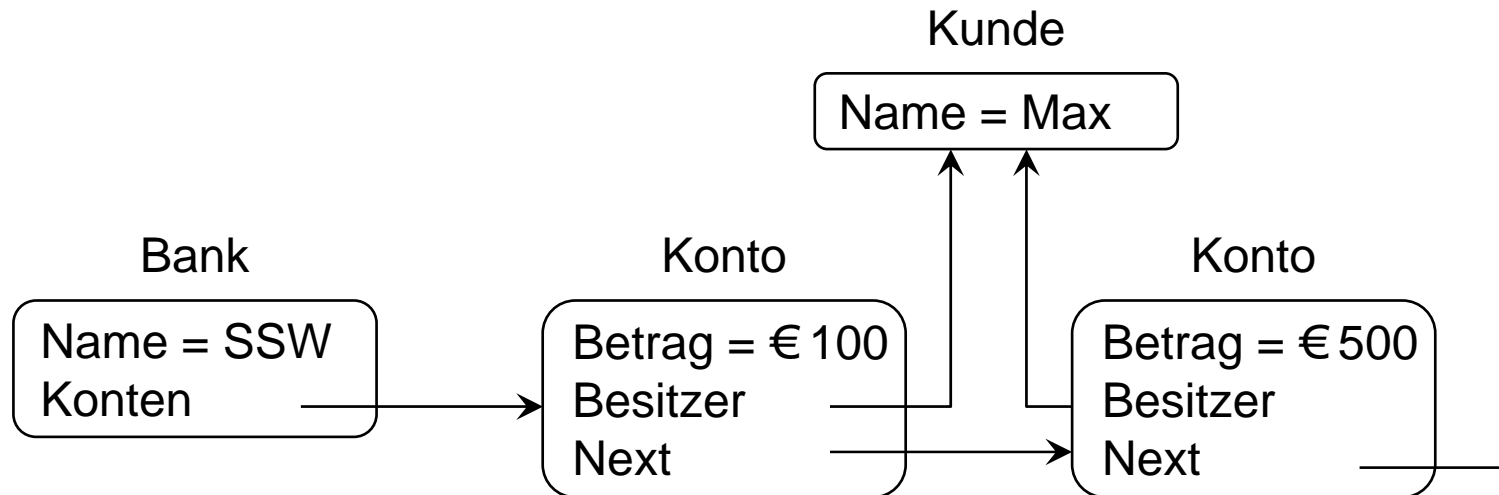
- Basisdatentypen (i nt, bool ean, doubl e, ...)
- Alle Arrays
- String
- Collections: ArrayLi st, Li nkedLi st, TreeSet, ...
- Date, Ti me, Cal endar, ...
- Alle Typen die Seri al i zabl e implementieren

Nicht serialisierbar:

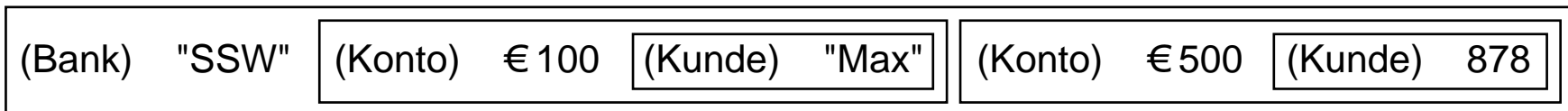
- Component, JComponent, etc → GUI Komponenten
- Thread, Cl assLoader, etc. → Objekte der Java-VM
- InputStream, Socket, Fi l e etc. → Objekte, die auf Betriebssystemressourcen zugreifen



Serialisierung von Objektgraphen



- Mehrere Referenzen auf das selbe Objekt:
 - 1. Besuch: Objekt vollständig schreiben
 - weiterer Besuch: nur ID des Objekts schreiben



Deserialisierung

- Einlesen eines Objekts
 - Allokation von Speicher und Initialisierung mit null
 - ClassNotFoundException falls Klasse nicht vorhanden
 - Einlesen des Objekts
 - Statische und transiente Felder werden ignoriert
- Erzeugt Kopie der transitiven Hülle
 - Identität mehrfach referenzierter Objekte bleibt erhalten
 - Aufruf des Standardkonstruktors für nicht serialisierbare Objekte

```
public class ObjectInputStream {  
    boolean readBoolean();  
    byte    readByte();  
    ...  
    Object  readObject();  
    ...  
}
```

```
ObjectInputStream in =  
    new ObjectInputStream(  
        new FileInputStream("data"));  
Person p = (Person) in.readObject();
```



Benutzerdefinierte Serialisierung

- Implementierung der optionalen Methoden
 - Serialisierung der Superklasse geschieht automatisch

```
private void writeObject(ObjectOutputStream stream)
    throws IOException;
private void readObject(ObjectInputStream stream)
    throws IOException, ClassNotFoundException;
```

- Ersetzung durch ein anderes Objekt
 - z.B. Ersetzung eines Remote-Objekts durch Stub

```
ANY-ACCESS-MODIFIER Object writeReplace()
    throws ObjectStreamException;
ANY-ACCESS-MODIFIER Object readResolve()
    throws ObjectStreamException;
```

- Implementierung des Interfaces Externalizable
 - Nur Identität der Klasse wird automatisch gespeichert

```
public void writeExternal(ObjectOutput out)
    throws IOException;
public void readExternal(ObjectInput in)
    throws IOException, ClassNotFoundException;
```



Zusammenfassung

- Datenströme
 - Verbinden Datenquelle mit Programm oder Programm mit Datensenke
- Byteorientierte Datenströme
 - InputStream und OutputStream
 - FileInputStream und FileOutputStream
- Zeichenorientierte Datenströme
 - Reader und Writer
- Gepufferte Datenströme
 - BufferedInputStream und BufferedOutputStream
 - BufferedReader und BufferedWriter
- Serialisierung
 - Plattformunabhängige Transformation des Objektgraphen in Bytes
 - Siehe auch RMI

