Einführung in die Informatik II
Ströme und Dateien

Prof. Bernd Brügge, Ph.D
Institut für Informatik
Technische Universität München

Sommersemester 2001

25. Juni 2001

Ziele dieser Vorlesung

- Sie verstehen das Konzept von Strömen
- Sie können die Java-Klassen **`Reader`**, **`Writer`**, **`InputStream`** und **`OutputStream`** benutzen
- Sie verstehen Java's Konzept für die plattformunabhängige Repräsentation von Dateien in hierarchischen Datei-Systemen.
- Sie verstehen das Konzept der Serialisierung/Deserialisierung von Objekten.
- Sie können Dateien entwerfen und implementieren
  - Textdateien
  - Binäre Dateien
  - Dateien von beliebigen Objekten

Einleitung

- In den Informatik-Systemen, die wir bisher implementiert haben, haben wir Klasssen *deklariert*, *instantiiert*, und über Methodenaufrufe den Attributen *Werte zugewiesen*. Die Werte waren allerdings nur während der Laufzeit des Systems verfügbar.
- 1. Häufig sollen die Werte bei einem erneuten Lauf des Systems wieder zur Verfügung stehen.
- 2. In vielen Fällen werden auch Werte von einem Informatik-System erzeugt, die von einem anderen System benötigt werden.
- 3. Bei interaktiven Systemen wollen wir außerdem bereits während der Laufzeit ***Daten mit der Umgebung austauschen***:
  - Wir wollen Werte über eine Tastatur eingeben lassen, andere Werte sollen auf dem Bildschirm erscheinen, Verzeichnisse sollen gelesen und beschrieben werden.
- Zur Speicherung von Werten und zur Interaktion mit der Umgebung führen wir jetzt die Konzepte Strom und Datei ein.

Datei

- Wir haben bereits den Datenaustausch mit der Umgebung des Systems zugelassen, allerdings sehr spärlich.
  - **`System.out.println()`**: Ausdrucken von Daten auf dem Bildschirm
  - **`getText()`**: Auslesen eines Textes aus einer graphischen Komponente vom Typ **`TextField`**.
- Unser Ziel ist jetzt die Modellierung von externen Speichern und der Interaktion von Informatik-Systemen mit ihrer Umgebung.
- **Definition Datei:** Eine Verwaltungseinheit zur Repräsentation von externen Daten nach gewissen Organisationsformen, die den Zugriff innerhalb des Informatik-Systems auf die Daten festlegen.

Copyright 2001 Bernd Brügge

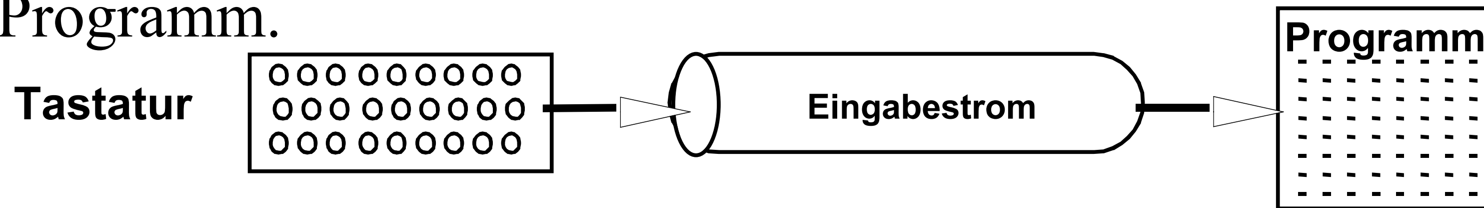
Strom

- Um auf Dateien innerhalb eines Informatik-Systems zugreifen zu können, führen wir den Begriff des Stroms ein.
  - **Definition Strom(*):** Die interne Repräsentation einer (externen) Datei oder eines Ein-/Ausgabe (E/A)-Gerätes in einem Informatik-System.
  - **Definition Eingabe:** Das Lesen der Daten von einer Datei oder von einem Eingabe-Gerät in einen Strom. Der Strom heißt dann *Eingabestrom*.
  - **Definition Ausgabe:** Das Schreiben von Daten eines Stroms auf eine Datei oder ein Ausgabe-Gerät. Der Strom heißt dann *Ausgabestrom*.
  - Ein Strom hat eine **Quelle** und eine **Senke**.

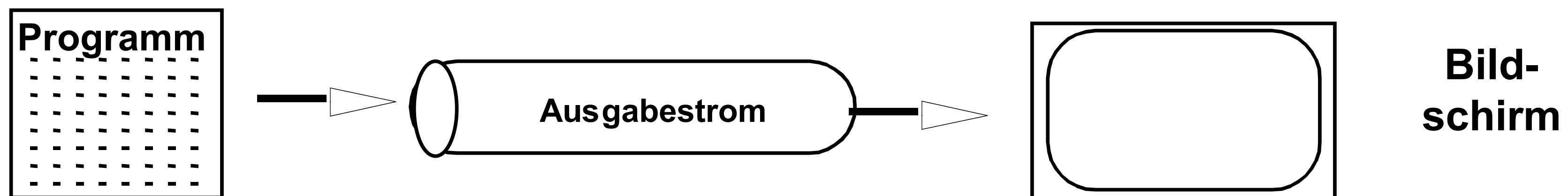
(*) Achtung: In Goos II hat der Begriff Strom eine andere Bedeutung: Bei Goos ist ein Strom ein sog. *Iterator* über Daten.

Beispiele von Eingabe- und Ausgabeströmen

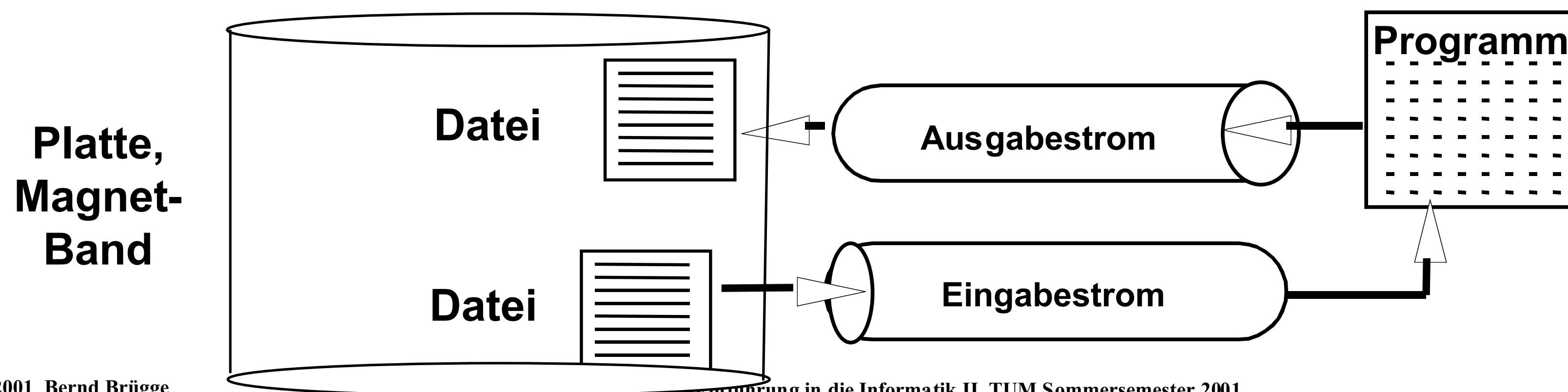
- Eine *Tastatur* ist eine Quelle für einen Eingabestrom von Zeichen an ein Programm.



- Ein *Bildschirm* ist ein Senke für einen Ausgabestrom von Zeichen, die von einem Programm kommen.



- Eine *Datei* ist Senke oder Quelle für Ströme von Zeichen



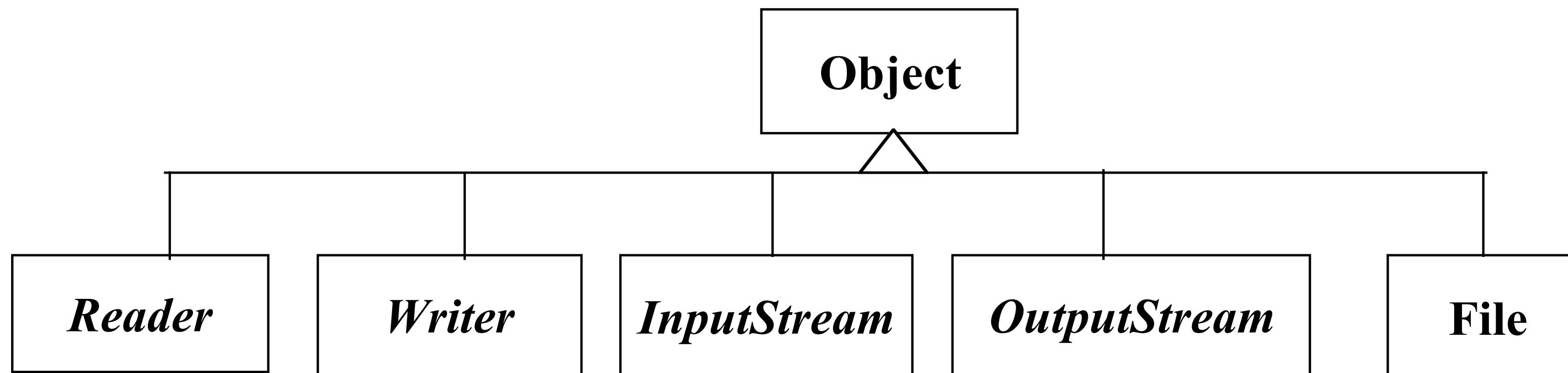
Copyright 2001 Bernd Brügge

Entwurf von Strömen

- Wir modellieren Ströme als Klassen. Allgemein können wir sagen:
- Ein Strom hat private Attribute
  - Name: Bezeichner der mit dem Strom verbundenen Quelle oder Senke
  - **`Marke`**: Zeiger auf das derzeitige Element (current element).
- Ein Strom stellt folgende Dienste bereit:
  - **`open()`**: Öffnen der Verbindung mit einer (externen) Datei
  - **`read()`**: Lesen eines Elements aus dem Strom.
  - **`write()`**: Schreiben eines Elements in den Strom.
  - **`close()`**: Schließen der Verbindung mit der Datei.
- Die Signatur der Dienste und die Implementierung von Strömen ist abhängig von der Programmiersprache; die Implementierung von Dateien ist außerdem oft vom Betriebssystem abhängig.
- Java unterstützt Ströme und Dateien.

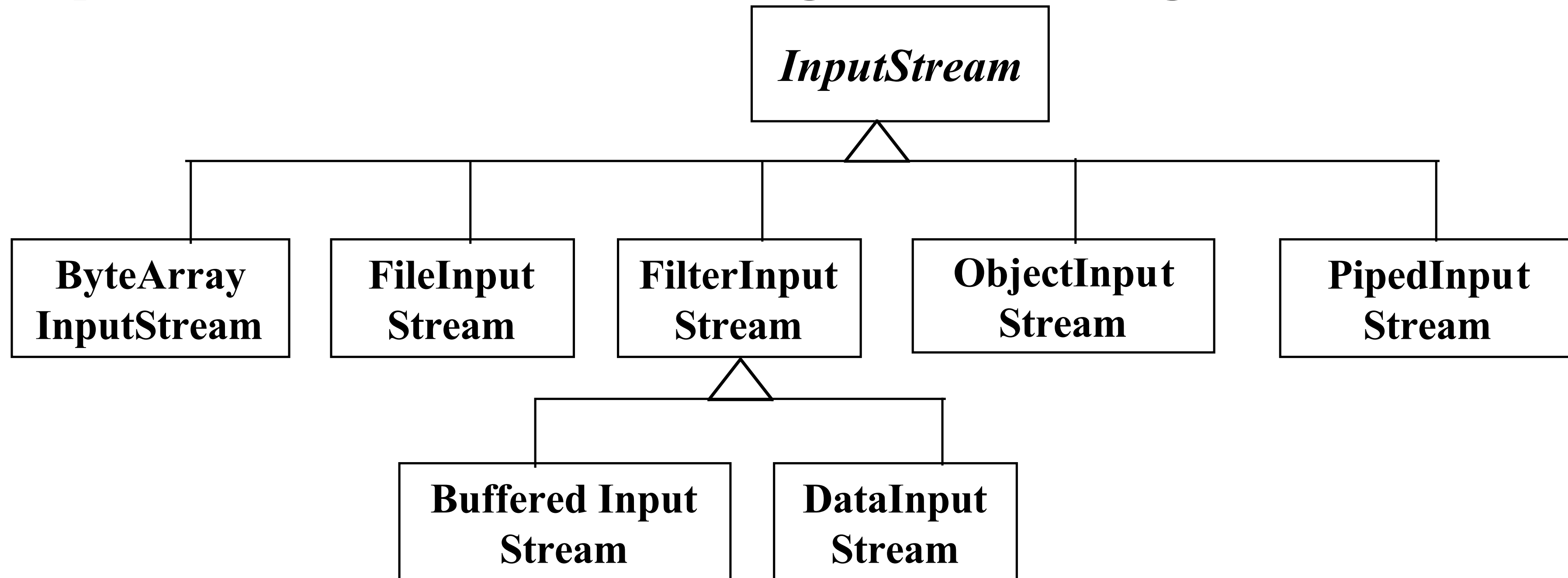
Ströme und Dateien in Java

- Java stellt eine große Anzahl von unterschiedlichen Strömen für Ein- und Ausgabe (**`Reader`**, **`Writer`**, **`InputStream`**, **`OutputStream`**) und eine Betriebssystem-unabhängige Repräsentation für Dateien (**`File`**) bereit, die alle im Paket **`java.io`** definiert sind.



Klasse	Beschreibung
`Reader`	Abstrakte Klasse für textuelle Eingabeströme
`Writer`	Abstrakte Klasse für textuelle Ausgabeströme
`InputStream`	Abstrakte Klasse für binäre Eingabeströme
`OutputStream`	Abstrakte Klasse für binären Ausgabeströme
`File`	Plattform-unabhängige Repräsentation von Dateien

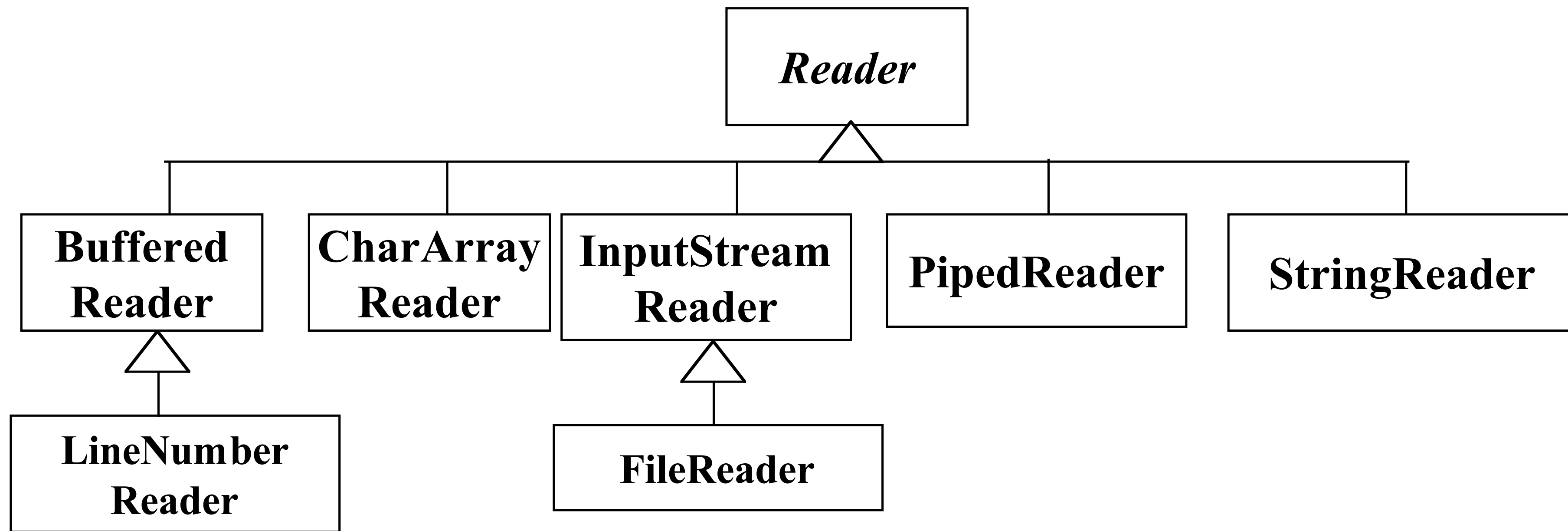
`InputStream`: *Modellierung binärer Eingabeströme*



Klasse	Beschreibung
`ByteArrayInputSteam`	Einlesen von Reihungen vom Typ **`byte`** als Strom
`FileInputStream`	Einlesen von Binär-Dateien
`FilterInputStream`	Ermöglicht gefiltertes Einlesen von Daten auf verschiedene Arten
`BufferedInputStream`	Ermöglicht das Puffern von Eingabedaten
`DataInputStream`	Lesen von vordefinierten elementaren Java-Typen
`ObjectInputStream`	*Deserialisieren* von Objekten
`PipedInputStream`	Lesen von Binärdaten aus einem anderen Thread (⇒ Info III)

Copyright 2001 Bernd Brügge

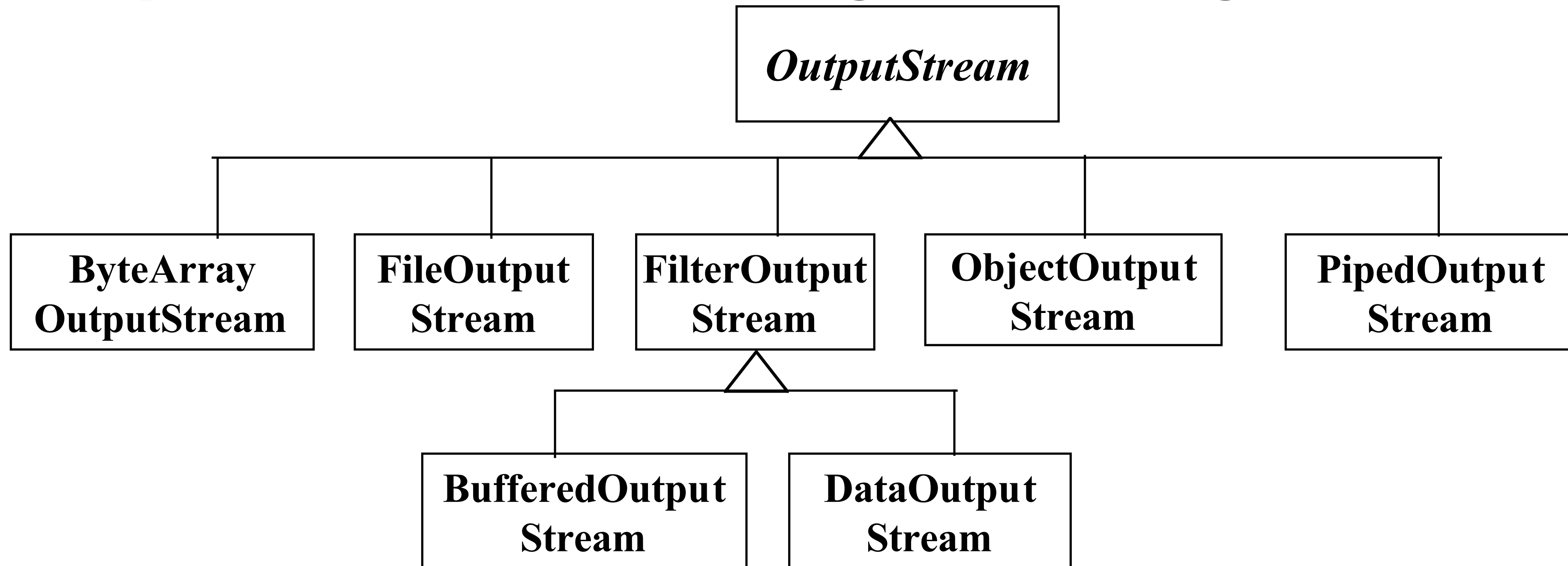
`Reader`: *Modellierung textueller Eingabeströme*



Klasse	Beschreibung
`BufferedReader`	Gepufferte Eingabe bei textuellen Eingabeströmen
`CharArrayReader`	Einlesen von Reihungen vom Typ **`char`** als Strom
`FileReader`	Einlesen von Text-Dateien
`PipedReader`	Lesen von Textdaten aus einem anderen Thread (⇒ Info III)
`StringReader`	Einlesen von Zeichenketten (**`String`**) als Strom
`LineNumberReader`	Zählt Anzahl der Text-Zeilen, die bereits gelesen wurden.

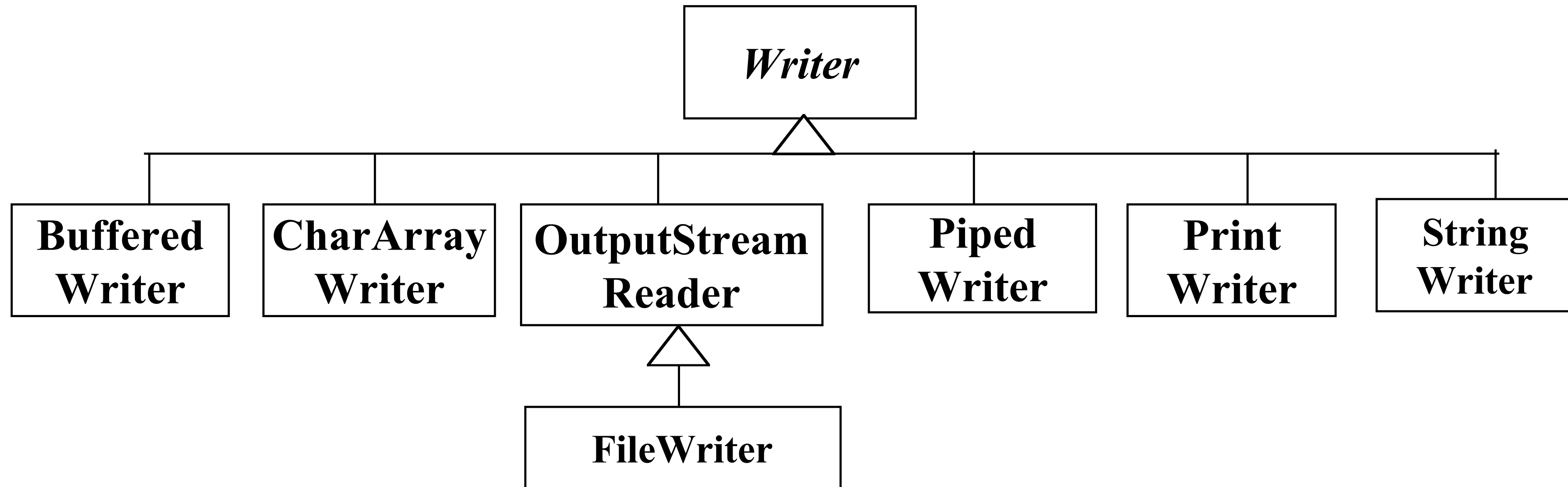
Copyright 2001 Bernd Brügge

`OutputStream`: *Modellierung binärer Ausgabeströme*



Klasse	Beschreibung
`ByteArrayOutputSteam`	Schreiben von Binärdaten in Reihungen vom Type **`byte`**
`FileOutputStream`	Schreiben von Bytes in Binär-Dateien
`FilterOutputStream`	Ermöglicht das gefilterte Schreiben von Daten auf verschiedene Arten
`BufferedOutputStream`	Ermöglicht das Puffern von Ausgabedaten
`DataOutputStream`	Schreiben von vordefinierten elementaren Java-Typen
`ObjectOutputStream`	*Serialisieren* von Objekten
`PipedOutputStream`	Weitergabe von Binärdaten an anderen Thread (⇒ Info III)

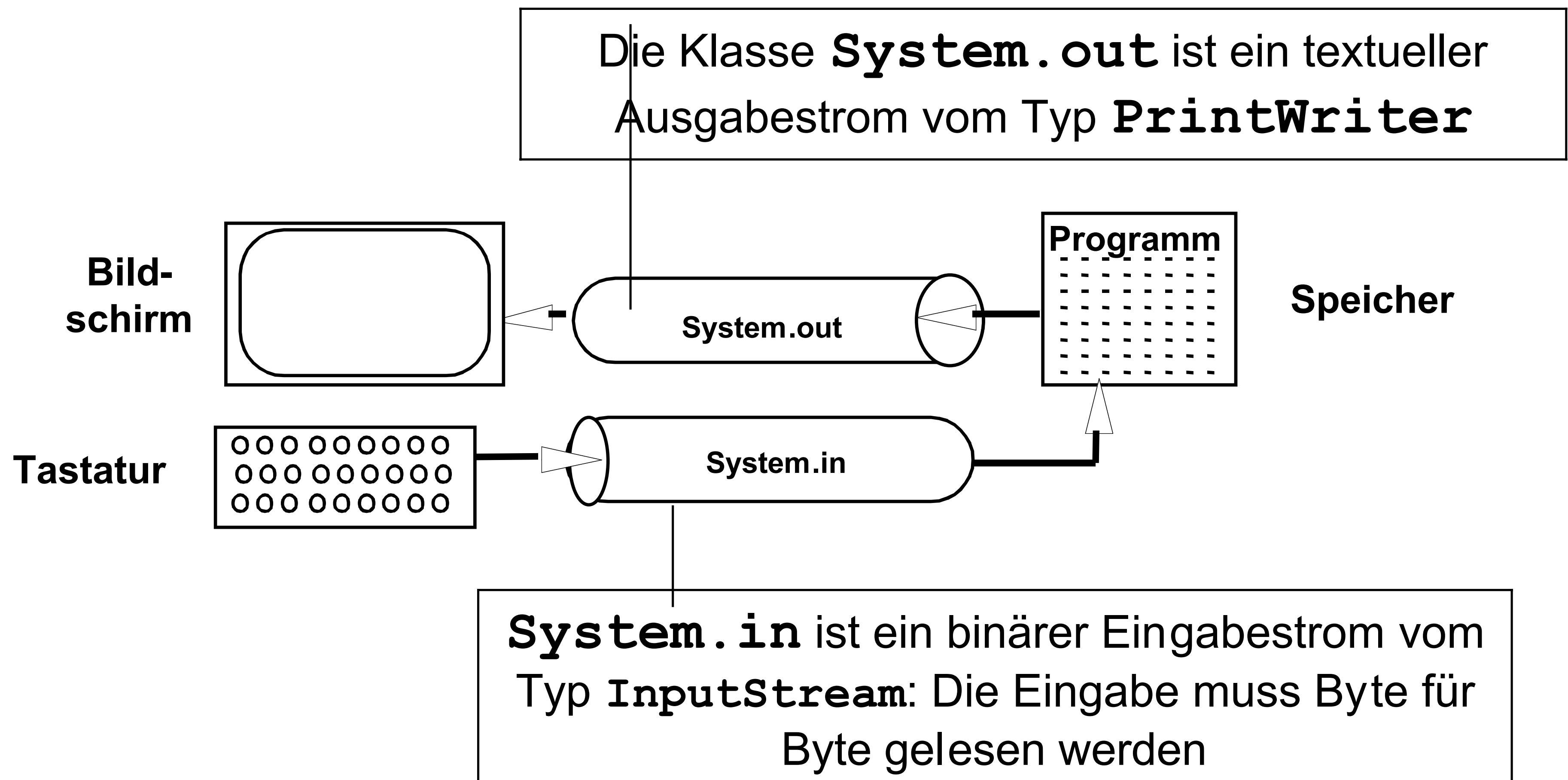
`Writer`: *Modelliert textuelle Ausgabeströme*



Klasse	Beschreibung
`BufferedWriter`	Gepufferte Ausgabe bei textuellen Ausgabeströmen
`CharArrayWriter`	Schreiben von Textdaten in Reihungen vom Typ `char`
`FileWriter`	Schreiben von Textzeichen in Text-Dateien
`PipedWriter`	Weitergabe von Textdaten an anderen Thread (⇒ Info III)
`PrintWriter`	Textuelle Ausgabe von Java's Basistypen
`StringWriter`	Schreiben von Textdaten in Zeichenketten (`String`)

Die Standardein-/-ausgabe in Java basiert auf Strömen

- Informelles Modell



2 wichtige Konzepte

- Zur Verbesserung der Laufzeit von Informatik-Systemen mit Strömen gibt es 2 wichtige Konzepte:
  - ***Puffern von Daten:*** Puffer werden benutzt, wenn das Ein-/Ausgabe-Gerät signifikant langsamer als der Prozessor ist.
  - ***Konkatenation von Strömen:*** Wenn die Ausgabe eines Stroms die Eingabe eines anderen Stroms ist.
    - Beispiel "Unix pipes": **`ls | more`**

Puffern von Daten

- **Definition Eingabepuffer (input buffer):** Eine Region im Speicher für die temporäre Speicherung von bereits gelesenen, aber noch nicht verarbeiteten Daten:
  - Anstatt ein Byte nach dem anderen von dem Eingabegerät zu lesen, wird eine große Anzahl von Bytes gleichzeitig in den internen Puffer gelesen.
  - Die Bytes werden dann einzeln bei jeder Lese-Operation aus dem Eingabepuffer ins Programm transferiert.
- **Definition Ausgabepuffer (output buffer):** Eine Region im Speicher für die temporäre Speicherung von zu schreibenden Daten.
  - Daten werden erst auf das Ausgabegerät geschrieben, wenn der Puffer voll ist (oder bei einer sogenannten **`flush()`**-Operation).

Puffern von Daten (2)

- Viele E/A-Geräte unterstützen den Transfer von Blöcken.
- Der Datentransfer zwischen Puffern und Objekten (z.B. beim Einlesen von Objektattributwerten aus einer Datei) ist im allgemeinen schnell, da sowohl Puffer als auch Objekte gewöhnlich im Programmspeicher angelegt werden.
- Puffern verbessert die Laufzeit von Programmen (natürlich auf Kosten des für die Puffer zusätzlich benötigen Speicherplatzes)

Copyright 2001 Bernd Brügge

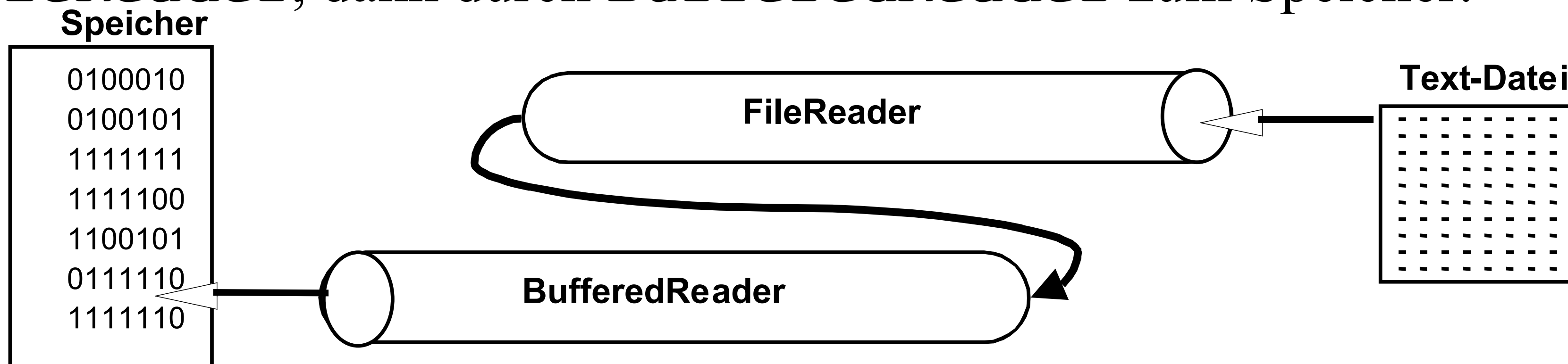
Konkatenation von Strömen

- Die Deklaration

```
BufferedReader inStream
      = new BufferedReader(new FileReader(fileName));
```

konkateniert zwei Ströme **`BufferedReader`** und **`FileReader`**.

- Das Programm kann dann **`inStream.readLine()`** benutzen, um die Text-Datei **`fileName`** zeilenweise - und nicht Zeichen für Zeichen - zu lesen.
- Interpretation: Die Daten fließen von der Text-Datei erst durch **`FileReader`**, dann durch **`BufferedReader`** zum Speicher.



- Analogie aus der Klempnerei: Konkatenation von Strömen ist nichts anderes, als ein Rohr aus zwei Teilrohren zusammenzusetzen.

Konkatenation von Strömen (2)

- Zur Konkatenation von Strömen ist es notwendig, dass der eine Strom bei seiner Erzeugung einen anderen Strom als Parameter erhält. Dies ist der Fall für **`BufferedReader`**:

```
Public class BufferedReader extends Reader {
   // Konstruktor
   public BufferedReader(Reader instream);
   //Instanz-Methode
   public String readLine() throws IOException;
}
```

`FileReader` ist Unterklasse von **`Reader`**

```
BufferedReader inStream
       = new BufferedReader(new FileReader(fileName));
```

Wann benutzen wir welchen Strom?

- Bei binärer Ein-/Ausgabe:
  - Unterklassen von **`InputStream`** und **`OutputStream`**.
- Bei Textueller Ein-Ausgabe:
  - Unterklassen von **`Reader`** und **`Writer`**.
- Beispiel: **`PrintWriter`** ist eine Unterklasse von **`Writer`**. Sie stellt Methoden zur textuellen Ausgabe von Objekten vom Typ **`int`**, **`long`**, **`float`**, **`double`**, **`String`** und **`Object`** bereit:

```
public void print(int i);          public void println(int i);
public void print(long l);         public void println(long l);
public void print(float f);        public void println(float f);
public void print(double d);       public void println(double d);
public void print(String s);       public void println(String s);
public void print(Object o);       public void println(Object o);
```

Verbindung von Dateien und Strömen

- Jede Programmiersprache muss ein Konzept bereitstellen, um Ströme mit Dateien zu verbinden.
  - In Java geschieht die Verbindung im Konstruktor der Strom-Klasse, die mit dem Dateinamen als Argument aufgerufen wird.
- Beispiel:

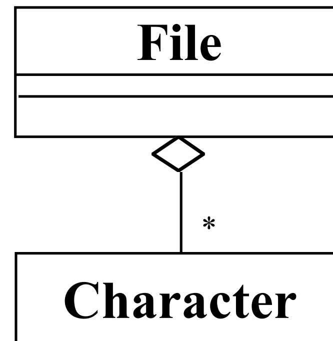
```
String fName = "/usr/bob/src/trivial.java";
FileWriter outS = new FileWriter (fName);
```

verbindet den Ausgabestrom **`outS`** mit einer Datei namens **`fName`**.

In Java unterscheiden wir Text-Dateien und Binär-Dateien

Text-Datei in Java

- **Definition Text-Datei** (text file): Eine Datei, in der die Daten vom Typ **`char`** (bzw. **`Character`**) sind.

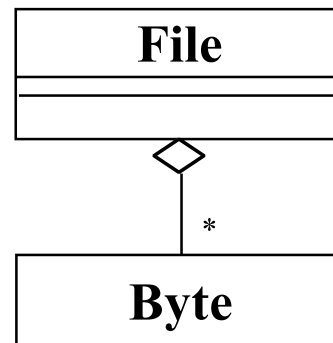


- Da die Codierung und Decodierung von **`char`**-Werten in Java auf Unicode basiert, sind Text-Dateien portierbar, d.h. auch auf verschiedenen Plattformen lesbar

- **Definition Text-Editor:** Ein Programm, dass Benutzern das interaktive Eintragen, Ändern und Löschen von Zeichen in einer Text-Datei ermöglicht.

Binär-Datei

- **Definition Binär-Datei** (binary file): Eine Datei, in der die Daten vom Typ **`byte`** (bzw. **`Byte`**) sind.



- Binär-Dateien sind im allgemeinen nicht portierbar, da verschiedene Rechner verschiedene Repräsentationen für Binär-Daten verwenden .
- Java-spezifische Binär-Dateien (z.B. **`.class`**-Dateien) sind auf allen Rechnern lesbar, die die JVM-Spezifikation einhalten, da Java die Repräsentation von binären Dateien selbst definiert.

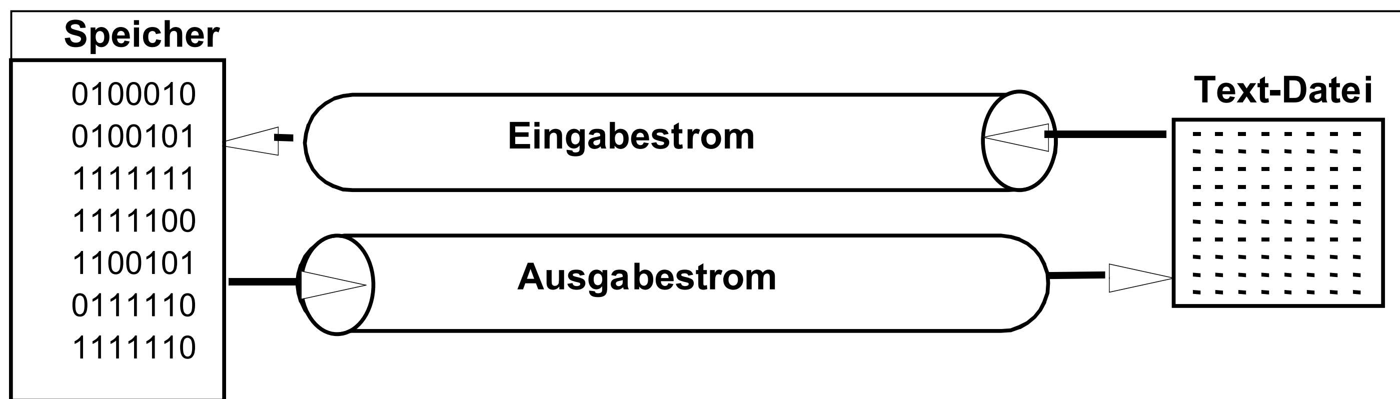
Was machen wir jetzt?

1. Kleine Fallstudie für die Benutzung von Strömen und Text-Dateien: Entwurf eines Text-Editors
    - Schreiben einer Text-Datei
    - Lesen einer Text-Datei
2. Schreiben und Lesen von Binär-Dateien
    - Schreiben einer Binär-Datei
    - Lesen einer Binär-Datei
3. Schreiben und Lesen von beliebigen Objekten
    - Serialisierung
    - Deserialisierung

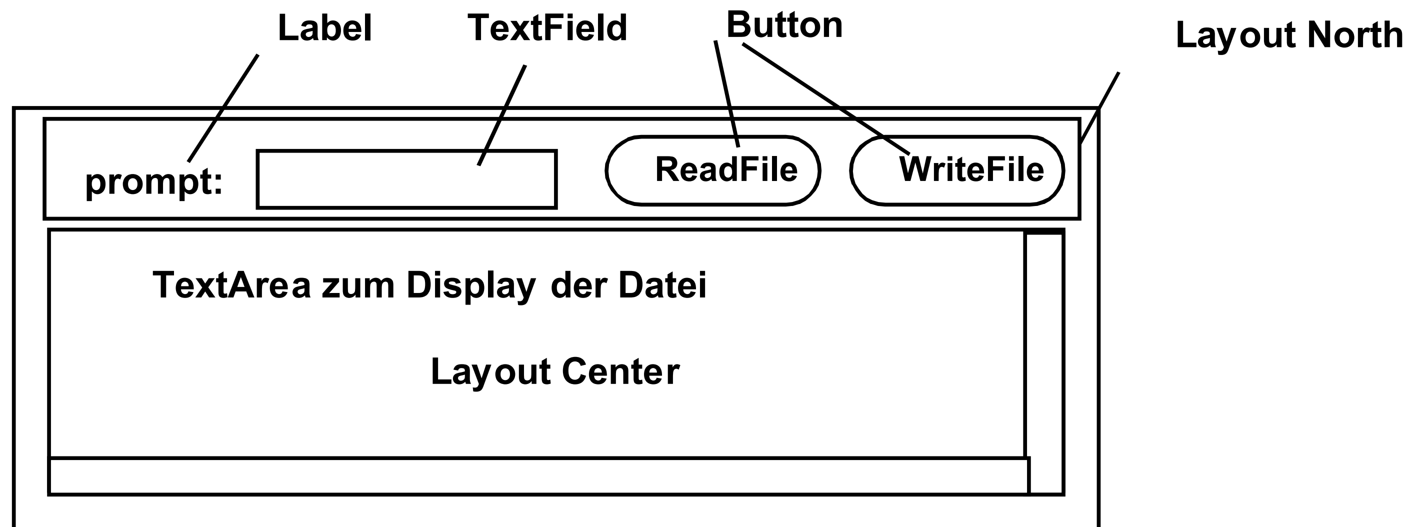
1. Entwurf eines Text-Editors

Problembeschreibung: Zwei wichtige Aufgaben eines Text-Editors sind das *Lesen von Text-Dateien* und das *Schreiben von Text-Dateien*.

***Lösung*:** Aus Zeitgründen machen wir nur eine "verkürzte Analyse" und konzentrieren uns dann vorwiegend auf den detaillierten Entwurf und die Implementierung

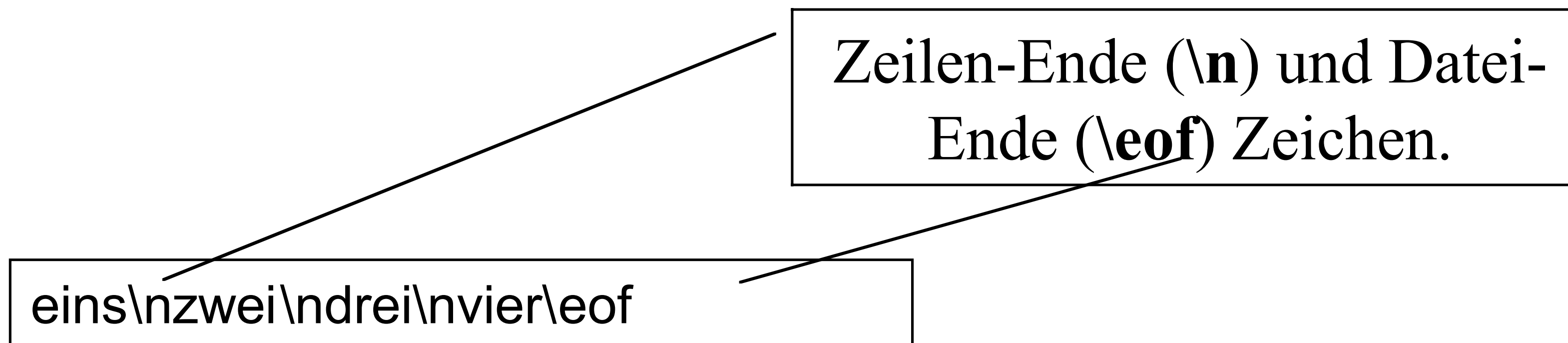


Modell der Bedienoberfläche



Format von Text-Dateien

- Text-Dateien haben folgendes Format:
  - Eine Folge von Zeichen, getrennt durch Null oder mehr *Zeilen-Ende Zeichen* **`\n`**.
  - Das Ende der Datei wird mit dem speziellen *Datei-Ende Zeichen* **`\eof`** angezeigt.



Copyright 2001 Bernd Brügge

Anwendungsfälle für den Text-Editor

Anwendungsfall "Schreiben einer Text-Datei":

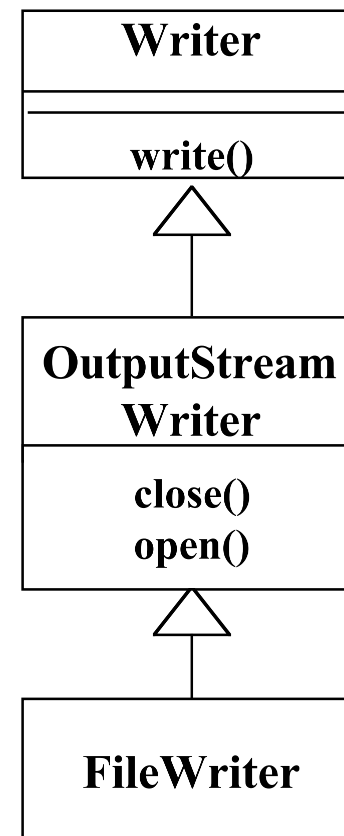
- Der Ereignisfluss besteht aus den folgenden 3 Schritten:
  - Verbinde den Ausgabestrom mit einer Text-Datei.
  - Schreibe die einzelnen Daten auf den Strom (in einer Schleife).
  - Schließe den Ausgabestrom.

Anwendungsfall "Lesen einer Text-Datei"

- Der Ereignisfluss besteht aus den folgenden 3 Schritten:
  - Verbinde den Eingabestrom mit der Text-Datei.
  - Lies die einzelnen Daten vom Strom (in einer Schleife).
  - Schließe den Eingabestrom.

Detaillierter Entwurf: Schreiben einer Text-Datei

- Weil wir eine Text-Datei schreiben wollen, schauen wir uns die Unterklassen von **`Writer`** an.



- Wir wählen **`FileWriter`**, denn diese Klasse wurde explizit für die Ausgabe in Text-Dateien entworfen.

Detaillierter Entwurf: Schreiben einer Text-Datei (2)

Spezifikation der Klasse **`FileWriter`**:

```
public class FileWriter extends OutputStreamWriter {
  public FileWriter (String fileName) throws IOException;
  public FileWriter (String fileName, boolean append)
    throws IOException;
}
```

Die Methode **`write()`** wird von der Superklasse **`Writer`** geerbt:

```
public void write (String s) throws IOException;
```

Da der Konstruktor von **`FileWriter`** und die Methode **`write()`** Ausnahmen vom Typ **`IOException`** werfen, werden wir sie nur innerhalb eines **`try-catch`** Gerüsts benutzen.

Allgemein gilt: Alle öffentlichen Methoden von Strömen werfen Ausnahmen vom Typ **`IOException`** (auch wenn wir sie im folgenden nicht immer zeigen).

Implementierung: Schreiben einer Text-Datei

`writeTextFile()` nimmt eine mehrzeilige Zeichenkette von Typ **`TextArea`** und schreibt sie auf eine Text-Datei mit Namen **`fileName`**.

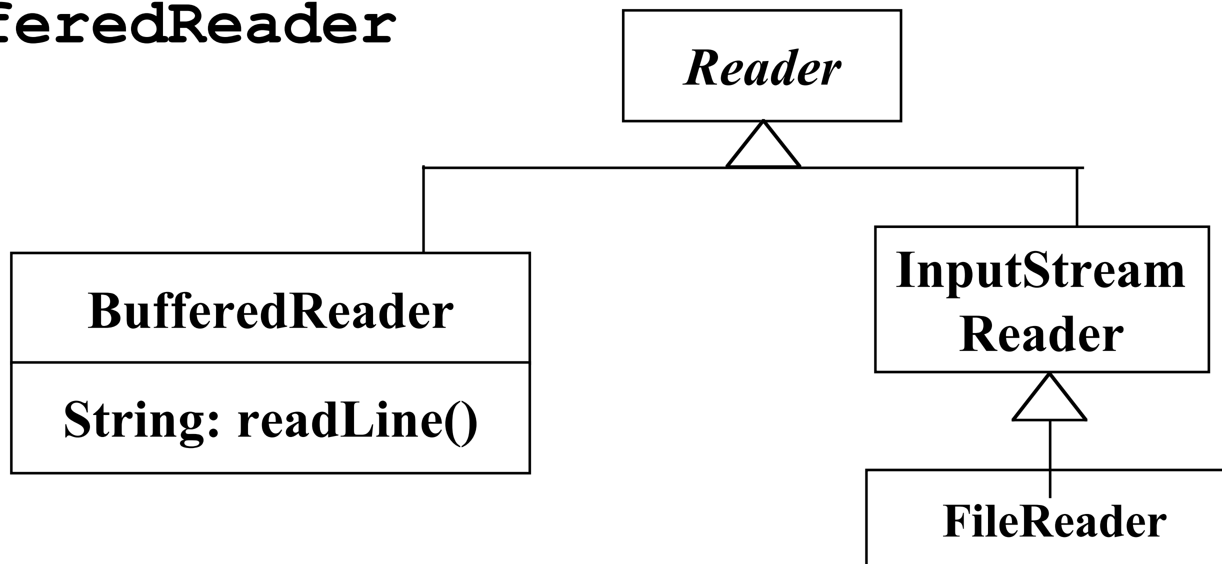
Verbindung des Ausgabestroms mit einer Text-Datei namens **`fileName`**.

```
private void writeTextFile(TextArea display, String fileName) {
  try {
    FileWriter outStream = new FileWriter(fileName);
    outStream.write(display.getText());
    outStream.close();
  } catch (IOException e) {
    display.setText("E/A-Fehler: " + e.getMessage() + "\n");
    e.printStackTrace();
  }
} // writeTextFile()
```

Abfangen der Ausnahme **`IOException`**.

Detaillierter Entwurf: Lesen einer Text-Datei

- Da es sich um Text-Dateien handelt, schauen wir uns in diesem Fall die Klasse **`Reader`** an.
- Wir müssen Zeichen (Typ **`char`**) aus einer Datei lesen: **`FileReader`**
- Ausserdem wollen wir die Zeichen zeilenweise lesen: **`BufferedReader`**



- Initialisierung durch Konkatenation der Ströme: **`new BufferedReader(new FileReader(filename))`**
- Lesen der Zeilen dann mit öffentlicher Methode **`readLine()`**

Implementierung: Lesen einer Text-Datei

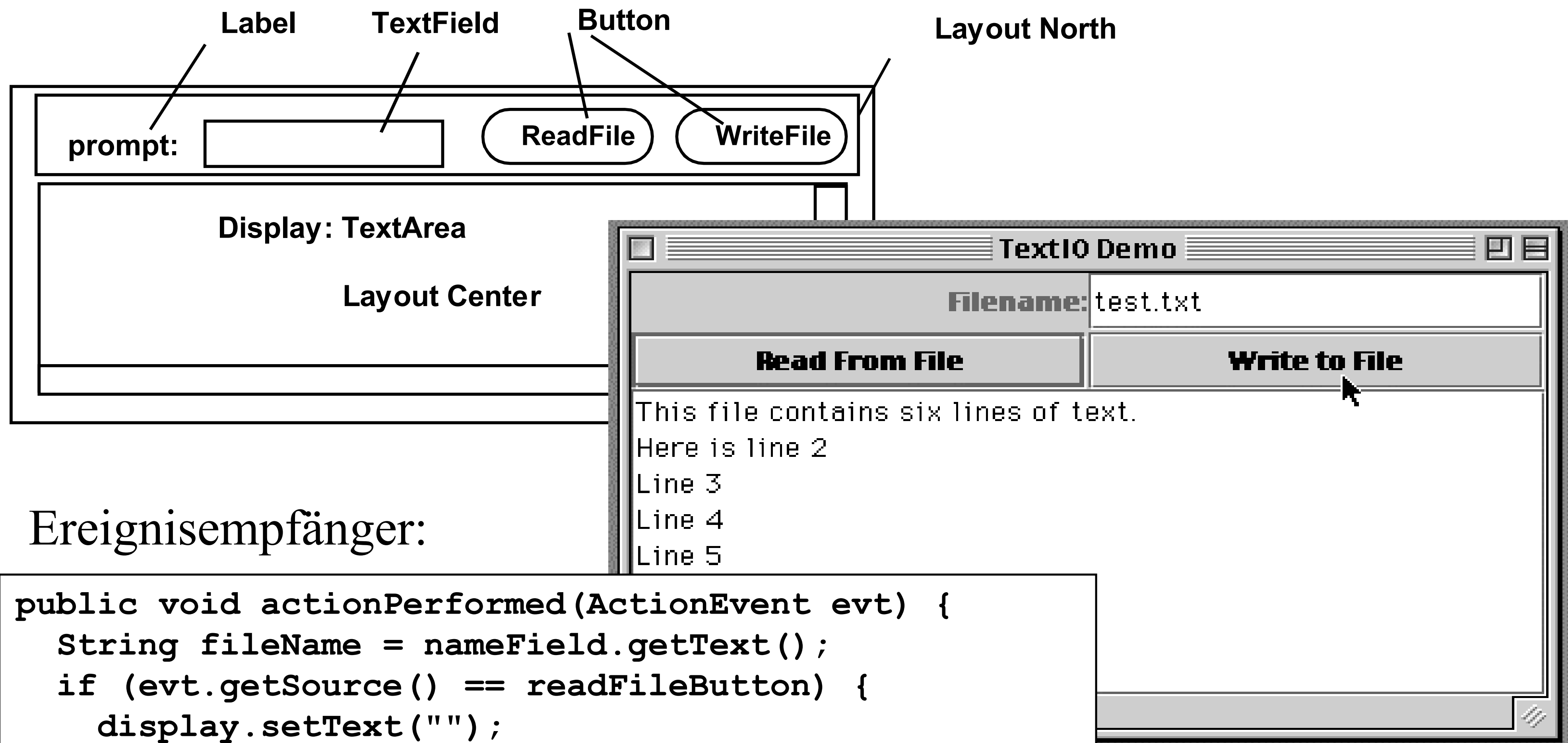
Konkatenation von Strömen des Typs **`BufferedReader`** und **`FileReader`**

bei *end-of-file* gibt **`readLine()`** null zurück

```
private void readTextFile(TextArea display, String fileName) {
  try {
    BufferedReader inStream
      = new BufferedReader(new FileReader(fileName));
    String line = inStream.readLine();  // Lies eine Zeile
    while (line != null) {              // solange noch was da ist
      display.append(line + "\n");      // Stelle die Zeile dar
      line = inStream.readLine();       // Lies die nächste Zeile
    }
    inStream.close();                   // Schließe den Strom
  } catch (FileNotFoundException e) {
    display.setText("IOERROR: File NOT Found: " + fileName + "\n");
    e.printStackTrace();
  } catch ( IOException e ) {
    display.setText("IOERROR: " + e.getMessage() + "\n");
    e.printStackTrace();
  }
} // readTextFile()
```

Eine nicht existierende Datei erzeugt beim Aufruf von **`new FileReader`** die Ausnahme **`FileNotFoundException`**

Implementierung des Text-Editors



Ereignisempfänger:

```
public void actionPerformed(ActionEvent evt) {
  String fileName = nameField.getText();
  if (evt.getSource() == readFileButton) {
    display.setText("");
    readTextFile(display, fileName);     // Folie 32
  }
  else writeTextFile(display, fileName); // Folie 30
} // actionPerformed()
```

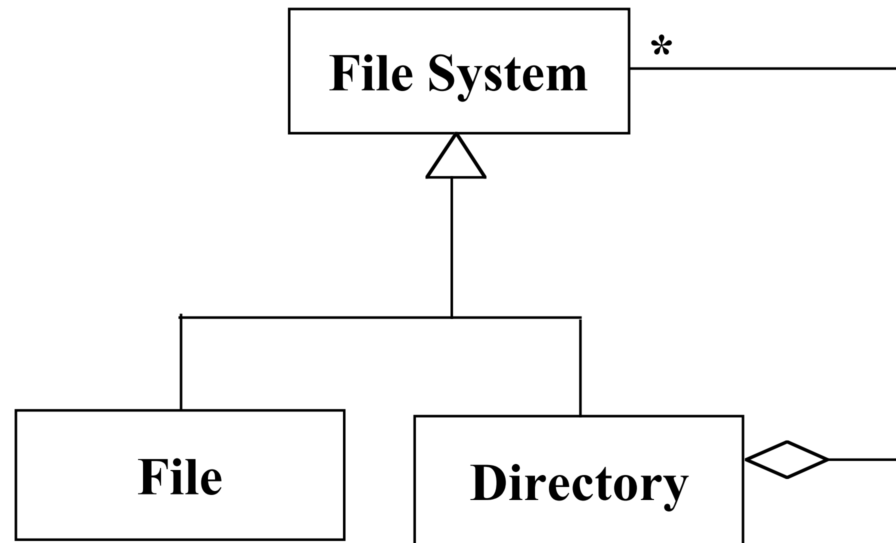
Robuste vs Benutzerfreundliche Programmierung

- Wie wir gesehen hatten, wird beimVersuch, einen Strom mit **`new FileReader(fileName)`** zu öffnen, die Ausnahme **`FileNotFoundException`** geworfen, wenn die Datei namens **`fileName`** nicht existiert.
  - Dies kommt vor, wenn Benutzer einen nicht existenten Dateinamen angeben oder die Datei nicht dort ist, wo sie sein sollte (was sehr häufig der Fall ist :-)
- Abfangen der Ausnahme ist ein Beispiel für robuste, aber benutzer-unfreundliche Programmierung.
- Gibt es eine Möglichkeit, diese Art von Fehler vorher zu entdecken, um den Benutzern bessere Fehlermeldungen geben zu können?
  - Die Klasse **`java.io.File`** enthält Methoden, die man dafür benutzen kann.

Datei-Systeme, Dateien und Verzeichnisse

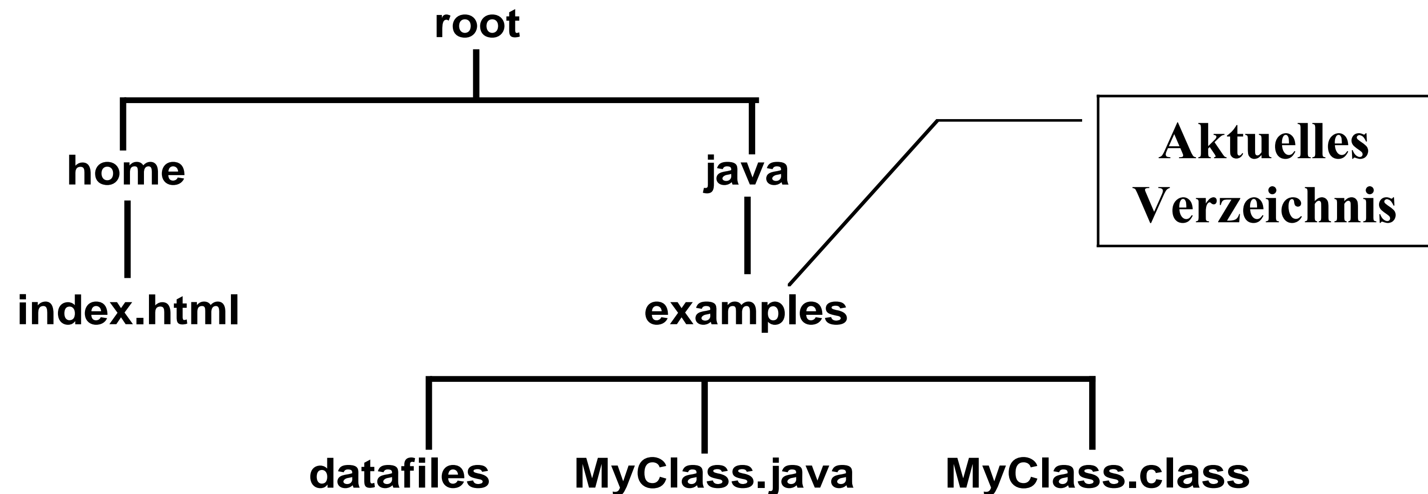
- Die Klasse `File` repräsentiert Dateien und Verzeichnissen in einer Betriebssystem- bzw. Plattform-unabhängigen Weise.
- Beschreibung von hierarchischen Datei-Systemen:
  - Ein ***Datei-System*** (file system) besteht aus ***Dateien*** (files) und ***Verzeichnissen*** (directories). Verzeichnisse können wieder aus Dateien und weiteren Verzeichnissen bestehen.
  - Jedes Datei-System hat ein Verzeichnis als Wurzel. Alle Dateien sind in der Hierarchie "ist Teil von" angeordnet.
  - Ein ***Pfadname*** ist die Beschreibung einer Datei in dieser Hierarchie. Das Verzeichnis, in dem sich ein Benutzer bzw. ein Programm befindet, wird ***aktuelles Verzeichnis*** (current directory) genannt.
    - Der ***absolute Pfadname*** beschreibt eine Datei ausgehend von der Wurzel des Datei-Systems. Ein ***relativer Pfadname*** beschreibt eine Datei relativ zum aktuellen Verzeichnis
- Wie sieht das entsprechende UML-Modell aus? Können wir ein Entwurfsmuster verwenden? Welches?

UML-Modell des Java-Dateisystems



Absoluter und relativer Pfadname

Gegeben sei das Java Programm **`MyClass.java`** in einem Unix-basierten Datei-System



Aktuelles Verzeichnis: **`/root/java/examples`**
Absoluter Pfadname: **`/root/java/examples/MyClass.java`**
Relativer Pfadname: **`MyClass.java`**

Die Klasse `File`

```
public class File extends Object implements Serializable {
  // Konstanten
  public static final String pathSeparator;
  public static final char pathSeparatorChar;
  public static final String separator;
  public static final char separatorChar;
  // Konstruktoren
  public File(String path);
  public File(String path, String name);
  // Öffentliche Instanz-Methoden
  public boolean canRead();   // Ist die Datei lesbar?
  public boolean canWrite();  // Ist die Datei schreibbar?
  public boolean delete();    // Lösche die Datei
  public boolean exists();    // Existiert die Datei?
  public String getParent();  // Elternknoten von Datei oder Verzeichnis
  public String getPath();    // Absoluter Pfadname der Datei
  public boolean isDirectory();    // Ist das Objekt ein Verzeichnis?
  public boolean isFile();    // Ist das Objekt eine Datei?
  public long lastModified(); // Wann wurde es letztes Mal modifziert?
  public long length();       // Wieviele Bytes enthält es?
  public String[] list();     // Inhalt dieses Verzeichnisses
  public boolean renameTo(File f); // Gib Datei neuen Namen f
}
```

Plattformunabhängigkeit:
Unix: **/**
Windows: ****
Mac OS: **:**

`File`-Methoden erzeugen keine Ausnahmen!

Benutzerfreundliche Programmierung

- Um den Benutzer vor unverständlichen (Fehler-)Meldungen zu schützen, schreiben wir jetzt zwei Methoden, in denen wir unsere eigenen Ausnahmen erzeugen:
  - Überprüfung der Lesbarkeit von Dateien
  - Überprüfung der Schreibbarkeit von Dateien

Überprüfung der Lesbarkeit von Dateien

Kreiere ein **`File`**-Objekt für die Datei **`fileName`**.

```
private boolean isReadableFile(String fileName) {
  try {
    File file = new File(fileName);
    if (!file.exists())
      throw (new FileNotFoundException("No such File:" + fileName));
    if (!file.canRead())
      throw (new IOException("File not readable: " + fileName));
    return true;
  } catch (FileNotFoundException e) {
    System.out.println("IOERROR: File NOT Found: " + fileName + "\n");
    return false;
  } catch (IOException e) {
    System.out.println("IOERROR: " + e.getMessage() + "\n");
    return false;
  }
} // isReadableFile
```

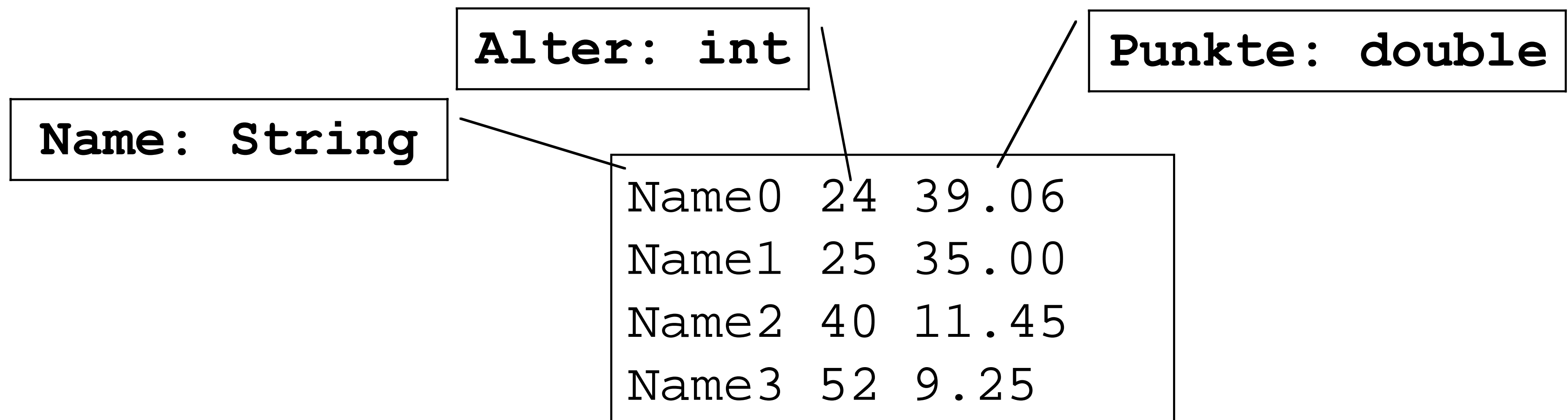
Prüfe Existenz und Lesbarkeit mit Methoden aus **`File`**

Überprüfung der Schreibbarkeit von Dateien

```
private boolean isWriteableFile(String fileName) {
  try {
    File file = new File(fileName);
    if (fileName.length() == 0)
      throw (new IOException("Invalid file name: " + fileName));
    if (file.exists() && !file.canWrite())
      throw (new IOException("IOERROR: File not writeable: " +
                                    fileName));
    return true;
  } catch (IOException e) {
    display.setText("IOERROR: " + e.getMessage() + "\n");
    return false;
  }
} // isWriteableFile()
```

2. Schreiben und Lesen von Binär-Dateien

- Binär-Dateien haben kein Datei-Ende Zeichen (**`\eof`**).
- Ereignisfluss beim Lesen bzw. Schreiben von Binärdaten:
  - Verbinde einen Strom mit der Datei.
  - Lies oder schreibe die Binärdaten in einer Schleife.
  - Schließe den Strom.
- Beispiel: Schreiben von Studenten-Daten



Detaillierter Entwurf

Erlaubt das Schreiben von Strings und Doubles

- Welche Ströme und Methoden verwenden wir?
  - Strom: Unterklasse von **`OutputStream`**.
  - Konstruktor: **`FileOutputStream(String filename)`**
  - Schreibe-Methoden aus **`DataOutputStream`**

```
public class DataOutputStream extends FilterOutputStream {
  // Konstructor
  public DataOutputStream(OutputStream out);
  // Ausgewählte öffentliche Instanzen-Methoden
  public void flush() throws IOException;
  public final void writeChars(String s) throws IOException;
  public final void writeDouble(double d) throws IOException;
  public final void writeInt(int i) throws IOException;
  public final void writeUTF(String s) throws IOException;
}
```

Unicode Text Format (UTF) ist ein Schema zum binären Codieren von Java's Unicode-Zeichenvorrat.

Schreiben von Binärdaten in eine Datei

Verbindung des Stroms mit der Binär-Datei:

```
DataOutputStream outStream
  = new DataOutputStream(new FileOutputStream (fileName));
```

Schreiben der Daten in die Datei:

```
for (int k = 0; k < 5; k++) {                              // 5 Studenten
  outStream.writeUTF("Name" + k);                          // Name
  outStream.writeInt((int)(20 + Math.random() * 25));      // Alter
  outStream.writeDouble(Math.random() * 500);              // Punkte
}
```

```
01010011001100100101010011001100
00010100110011001011010100110011
...
```

Interpretierbar z.B. als
- zwei 32-bit-Werte (z.B. **`int`**)
- ein 64-bit-Wert (z.B. **`double`**)
- acht 8-bit-ASCII-Zeichen

Die Binär-Datei enthält keine Informationen, wie die Binärdaten zu interpretieren sind $\Rightarrow$ große Sorgfalt beim Einlesen erforderlich!

Copyright 2001 Bernd Brügge

Methode zum Schreiben von Binärdaten

```
private void writeBinary( String fileName )  {
  try {
    DataOutputStream outStream    // Öffne den Strom
      = new DataOutputStream(new FileOutputStream(fileName));
    for (int k = 0; k < 5; k++) { // 5 Studenten
      String name = "Name" + k;
      outStream.writeUTF("Name" + k);                    // Name
      outStream.writeInt((int)(20 + Math.random() * 25)); // Alter
      outStream.writeDouble(5.00 + Math.random() * 10);   // Punkte
    } // for
    outStream.close();            // Schliesse den Strom
  } catch (IOException e) {
    display.setText("IOERROR: " + e.getMessage() + "\n");
  }
} // writeBinary()
```

Lesen von Binärdaten aus einer Datei

- Strom: Unterklasse von **`InputStream`**
- Konstruktor: **`FileInputStream(String filename)`**
- Lese-Methoden aus **`DataInputStream`**

Pendant zu **`DataOutputSteam`**

```
public class DataInputStream extends FilterInputStream {
  // Instanz-Methoden
  public final boolean readBoolean() throws IOException;
  public final byte readByte() throws IOException;
  public final char readChar() throws IOException;
  public final double readDouble() throws IOException;
  public final float readFloat() throws IOException;
  public final int readInt() throws IOException;
  public final long readLong() throws IOException;
  public final short readShort() throws IOException;
  public final String readUTF() throws IOException;
}
```

Schleife zum Lesen von Binärdaten

- Da Binär-Dateien kein **`\eof`**-Zeichen haben, müssen wir das Ende der Datei mit der Ausnahme **`EOFException`** abfangen.

```
try {
  while (true) {                              // Unendliche Schleife
    String name = inStream.readUTF();  // Lies Studenten
    int alter = inStream.readInt();
    double punkte = inStream.readDouble();
    display.append(name + " " + alter + " " + punkte + "\n");
  } // while
} catch (EOFException e) {}              // bis zum Ende der Datei
```

Methode zum Lesen von Binärdaten aus einer Datei

```
private void readBinary(String fileName) {
  try {
    DataInputStream inStream
      = new DataInputStream(new FileInputStream(fileName));
    display.setText("Name   Alter  Note\n");
    try {
      while (true) {
        String name = inStream.readUTF();
        int alter = inStream.readInt();
        double note = inStream.readDouble();
        display.append(name + "   " + alter + "   " + note + "\n");
      } // while
    } catch (EOFException e) { // bis zur EOFException
    } finally {
      inStream.close();        // Schließe den Strom
    }
  } catch (FileNotFoundException e) {
    display.setText("IOERROR: File NOT Found: " + fileName + "\n");
  } catch (IOException e) {
    display.setText("IOERROR: " + e.getMessage() + "\n");
  }
} // readBinary()
```

Konkatenation der Ströme

geschachtelter **`try`**-Block

`finally` bezieht sich auf den inneren **`try`**-Block

3. Schreiben und Lesen von beliebigen Objekten

- ***Objekt-Serialisierung*** ist der Prozess des Schreibens eines Objektes auf einen Strom als eine Serie von Bytes.
- ***Objekt-Deserialisierung*** ist der Prozess des Lesens eines Objektes von einem Strom als eine Serie von Bytes.
- Objekte serializiert man mit **`ObjectOutputStream`** und deserialisiert sie mit **`ObjectInputStream`**

```
public class ObjectOutputStream extends OutputStream
  implements ObjectOutput {
  public final void writeObject(Object obj) throws IOException;
}
public class ObjectInputStream extends InputStream
  implements ObjectInput {
  public final Object readObject()
    throws IOException, ClassNotFoundException;
}
```

- Um ein Objekt zu serialisieren, muss es als Argument beim Aufruf der **`writeObject()`**-Methode übergeben werden. Zur Deserialisierung eines Objekts wird die **`readObject()`**-Methode aufgerufen.

Die Klasse **`Student`** *als serialisierbare Klasse*

```
public class Student implements java.io.Serializable {
  private String name;
  private int alter;
  private double punkte;

  public Student() {}
  public Student (String fname, int yr, double fpunkte) {
    name = fname;
    alter = yr;
    punkte = fpunkte;
  }
  public String toString() {
    return name + "\t" + alter + "\t" + punkte;
  }
} // Student
```

- **Wichtig**: Die Java Schnittstelle **`Serializable`** hat weder Methoden noch Konstanten (sog. *Marker Interface*)
- Klassen müssen diese Schnittstelle trotzdem "implementieren", um anzuzeigen, dass sie serialisiert und deserialisiert werden dürfen.

Dateiein-/-ausgabe von serialisierbaren Objekten

```
public void writeToFile (FileOutputStream outStream)
  throws IOException {
  ObjectOutputStream ooStream = new ObjectOutputStream(outStream);
  ooStream.writeObject(this);
  ooStream.flush();
} // writeToFile()

public void readFromFile (FileInputStream inStream)
  throws IOException, ClassNotFoundException {
  ObjectInputStream oiStream = new ObjectInputStream(inStream);
  Student s = (Student) oiStream.readObject();
  this.name = s.name;
  this.alter = s.alter;
  this.punkte = s.punkte;
} // readFromFile()
```

Schreibe das (serialisierte) Objekt auf den Strom.

Lies das (deserialisierte) Objekt vom Strom.

Schreiben von Objekten als Binärdaten in eine Datei

```
private void writeBinaryRecords (String fileName) {
  try {
    FileOutputStream outStream = new FileOutputStream(fileName);
    for (int k = 0; k < 5; k++) {       // Erzeuge 5 Zufallsobjekte
      String name = "name" + k;
      int alter = (int)(2000 + Math.random() * 4);
      double punkte = Math.random() * 12;
      Student student = new Student(name, alter, punkte); // Object
      display.append("Output: " + student.toString() +  "\n");
      student.writeToFile(outStream) ;  // Schreibe sie in die Datei
    } // for
    outStream.close();
  } catch (IOException e) {
    display.append("IOERROR: " + e.getMessage() + "\n");
  }
} // writeBinaryRecords()
```

führt Serialisierung aus

Lesen von Objekten als Binärdaten aus einer Datei

```
private void readBinaryRecords(String fileName) {
  try {
    FileInputStream inStream = new FileInputStream(fileName);
    display.setText("Name\talter\tpunkte\n");
    try {
      while (true) {                          // Unendliche Schleife
        Student student = new Student();      // Kreiere Student
        student.readFromFile(inStream);       // und lies die Daten
        display.append(student.toString() + "\n");
      }
    } catch (EOFException e) {}               // EOFException (Dateiende)
    inStream.close();
  } catch (FileNotFoundException e) {
    display.append("IOERROR: File NOT Found: " + fileName + "\n");
  } catch (IOException e) {
    display.append("IOERROR: " + e.getMessage() + "\n");
  } catch (ClassNotFoundException e) {
    display.append("ERROR: Class NOT found " + e.getMessage() + "\n");
  }
} // readbinaryRecords()
```

führt Deserialisierung aus

Zusammenfassung

- Eine ***Datei*** ist eine Sammlung von Daten, die extern auf einem Sekundärspeicher (Platte, CD, Band etc.) gespeichert sind.
- Ein ***Strom*** ist ein Objekt, das Daten von Geräten, Dateien oder anderen Objekten holt bzw. an Geräte, Dateien oder andere Objekte liefert:
  - Ein ***Eingabestrom*** liefert Daten von einer externen Quelle an ein Programm.
  - Ein ***Ausgabestrom*** liefert Daten vom Programm an eine externe Senke.
- Ein ***Puffer*** ist ein Bereich im Hauptspeicher, der temporär genutzt wird, um Daten während der Ein- oder Ausgabe zu speichern.
- Problem des detaillierten Entwurfs der Ein-/Ausgabe für Java-Programme:
  - Welche Ströme und Methoden aus der Java-Klassenbibliothek soll ich benutzen?
  - Woher weiß ich, ob es die Klassen gibt, die ich brauche?
  - Wo finde ich sie?

Zusammenfassung (2)

- allgemeiner Algorithmus für Implementierung von Ein-/Ausgabe:
  - Verbinde den gewählten Ein-/Ausgabestrom mit der Datei
  - Führe die Ein-/Ausgabe durch
  - Schließe den Ein-/Ausgabestrom.
- Die Klasse **`java.io.File`** erlaubt eine Überprüfung von Dateien, bevor man sie mit dem gewählten Strom verbindet.

  Analogie: *"Überprüfen der Wassertemperatur im Schwimmbad, ohne dass man gleich reinspringen muss."*
- **`File`** erlaubt eine plattformunabhängige Behandlung von Datei-Systemen und Dateien.
- ***Serialisierung*** erlaubt das Schreiben eines beliebigen Objekts auf einen (binären) Ausgabestrom. ***Deserialisierung*** erlaubt das Einlesen serialisierter Objekte aus einem (binären) Eingabestrom.

Copyright 2001 Bernd Brügge