Script generated by TTT

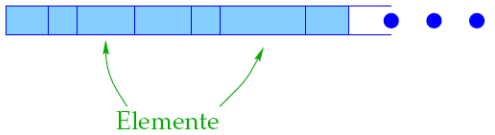
Title: Westermann: Einführung in die Informatik (30.01.2012)

Date: Mon Jan 30 17:23:43 CET 2012

Duration: 53:41 min

Pages: 41

Vorstellung:



- Sowohl Ein- wie Ausgabe vom Terminal oder aus einer Datei wird als Strom aufgefasst.
- Ein Strom (Stream) ist eine (potentiell unendliche) Folge von Elementen.
- Ein Strom wird gelesen, indem links Elemente entfernt werden.
- Ein Strom wird geschrieben, indem rechts Elemente angefügt werden.

Unterstützte Element-Typen:

- Bytes;
- Char (16 Bit Unicode-Zeichen).

Achtung:

- Alle Bytes enthalten 8 Bit
- Intern stellt Java 16 Bit pro Unicode-Zeichen bereit ...
- standardmäßig benutzt Java (zum Lesen und Schreiben) den Zeichensatz Latin-1 bzw. ISO8859_1.
- Diese externen Zeichensätze benötigen ein Byte pro Zeichen.

Orientierung:

- Will man mehr oder andere Zeichen (z.B. chinesische), kann man den gesamten Unicode-Zeichensatz benutzen.
- Wieviele Bytes dann extern für einzelne Unicode-Zeichen benötigt werden, hängt von der benutzten Codierung ab ...
- Die gelesenen Bytes werden gemäß der Codierung in Unicode-Zeichen umgewandelt.
- Dies wird von Java in den Klassen `InputStreamReader`, `OutputStreamWriter` unterstützt.

Problem 1: Wie repräsentiert man Daten, z.B. Zahlen?

- binär codiert, d.h. wie in der Intern-Darstellung
  $\Longrightarrow$ vier Byte pro `int`;
- textuell, d.h. wie in Java-Programmen als Ziffernfolge im Zehner-System (mithilfe von Latin-1-Zeichen für die Ziffern)
  $\Longrightarrow$ bis zu elf Bytes pro `int`.

	Vorteil	Nachteil
binär	platzsparend	nicht menschenlesbar
textuell	menschenlesbar	platz-aufwendig

Problem 2:

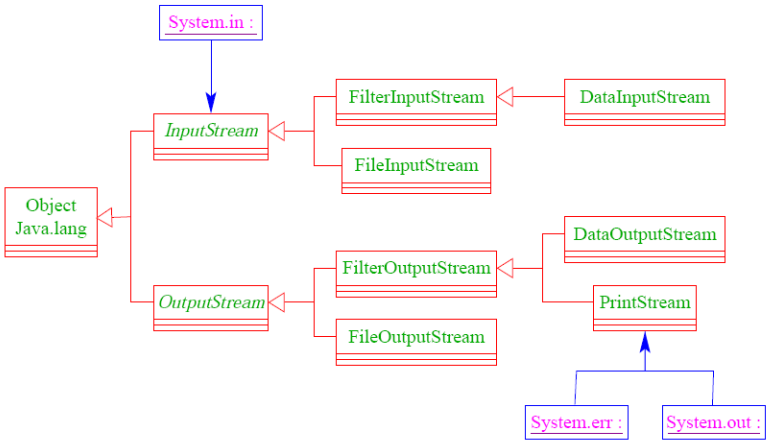
Wie schreibt bzw. wie liest man die beiden unterschiedlichen Darstellungen?

Dazu stellt Java im Paket `java.io` eine Vielzahl von Klassen zur Verfügung ...

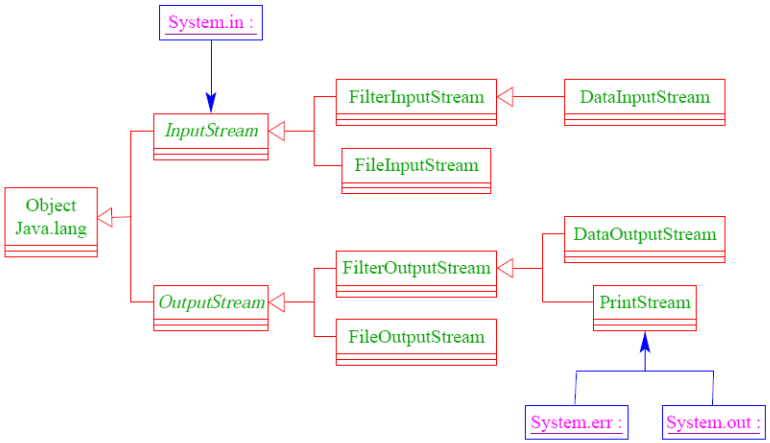
Java unterscheidet zwischen Byte-Streams (Input/OutputStream) und Character-Streams (Reader/Writer)

1.1 Byteweise Ein- und Ausgabe

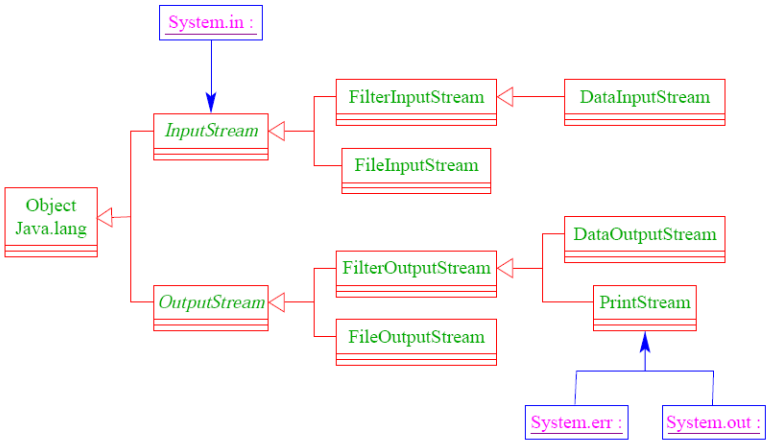
Zuerst eine unvollständige Übersicht ...



- InputStream bzw. OutputStream: einfache Ein- und Ausgabe, mehrfach überladenen Methoden `read` und `write` basierend auf abstrakter Instanz, die nur ein Byte liest oder schreibt.
- Streams, die nicht reine Bytefolgen lesen bzw. schreiben, sondern die Daten (vor-)verarbeiten, werden abgeleitet von den Oberklassen FilterInputStream und FilterOutputStream, die aber keine zusätzliche Funktionalität zur Verfügung stellen.
- DataInputStream und DataOutputStream: stellen Lese- und Schreib-Methoden für die Java-Basistypen und für String zur Verfügung.
- PrintStream: zum Ausdrucken von Texten und eingebauten Datentypen im Klartext. Intern arbeitet PrintStream mit Writern zur Dekodierung.



- InputStream bzw. OutputStream: einfache Ein- und Ausgabe, mehrfach überladenen Methoden `read` und `write` basierend auf abstrakter Instanz, die nur ein Byte liest oder schreibt.
- Streams, die nicht reine Bytefolgen lesen bzw. schreiben, sondern die Daten (vor-)verarbeiten, werden abgeleitet von den Oberklassen FilterInputStream und FilterOutputStream, die aber keine zusätzliche Funktionalität zur Verfügung stellen.
- DataInputStream und DataOutputStream: stellen Lese- und Schreib-Methoden für die Java-Basistypen und für String zur Verfügung.
- PrintStream: zum Ausdrucken von Texten und eingebauten Datentypen im Klartext. Intern arbeitet PrintStream mit Writern zur Dekodierung.



Beispiel:

```
import java.io.*;
public class FileCopy {
public static void main(String[] args) throws IOException {
   FileInputStream file = new FileInputStream(args[0]);
   int t;
   while(-1 != (t = file.read()))
      System.out.print((char)t);
   System.out.print("\n");
   } // end of main
} // end of FileCopy
```

- Die grundlegende Klasse für `byte`-Eingabe heißt `InputStream`.
- Diese Klasse ist abstrakt.
- Trotzdem ist `System.in` ein Objekt dieser Klasse

Nützliche Operationen:

- `public int available()` :
  gibt die Anzahl der verfügbaren Zeichen im Datenstrom zurück, die sofort ohne Blockierung gelesen werden können;
- `public int read() throws IOException` :
  liest ein Byte vom Input als `int`-Wert; ist das Ende des Stroms erreicht, wird -1 geliefert;
- `void close() throws IOException` : schließt den Eingabe-Strom.

Beispiel:

```
import java.io.*;
public class FileCopy {
public static void main(String[] args) throws IOException {
   FileInputStream file = new FileInputStream(args[0]);
   int t;
   while(-1 != (t = file.read()))
      System.out.print((char)t);
   System.out.print("\n");
   } // end of main
} // end of FileCopy
```


- Das Programm interpretiert das erste Argument in der Kommando-Zeile als Zugriffspfad auf eine Datei.
- Sukzessive werden Bytes gelesen und als `char`-Werte interpretiert wieder ausgegeben.
- Das Programm terminiert, sobald das Ende der Datei erreicht ist.

Achtung:

- `char`-Werte sind intern 16 Bit lang ...
- Ein Latin-1-Text wird aus dem Input-File auf die Ausgabe geschrieben, weil ein Byte/Latin-1-Zeichen `xxxx xxxx`
  - intern als `0000 0000 xxxx xxxx` abgespeichert und dann
  - extern als `xxxx xxxx` ausgegeben wird :-)

Beispiel:

```
import java.io.*;
public class FileCopy {
public static void main(String[] args) throws IOException {
   FileInputStream file = new FileInputStream(args[0]);
   int t;
   while(-1 != (t = file.read()))
      System.out.print((char)t);
   System.out.print("\n");
   } // end of main
} // end of FileCopy
```

Beispiel:

```
import java.io.*;
public class FileCopy {
public static void main(String[] args) throws IOException {
   FileInputStream file = new FileInputStream(args[0]);
   int t;
   while(-1 != (t = file.read()))
      System.out.print((char)t);
   System.out.print("\n");
   } // end of main
} // end of FileCopy
```


- Das Programm interpretiert das erste Argument in der Kommando-Zeile als Zugriffspfad auf eine Datei.
- Sukzessive werden Bytes gelesen und als `char`-Werte interpretiert wieder ausgegeben.
- Das Programm terminiert, sobald das Ende der Datei erreicht ist.

Achtung:

- `char`-Werte sind intern 16 Bit lang ...
- Ein Latin-1-Text wird aus dem Input-File auf die Ausgabe geschrieben, weil ein Byte/Latin-1-Zeichen `xxxx xxxx`
  - intern als `0000 0000 xxxx xxxx` abgespeichert und dann
  - extern als `xxxx xxxx` ausgegeben wird :-)

Erweiterung der Funktionalität:

- In der Klasse `DataInputStream` gibt es spezielle Lese-Methoden für jeden Basis-Typ.

Unter anderem gibt es:

- `public byte readByte() throws IOException;`
- `public char readChar() throws IOException;`
- `public int readInt() throws IOException;`
- `public double readDouble() throws IOException.`

Beispiel:

```
import java.io.*;
public class FileCopy {
public static void main(String[] args) throws IOException {
   FileInputStream file = new FileInputStream(args[0]);
   DataInputStream data = new DataInputStream(file);
   int n = file.available(); char x;
   for(int i=0; i<n/2; ++i) {
      x = data.readChar();
      System.out.print(x);
      }
   System.out.print("\n");
   } // end of main
} // end of erroneous FileCopy
```

Beispiel:

```
import java.io.*;
public class FileCopy {
public static void main(String[] args) throws IOException {
   FileInputStream file = new FileInputStream(args[0]);
   DataInputStream data = new DataInputStream(file);
   int n = file.available(); char x;
   for(int i=0; i<n/2; ++i) {
      x = data.readChar();
      System.out.print(x);
      }
   System.out.print("\n");
   } // end of main
} // end of erroneous FileCopy
```


Beispiel:

```
import java.io.*;
public class FileCopy {
public static void main(String[] args) throws IOException {
   FileInputStream file = new FileInputStream(args[0]);
   DataInputStream data = new DataInputStream(file);
   int n = file.available(); char x;
   for(int i=0; i<n/2; ++i) {
      x = data.readChar();
      System.out.print(x);
      }
   System.out.print("\n");
   } // end of main
} // end of erroneous FileCopy
```

15

Beispiel:

```
import java.io.*;
public class FileCopy {
public static void main(String[] args) throws IOException {
   FileInputStream file = new FileInputStream(args[0]);
   DataInputStream data = new DataInputStream(file);
   int n = file.available(); char x;
   for(int i=0; i<n/2; ++i) {
      x = data.readChar();
      System.out.print(x);
      }
   System.out.print("\n");
   } // end of main
} // end of erroneous FileCopy
```

15

... führt i.a. zur Ausgabe: ?????????

Der Grund:

- `readChar()` liest nicht ein Latin-1-Zeichen (i.e. 1 Byte), sondern die 16-Bit-Repräsentation eines Unicode-Zeichens ein.
- Das Unicode-Zeichen, das zwei Latin-1-Zeichen hintereinander entspricht, ist (i.a.) auf unseren Bildschirmen nicht darstellbar. Deshalb die Fragezeichen ...

Beispiel:

```
import java.io.*;
public class File2FileCopy {
public static void main(String[] args) throws IOException {
   FileInputStream fileIn = new FileInputStream(args[0]);
   FileOutputStream fileOut = new FileOutputStream(args[1]);
   int n = fileIn.available();
   for(int i=0; i<n; ++i)
      fileOut.write(fileIn.read());
   fileIn.close(); fileOut.close();
   System.out.print("\t\tDone!!!\n");
   } // end of main
} // end of File2FileCopy
```


Beispielsweise gibt es:

- `void writeByte(int x) throws IOException;`
- `void writeChar(int x) throws IOException;`
- `void writeInt(int x) throws IOException;`
- `void writeDouble(double x) throws IOException.`

Beachte:

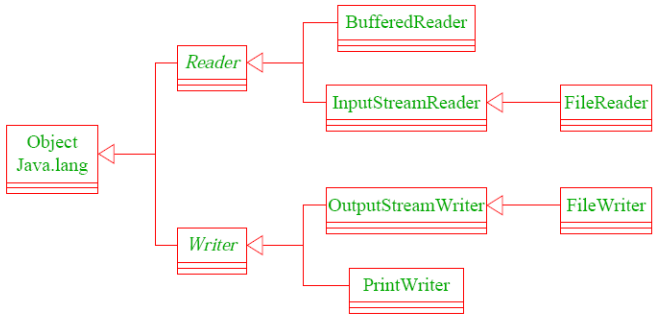
- `writeChar()` schreibt genau die Repräsentation eines Zeichens, die von `readChar()` verstanden wird, d.h. 2 Byte.

Beispiel:

```
import java.io.*;
public class Numbers {
public static void main(String[] args) throws IOException {
   FileOutputStream file = new FileOutputStream(args[0]);
   DataOutputStream data = new DataOutputStream(file);
   int n = Integer.parseInt(args[1]);
   for(int i=0; i<n; ++i)
      data.writeInt(i);
   data.close();
   System.out.print("\t\tDone!\n");
   } // end of main
} // end of Numbers
```


1.2 Textuelle Ein- und Ausgabe

Wieder erstmal eine Übersicht über hier nützliche Klassen ...



- Die Klasse `Reader` ist abstrakt.

Wichtige Operationen:

- `public boolean ready() throws IOException` : liefert `true`, sofern ein Zeichen gelesen werden kann;
- `public int read() throws IOException` : liest ein (Unicode-)Zeichen vom Input als `int`-Wert; ist das Ende des Stroms erreicht, wird -1 geliefert;
- `void close() throws IOException` : schließt den Eingabe-Strom.

Beispiel:

```
import java.io.*;
public class CountLines {
public static void main(String[] args) throws IOException {
   FileReader file = new FileReader(args[0]);
   BufferedReader buff = new BufferedReader(file);
   int n=0;
   while(null != buff.readLine())
    n++;
   buff.close();
   System.out.print("Number of Lines:\t\t"+ n);
   } // end of main
} // end of CountLines
```


- Die Objekt-Methode `readLine()` liefert `null`, wenn beim Lesen das Ende der Datei erreicht wurde.
- Das Programm zählt die Anzahl der Zeilen einer Datei
- Wieder stehen analoge Klassen für die Ausgabe zur Verfügung.
- Die grundlegende Klasse für textuelle Ausgabe heißt `Writer`.

2 Programmierfehler und ihre Behebung

(kleiner lebenspraktischer Ratgeber)

Grundsätze:

- Jeder Mensch macht Fehler
- ... insbesondere beim Programmieren.
- Läuft ein Programm, sitzt der Fehler tiefer.
- Programmierfehler sind Denkfehler.
- Um eigene Programmierfehler zu entdecken, muss nicht ein Knoten im Programm, sondern ein Knoten im Hirn gelöst werden.

2.1 Häufige Fehler und ihre Ursachen

- Das Programm terminiert nicht.

  Mögliche Gründe:

  - In einer Schleife wird die Schleifen-Variable nicht modifiziert.

```
...
String t = file.readLine();
while(t! = null)
   System.out.println(t);
...
```


- ... oder einem Objekt ohne passenden Konstruktor:

```
import java.io.*;
class A {
   public String name;
   public A (String name) {this.name = name;}
}
class AA {
   public static void main(String[] args) {
      A a = new A();
      System.out.println(a);
      System.out.println(a.name);
   }
}
```


- Ein Funktionsaufruf hat überhaupt keinen Effekt ...

```
public static void reverse (String [] a) {
   int n = a.length();
   String [] b = new String [n];
   for (int i=0; i<n; ++i) b[i] = a[n-i-1];
   a = b;
}
```

- Eine bedingte Verzweigung liefert merkwürdige Ergebnisse.

  Mögliche Gründe:

  - `equals()` mit == verwechselt?
  - Die `else`-Teile falsch organisiert?

2.2 Generelles Vorgehen zum Testen von Software

(1) Feststellen fehlerhaften Verhaltens.

Problem: Auswahl eines geeigneten Test-Scenarios

Black-Box Testing: Klassifiziere Benutzungen!
Funktionsorientiertes Testen basierend auf einer Spezifikation, ohne Kenntnis der inneren Funktionsweise des zu testenden Systems.

White-Box Testing: Klassifiziere Berechnungen!

Testfälle werden nicht aus der Spezifikation hergeleitet sondern aus dem Programm selbst. Getestet werden kann nur die Korrektheit eines Systems, nicht, ob es eine geforderte Semantik erfüllt.

Mögliche Klassifikationen:

- besuchten Programm-Punkten,
- benutzten Datenstrukturen oder Klassen
- benutzten Methoden, geworfenen Fehler-Objekten ...

Beispiel: `int find(int[] a, int x);`

Black-Box Test: Klassifizierung denkbarer Argumente:

1. `a == null;`
2. `a != null` :

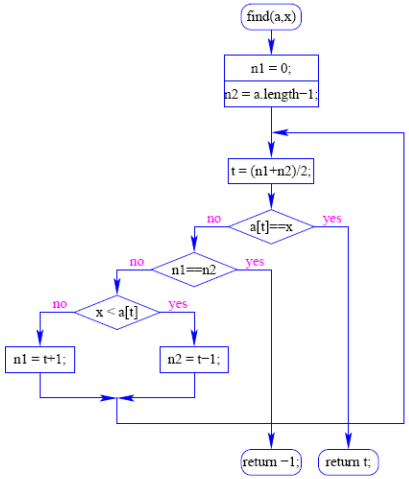
2.1. `x` kommt in `a` vor $\Longrightarrow$ `a == [42], x == 42`

2.2. `x` kommt nicht in `a` vor $\Longrightarrow$ `a == [42], x == 7`

Achtung:

Es wird nur das Aussenverhalten in Bezug auf den geforderten Funktionsumfang basierend auf einer Spezifikationen betrachtet.

White-Box Test: Klassifizierung von Berechnungen:



- Eine Menge von Test-Eingaben überdeckt ein Programm, sofern bei ihrer Ausführung sämtliche interessanten Stellen (hier: Programm-Punkte) mindestens einmal besucht werden.
- Die Funktion `find()` wird überdeckt von:

```
a == [42]; x == 42;
a == [42]; x == 7;
a == [7, 42]; x == 42;
a == [7, 42]; x == 3;
```

(2) Eingrenzen des Fehlers im Programm.

- Leicht, falls der Fehler eine nicht abgefangene `exception` auslöste
- Schwer, falls das Programm stumm in eine Endlos-Schleife gerät ... $\Longrightarrow$ Einfügen von Test-Ausgaben, Breakpoints.

(4) Verstehen des Fehlers.

Problem: Lösen des Knotens im eigenen Hirn. Oft hilft:

- Das Problem einer anderen Person schildern ...
- Eine Nacht darüber schlafen ...