

# JAVA für Einsteiger

## Streams und Dateien

© eden market

Autor: Norman Lahme

# Gliederung

1. Einleitung
2. Bildschirmausgabe und Tastatureingabe
3. Laden und Speichern von Texten
4. Laden und Speichern von Objekten

# Gliederung

1. *Einleitung*
2. Bildschirmausgabe und Tastatureingabe
3. Laden und Speichern von Texten
4. Laden und Speichern von Objekten

# Einleitung

- In diesem Modul wird betrachtet,
  - wie man Eingaben von einer beliebigen Datenquelle holt und
  - wie man Ausgaben an ein beliebiges Ziel schickt
- Quellen und Ziele können sein
  - Tastatur / Bildschirm
  - Dateien
  - Netzwerkverbindungen (hier nicht behandelt), ...

# Einleitung

- Der Mechanismus für das Senden und Empfangen von Daten ist unabhängig von der Art der Quelle und des Ziels

# Einleitung

- Für diese Abstraktion bietet Java die folgenden Klassen
  - InputStream: Lesen von Bytes (Quelle egal)
  - OutputStream: Schreiben von Bytes (Ziel egal)
  - Reader: Lesen von Zeichen (Unicode)
  - Writer: Schreiben von Zeichen (Unicode)

# Einleitung

- Von diesen vier Klassen gibt es insgesamt über 60 verschiedene Subklassen, die jeweils einen anderen Streamtypen beschreiben
- Im folgenden sollen die wichtigsten Klassen rein anwendungsorientiert betrachtet werden

# Gliederung

1. Einleitung
2. **Bildschirmausgabe und Tastatureingabe**
3. Laden und Speichern von Texten
4. Laden und Speichern von Objekten

# BildschirmAusgabe

- Für die BildschirmAusgabe wurde bereits die Methode `System.out.println` vorgestellt
- Nun soll diese näher betrachtet werden

# BildschirmAusgabe

- Die Klasse System besitzt eine Klassenvariable out, die wiederum eine Referenz auf ein Objekt der Klasse PrintStream besitzt
- Diese Klasse ist indirekt Subklasse von OutputStream – stellt also einen Ausgabestrom dar

# BildschirmAusgabe

Wesentliche Methoden von `PrintStream`:

- `public void print(Object o)`
- `public void println(Object o)` → mit „Return“
- `public void print(int i)`
- `public void println(int i)` → mit „Return“
- ... (für weitere Datentypen)

# Beispiel

```
public class Test {  
    public static void main (String [] args) {  
        System.out.println("Textausgabe");  
        System.out.print(200);  
        System.out.println(0);  
    }  
}
```

# Tastatureingabe

- Die Klassenvariable in der Klasse System beinhaltet ein Objekt der Klasse BufferedInputStream, um Bytes von der Tastatur einzulesen

# Tastatureingabe

- Um komfortabel Unicode- Zeichen einlesen zu können, ist ein Objekt der Klasse `BufferedReader` zu erzeugen:

```
BufferedReader in = new BufferedReader(  
    new InputStreamReader(System.in));
```

# Tastatureingabe

Wesentliche Methode von `BufferedReader`:

- `public String readLine()` throws `IOException`
  - null, falls Eingabestrom zuende
- `public void close()` throws `IOException`
  - nur bei Dateien relevant (s. nächstes Kapitel)

# Beispiel

```
import java.io.*;

public class Test {
    public static void main(String[] args) {
        try {
            BufferedReader in = new BufferedReader(
                new InputStreamReader(System.in));
            String input = in.readLine();
            System.out.println("Ihre Eingabe: " + input);
        } catch (Exception e) {}
    }
}
```

# Übung 1 Aufgabe 1

# Gliederung

1. Einleitung
2. Bildschirmausgabe und Tastatureingabe
- 3. *Laden und Speichern von Texten***
4. Laden und Speichern von Objekten

# Text aus einer Datei einlesen

- Für das Einlesen von Texten aus Dateien bietet Java die Klasse `FileReader`
- Konstruktor:
  - `public FileReader(String filename)`

# Text aus einer Datei einlesen

- Für ein komfortableres Einlesen bietet es sich jedoch wieder an, aus einem FileReader-Objekt ein BufferedReader-Objekt zu erzeugen:  

```
BufferedReader in = new BufferedReader(  
    new FileReader("filename"));
```
- Auf diesem kann dann die bekannte Methode `readLine` aufgerufen werden

# Beispiel

```
import java.io.*;
public class Test {
    public static void main(String[] args) {
        try {
            BufferedReader in = new BufferedReader(
                new FileReader("test.txt"));
            String str;
            while ((str = in.readLine()) != null)
                System.out.println(str);
            in.close();
        } catch (Exception e) {}
    }
}
```

# Text in eine Datei schreiben

- Analog zur Klasse FileReader existiert eine Klasse FileWriter
- Konstruktoren:
  - `public FileWriter(String filename, boolean append) throws IOException;`
    - `append = true` → neuen Text hinten anhängen
    - `append = false` → Datei überschreiben

# Text in eine Datei schreiben

- Auch hier ist es sinnvoll, zum `FileWriter`- Objekt ein `BufferedWriter`- Objekt zu erzeugen:

```
BufferedWriter out = new BufferedWriter(  
    new FileWriter("filename",false));
```

- Dies hat zudem den Vorteil, dass Schreibvorgänge zunächst auf einem Puffer operieren, dessen Inhalt in einem Schreibvorgang auf den Datenträger übertragen werden kann

# Text in eine Datei schreiben

Wesentliche Methoden von `BufferedWriter`:

- `public void write(String s) throws IOException`
- `public void newLine() throws IOException`
  - Fügt einen Zeilenumbruch ein
- `public void flush() throws IOException`
  - Schreibt den Puffer-Inhalt in die Datei
- `public void close() throws IOException`

# Beispiel

```
import java.io.*;
public class Test {
    public static void main(String[] args) {
        try {
            BufferedW riter out =new BufferedW riter(
                new FileW riter("test.txt", false));
            out.w rite("Z eile 1");
            out.new Line();
            out.w rite("Z eile 2");
            out.flush();
            out.close();
        } catch (E xception e) {}
    }
}
```

# Gliederung

1. Einleitung
2. Bildschirmausgabe und Tastatureingabe
3. Laden und Speichern von Texten
4. *Laden und Speichern von Objekten*

# Objekte in eine Datei schreiben

- Java bietet sogar die Möglichkeit, komplexe Objekte zu speichern
- Hierzu wird das zu speichernde Objekt in einen Bytestrom konvertiert (Serialisierung)
  - Der Programmierer braucht jedoch nicht zu wissen, wie dies geschieht

# Objekte in eine Datei schreiben

- Anforderung an die Klasse des zu speichernden Objektes:
  - Sie muss das Interface `java.io.Serializable` implementieren
  - Sollen bei manchen Attributen die Attributwerte nicht mit abgespeichert werden, so sind die Attribute als transient zu deklarieren
    - `private transient Frame theMainFrame;`

# Objekte in eine Datei schreiben

- Zur Speicherung binärer Daten existiert die Klasse `FileOutputStream`
- Konstruktoren:
  - `public FileOutputStream(String filename, boolean append)`  
throws `FileNotFoundException`;
    - `append = true` → neues Objekt hinten anhängen
    - `append = false` → Datei überschreiben

# Objekte in eine Datei schreiben

- Den Serialisierungsvorgang kapselt die Klasse `ObjectOutputStream`
- Ein Objekt dieser Klasse ist auf Basis des `FileOutputStream`- Objektes zu erzeugen:

```
ObjectOutputStream out =  
    new ObjectOutputStream(  
        new FileOutputStream("filename", false));
```

# Objekte in eine Datei schreiben

Wesentliche Methoden von `ObjectOutputStream`:

- `public void writeObject(Object o) throws IOException`
- `public void flush() throws IOException`
  - Schreibt den Puffer-Inhalt in die Datei
- `public void close() throws IOException`

## Beispiel (1/2)

```
import java.io.*;
public class Autor implements Serializable {
    private String name;
    private String vorname;
    public Autor(String name, String vorname) {
        this.name = name;
        this.vorname = vorname;
    }
}
```

## Beispiel (2/2)

```
import java.io.*;
public class Test {
    public static void main(String[] args) {
        Autor a = new Autor("Monson-Haefel", "Richard");
        try {
            ObjectOutputStream out = new ObjectOutputStream (
                new FileOutputStream("test.obj"));
            out.writeObject(a);
            out.flush();
            out.close();
        } catch (Exception e) {}
    }
}
```

# Objekte aus einer Datei lesen

- Um ein Objekt aus einer Datei zu lesen, ist aus dem dort abgelegte Bytestrom ein Objekt zu erzeugen (Deserialisierung)
- Auch hier braucht der Programmierer braucht nicht wissen, wie dies geschieht

# Objekte aus einer Datei lesen

- Zum Einlesen binärer Daten existiert die Klasse `FileInputStream`
- Konstruktoren:
  - `public FileInputStream(String filename) throws FileNotFoundException;`

# Objekte aus einer Datei lesen

- Den Deserialisierungsvorgang kapselt die Klasse `ObjectInputStream`
- Ein Objekt dieser Klasse ist auf Basis des `FileInputStream`- Objektes zu erzeugen:

```
ObjectInputStream in =  
    new ObjectInputStream(  
        new FileInputStream("filename"));
```

# Objekte aus einer Datei lesen

Wesentliche Methoden von `ObjectInputStream`:

- `public Object readObject() throws OptionalDataException, ClassNotFoundException, IOException`
- `public void close() throws IOException`

# Beispiel

```
import java.io.*;
public class Test {
    public static void main(String[] args) {
        Autor a;
        try {
            ObjectInputStream in = new ObjectInputStream(
                new FileInputStream("test.obj"));
            a = (Autor) in.readObject();
            in.close();
            System.out.println(a);
        } catch (Exception e) {}
    }
}
```

# Anmerkung

- Wird zwischen dem Speichern eines Objektes und dem Laden desselben eine Änderung an der Klassendefinition vorgenommen, so scheitert der Ladeversuch

# Übung 1 Aufgabe 2