Java im Softwarepraktikum

Literatur

[AG98] Ken Arnold and James Gosling.
The Java Programming Language.
Addison-Wesley, Reading, MA, USA, second edition, 1998.

[Fla99] David Flanagan.
Java in a Nutshell: A Desktop Quick Reference.
O'Reilly, third edition, 1999.

Tutorial: `http://java.sun.com/docs/books/tutorial/`

API-Doku: `file:/home/dbs/share/jdk1.3/docs/api/index.html`

Hello world!

```java
/**
 * This class implements the famous hello world program.
 */
public class HelloWorld {

    public static void main(String[] args) {
        // print string to stdout
        System.out.println("Hello World!");
    }
}
```


Übersetzen und Ausführen von Programmen

Java 2/JDK 1.3 für Sun Solaris:

- Umgebungsvariablen:
  - PATH → `/home/dbs/share/jdk1.3/bin`
  - CLASSPATH → Verzeichnis von weiteren `.class`–Dateien
- `javac HelloWorld.java` (erzeugt `HelloWorld.class`)
- `java HelloWorld` (führt die Methode main in der Klasse `HelloWorld` aus)
- Schnelle Compiler-Alternative: `/home/dbs/bin/jikes`

(GNU–)Makefile

Dateiname: 'Makefile', Aufruf mit: 'gmake'

```
JAVAC = javac
JAVACFLAGS = -depend

SRCFILES := $(wildcard *.java)
CLASSFILES := $(SRCFILES:.java=.class)

# compile all java files in current directory
all: $(CLASSFILES)

# compile java file to class file
%.class: %.java
	$(JAVAC) $(JAVACFLAGS) $^
```

Modularisierung durch Packages

- Kleinster Baustein ist die Klasse
- Variablen und Methoden sind ausschließlich in Klassen enthalten (keine globalen Variablen!)
- Klassen werden in packages zusammengefaßt
- Packages bilden einen hierarchischen Namensraum, z.B. `java.lang.String`
- Package–Hierarchie wird auf Verzeichnisstrukturen abgebildet, z.B. `.../java/lang/String.class`
- Weltweit eindeutige Namen durch Verwendung von domain–Namen

Packages – Beispiel

```
package DE.uni_bremen.swp2001.gruppe1;

import java.util.Hashtable;
import java.io.*;
import DE.uni_bremen.swp2001.gruppe2.List;

/** stack implementation using a list */
public class Stack {
    List list;
    ...
}
```


Testen von Klassen

- Modultest (Klasse Stack enthält `main`–Methode):
  `java DE.uni_bremen.swp2001.gruppe1.Stack`

```
public class Stack {
    public void push(Object o ) { ... }
    public Object pop () { ... }

    public static void main(String[] args) {
        Stack stack = new Stack();
        String str = "aString";
        stack.push(str);
        if (! str.equals(stack.pop()))
            System.err.println("pop != push");
    }
}
```


Datentypen

- Primitive Datentypen: boolean, char, byte, short, int, long, float, double
- Wrapper–Klassen: Boolean, Character, Integer, Double, . . .
  (für Objektreferenzen und Operationen auf Datentyp)
- Referenztypen:
  - Objekttypen: `Button b; Stack expr; DataInputStream in;`
  - Felder: `int[] score; Hashtable[][] assoc;`
- Referenztypen werden dynamisch angelegt und automatisch freigegeben:
  - `b = new Button("Ok");`
  - `score = new int[10];`

Konstanten

- Konstantendeklaration mit **static final**

```
public class Constants {
    public static final int N = 8;
    public static final char BLANK = ' ';
    public static final String MO = "Monday";
    public static final int [] DAYS_IN_MONTH =
            { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
}
```


Felder

```
class Chess {
    static final int N = 8;
    static final int PIECE_EMPTY = 0;

    public int [][] getNewBoard() {
        int [][] board = new int[N][N];

        for (int i = 0; i < N; i++)
            for (int j = 0; j < N; j++)
                board[i][j] = PIECE_EMPTY;

        return board;
    }
}
```


Strings I

- Zeichensatz ist Unicode (16-bit, enthält Latin-1)
- `String`–Objekte werden durch Java besonders unterstützt

```
String name = "James Gosling";
name += " and Ken Arnold";
System.out.println(name + 2);
```

- implizite Typumwandlung für String–Konkatenation: +
- Überschreiben der `toString()`–Methode

Strings II

- String → int: `Integer.parseInt(String s)`
- Test auf Gleichheit: `if ( s1.equals(s2) ) { ... }`
- == testet auf Identität!
- Strings sind *read–only* → `StringBuffer`

Klassen und Objekte

```
class Point {
    private double x, y;

    public Point(double x1, double y1) {
        x = x1;
        y = y1;
    }

    public double getX() {
        return x;
    }
}
```

- Typ eines Objektes ist die Klasse
- Klassen beinhalten *Felder* und *Methoden*
- Objektinstanzen werden durch `new` erzeugt
- Variablen sind Referenz auf Objekt oder `null`

```
Point origin = new Point(0,0);
System.out.println(origin.getX());
```


Sichtbarkeit / Information hiding

- `public`, `private`, `protected` und default für Felder und Methoden:
  1. `public`: überall sichtbar
  2. `private`: nur in der Klasse
  3. `protected`: in der Klasse, ihren Unterklassen *und* im selben package
  4. default: innerhalb eines package
- Klassen implementieren abstrakten Datentyp: Felder immer `private` oder `protected`, Zugriff nur über `public`–Methoden

Weitere Modifizierer

- `final`: Klassen können nicht vererbt, Methoden nicht überschrieben werden (Sicherheit und Effizienz), z.B.: `public final class Socket { ... }`
- `static`: Klassenvariablen und -methoden, unabhängig von Instanzen

  `public static Point origin = new Point(0, 0);`
- `abstract`: Methoden müssen in abgeleiteten Klassen implementiert werden
- `native`: Methodenrumpf ist extern definiert (z.B. in C)
- `synchronized`: wechselseitiger Ausschluß bei gleichzeitigem Zugriff (durch Threads) auf Objekt oder Klasse

Initialisierung von Objekten

- durch Konstruktor oder einfache Ausdrücke

```
class Calendar {
    private static int [] daysInMonth =
            { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

    private Map holidayNames = new HashMap();

    public Calendar(int month, int year) {
        holidayNames.put(new Integer(1), "Neujahr");
        ...
    }
}
```

Methoden

- Parameterübergabe immer *call by value*
- bei Referenztypen: Seiteneffekte auf Objekt möglich

```
public void setFocus(Button b) {
    b.setBackColor(Colors.RED);
}
```

- *Overloading*: Methoden gleichen Namens haben unterschiedliche Parameter

```
public void setFocus(Button b, int col) {
    b.setBackColor(col);
}

...
setFocus(btn);
setFocus(btn, Colors.GREEN);
```


This

- Referenz auf aktuelles Objekt: `this`

```
public boolean samePoint(Point p2) {
    return this == p2;
}
```

- Expliziter Konstruktoraufruf:

```
public Point(double x, double y) {
    fX = x;
    fY = y;
}

public Point() {
    this(0, 0);
}
```


Vererbung

```
class ColorPoint extends Point {
    private int fColor;

    public ColorPoint(double x, double y,
                      int color) {
        super(x, y);
        fColor = color;
    }

    public int getColor() {
        return fColor;
    }
}
```

- Unterklasse erbt Felder und Methoden der Oberklasse
- (Indirekter) Vorgänger *aller* Klassen ist `Object`
- Einfachvererbung in bezug auf Implementierungen
- "Mehrfachvererbung" durch Interfaces

Test und Umwandlung von Klassentypen

- Test auf Gleichheit (Überschreiben der Methode in Klasse `Object`):

```
public boolean equals(Object obj) {
    if ((obj != null) && (obj instanceof Point)) {
        Point p = (Point) obj;
        return (this.x == p.x) && (this.y == p.y);
    }
    return false;
}
```

Abstrakte Klassen

- Definition eines gemeinsamen Protokolls,
  gibt allgemeine Teile für Schnittstelle und Implementierung vor

- abstrakte Klassen können nicht instanziiert werden,
  abstrakte Methoden müssen überschrieben werden.

```
abstract class GenericPoint {
    private double x, y;

    abstract public void moveTo(GenericPoint p);
}
```


Interfaces I

```
interface Drawable {
    void drawOn(Window w);
}
```

- Definition von Schnittstellen, Implementierung bleibt offen
- Enthält nur Methodendeklarationen und Konstanten
- Vererbung durch `extends` möglich
- Klassen können beliebig viele Interfaces implementieren

Interfaces II

```
class ColorPoint extends Point implements Drawable {
    ...
    // implement Drawable interface
    public void drawOn(Window w) {
        w.drawPoint(x, y, color);
    }
}
```

- Interface–Typen und Klassentypen sind gleichberechtigt

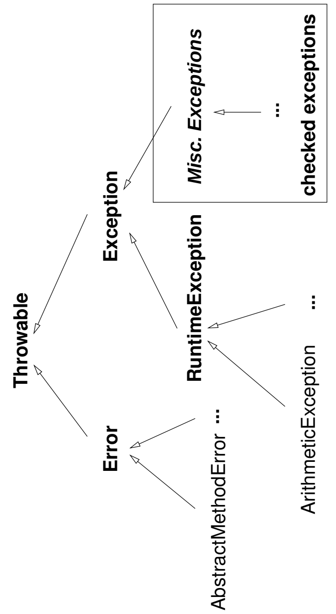
```
Drawable d = new ColorPoint(10, 20, 0);
```


Exceptions

```
try {
    // open file
    // read file
    // process file
} catch(IOException e) {
    // handle error
} finally {
    // close file
}
```

- trennen Fehlerbehandlung von Algorithmen
- Auftreten einer Exception ändert automatisch den Kontrollfluß
- Fehler können gezielt behandelt werden (Exception–Objekte)
- Fehlerbehandlung kann vom Compiler erzwungen werden

Exception–Klassen



- *checked exceptions* müssen deklariert oder abgefangen werden
- *unchecked exceptions* können in Methoden ohne Deklaration auftreten
- Exception–Objekte für Anwendungen werden von der Klasse `Exception` abgeleitet

```
class InvalidWeekDayException extends Exception {
  ...
}
```


Exceptions erzeugen

- Exceptions werden mittels `throw` ausgelöst
- Methoden müssen deklarieren, welche (checked) Exceptions erzeugt werden können

```
public String getNameOfDay(int d) throws InvalidWeekDayException {
    if ( d < 0 || d > 6 )
        throw new InvalidWeekDayException("Day = " + d);
    ...
}
```


Exceptions behandeln

```
String name;
int d;
try {
    d = ...  // read number of day from input
    name = getNameOfDay(d);
    System.out.println("Name of day is " + name);
} catch(InvalidWeekDayException e) {
    System.err.println("Your input is an invalid day number. " +
        e.getMessage());
} finally {
    System.out.println("Thanks for your input");
}
```


I/O

- Klassen im package `java.io`
- Streams abstrahieren von Details des Betriebssystems
- arbeiten einheitlich auf Dateien, Strings, ByteArrays, . . .
- Filter erweitern Streams um spezielle Eigenschaften (z.B. `BufferedOutputStream`, `LineNumberInputStream`)

Dateien kopieren

```java
import java.io.*;

class FileCopy {
  public static void main(String[] argv) {
    FileInputStream in = null;
    FileOutputStream out = null;
    int c;
    try {
      in = new FileInputStream(argv[0]);
      out = new FileOutputStream(argv[1]);

      // copy each character
      while ( (c = in.read()) >= 0 )
        out.write(c);

    } catch (IOException e) {
      System.err.println("copy error: " +
                         e.getMessage());
    } finally {
      try {
        if ( in != null ) in.close();
        if ( out != null ) out.close();
      } catch (IOException e) {
        /* empty */
      }
    }
  }
}
```


Collection–Klassen

- Interfaces im package `java.util`: `Collection`, `Set`, `List`, ...
- Verschiedene Implementierungen, z.B.: `HashSet`, `TreeSet`, ...
- heterogene Kollektionen möglich, Elemente müssen nur Untertyp von `Object` sein (Klassen für Basistypen verwenden: Integer, Boolean, ...)
- `ArrayList`:
  - Größe zur Laufzeit änderbar
  - nicht so sicher wie Felder, da Elemente nur als Typ `Object` enthalten sind (Typumwandlung nötig):

```java
List polygon = new ArrayList();
Point p = (Point) polygon.get(12);
```


Iterator interface

- Einheitlicher Zugriff auf Elemente beliebiger Collections
- Alle Collections bieten Iterator über ihre Elemente `Iterator iterator()`:
  boolean hasNext();
  Object next();
  void remove();
- Abstrahiert von Details der Iteration

```java
Iterator graphIter = graph.iterator();
while ( graphIter.hasNext() ) {
    Node n = (Node) graphIter.next();
    // process node n
}
```


Wie geht's weiter?

- Vertraut machen mit den Java Standard-Bibliotheken:
  `file:/home/dbs/share/jdk1.3/docs/api/index.html`
- Programmcode von Profis anschauen:
  `file:/home/dbs/share/jdk1.3/src`
- Übungen mitmachen, Beispiele ausprobieren