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!

```
1 /**
2  * This class implements the famous hello world program.
3  */
4 public class HelloWorld {
5
6     public static void main(String[] args) {
7         // print string to stdout
8         System.out.println("Hello World!");
9     }
10 }
```

Übersetzen und Ausführen von Programmen

Java 2/JDK 1.3 für Sun Solaris:

- Umgebungsvariablen:
  - PATH $\rightarrow$ `/home/dbs/share/jdk1.3/bin`
  - CLASSPATH $\rightarrow$ 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

```
1  package DE.uni_bremen.swp2001.gruppe1;
2
3  import java.util.Hashtable;
4  import java.io.*;
5  import DE.uni_bremen.swp2001.gruppe2.List;
6
7  /** stack implementation using a list */
8  public class Stack {
9        List list;
10       ...
11 }
```

Testen von Klassen

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

```
1  public class Stack {
2      public void push(Object o ) { ... }
3      public Object pop () { ... }
4
5      public static void main(String[] args) {
6          Stack stack = new Stack();
7          String str = "aString";
8          stack.push(str);
9          if ( ! str.equals(stack.pop()) )
10             System.err.println("pop != push");
11     }
12 }
```

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**

```
1 public class Constants {
2       public static final int N = 8;
3       public static final char BLANK = ' ';
4       public static final String MO = "Monday";
5       public static final int [] DAYS_IN_MONTH =
6               { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
7 }
```

Felder

```
1  class Chess {
2      static final int N = 8;
3      static final int PIECE_EMPTY = 0;
4
5      public int [][] getNewBoard() {
6          int [][] board = new int[N][N];
7
8          for (int i = 0; i < N; i++)
9              for (int j = 0; j < N; j++)
10                 board[i][j] = PIECE_EMPTY;
11
12         return board;
13     }
14 }
```

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

Klassen und Objekte

```
1  class Point {
2      private double x, y;
3
4      public Point(double x1, double y1) {
5          x = x1;
6          y = y1;
7      }
8
9      public double getX() {
10         return x;
11     }
12 }
```

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

```
1 Point origin = new Point(0,0);
2 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

```
1  class Calendar {
2      private static int [] daysInMonth =
3              { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
4
5      private Map holidayNames = new HashMap();
6
7      public Calendar(int month, int year) {
8          holidayNames.put(new Integer(1), ”Neujahr”);
9              ...
10     }
11 }
```

Methoden

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

```
1 public void setFocus(Button b) {
2       b.setBackColor(Colors.RED);
3 }
```

- *Overloading*: Methoden gleichen Namens haben unterschiedliche Parameter

```
1 public void setFocus(Button b, int col) {
2       b.setBackColor(col);
3 }
4
5 ...
6 setFocus(btn);
7 setFocus(btn, Colors.GREEN);
```

This

- Referenz auf aktuelles Objekt: `this`

```
1 public boolean samePoint(Point p2) {
2       return this == p2;
3 }
```

- Expliziter Konstruktoraufruf:

```
1 public Point(double x, double y) {
2       fX = x;
3       fY = y;
4 }
5
6 public Point() {
7       this (0, 0);
8 }
```

Vererbung

```
1  class ColorPoint extends Point {
2      private int fColor;
3
4      public ColorPoint(double x, double y,
5                        int color) {
6          super(x, y);
7          fColor = color;
8      }
9
10     public int getColor() {
11         return fColor;
12     }
13 }
```

- 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`):

```
1 public boolean equals(Object obj) {
2       if  ( ( obj  != null) && (obj  instanceof Point) ) {
3            Point  p = (Point)  obj;
4            return (this.x == p.x) && (this.y == p.y);
5       }
6       return false;
7 }
```

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.

```
1 abstract class GenericPoint {
2       private double x, y;
3
4       abstract public void moveTo(GenericPoint p);
5 }
```

Interfaces I

```
1 interface Drawable {
2       void drawOn(Window w);
3 }
```

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

Interfaces II

```
1 class ColorPoint extends Point implements Drawable {
2       ...
3       // implement Drawable interface
4       public void drawOn(Window w) {
5           w.drawPoint(x, y , color );
6       }
7 }
```

- Interface–Typen und Klassentypen sind gleichberechtigt

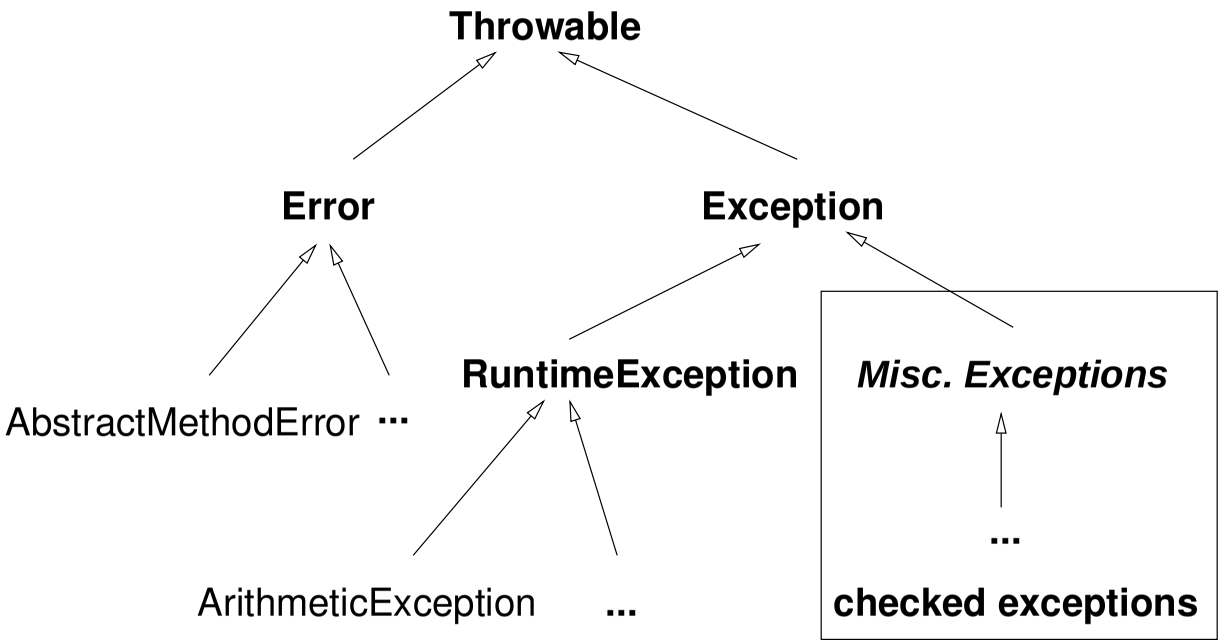
  Drawable d = **new** ColorPoint(10, 20, 0);

Exceptions

```
1 try {
2       // open file
3       // read file
4       // process file
5 } catch(IOException e) {
6       // handle error
7 } finally {
8       // close file
9 }
```

- 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

```
1 class InvalidWeekDayException extends Exception {
2  ...
3 }
```

Exceptions erzeugen

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

```
1 public String getNameOfDay(int d) throws InvalidWeekDayException {
2       if ( d < 0 || d > 6 )
3           throw new InvalidWeekDayException("Day = " + d);
4       ...
5 }
```

Exceptions behandeln

```
1  String name;
2  int d;
3  try {
4      d = ...   // read number of day from input
5      name = getNameOfDay(d);
6      System.out.println("Name of day is " + name);
7  } catch(InvalidWeekDayException e) {
8      System.err.println("Your input is an invalid day number. " +
9          e.getMessage());
10 } finally {
11     System.out.println("Thanks for your input");
12 }
```

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

```
1  import java.io.*;
2
3  class FileCopy {
4      public static void main(String[] argv) {
5          FileInputStream in = null;
6          FileOutputStream out = null;
7          int c;
8          try {
9              in = new FileInputStream(argv[0]);
10             out = new FileOutputStream(argv[1]);
11
12             // copy each character
13             while ((c = in.read()) >= 0)
14                 out.write(c);
15
16         } catch (IOException e) {
17             System.err.println("copy error: " +
18                                e.getMessage());
19         } finally {
20             try {
21                 if (in != null) in.close();
22                 if (out != null) out.close();
23             } catch (IOException e) {
24                 /* empty */
25             }
26         }
27     }
28 }
```

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):

```
1 List  polygon = new ArrayList();
2 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

```
1 Iterator graphIter = graph.iterator();
2 while ( graphIter.hasNext() ) {
3       Node n = (Node) graphIter.next();
4       // process node n
5 }
```

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