Einführung in die Programmiertechnik

Java

Warum Java?

- Portabilität: Programme können unverändert auf verschiedenen Systemen ausgeführt werden
  - ohne Neuübersetzung
- Integration in Webbrowser: Applets
- “Modernisiertes C++”
  - einfachere Syntax
  - Umschulung von C++-Programmierern auf Java leicht möglich
- Schul-/Universitätssprache
  - gleiche Konzepte wie Pascal, Modula, Oberon
  - +OOP
  - +praxisrelevant
- Implementierung/Entwicklungsumgebung kostenlos
  - NetBeans, Eclipse, BlueJ

Geschichte

- Seit 1991 unter Leitung von James Gosling entwickelt
  - ursprünglich OAK (Object Application Kernel, nach der Eiche vor Goslings Büro)
  - Sprache für eingebettete Systeme
- Seit 1994: Integration in Webbrowser
  - Java: Hauptinsel Indonesiens; amer. synonym für guten Kaffee
  - “Oak” war eingetragenes Warenzeichen
- Seit 1995: Java Development Kit (JDK) verfügbar
  - Java 1.0: 1996
  - Java 1.1: 1997 (inner classes)
  - Java 1.2: 1998, Umbenennung in Java 2
    - Swing, Reflection, JIT, strictfp
  - Java 1.3: 2000 (HotSpot VM, RMI auf Basis von CORBA)
  - Java 1.4: 2002 (assert, reguläre Ausdrücke, XML, ...)
  - Java 5: 29. 9. 2004 (generics, autoboxing, annotations, Aufzählungstypen, ...)
  - Java 6: 11. 12. 2006 (OS-spezifische Funktionen, bessere Werkzeugintegration, ...)

Lexik

- Basiert auf Unicode
  - alle Schlüsselwörter, alle Klassen, Funktionen der Standardbibliothek beschränken sich auf ASCII
  - Unicode-Escapes im Quelltext: \uXXXX
- Bezeichnersyntax: Beginnend mit JavaLetter, dann JavaLetterOrDigit
  - JavaLetter: falls Character.isJavaIdentifierStart(code) gilt
    - ➡ isLetter(code) (Ll, Lu, Lt, Lm, Lo) oder
    - ➡ getType(code)==LETTER_NUMBER (Nl) oder
    - ➡ code ist Währungssymbol (Sc?) oder
    - ➡ code ist “connecting punctuation character” (Pc?)
  - JavaLetterOrDigit: falls Character.isJavaIdentifierPart(code) gilt
    - ➡ isJavaIdentifierStart(code) (?) oder
    - ➡ code ist Ziffer (Nd?) oder
    - ➡ code ist “combining mark” (Mc, Mn) oder
    - ➡ isIdentifierIgnorable(code)

Lexik (2)

- Zeilenende: CR (U+000D), LF (U+000A), oder CR+LF
  - beendet //-Kommentar
- Leerzeichen: SP (U+0020), HT (Tab, U+0009), FF (U +000C), Zeilenende
- Kommentare:
  - /* Blockkommentare */
    - /** JavaDoc-Kommentar */
  - // Zeilenendekommentare
- Trennzeichen: ( ) { } [ ] ; , .
- Operatoren: = > < ! ~ ?: == <= >= != && || ++ -- + - * / & | ^ % << >> >>> += -= *= /= &= |= ^= %= <<= >>= >>>=

Lexik (3): Schlüsselwörter

```
abstract    continue    for          new          switch
assert      default     if           package      synchronized
boolean     do          goto         private      this
break       double      implements   protected    throw
byte        else        import       public       throws
case        enum        instanceof   return       transient
catch       extends     int          short        try
char        final       interface    static       void
class       finally     long         strictfp     volatile
const       float       native       super        while
```

Lexik (4): Literale

- Ganze Zahlen: dezimal, hexadezimal, oktal, long
  - 42, 0xFF, 007, 42L, 0xC0B0L
- Gleitkommaliterale: float, double, hexadezimal
  - 1e1f 2.7 .3f 0f 3.14d 6.022137e+23f
  - 0x10p3, 0XAP0D
- Zeichen: ‘c’ ‘ ‘ ‘ж’ ‘☂’
  - Escape-Sequenzen: \b \t \n \f \r \” \’ \\ \007 (\uXXXX)
- Zeichenketten: “Hallo”
  - String muss vor Zeilenende enden
  - Zusammenfügen eines mehrzeiligen Strings: Addition
    “Hallo, “+
    “Welt”
- boolean: true, false
- Objekte: null

Datentypen

- Statische Typüberprüfung: jeder Ausdruck hat Typ
  - Compilerfehler bei Typverletzung
- Einfache Datentypen (Wertetypen):
  - boolean, char, byte, short, int, long, float, double
  - Wertebereiche plattformunabhängig:
    - char: Unicode (16 bit)
    - byte: 8 bit
    - short: 16 bit
    - int: 32 bit
    - long: 64 bit
    - float: 32 bit
    - double: 64 bit
  - Zuweisung, Parameterübergabe etc. erzeugt Kopien dieser Werte
- Referenz-Datentypen:
  - Felder (arrays), Klassen (classes), Schnittstellen (interfaces)
  - Java 5: Aufzählungstypen (enumerations)

Variablen

- Variablendeklaration: Erklärung von Variablentyp und Variablenname, optional Initialisierung
  - int x;
  - double y = 7.0;
  - boolean fertig = false;
- Deklaration in Funktionen vor erster Verwendung
  - Objektkomponenten (members) in beliebiger Reihenfolge deklarierbar
- Initialisierung stets vor erster Verwendung
  - Objektfelder, Arrayelemente werden automatisch null-initialisiert
    - 0, 0L, 0.0f, 0.0d, \u0000, false, null
  - lokale Variablen müssen explizit einen Wert erhalten
    - statische Überprüfung der Initialisierung

Referenztypen

- Variablen halten nicht den Wert selbst, sondern nur eine Referenz auf den Wert
- Referenzwertige Variablen sind entweder **null** oder verweisen auf ein Objekt
  - Zuweisung aus anderer Referenzvariable
  - Ergebnis eines Funktionsrufs
  - Zuweisung aus Operator **new**

Arrays

- Sei T ein Typ, dann ist T[] der dazu gehörige Array-Typ
- Erzeugung eines Arrays:
  - int[] var = new int[100];
    - null-initialisiert
    - Zahl der Elemente kann dynamisch festgelegt werden
    - Alternative Variablensyntax: int var[]
  - int[] var = { 2, 3, 5, 7, 11 };
    - Initialisierung durch Aufzählung der Werte
  - Größe nachträglich nicht änderbar
- Typkompatibilität: Zuweisung möglich unabhängig von Zahl der Elemente

```
int[] a = { 1, 2, 3};
int[] b = new int[100];
a = b;
```

Arrays (2)

- Typkompatibilität: Zuweisung möglich, wenn Elementetyp der gleiche ist
  - Falls Elementetyp Referenztyp ist: Zuweisung auch möglich, wenn Elementetyp zuweisbar ist
    String[] x = { “Hallo”, “Welt”};
    Object[] y = x;
- Elementzugriff: f[index]
  - Indizierung beginnt bei 0
  - Zahl der Elemente: f.length (keine! Funktion)

```
int[] squares = new int[100];
for(int i = 0; i < squares.length; i++)
    squares[i] = i*i;
```

- Mehrdimensionale Arrays: T[][] ist Array aus T[]
  - int[][] int4Bsp = new int[42][42];
  - int[][] jagged = { {1}, {1,1}, {1, 2, 1}, {1, 3, 3, 1}};

Methoden

- methode ::= attribut* rtyp name
  ‘(‘ [parameterliste] ‘)’ [throws]
  körper
- attribut ::= ‘public’ | ‘protected’ | ‘private’
  | ‘abstract’ | ‘static’ | ‘final’
  | ‘synchronized’ | ‘native’ | ‘strictfp’
- rtype ::= ‘void’ | typ
- parameterliste ::= (parameter ‘,’)* parameter
  - eigentlich: Spezialsyntax für letzten Parameter (...)
- parameter ::= [‘final’] typ name
- körper ::= block | ‘;’
- block ::= ‘{‘ anweisung* ‘}’

Methoden (2)

```
int fib(int n) {
    if (n <= 1)
        return 1;
    return fib(n-1) + fib(n-2);
}
```

- Typ des Ausdrucks in der return-Anweisung muss kompatibel mit Rückgabetyp sein
- Ausdruck hinter return ist erforderlich gdw. Rückgabetyp nicht void ist.

Klassen

- Vereinfachte Syntax:

  klasse ::= ‘class’ name ‘{‘ (feld | methode) * ‘}’

  - eigentlich: + Klassenattribute, + Konstruktoren, + Klasseninitialisierung, + Exemplarinitialisierung

  feld ::= attribut* typ felddekl (‘,’ felddekl) * ‘;’

  felddekl ::= name [‘=’ wert]

  attribut ::= ‘public’ | ‘protected’ | ‘private’ | ‘static’
  | ‘final’ | ‘transient’ | ‘volatile’

```
class Punkt {
    int x;
    int y;
}
```

Klassen (2)

```
class Kreis {
    double radius;
    Punkt mitte;
    double fläche() {
        return Math.PI * radius * radius;
        // alternativ: return Math.PI * this.radius * this.radius;
    }
}
```

Klassen (3)

- Exemplarerzeugung: new T()
  - Kreis a = new Kreis();
  - Objektfelder automatisch null-initialisiert
- Referenzsemantik
  - Kreis b = a;
    a.radius = 7;
    b.radius = b.radius * 2;
  - Erzeugung von Kopien: Methode .clone für alle Objekte:
    Kreis b = (Kreis)a.clone();

static

- Normalerweise: Felder, Methoden gelten für Objekte
  - Feldwerte sind pro Exemplar vorhanden
  - Methoden werden für ein Exemplar aufgerufen
    - this gibt aktuelles Objekt an
- static: Felder, Methoden gelten für die ganze Klasse
  - static int i;
    - z.B. Math.PI (Math ist eine Klasse)
  - static int fib(int n) {
    if (n <= 1) return 1;
    return fib(n-1)+fib(n-2);
    }
- Zugriff auch nicht-statische Felder nur in nicht-statischen Methoden erlaubt
  - oder mit expliziter Angabe des Objekts
  - nicht-statische Methoden können auch auf statische Felder zugreifen

Zugriffssteuerung

- “access control”
  - “control” oft mit (falsch?) “Kontrolle” übersetzt: besser “Steuerung”
  - “access control” oft falsch mit “Sichtbarkeit” übersetzt
    - Sichtbarkeit: Felder wären nicht bekannt
      - In C++ handelt es sich bei gleichnamigem Konstrukt um Sichtbarkeitsbeschränkungen (visibility)
    - Zugriffssteuerung: Felder sind bekannt, aber nicht zugänglich
- Ziel: Durchsetzung der Kapselung
  - Anwender einer Klasse sollen nur ausgewählte Felder lesen und schreiben, nur ausgewählte Methoden rufen
  - Garantie von Invarianten
- Java-Zugriffssteuerung:
  - public: Feld/Methode von außen zugänglich
  - private: Feld/Methode nur innerhalb der Klasse zugänglich
  - ... (protected, “package-private”)

Zugriffssteuerung (2)

- Zugriffsmethoden (getter/setter)
  - genauere Steuerung des Zugriffs/Invarianten
  - Methode zum Lesen von “a” heißt per Konvention getA();
  - Methode zum Schreiben: setA();

```
public class Person{
    private String name; //read-only
    private int age; // monoton wachsend
    public String getName() {
        return name;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int newAge) {
        if (age < newAge) age = newAge;
    }
```

Überladung

- *overloading*: Verwendung des gleichen Namens/Operators für verschiedene Zwecke
- Operatorüberladung: Operatoren funktionieren mit verschiedenen Parametertypen, liefern verschiedene Ergebnistypen
  - T op T liefert i.d.R. T (Ausnahme: relationale Operatoren)
  - Automatische Typumwandlung bei gemischten Operandentypen
    - Java: Addition (+) funktioniert für string+T, T+string
- Methodenüberladung: gleicher Methodenname, verschiedene Parametertypen
  - Signatur (*signature*): Folge der Parametertypen
  - gleiche Signatur: gleiche Zahl von Parametern, jeweils gleiche Parametertypen
    - Ergebnistyp, Parameternamen irrelevant

Überladung (2)

```
class Datum {
    private int jahr=2006, monat = 1, tag = 19;
    //...
    public void add(int tage) { ... }
    public void add(int tage, int monate) { ... }
    public void add(int tage, int monate, int jahre) {...}
}
```

Konstruktoren

- Initialisierung von Objekten:
  - Bereitstellung von Speicher
  - Null-Initialisierung aller Felder
  - Aufruf des Konstruktors
- Konstruktorsyntax: ähnlich zu Methodensyntax, aber
  - Kein Rückgabetyp (auch nicht void)
  - Methodenname gleich Klassenname
  - Kein expliziter Aufruf (außer in Konstruktor selbst)
- Abarbeitung des Konstruktors:
  1. Aufruf des Basisklassenkonstruktors
  2. Ausführung expliziter Feldinitialisierungen
  3. Ausführung des Körpers des Konstruktors

Konstruktoren (2)

- Standardkonstruktor (*default constructor*):
  - parameterlos
  - aufgerufen bei new T();
  - implizit definiert, falls ansonsten keine Konstruktoren definiert werden
    - implizit public, leerer Körper
-

Konstruktoren (3)

```
class Datum {
    private int tag, monat, jahr;
    public Datum() { tag = 19; monat = 1; jahr = 2006; }
    public Datum(int t) { tag = t; monat = 1; jahr = 2006; }
    public Datum(int t, int m) { tag = t; monat = m; jahr = 2006; }
    public Datum(int t, int m, int j) { tag = t; monat = m; jahr = j; }
}
//...
Datum d = new Datum();
Datum d2 = new Datum(28, 2);
```

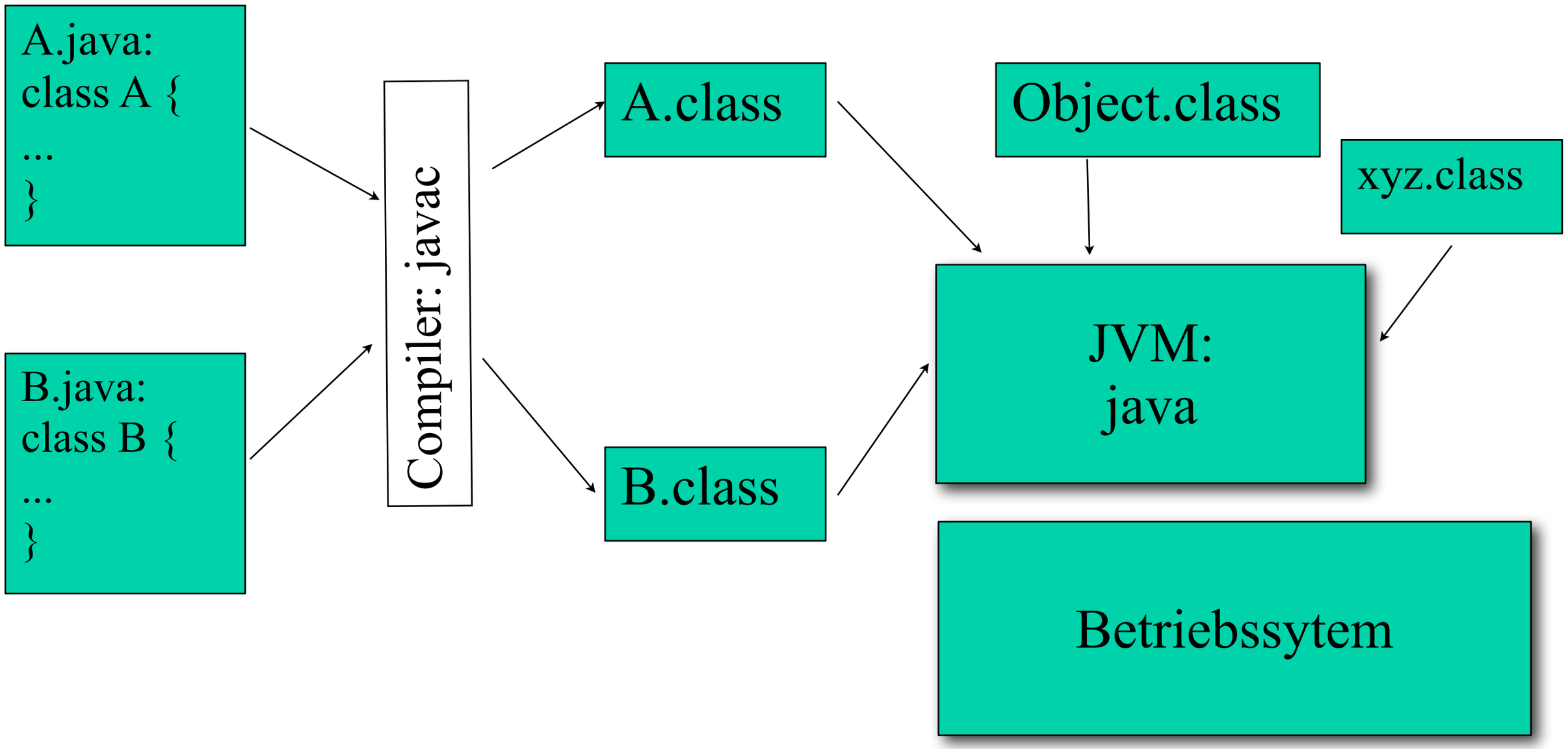
Programme

- Kein eigentlicher Programmbegriff
- Verschiedene Ausführungsumgebungen für Java-Code:
  - eigenständiges Programm
  - Applet im Webbrowser
  - Servlet im Webserver
  - Enterprise Java Bean
- Programm-Modus: Eine Klasse enthält das Hauptprogramm
  - public static void main(String[] args);
  - Klasse kann weitere statische oder Objektmethoden enthalten

Java-Werkzeugkette

- Quellcode in Dateien X.java
  - höchstens eine Klasse in der Datei darf public sein oder von anderen Klasse des gleichen *package* referenziert werden; diese Klasse muss dann den Namen X haben
  - i.d.R. insgesamt eine Klasse pro Datei, class Foo in Foo.java
- Übersetzung des Quelltexts in Java-Bytecode: javac
  - Kommandozeilenargumente: Namen von .java-Dateien
- Ausführung des Bytecodes: java
  - Kommandozeilenargument: Name der Klasse, die main enthält

Java-Werkzeugkette (2)



Pakete

- *packages*: Sammlung zusammengehöriger Klassen
- Ziel: Strukturierung des Namensraums von Klassen
  - Vermeidung/Behebung von Namenskonflikten
- Java-Standardbibliothek ist strukturiert in Pakete
  - java.lang: Kernsprache (z.B. class Object)
  - java.util: allgemeine Hilfsklassen (z.B. Container: Vector)
  - java.io: Ein-/Ausgabe
  - java.awt: GUI (Abstract Windowing Toolkit)
  - java.net: Netzkommunikation
- Konvention: Eigene Pakete entsprechen umgedrehtem DNS-Namen des Besitzers
  - org.apache: Projekte der Apache-Foundation
  - org.omg: Klassen der OMG (Object Management Group)
  - com.sun: Nicht-öffentliche Klassen von Sun

Verwendung von Paketen

- “Normalform”: jede Klasse hat vollqualifizierten Namen
  - java.io.File f = null;
- Import-Anweisungen: Kurzschreibweise für Klassen in importiertem Paket
  - Importanweisungen am Anfang der Datei

```
import java.io.File;
//...
File f = null;
```

  - Alternativ: Import aller Klassen eines Pakets

```
import java.io.*;
```

  - java.lang.* ist stets implizit importiert

Verwendung von Paketen (2)

- Importierte Klassen müssen im Suchpfad von Java sein
- Umgebungsvariable CLASSPATH: Definition des Suchpfads
  - Standard-Java-Klassen werden immer gefunden
  - CLASSPATH nicht gesetzt: aktuelles Verzeichnis ist auch im Suchpfad
  - Alternativ: Kommandozeilenargument -classpath, -cp für javac, java
- Abfrage des Suchpfads im Programm:
  - System.getProperty(“java.class.path”);

java.lang

- Kernklassen der VM: Object, String, System, Class, Runtime, ...
- Hüllklassen für Wertetypen: Boolean, Character, Number, Integer, Float, Double
- Math: Sammlung statischer mathematischer Funktionen
  - sin, cos, log, abs, ...

Definition eigener Pakete

- Deklaration des Paketnamens mit package
  - package de.uni_potsdam.hpi.pt1;
- Verzeichnisstruktur muss Paketstruktur entsprechen
- Zugriffssteuerung “package private”: Felder, Methoden ohne explizite Zugriffsdeklaration sind in allen Klassen des Pakets sichtbar

Ausdrücke

- Vorrang (*precedence*), Assoziativität wird durch Grammatik definiert

  AdditiveExpression ::= MultiplicativeExpression

  | AdditiveExpression ‘+’ MultiplicativeExpression

  | AdditiveExpression ‘-’ MultiplicativeExpression

- primary expression: Literale, this, Methodenruf, Feldzugriff, Exemplarerzeugung, ...
  - 4, a.b, (7*x), System.out.println(45), new Foo(109)
- Postfix-Ausdrücke: x++, a.y--
- Unäre Operatoren: +a, -b, !c, ~d, ++e, --f
- Typkonvertierungen (*cast expression*): (Typ)Wert
- Multiplikationsoperationen: a*b, a/b, a%b
- Additionsoperationen

Ausdrücke (2)

- Shift-Operationen
- Vergleichsoperationen: a<b, a>b, a<=b,
  a>=b, a instanceof b
- Gleichheitsoperatoren: a==b, a!=b
- Bitweise Operatoren
- logische Operatoren
- Zuweisungsoperatoren

Ausdrücke (3)

- Automatische Typumwandlungen:
  - byte → short → int → long → float → double
  - Konvertierung von abgeleiteter Klasse in Basisklasse
  - Konvertierung von Klasse in implementiertes Interface

Einfache Anweisungen

- Leeranweisung
  - ;
- Ausdrucksanweisung (*expression statement*)
  - *expression*;
- Rückkehranweisung
  - return *expression*;
- Schleifensteuerung
  - break;
  - break *label*;
  - continue;
  - continue *label*;
- Auslösen von Ausnahmen
  - throw *expression;*
- assert

Blöcke

- Gruppierung von Variablendeklarationen und Anweisungen durch geschweifte Klammern
  - Gesamter Block ist syntaktisch wie eine Anweisung

```
{
    ...
}
```

- Variablendeklarationen sind sichtbar von der Deklaration bis zum Ende des Blocks (lokale Variablen)

Alternativ-Anweisung

- if (bedingung) $anweisung_1$ else $anweisung_2$

switch-Anweisung

```
static int tageProMonat(int j, int m) {
   switch(m) {
      case 1: case 3: case 5: case 7:
      case 8: case 10: case 12:
         return 31;
      case 2:
         return schaltJahr(j) ? 29 : 28;
      default:
         return 30;
   }
}
```

switch-Anweisung (2)

SwitchStatement ::= ‘switch’ ‘(‘ Expression ‘)’
‘{‘ SwitchBlockStatementGroup* SwitchLabel* ‘}’

SwitchBlockStatementGroup ::= SwitchLabel+ BlockStatement*

SwitchLabel ::= ‘case’ ConstantExpression
| ‘case’ EnumConstantName
| ‘default’

ConstantExpression ::= Expression

- Typ des switch-Ausdrucks: char, byte, short, int, Character, Byte, Short, Integer, oder ein enum-Typ
- ConstantExpression: muss Konstante sein
  - Zuweisungskompatibel zu Typ des switch-Ausdrucks
  - darf nicht null sein
- Alle case-Werte müssen verschieden sein
- Es darf höchstens ein default-Label geben

Konstante Ausdrücke

- *compile-time constant expression*
- Literale primitiver Typen, sowie String-Literale
- Anwendungen von Operatoren auf Konstanten:
  - Typkonvertierungen zu primitiven Typen und String-Literalen
  - unäre Operatoren +, -, ~, ! (nicht: ++, --)
  - binäre Operatoren *, /, %, +, -, <<, >>, >>>, ==, !=, &, ^, |, &&, ||
  - ternärer Operator ?:
- Klammerung konstanter Ausdrücke
- als final deklarierte Variablen mit konstantem Wert:
  - final int i = 3600*24*7;
  - Zuweisung an Variable nicht erlaubt
  - final-Variablen können auch mit nicht-konstantem Wert initialisiert werden

while-Schleife

- while (*Expression*) *Statement*
  - Expression muss Typ boolean haben (Java 5: auch Boolean)
  - kein explizites Semikolon: entweder Semikolon aus einfacher Anweisung, oder Angabe eines Blocks
- Abwechselnd: Test von Expression, dann Statement
  - Schleife beendet falls Expression false
- Abbruch durch break
- Fortsetzen mit der nächsten Überprüfung von Expression: continue

do-while-Schleife

- do *Statement* while (*Expression*) ;
- Anweisung wird wenigstens einmal ausgeführt, dann abwechselnd Bewertung vom Ausdruck, Wiederholung der Anweisung...

for-Schleife

```
for (int i = 0; i < N; i++)
   int2Bsp[i| = i;
for(a = 1, b = 2; a < 10; a++, b++)
   System.out.println(a*b);
```

for-Schleife: Syntax

- for (*ForInit* ; *Test* ; *ForUpdate*) *Statement*
  - ForInit, Test, Update sind alle optional
  - Statement kann auch Leeranweisung sein
- ForInit: $Init_1$, $Init_2$, ...
  - entweder Liste von Ausdrucksanweisungen, oder eine Variablendeklaration
    - Deklarierte Variablen nur innerhalb von for-Schleife sichtbar
- ForUpdate: $Update_1$, $Update_2$, ...
  - Liste von Ausdrucksanweisungen

for-Schleife: dynamische Semantik

Init$_1$*; Init*$_2$*; ...*
while(*Test*) {
    Statement
    Update$_1$*; Update*$_2$*; ...*
}

enhanced for statement

- Java 5
- for (*Type Identifier* : *Expression*) *Statement*

```
int[] primes = { 2, 3, 5, 7, 11, 13 };
for (int prime: primes)
    System.out.println(prime);
```

- Expression muss ein Array sein
  - (oder ein Objekt, das java.lang.Iterable implementiert)
- Typ muss Elementetyp des Felds sein
- Semantik:

```
Type[] a = Expression;
for(int index = 0; index < a.length; index++) {
    Type Identifier = a[index];
    Statement
}
```

break und continue mit Sprungmarke

Verlassen/Fortsetzen von äußeren Schleifen

```
haufen:for(int h = 0; h < heuhaufen.length; h++) {
   for (int halm = 0; halm < heuhaufen[h].length; halm++) {
       if (heuhaufen[h][halm].ist_nadel()) {
           System.out.println("Nadel in Haufen "+h+" gefunden");
           break haufen;
       }
   }
}
```

Klassen

- Wiederholung:
  - Felder (members)
  - Methoden
  - Konstruktoren, new
  - statische Felder und Methoden
  - Zugriffssteuerung
  - final

Vererbung

- Java: nur Einfachvererbung

```
class Abgeleitet extends Basisklasse {
    // neue Felder, Methoden, Konstruktoren
}
```

- Basisklasse ist java.lang.Object, falls keine andere angegeben ist
  - Alle Klassen erben letztlich von java.lang.Object
- Zugriffsrecht “protected”: protected-Felder und -Methoden sind nur in Klasse selbst und allen abgeleiteten Klassen zugänglich

Typkompatibilität und Polymorphie

- Variablen der Basisklasse können Objekte der Ableitung aufnehmen

  Abgeleitet a = new Abgeleitet();

  Basisklasse b = a;

- Umgekehrte Zuweisung nur mit expliziter Typkonvertierung (cast) möglich

  a = (Abgeleitet)b;

  – “checked cast”: falls referenziertes Objekt nicht Exemplar der Klasse Abgeleitet, erscheint java.lang.ClassCastException

Späte Bindung in Java

- Alle Exemplarmethoden sind “virtuell”
  - späte Bindung (*late binding*)
- Bindung erfolgt auf Basis von aktuellem Objekt (this), Methodenname und Methodensignatur
  - z.B. toString: Definiert in java.lang.Object
  - In

    Object o = new Point(10,7);

    o.toString();

    wird Object.toString gerufen, es sei denn, Point überschreibt (*overrides*) toString.
- Konvention: Sofern möglich, immer auf späte Bindung vertrauen, anstatt in den spezialisierten Typ zu casten
  - i.d.R. bei Containertypen nicht möglich

final-Methoden

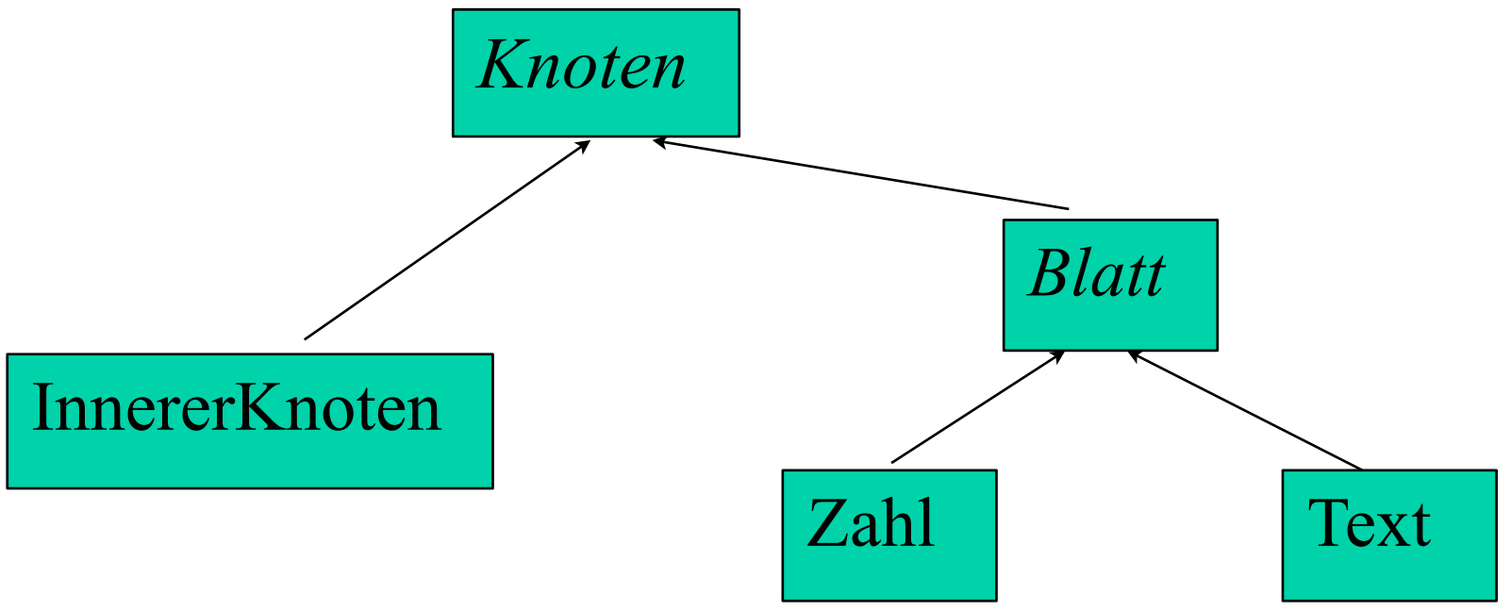
- Wunsch: Abgeleitete Klassen sollen gewisse Methoden verwenden, aber nicht überschreiben dürfen
  - z.B. fiktive Ausgabemethode toHTML soll sicherstellen, dass auch für abgeleitete Methoden immer gültiges HTML ausgegeben wird
- public final String toHTML();
- Oft verwendet zur Steigerung der Performance:
  - JVM muss u.U. keine späte Bindung mehr durchführen, weil Zielmethode schon zur Compile-Zeit bekannt ist

final-Klassen

- final class X
  - Ableitungen von X sind nicht mehr erlaubt
- Beispiele: java.lang.System, java.lang.String, ...
  - zur Wahrung von Invarianten
  - aus Performance-Gründen

Abstrakte Klassen und Methoden

- Ziel: Typunabhängige (*generic*) Algorithmen
  - Alle Objekte einer Oberklasse sollen die gleichen Methodensignaturen anbieten
  - aber: keine Realisierung der Methode in der Oberklasse vorstellbar
- Beispiel: Binärer Baum
  - Werte verschiedener Typen in den Blättern
  - innere Knoten: Verweis auf Kindknoten



```
abstract class Knoten {
    public abstract int tiefe();
    public abstract boolean istBlatt();
    public abstract void print();
}
class InnererKnoten extends Knoten {
    public Knoten links, rechts;
    public boolean istBlatt() { return false; }
    public int tiefe() {
        int tlinks = links.tiefe(), trechts = rechts.tiefe();
        return tlinks > trechts ? tlinks+1 : trechts+1;
    }
    public void print() {
        System.out.print("("); links.print();
        System.out.print(", "); rechts.print(); System.out.print(")");
    }
}
```

```
abstract class Blatt extends Knoten {
   public bool istBlatt() { return true; }
   public int tiefe() { return 1; }
}
class Zahl extends Blatt {
   int value;
   public void print() { System.out.print(value); }
}
class Text extends Blatt {
   String value;
   public void print() { System.out.print(value); }
}
```

Abstrakte Klassen (2)

- Verallgemeinerung des Beispiels: Definition von Blatt als parametrisierten (*generic*) Typen
- Abstrakte Methoden: Implementierung muss nicht angegeben werden
- Abstrakte Klassen: Es können keine Exemplare der Klasse gebildet werden
  - Klasse muss abstrakt sein, wenn eine Methode abstrakt ist

Schnittstellen

- *interfaces*
- “reine” abstrakte Klassen: Lediglich Festlegung von Signaturen
  - keine Methoden-Definitionen
  - keine Objektfelder
  - statische Felder sind erlaubt
  - alle Methoden implizit “public”
- Klassen können nur eine Basisklasse haben (Einfachvererbung), aber beliebig viele Schnittstellen implementieren
  - Ziel: Klassifikation anhand gemeinsamer Eigenschaften

Schnittstellen (2)

- Beispiel: Ordnungsrelationen
  - Objekte unterliegen u.U. einer Ordnungsrelation
  - Sortierverfahren benötigen lediglich Kenntnis der Ordnungsrelation, nicht aber der konkreten Bedeutung einzelner Klassen

```
interface Geordnet {
    boolean kleinerGleich(Geordnet zweitesObjekt);
}
class Account implements Geordnet {
    public boolean kleinerGleich(Geordnet zweitesObjekt_G) {
        Account zweitesObjekt = (Account)zweitesObjekt_G;
        return this.UID <= zweitesObjekt.UID;
    }
}
// Verwendung:
Geordnet a,b; // a = new Account(); ...
 a.kleinerGleich(b);
```

Ausnahmebehandlung

- Ausnahmen sind Objekte, deren Klassen von java.lang.Throwable abgeleitet sind
- Auslösen von Ausnahmen:
  - throw <objekt>;
  - i.d.R.: throw new T(params);
  - im catch-Block auch: throw;
- Abfangen von Ausnahmen

```
try{
    anweisungen;
}catch(Ausnahmeklasse variable) {
    behandlung;
}
//weitere catch-Blöcke
finally{
    anweisungen
}
```

Semantik der Ausnahmebehandlung

- sowohl catch-Blöcke als auch finally-Block sind optional
  - eins von beiden muss aber vorhanden sein
- falls der finally-Block vorhanden ist, wird er immer ausgeführt (egal, ob in dem try-Block oder im catch-Block eine Ausnahme auftritt)
  - der finally-Block wird als letztes ausgeführt
- falls im try-Block eine Ausnahme auftritt, wird nach einem passenden catch-Block gesucht
  - von oben nach unten (links nach rechts)
  - entsprechend der Klassenhierarchie
- falls kein catch-Block gefunden wird, wird die Ausnahme “nach außen” weitergeleitet

Statische Überprüfung von Ausnahmen

- Jede Methode kann Liste der potentiell auftretenden Ausnahmen deklarieren
  - Basisklasse schließt alle Ableitungen ein

```
public void foo() throws FileNotFoundException, EOFException
{
    ...
}
```

- Für jede Anweisung wird Menge der potentiell auftretenden Ausnahmen bestimmt
  - Betrachte Menge der gerufenen Methoden
  - Bilde Vereinigung aller an diesen Methoden auftretenden Ausnahmen
  - Streiche alle Ausnahmen, für die es eine Behandlung (ohne re-throw) gibt
  - Liste der in einer Funktion deklarierten Ausnahmen muss in deklarierten Ausnahmen enthalten sein

Statische Überprüfung von Ausnahmen (2)

- Annahme: Klassen A, B, C
  - B extends A, C extends A
  - abstract void f1() throws B;
  - abstract void f2() throws C;
- Betrachten Methode

```
void g() {
    f1();
    f2();
}
```

- Fehler: Ausnahmen B und C sind möglich, werden aber weder behandelt noch deklariert

- Lösung 1: Deklarieren der Ausnahmen
  - Problem: evtl. lange Liste

```
void g() throws B, C {
    f1();
    f2();
}
```

- Lösung 2: Deklarieren einer Basisklasse
  - Problem: alle Rufer von g() müssen nun A behandeln

```
void g() throws A{
    f1();
    f2();
}
```

- Lösung 3: Manche Ausnahmen behandeln
  - Problem: u.U. keine adäquate Behandlung denkbar

```
void g() throws B {
    try{
        f1();
        f2();
    } catch (C) {
        // Behandlung
    }
}
```

- i. allg. Kombination aus diesen Möglichkeiten
  - Ziel: Behandlung aller “behandelbaren” Ausnahmen, Weiterleitung aller Ausnahmen, die der Rufer behandeln kann
  - u.U. Umsetzung einer Ausnahme auf eine andere, zur Vereinfachung der Behandlung im Rufer

Vordefinierte Ausnahmen

- Basisklassen:
  - Throwable
    - Error
    - Exception
      - RuntimeException
- “unchecked exceptions”: Ausnahmen müssen nicht an Methode deklariert werden, wenn sie von Error oder RuntimeException ableiten
  - Manche Ausnahmen können in nahezu jeder Anweisung auftreten; es wäre redundant, sie jedes Mal deklarieren zu müssen
  - Beispiele (Error): OutOfMemoryError, StackOverflowError, AssertionError
  - Beispiele (RuntimeException): NullPointerException, ArithmeticException, NegativeArraySizeException

Weitere Java-Konzepte

- generic types
- enums
- boxing
- Threads, synchronized
- class initializer, instance initializer
- annotations
- JavaDoc
- Bibliothek
- Bibliothek
- Bibliothek