Java: Eine Übersicht

Dennis Giffhorn

Lehrstuhl für Programmierparadigmen
Universität Karlsruhe

Allgemeines

- Objektorientiert
- Syntaxfamilie von C/C++
- Statisch getypt
- Entwickelt von Sun Microsystems

```
class HelloWorld {
    public static void main(String[] args) {
        String msg = "Hello World!";
        System.out.println(msg);
    }
}
```

Allgemeines

- 1996 Version 1.0, momentan Version 1.7
- Gesamte Java-Technologie ist Open Source (unter GPL 2), es ist aber auch eine kommerzielle Lizenz erhältlich
- Umfangreiche Klassenbibliothek

Virtuelle Maschine (JVM)

- Quellcode $\longrightarrow$ Bytecode
- Wird von der JVM ausgeführt
  - Portabler Code
  - Überwachte Ausführung (Einhaltung von Sicherheitseinstellungen, keine Pufferüberläufe)
  - Garbage Collection
  - Erlaubt gute Fehlerbehandlung
- Frei erhältliche Spezifikation

Just-in-Time Compiler (JIT)

Die gängigsten JVM verwenden JIT-Compiler

- Bytecode einer Methode wird unmittelbar vor ihrer ersten Ausführung kompiliert
- Vor- und Nachteile:
  - Overhead, da zur Laufzeit kompiliert wird
  - Dyn. Optimierungen für bestimmte Eingaben oder Hot Spots

Java-Objekte

- Instanzen einer Klasse
- Zugriff nur über Objektreferenz

```
A a = new A();
A b = a;
a = new A();
```

- Keine Pointer wie in C++
  $\longrightarrow$ keinen expliziten Zugriff auf Adressen
  $\longrightarrow$ keine Pointer-Arithmetik
- Keine Referenzen auf Referenzen
- Java-Referenzen sind aber wiederverwendbar
  $\longrightarrow$ Unterschied zu Referenzen in C++

Java-Objekte

- Objekte liegen immer auf dem Heap
  - Stack enthält lokale primitive Variablen und Referenzen
  - Speicherverwaltung durch Garbage Collection
    $\rightarrow$ Manuelle Speicherverwaltung in Java beinahe unnötig

Java-Objekte

Methodenaufrufe: Call-by-Value

```
class A { int x; }
class B {
  void foo(A a) { A b = new A(); b.x = 17; a = b;}

  void bar() {
    A a = new A();
    a.x = 42;
    foo(a); // a.x = 42
  }
}
```

Java-Objekte

Vorsicht: Keine echten Kopien von Objekten!
$\longrightarrow$ kopiert wird nur die Referenz auf a

```
class A { int x; }
class B {
  void foo(A a) { a.x = 17;}

  void bar() {
    A a = new A();
    a.x = 42;
    foo(a); // a.x = 17 !
  }
}
```

Vererbung

Einfachvererbung, es können aber mehrere Interfaces implementiert werden

```
class A { int x; }
interface I { public void setX(int y); }
interface J { public int getX(); }

class B extends A implements I, J {
   int x;
   public void setX(int y) { x = y; }
   public int getX() { return super.x; }
}
```

Vererbung

- Zugriffskontrolle durch `public`, `protected`, `default` (leer) und `private`.

Vererbung

- Zugriffskontrolle durch `public`, `protected`, `default` (leer) und `private`.
- Jede Methode ist prinzipiell in Unterklassen redefinierbar
  $\longrightarrow$ muss man explizit ausschließen

```
class A {
    int x;
    private void setX(int y) { ... }
    public final int getX() { ... }
}
```

  $\longrightarrow$ Wird nur selten gemacht

Parametrischer Polymorphismus

‘Generics’, seit Java 1.5

- Vor Java 1.5

```
List l = new LinkedList();
l.add(new Integer(5));
l.add(``String'');
...
Integer i = (Integer) l.get(0); // Typecasts
Integer j = (Integer) l.get(1); // Programmabsturz
```

Parametrischer Polymorphismus

Sicher durch Generics

```
class LinkedList<T> {
  void add(T t) { ... }
  T poll() { ... }
}
```

Parametrischer Polymorphismus

Sicher durch Generics

```
class LinkedList<T> {
  void add(T t) { ... }
  T poll() { ... }
}

List<Integer> l = new LinkedList<Integer>();
l.add(new Integer(5));
l.add(``String'');    // verbietet der Compiler
...
Integer i = l.get(0); // keine Typecasts mehr
```

Parametrischer Polymorphismus

Sicher durch Generics

```
class LinkedList<T> {
  void add(T t) { ... }
  T poll() { ... }
}

List<Integer> l = new LinkedList<Integer>();
l.add(new Integer(5));
l.add(``String'');    // verbietet der Compiler
...
Integer i = l.get(0); // keine Typecasts mehr
```

Nicht so mächtig wie Templates in C++, dafür typsicher

Typsystem

Nicht alles in Java ist ein Objekt.

- Referenzdatentypen (Oberklasse Object)
- Primitive Datentypen (int, char, boolean, ...)

```
int i = 5;
Object o = i; // vor Java 1.5 vom Compiler verboten
```

Typsystem

Nicht alles in Java ist ein Objekt.

- Referenzdatentypen (Oberklasse Object)
- Primitive Datentypen (int, char, boolean, ...)

```
int i = 5;
Object o = i; // vor Java 1.5 vom Compiler verboten
```

Problem: Verwendung von Containern

```
List<int> l = new LinkedList<int>(); // kein Objekt
l.add(5);            // 5 ist kein Objekt
int j = l.poll(); // j ist kein Objekt
```

Typsystem

Nicht alles in Java ist ein Objekt.

- Referenzdatentypen (Oberklasse Object)
- Primitive Datentypen (int, char, boolean, ...)

```
int i = 5;
Object o = i; // vor Java 1.5 vom Compiler verboten
```

Stattdessen:

```
List<Integer> l = new LinkedList<Integer>();
l.add(new Integer(5));
Integer i = l.poll();
int j = i.intValue();
```

Autoboxing/-unboxing

Automatische Konvertierung zwischen Basis- und Objekt-Typen:
Vorher:

```
Integer i = new Integer(3);
int j = i.intValue();
```

Autoboxing/-unboxing

Automatische Konvertierung zwischen Basis- und Objekt-Typen:
Nachher:

```
Integer i = 3;
int j = i;
```

Autoboxing/-unboxing

Automatische Konvertierung zwischen Basis- und Objekt-Typen:
Nachher:

```
Integer i = 3;
int j = i;
```

Besonders nützlich im Zusammenspiel mit generischen Containern:

```
List<Integer> l = new LinkedList<Integer>();
l.add(new Integer(5));
Integer i = l.poll();
int j = i.intValue();
```

Autoboxing/-unboxing

Automatische Konvertierung zwischen Basis- und Objekt-Typen:
Nachher:

```
Integer i = 3;
int j = i;
```

Besonders nützlich im Zusammenspiel mit generischen Containern:

```
List<Integer> l = new LinkedList<Integer>();
l.add(new Integer(5));
Integer i = l.poll();
int j = i.intValue();
```

wird zu:

```
List<Integer> l = new LinkedList<Integer>();
l.add(5);
int j = l.poll();
```

Threads

Java bietet von Beginn an Threads, um nebenläufige Ausführungen zu ermöglichen

- Kommunikation über Shared Memory
  $\rightarrow$ alle Threads laufen in derselben JVM

```
class MyThread extends Thread {
   public void run() { Main.x = 1; }
}
class Main {
   static int x = 0;

   static void foo() {
      MyThread t = new MyThread();
      t.start();
      System.out.println(x); // 0 oder 1
   }
}
```

Threads

- Ursprünglich für GUI-Programmierung gedacht: Keine Blockade duch Warten auf Benutzereingaben
  $\longrightarrow$ ältere JVM belegten nur einen Prozessor
- Mittlerweile verteilt JVM Threads auf andere Prozessoren, falls vorhanden
  $\longrightarrow$ echte parallele Ausführung
- Es ex. sogar JVM für Cluster (zB. Cluster VM von IBM)

enum

enum: Menge von symbolischen Konstanten
Häufig sieht man Implementierungen folgender Art:

```
static final int AMPEL_ROT = 1;
static final int AMPEL_ROTGELB = 2;
static final int AMPEL_GELB = 3;
static final int AMPEL_GRUEN = 4;
int lights = AMPEL_ROT;
```

Nachteile:

- Fehleranfällig bei Änderung und Erweiterungen
- `lights` kann ungültige Werte annehmen
- Der Inhalt von `lights` hat keinen Dokumentationswert: `System.out.println(lights)` liefert '1'

enum

daher: Unterstützung für `enum` in der Sprache

```
enum Ampel { ROT, ROTGELB, GELB, GRUEN };

Ampel lights = Ampel.ROT;
switch(lights) {
  case ROT:      lights=Ampel.ROTGELB; break;
  case ROTGELB: lights=Ampel.GRUEN;   break;
  case GRUEN:   lights=Ampel.GELB;    break;
}
```

Vorteile:

- beseitigt alle o.g. Nachteile
- abgeschlossene Menge
  $\longrightarrow$ Compiler warnt im `switch` wegen fehlendem `GELB`
  (muss man aber explizit einstellen)

`enum`: Technische Realisierung (konzeptuell)

`enum` wird zur Klasse, Werte zu Objekten:

```
final class Ampel ... {
  private final String name;
  private Ampel(String name) { this.name = name; }

  static final Ampel ROT = new Ampel("ROT");
  static final Ampel ROTGELB = new Ampel("ROTGELB");
  static final Ampel GRUEN = new Ampel("GRUEN");
  static final Ampel GELB = new Ampel("GELB");

  Ampel[] values() {
    return new Ampel[]{ ROT,ROTGELB,GRUEN,GELB };
  }
  Ampel valueOf(String s) {
    ...
}
```

Eigene Members in `enum`

Durch Realisierung als Klassen sind enums in Java sehr mächtig

- Eigener Konstruktor (Parameter hinter enum-Konstanten)
- Beliebige Attribute
- Beliebige Methoden

```
enum Month {
  JAN(31), FEB(28), MAR(31), APR(30), MAY(31), JUN(30),
  JUL(31), AUG(31), SEP(30), OCT(31), NOV(30), DEC(31);

  private int days;

  Month(int days) { this.days = days; }

  int getDays(int year) { return days; }
}
```

Konstantenspezifische Methoden

```
enum Month {
  JAN(31),
  FEB(28) {
     int getDays(int y) {return (y % 4 == 0 ? 29 : 28);}
  },
  MAR(31), APR(30), MAY(31), JUN(30),
  JUL(31), AUG(31), SEP(30), OCT(31), NOV(30), DEC(31);

  private int days;

  Month(int days) { this.days = days; }

  int getDays(int year) { return days; }
}
```

Referenzen

- Java Language Specification
  http://java.sun.com/docs/books/jls/
- Java Virtual Machine Specification
  http://java.sun.com/docs/books/jvms/
- Thinking in Java
  http://www.mindview.net/Books/TIJ/
- Effective Java
  http://java.sun.com/docs/books/effective/