Spracherweiterungen in Java 5

Dennis Giffhorn

Lehrstuhl für Programmierparadigmen
Universität Karlsruhe

17. April 2008

Spracherweiterungen in Java 5

- Generics (s. Vorlesung)
- Annotations
- Enumerations
- Methoden mit variabler Anzahl von Parametern
- Erweiterte `for`-Schleife
- Autoboxing / -unboxing
- Static Imports
- Überschreiben von Methoden-Rückgabetypen

Generics

Generics werden in der Vorlesung vorgestellt,
hier nur kurz die Idee.

- 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
```

Generics

- Sicher durch Generics

```
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
```

Annotations

Annotations bieten die Möglichkeit, in einem Programm struktuiert zusätzlich Daten unterzubringen.

```
@Author(name="Mirko Streckenbach")
class Test {
  @LastChanged(date="2006-01-27")
  public static void main(String[] args) {
    Author a=Test.class.getAnnotation(Author.class);
    System.out.println("written by "+a.name());
  }
}
```

Annotations

Annotations bieten die Möglichkeit, in einem Programm struktuiert zusätzlich Daten unterzubringen.

```
@Author(name="Mirko Streckenbach")
class Test {
  @LastChanged(date="2006-01-27")
  public static void main(String[] args) {
    Author a=Test.class.getAnnotation(Author.class);
    System.out.println("written by "+a.name());
  }
}
```

Annotations

Annotations bieten die Möglichkeit, in einem Programm struktuiert zusätzlich Daten unterzubringen.

```
@Author(name="Mirko Streckenbach")
class Test {
  @LastChanged(date="2006-01-27")
  public static void main(String[] args) {
    Author a=Test.class.getAnnotation(Author.class);
    System.out.println("written by "+a.name());
  }
}
```

Annotations: Anwendungen

Anwendungen:

- Die Daten können von Tools weiterverarbeitet oder zur Laufzeit (Reflection) abgefragt werden.
- Nützlich in allen Bereichen, wo Source-Code automatisch verarbeitet wird:
  - JavaDoc
  - JUnit
  - Revisionskontrolle (vgl. `ident`)
- Formaler Mechanismus für bisherige Konventionen
  - Namenskonvention für Testmethoden in JUnit
  - Strukturierte Kommentare für Autor, Datum, etc. . .
- Annotations für den Compiler, z.B. um bekannte Warnungen zu unterdrücken (`@SuppressWarnings`)

Annotations: Spezifikation

Annotations können ähnlich zu Interfaces spezifiziert werden:

```
@Author(name="Mirko Streckenbach")
class Test {
  ...
}

@Retention(value=RetentionPolicy.RUNTIME)
@interface Author {
 String name();
 String company() default "unspecified";
};
```

- `@Retention` ist eine Meta-Annotation und spezifiziert den "Lebensraum" der Annotation.

Annotations: Spezifikation

```
@Retention(value=RetentionPolicy.RUNTIME)
@interface Author {
 String name();
 String company() default "unspecified";
};
```

- Methoden dürfen keine Parameter und Exceptions haben
- primitive Typen, String, Class, Enums, Annotationen
- Arrays dieser Typen
- Im Prinzip: Typ + Name + optionale Default-Werte

Annotations: Spezifikation

Annotiert werden können:

- Klassen und Interfaces
- Konstruktoren, Methoden und Fields
- Parameter und lokale Variablen
- andere Annotations und Packages

enum

enum: Menge von symbolischen Konstanten
In Java bisher übliche Implementierung:

```
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
  $\rightarrow$Compiler warnt im `switch` wegen fehlendem `GELB`
  (muss man aber explizit einstellen)
- implementieren Comparable und Serializable

`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`

- 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;
  } }
```

Methoden mit variabler Anzahl von Parametern

Bisher:

```
String join(String concat, String[] args);
```

Zum Aufruf von `join` muss man ein Array erzeugen:

```
String[] str =
    new String[] {"http:","","www.uni-karlsruhe.de"};

join("/", str);
```

Methoden mit variabler Anzahl von Parametern

Syntax:

```
String join(String concat, String... args);
```

`join` kann dann mit einer beliebigen Zahl von Argumenten aufgerufen werden:

```
join(" ","Hello","World");
join("/","http:","","www.uni-karlsruhe.de");
```

Technische Realisierung: Im Rumpf von `join` ist `args` einfach ein Array von `String`; das Array wird an der Aufrufstelle erzeugt.

Methoden mit variabler Anzahl von Parametern

Leider schafft das kleinere Probleme:

```
void f(Object o) { ... }
void f(Object[] o) { ... }
void g(Object o) { ... }
void g(Object... o) { ... }
...
void foo(String[] args) {
  f(args); // eindeutig: f(Object[])
  g(args); // mehrdeutig
}
```

Lösung:

```
g((Object)args);   // -> g(Object)
g((Object[])args); // -> g(Object...)
```

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 l = new LinkedList();
l.add(0, new Integer(42));
int x = ((Integer)l.get(0)).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 l = new LinkedList();
l.add(0, new Integer(42));
int x = ((Integer)l.get(0)).intValue();
```

wird zu:

```
List<Integer> l = new LinkedList<Integer>();
l.add(0, 42);
int x = l.get(0);
```

Erweiterte `for`-Schleife

Mit der `foreach`-Schleife kann direkt über die Elemtente eines Containers iteriert werden:

```
void main(String[] args) {
   for(String s : args)
      System.out.println(s);
}
```

Außer über Arrays kann über alle Objekte iteriert werden, die das Interface `Iterable<E>` implementieren.

```
public interface Iterable<E> {
   Iterator<E> iterator();
}
```

Die Java Container Klassen implementieren dieses Interface.

static Imports

Statische Methoden und Felder können einzeln importiert und direkt benutzt werden:

- Bisher:

```
double y = Math.sqrt(14.0);
```

static Imports

Statische Methoden und Felder können einzeln importiert und direkt benutzt werden:

- Mit static imports:

```
import static Math.sqrt;
...
double y = sqrt(14.0);
```

static Imports

Besonders praktisch im Zusammenhang mit Enums:

```
import static verkehr.Ampel.*;
...
Ampel a = ROT;
```

Überschreiben von Methoden-Rückgabetypen

Eine Methodenredifinition in einer Unterklasse darf einen stärkeren Rückgabewert haben als die ursprüngliche Definition (*covariant returns*)

Häufige Anwendung ist `clone`-Methode
Bisher:

```
class Test {
    Object clone() { ...  }
}
```

Überschreiben von Methoden-Rückgabetypen

Eine Methodenredifinition in einer Unterklasse darf einen stärkeren Rückgabewert haben als die ursprüngliche Definition (*covariant returns*)

Häufige Anwendung ist `clone`-Methode
Jetzt:

```
class Test {
    Test clone() { ... }
}
```

Überschreiben von Methoden-Rückgabetypen

Generell:

```
class A { }
class B extends A { }
```

Vorher:

```
class ObserverA {
    A data;
    A getSubject() { return data; }
}
class ObserverB extends ObserverA {
    B data;
    A getSubject() { return data; }
}
```

Überschreiben von Methoden-Rückgabetypen

Generell:

```
class A { }
class B extends A { }
```

Nachher:

```
class ObserverA {
    A data;
    A getSubject() { return data; }
}
class ObserverB extends ObserverA {
    B data;
    B getSubject() { return data; }
}
```

Überschreiben von Methoden-Rückgabetypen

Technische Realisierung: Compiler erzeugt zusätzliche Wrapper-Methode.
Idee (funktioniert nur im Byte-Code):

```
class ObserverB extends ObserverA {
    B data;
    A getSubject() { return data; }

    B getSubject() {
        A a = getSubject();
        return (B) a;
    }
}
```

Zusammenfassung

Java 5 bietet:

- Zwei “große” Erweiterungen:
  Generics und Annotations
- Mehrere kleine Erweiterungen

Fazit:

- Generics und Enums holen nur Ausgelassenes nach.
- Nur Annotations sind wirklich neu,
  ansonsten viel syntactic sugar.
- Viele Möglichkeiten zur Verbesserung der Übersichtlichkeit von Quelltexten.
- Ursprüngliche Simplizität von Java geht verloren.
- Anpassung alten Codes ist nicht trivial (Generics).
- Technische Realisierung ist teilweise kompliziert.

Weiterführendes

- Annotations: Erklärungen von Sun
  `http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html`
- Java 5 Erweiterungen:
  `http://java.sun.com/j2se/1.5.0/docs/guide/language/`