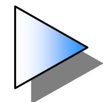
Autoboxing und statische Imports

Roman Seibold

Autoboxing und statische Imports – Inhalt



Autoboxing

statische Imports

Motivation (1)

- In Java "endet die Objektorientierung" an zwei Stellen etwas unbefriedigend:
  - Collections "verschlucken" die Typisierung (→ Abhilfe durch generische Typen).
  - Nicht alles ist ein Objekt.
- Mit *nicht alles ist ein Objekt* ist gemeint, dass die Sprache Unterschiede zwischen Referenztypen und elementaren ("primitiven") Typen macht:
  - Elementare Typen können nicht in Collections verwendet werden, man muss sie in Wrapper-Objekte hüllen.
  - Wrapper-Objekten kann man nicht arithmetisch verwenden. Man muss die elementaren Werte wieder "enthüllen".

Motivation (2)

```
public static void main(String[] args)
{
  int[] werte = { 5, 13, 12, 40, 30 };

  Collection collection = new ArrayList();

  for (int i = 0; i < werte.length; i++)
  {
    collection.add(new Integer(werte[i]));
  }

  System.out.println(summe(collection));
}

private static int summe(Collection collection)
{
  int summe = 0;

  for (Iterator iter = collection.iterator(); iter.hasNext();)
  {
    summe += ((Integer) iter.next()).intValue();
  }

  return summe;
}
```

Einpacken

Auspacken

Boxing

- Boxing ist das Verpacken, Unboxing das Entpacken von elementaren Typen in/von ihren entsprechenden Wrapper-Objekten.
- Java 5.0 versucht nach gewissen Regeln dies automatisiert zu tun. Man spricht dann von "Autoboxing".
- Autoboxing behebt also nicht den "Mangel", dass in Java nicht alles ein Objekt ist, sondern verdeckt ihn. Elementare Typen existieren (u.a. aus Performance-Gründen) weiterhin wie gehabt.

Ein einfaches Beispiel (1)

```
private static void benutzeReferenztyp(Integer i)
{
  System.out.println(i);
}

private static void nutze()
{
  benutzeReferenztyp(5);
}
```

- Dieses Programm erzeugt mit einem Compiler < 5.0 eine Fehlermeldung:

```
benutzeReferenztyp(java.lang.Integer) cannot be applied to (int)
    benutzeReferenztyp(5);
```

© aformatik Training & Consulting GmbH & Co. KG

Ein einfaches Beispiel (2)

- Java 5.0 versucht, bevor ein Fehler erzeugt wird, elementare Typen zu konvertieren.
- Im Beispiel wird das **`int`**-Literal in ein **`Integer`**-Objekt mit Value 5 konvertiert.
- Nach dieser Konversion ist der Methodenaufruf gültig.

```
private static void benutzeReferenztyp(Integer i)
{
  System.out.println(i);
}

private static void nutze()
{
  benutzeReferenztyp(new Integer(5));
}
```

unsichtbare Konvertierung

Konvertierungsregeln Boxing

Ausgangstyp	konvertiert in
byte	**Byte**
short	**Short**
char	**Character**
int	**Integer**
long	**Long**
float	**Float**
double	**Double**
boolean	**Boolean**

- Dabei ist vorgegeben (Beispiel **int**):

```
int p → Integer r mit r.intValue() == p
```

© aformatik Training & Consulting GmbH & Co. KG

Konvertierungsregeln Unboxing

Ausgangstyp	konvertiert in
`Byte`	`byte`
`Short`	`short`
`Character`	`char`
`Integer`	`int`
`Long`	`long`
`Float`	`float`
`Double`	`double`
`Boolean`	`boolean`

- Verbirgt sich hinter dem Ausgangstyp kein Objekt (**`null`**-Referenz) wird eine **`NullPointerException`** ausgelöst.

Andere Konvertierungen

- Der Autoboxing-Mechanismus ist in die normalen Typkonvertierung/Typangleichungen von Java eingebettet.

```
Integer i = 1;
long l = i + 1; // l hat den Wert 2
```

- Das funktioniert aber nur innerhalb der "Spielregeln" des jeweiligen Zieltyps.

```
Object o = 5;       // 5 -> new Integer(5)
Number n = 15.6;    // 15.6 -> new Double(15.6)

Long l = 5;         // Fehler!
```

- D.h. die Typkonvertierung erfolgt gemäß den Standardmechanismen *nach* Durchführung des Boxings.

Beispiele

```
Long l = 5;         // Fehler!
Long l = (long) 5; // Korrekt.
Long l = 5L;        // Korrekt.

int  i = 97;
char a = (Character) (char) i;
System.out.println(a);            // gibt 'a' aus

Long l = (long) (Long) (long) (int) 1; // ohne Worte...

Collection collection = new ArrayList();
collection.add(5); // Warning: "unchecked call" -
                   // funktioniert aber.

Number n = true; // Fehler (Boolean erbt nicht von Number)
Object o = true; // Korrekt.

int i = null;      // Fehler: "incompatible types"
Integer l = null;
int n = l;         // Korrekt, aber Laufzeitfehler
                   // (NullPointerException)
```

Arrays

- Für Array-Typen sind dem Autoboxing Grenzen gesetzt:

```
int[] a = { 1, 2, 3 };
Integer[] b = a; // Fehler:
                 // "incompatible types int[], Integer[]"

// Das funktioniert nur so:
int[] a = { 1, 2, 3 };
Integer[] b = new Integer[a.length];

for (int i = 0; i < a.length; i++)
{
  b[i] = a[i];
}
```

Identitäten (1)

- Autoboxing verändert nichts an den bisherigen Spielregeln für Wrapper-Objekte.
- Die Spezifikation fordert allerdings, dass für zwei Referenzen **`r1`** und **`r2`**, die aus dem Boxing eines Wertes **`p`** entstehen, gilt: **`r1 == r2`**.

```
Integer i1 = 5;
Integer i2 = 5;
int n1     = 5;
int n2     = 5;

System.out.println(i1.equals(i2)); // true
System.out.println(i1 == i2);      // true
System.out.println(n1 == n2);      // true
```

© aformatik Training & Consulting GmbH & Co. KG

Identitäten (2)

- Leider gelten für die geforderte Identität eine Reihe von Einschränkungen:
  - **`true`**, **`false`**
  - ASCII-Wertebereich
  - Zahlenwertebereich von –128 bis +127
- Das implementierte Verhalten führt daher zu einer Überraschung, die bei bloßem Lesen übersehen werden kann:

```
Integer i1 = 128;
Integer i2 = 128;
int m1     = i1;
int m2     = i2;

System.out.println(i1 == i2); // false !!!
System.out.println(m1 == m2); // true
```

Identitäten (3)

- Hinter den Kulissen verwendet der Java-Compiler für das Boxing den Mechanismus der korrespondierenden **`valueOf()`**-Methoden der entsprechenden Wrapper-Klassen.

```
Integer i1 = 127;
Integer i2 = Integer.valueOf(127);

System.out.println(i1 == i2); // true
```

```
Byte b1 = 127;
Byte b2 = Byte.valueOf((byte) 127);

System.out.println(b1 == b2); // true
```

```
Float f1 = 127.0f;
Float f2 = Float.valueOf(127.0f);

System.out.println(f1 == f2); // false !
```

© aformatik Training & Consulting GmbH & Co. KG

Identitäten (4)

- Also sieht die Konversion tatsächlich so aus:

```
Integer i1 = 127;
```

```
Integer i1 = Integer.valueOf(127); // ja

Integer i1 = new Integer(127);      // nein
```

- Die Identitätskriterien sind in den jeweiligen Konversions-Methoden spezifiziert bzw. in der JLS:

... as this method is likely to yield significantly better space and time performance by **caching frequently requested values**.

Less memory-limited implementations might cache all characters and shorts, as well as integers and longs in the range of -32K - +32K.

Beispiel mit "vollem" Einsatz von Java 5.0

```
public static void main(String[] args)
{
  int[] werte = { 5, 13, 12, 40, 30 };

  Collection<Integer> collection = new ArrayList<Integer>();

  for (int wert : werte)
  {
    collection.add(wert);
  }

  System.out.println(summe(collection));
}

private static int summe(Collection<Integer> collection)
{
  int summe = 0;

  for (int wert : collection)
  {
    summe += wert;
  }

  return summe;
}
```

Auto-Boxing

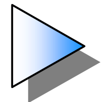
Auto-Unboxing

Fazit

- Im letzten Beispiel mit typisierten Collections und neuer `for`-Schleife sieht man deutlich, in welche Richtung die 5.0-Spracherweiterungen zielen:
  - Einfacherer (weniger) Code
  - Vordergründig einfacher lesbarer Code
  - Rückwärtskompatibilität
- Dies ist, v.a. beim Autoboxing, mit einigen Merkwürdigkeiten erkauft:
  - Probleme bei Wert-Identität
  - Konvertierungen (`Long l = 5;` produziert Fehler)
- Autoboxing ist eine Technik "for convenience", keine Reform des Paradigmas der Sprache.

Autoboxing und statische Imports – Inhalt

Autoboxing


statische Imports

Motivation (1)

- Der Umgang mit Service-Methoden (z.B. Mathematischen Routinen) ist in Java durch die konsequente Ansiedlung dieser Methoden an einen Typ nicht so einfach wie in anderen Programmiersprachen (z.B. C).
- Beispiel: Berechnung der Rotation eines Punktes

$$x' = x \cdot \cos\theta + y \cdot \sin\theta$$

$$y' = -x \cdot \sin\theta + y \cdot \cos\theta$$

```
double theta = ...;
double x = ...;
double y = ...;

double x1 = x * Math.cos(theta) + y * Math.sin(theta);
double y1 = -x * Math.sin(theta) + y * Math.cos(theta);
```

Wird von vielen als überflüssig empfunden

Motivation (2)

- Konstantenvererbung
  - Vor der Einführung von Enums wurden symbolische Konstanten gerne in Interfaces ausgelagert, die die benutzenden Klassen dann implementiert haben → Qualifizierung durch Typnamen entfällt
  - Beispiel: **`javax.swing.WindowConstants`**, wird von **`JFrame`** und **`JDialog`** "implementiert", beinhaltet aber nur symbolische Konstanten.

```
this.setDefaultCloseOperation(DISPOSE_ON_CLOSE);
```

statt

```
this.setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
```

Statische Imports

- Den Wünschen nach Vereinfachung der Funktions-Benutzung bzw. der Vermeidung von Interface-Konstanten-Vererbung wird mit den statischen Imports Rechnung getragen.
- Statische Elemente einer Klasse können durch einen statischen Import im Geltungsbereich einer anderen Klasse sichtbar gemacht werden:

```
import static java.lang.Math.*;
```

- Mit diesem Statement sind alle statischen Member (Methoden und Attribute) in der enthaltenden Klasse ohne Qualifizierung sichtbar.

Beispiele

```
import static java.lang.Math.*;
```

```
double theta = ...;
double x = ...;
double y = ...;

double x1 = x * cos(theta) + y * sin(theta);
double y1 = -x * sin(theta) + y * cos(theta);
```

```
import static javax.swing.WindowConstants.*;
```

```
// ist jetzt auch ohne Vererbung möglich
this.setDefaultCloseOperation(DISPOSE_ON_CLOSE);
```

Qualifizierte Imports

- Sinnigerweise wird man, um zu viele (potentielle) Überdeckungen zu vermeiden, nur qualifiziert importieren.
- Qualifizierte Imports funktionieren, analog zu normalen Imports, durch gezieltes Ansprechen der importierten Entität, hier allerdings bis zur Ebene der Klassenelemente:

```
import static java.lang.Math.sin;
import static java.lang.Math.cos;
```

```
double x1 = x * cos(theta) + y * sin(theta);
double y1 = -x * sin(theta) + y * cos(theta);

double tutNicht = sqrt(x * x); // Compile-Fehler
```

```
import static javax.swing.WindowConstants.DISPOSE_ON_CLOSE;
```

Überdeckungen

- Lokal definierte Klassenelemente überdecken (Namensgleichheit natürlich vorausgesetzt!) importierte.
- Der Compiler erzeugt hierzu keine Warnung, analog zu normalen Imports.
- Compile-Fehler gibt es erst, wenn durch die Imports Namensüberdeckungen (Ambiguitäten) entstehen.
- Ambiguitäten tauchen aber erst bei *Benutzung* importierter Elemente auf, ein potentieller Konflikt aufgrund der Import-Liste genügt nicht.

© aformatik Training & Consulting GmbH & Co. KG

Beispiel für Überdeckungen

LoggingDienst
+log(text : String) +log(wert : double) +log(wert : int)

Mathematik
+log(wert : double) : double

```
import static logging.LoggingDienst.log;
import static rechnen.Mathematik.log;
```

- Dies erzeugt noch keinen Fehler!
- Erst die Benutzung im Programmcode:

```
log(0.0);
```

```
reference to log is ambiguous, both method log(double) in
logging.LoggingDienst and method log(double) in rechnen.Mathematik
match
    log(0.0);
```

Anwendungshinweise

- Das neu eingeführte Konzept der Enumerations (Enums) wird das Erben von Konstanten-Interfaces obsolet machen.
- Statische Imports nur dann, wenn durch intensiven Gebrauch von Funktionen (hier: statische Methoden mit Rückgabewert) der Quelltext durch die Qualifizierung mit dem Klassennamen schwer lesbar wird.
- Ansonsten gilt: Java ist eine objektorientierte Sprache. In der Objektorientierung sind sämtliche Elemente immer einer Klasse bzw. deren Objekten zugeordnet. **Diese bewusste Zuordnung sollte möglichst selten verschleiert werden**.

Autoboxing und statische Imports

Fragen