Objektorientierte Programmierung

Kapitel 19: Wrapper-Klassen

Stefan Brass

Martin-Luther-Universität Halle-Wittenberg

Wintersemester 2014/15

http://www.informatik.uni-halle.de/~brass/oop14/

Inhalt

Einführung und Motivation (1)

- Die Typen in Java gliedern sich in
  - Primitive Typen und
  - Referenz-Typen.
- Für Referenz-Typen ist `Object` ein gemeinsamer Obertyp.
- D.h. man kann einen beliebigen Wert eines Referenztyps (Klasse, Interface, Array) einer Variablen vom Typ `Object` zuweisen.
  - Z.B. für Methoden praktisch: An einen Parameter vom Typ `Object` kann man jeden Wert eines Referenztyps binden.
  - Wenn man allgemein verwendbare Datenstrukturen wie z.B. Listen programmiert, kann man als Element-Typ `Object` verwenden, und jeden Referenztyp speichern.

Einführung und Motivation (2)

- Für Werte primitiver Typen geht dies zunächst nicht:
  - Sie sind grundlegend anders als Referenztypen.

    Z.B. belegen Variablen von einem Referenztyp alle gleich viel Speicher (die genaue Größe ist nicht vorgeschrieben, 32 Bit wären aber typisch). Ein `double` benötigt (mindestens) 64 Bit, ein `byte` dagegen nur 8 Bit.
  - `Object` ist kein Obertyp von primitiven Typen.
- Damit man Werte primitiver Typen aber doch an Methoden übergeben kann, die einen Parameter vom Typ `Object` haben, wurden die "Wrapper-Klassen" eingeführt:
  - z.B. kann man einen `int`-Wert in einem Objekt der Klasse `Integer` "verpacken"/"einwickeln" (engl. "to wrap").

    Deutsch werden sie manchmal auch als "Hüllklassen" genannt.
  - Es gibt eine solche Klasse für jeden primitiven Typ.

Einführung und Motivation (3)

- Im Prinzip enthält ein Objekt vom Typ `Integer` also einen einzelnen `int`-Wert:
  - Diesen Wert muß man bei der Objekterzeugung angeben (Parameter des Konstruktors), z.B.

```
Integer o = new Integer(3);
```

  - Man kann den Wert mit der Methode `intValue()` wieder abfragen:

```
int i = o.intValue(); // i ist 3
```

    Die Methode heißt `intValue()` und nicht einfach `value()`, weil es auch andere Methoden gibt, mit denen man automatisch eine Typumwandlung durchführen kann, s.u.

- Der im Objekt gespeicherte Wert ist nicht änderbar.

Einführung und Motivation (4)

- Da `Integer` eine Klasse ist, vergleicht `==` die Referenzen und nicht den Inhalt:

```
Integer o1 = new Integer(3);
Integer o2 = new Integer(3);
if(o1 == o2)
    System.out.println("gleich");
else
    System.out.println("verschieden");
```

  Dieses Testprogramm druckt "`verschieden`" aus.

- Die Methode `equals(...)` vergleicht wie üblich den Inhalt:

```
if(o1.equals(o2))
    System.out.println("Aber equals ist true!");
```

  Die Bedingung ist wahr, die Ausgabe wird also ausgeführt.

Einführung und Motivation (5)

- Tabelle der Wrapper-Klassen:

Primitiver Typ	Wrapper-Klasse
`boolean`	`Boolean`
`char`	`Character`
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`

Dem Namen der Klasse ist jeweils ein Link zur Dokumentation hinterlegt.

Meist heißt die Klasse wie der primitive Typ, nur mit großem Anfangsbuchstaben. Ausnahmen sind nur `Integer` und `Character`.

Einführung und Motivation (6)

- Es gibt eine gemeinsame abstrakte Oberklasse `Number` für `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`.

  [http://docs.oracle.com/javase/7/docs/api/java/lang/Number.html]

- Diese Klasse hat noch weitere Unterklassen, z.B. `BigInteger` und `BigDecimal` für beliebig große Zahlen.

- Jede Wrapper-Klasse `T` implementiert das Interface `comparable<T>`, (mit der Methode `compareTo(...)`).

  Und das Interface `Serializable`.

- Die Klassen haben auch (s.u.)
  - statische Methoden für den jeweiligen primitiven Typ, z.B. `parseInt(...)` zur Umwandlung von Strings in den Typ,
  - Konstanten für Begrenzungen des Zahlbereichs.

Inhalt

Methoden (1)

- Alle numerischen Wrapper-Klassen erlauben es, den im Objekt gespeicherten Wert mit verschiedenen Typen abzufragen:
  - `byte byteValue()`
  - `short shortValue()`
  - `int intValue()`
  - `long longValue()`
  - `float floatValue()`
  - `double doubleValue()`
- Natürlich wird man im Normalfall die Methode für den jeweils gespeicherten Typ verwenden.

  Man kann so aber eine Typ-Umwandlung durchführen, z.B. runden. Wie bei einem expliziten Typ-Cast werden im Notfall Bits abgeschnitten und der Wert möglicherweise zerstört. Es gibt dabei keine Exception.

Methoden (2)

- Für die beiden anderen Klassen bekommt den gespeicherten Wert entsprechend mit
  - `charValue()` für Objekte der Klasse `Character`
  - `booleanValue()` für Objekte der Klasse `Boolean`.
- Außerdem gibt es natürlich die von der Klasse `Object` ererbten Methoden, insbesondere
  - `boolean equals(Object obj)`
  - `String toString()`
- Und die Methode des `comparable<T>`-Interfaces, z.B. für die Klasse `Integer`:
  - `int compareTo(Integer o)`

    0/positiv/negativ, falls Wert in `this` gleich/größer/kleiner als Wert in `o`.

Erzeugung von Objekten

- Alle Wrapper-Klassen (mit Ausnahme von `Character`) haben zwei Konstruktoren (`Character` nur den ersten):
  - einen mit einem Argument von dem jeweiligen primitiven Typ,

    `Float` hat zusätzlich einen Konstruktor mit `double`-Argument.
  - einen mit einem Argument vom Typ `String`, z.B.

    ```
    Integer obj = new Integer("123");
    ```

    Bei den Zahl-Typen gibt es notfalls eine `NumberFormatException`.
- Außerdem gibt es eine statische Methode `valueOf(...)`, die ein Objekt der jeweiligen Wrapper-Klasse liefert:

  ```
  Integer obj = Integer.valueOf(3);
  ```

  Wie beim Konstrukor gibt es Varianten mit dem jeweiligen Typ, und mit einem String als Parameter (außer bei `Character`). Bei den ganzzahligen Typen gibt es noch eine dritte Variante, bei der man die Basis wählen kann (z.B. 8 für oktal). Im Unterschied zum Konstruktor erhält man nicht unbedingt ein neues Objekt.

Umwandlung zwischen Wert und String (1)

- Eine Umwandlung von einer Zeichenkette in ein Objekt der Wrapper-Klasse ist möglich mit
  - Konstruktor mit Parameter vom Typ `String`,
  - statische Methode `valueOf(String s)`.
- Eine Umwandlung von einem Objekt der Wrapper-Klasse in einen String ist möglich mit
  - Methode `toString()`: Von `Object` geerbt.
- Die Klasse enthält aber auch statische Methoden, um direkt zwischen einem Wert des primitiven Typs und `String` umzuwandeln (siehe nächste Folie).

Umwandlung zwischen Wert und String (2)

- Eine Umwandlung von einer Zeichenkette in einen Wert des primitiven Typs ist möglich mit der statischen Methode
  - `static T parseT(String s)`, wobei `T` durch den primitiven Typ zu ersetzen ist, z.B.

    ```
    int i = Integer.parseInt("123");
    ```

    Nur `Character` hat das nicht. Die ganzzahligen Typen haben außerdem eine Variante mit zweiten Argument für die Basis, sowie die Methode `decode(String s)`, die einen Präfix wie `0x` für Basis 16 berücksichtigt.
- Die Umwandlung eines Wertes eines primitiven Typs in eine Zeichenkette ist möglich mit der statischen Methode
  - `static String toString(T n)`, wobei `T` der Typ ist.

    Die Klassen `Integer`/`Long` haben außerdem eine Variante mit zweitem Argument für die Basis, sowie Methoden wie `toHexString(int i)`, `toOctalString(int i)`, `toBinaryString(int i)` für spezielle Basen.

Weitere Statische Methoden (1)

- Die Klasse `Character` statische Methoden u.a. zur
  - Unicode-Unterstützung, z.B.
    `static int toCodePoint(char high, char low)`

    Viele Methoden sind dupliziert, und arbeiten entweder mit dem vollen Unicode-Bereich (`int`), oder mit UTF-16 Einheiten (`char`).
  - Zeichen-Klassifizierung, z.B.
    `static boolean isLetter(char ch)`
  - Umwandlung zwischen Groß- und Kleinbuchstaben, z.B.
    `static char toUpperCase(char ch)`
  - Umwandlung zwischen Ziffern und Zahlen, z.B.
    `static int digit(char ch, int radix)`

    Die umgekehrte Abbilung erhält man mit
    `static char forDigit(int d, int radix)`.

Weitere Statische Methoden (2)

- `Integer` hat verschiedene statische Funktionen zum Arbeiten mit ganzen Zahlen auf Bitebene.

  Z.B. liefert "`static int bitCount(int i)`" die Anzahl von 1 Bits, und "`static int numberOfLeadingZeros(int i)`" die Anzahl führender Nullen.

- `Double` hat
  - Methoden `isNaN` und `isInfinite` zum Test auf spezielle Werte,

    Es gibt diese Methoden jeweils als normale Methode ohne Parameter (dann wird der Wert aus dem Objekt genommen), und als statische Methode mit `double`-Parameter.

  - Methoden `doubleToLongBits` und `longBitsToDouble`, um zwischen `double`-Wert und interner Bit-Darstellung umrechnen zu können.

Konstanten

- Die Wrapper-Klassen für die numerischen Typen haben Konstanten für die Begrenzung des Zahlbereiches:
  - `MIN_VALUE`: Kleinstmöglicher Wert

    Z.B. ist `Integer.MIN_VALUE` $= -2^{31} = -2\,147\,483\,648$.
  - `MAX_VALUE`: Größtmöglicher Wert

    Z.B. ist `Integer.MAX_VALUE` $= -^{31} - 1 = 2\,147\,483\,647$.
  - `SIZE`: Speichergröße in Bits

    Z.B. ist `Integer.SIZE` $= 32$.
- Für die Gleitkomma-Typen gibt es weitere Konstanten, z.B. `NaN` ("not a number": Fehlerwert).
- Die Klasse `Character` hat viele Konstanten für die Unicode-Unterstützung.

Inhalt

Boxing und Unboxing (1)

- Wenn man vor Java 5 mit dem Wert im Objekt einer Wrapper-Klasse rechnen wollte, musste man
  - explizit den Wert aus dem Objekt herausholen, z.B. mit der Methode `intValue()`,
  - dann mit dem Wert des primitiven Typs rechnen,
  - anschließend explizit ein neues Objekt erzeugen.
- Das sieht z.B. so aus (mit Variable `obj` vom Typ `Integer`):

```
obj = Integer.valueOf(obj.intValue() + 1);
```

- Seit Java 5 fügt der Compiler den Methoden-Aufruf und die Objekt-Erzeugung automatisch ein:

```
obj = obj + 1;
```

Boxing und Unboxing (2)

- Intern geschehen bei

```
obj = obj + 1;
```

  die gleichen drei Schritte, wie oben genannt: Wert auspacken, eins addieren, neues Objekt erzeugen mit dem neuen Wert.

  Die Objekte der Wrapper-Klassen sind ja unveränderlich. Man kann also nicht einfach in dem existierenden Objekt den Wert ändern, sondern es muss ein neues Objekt erzeugt werden. Wenn es keine anderen Referenzen auf das alte Objekt gibt, wird es wahrscheinlich nach einiger Zeit vom "Garbage Collector" eingesammelt und recycled.

- Tatsächlich kann man sogar

```
obj++;
```

  schreiben: Auch hier wird ein neues `Integer`-Objekt mit dem um eins größeren Wert erzeugt und in `obj` gespeichert.

  D.h. die Referenz in `obj` zeigt jetzt auf das neue Objekt.

Boxing und Unboxing (3)

- Das "Aus- und Einpacken" ist auch einzeln nützlich.
- Eine Methode mit Parameter vom Typ `Object`, z.B.

  ```
  static void m(Object o) { ... }
  ```

  kann man auch mit Werten beliebiger primitiver Typen aufrufen, ohne explizit eine Umwandlung in ein Objekt aufschreiben zu müssen: `m(5);`
- Intern wird ein Objekt vom Typ `Integer` erzeugt (mit dem Wert 5), und dieses Objekt an die Methode übergeben.

  `Object` ist (indirekte) Oberklasse aller Klassen, auch von `Integer`.
- Dies ist das "Auto-Boxing" (oder einfach "Boxing").
- Das Gleiche geschieht bei der einfachen Zuweisung

  ```
  Integer obj = 5;
  ```

Boxing und Unboxing (4)

- Das automatische Auspacken ("Auto-Unboxing" oder einfach "Unboxing") ist z.B. nützlich, wenn man ein Objekt einer Wrapper-Klasse mit einem Wert eines primitiven Typs vergleicht:

```
if(obj == 100) { ... }
```

- Das ist kein Typfehler, sondern es wird der in `obj` gespeicherte Wert mit 100 verglichen.

  Natürlich ist es nur dann kein Typfehler, wenn `obj` einer numerischen Wrapper-Klasse angehört. Das Auto-Unboxing ist eine Spezialbehandlung der Wrapper-Klassen, die man nicht für eigene Klassen haben kann. Falls `obj` die Null-Referenz enthält, gibt es eine `NullPointerException`.

- Wenn natürlich auf beiden Seiten vom `==` ein Objekt einer Wrapper-Klasse steht, werden die Referenzen verglichen, nicht die gespeicherten Werte.

Boxing und Unboxing (5)

- Weil dagegen bei <= u.s.w. immer ausgepackt wird, kann folgende paradoxe Situation auftreten:
  - `x <= y` und
  - `x >= y`, aber
  - `x != y` (wenn `x` und `y` den gleichen Wert enthalten).
- Überraschung: Autoboxing erzeugt nicht immer neue Objekte, sondern nimmt für kleine Zahlen vorgefertigte Objekte:

```
Integer o1 = 5; // implizit: Integer.valueOf(5);
Integer o2 = 5;
if(o1 == o2) // ist wahr!
    System.out.println("Gleiches Objekt!");
```

Dies gilt für alle Werte der Typen `boolean` und `byte`, für `char`-Werte, die ASCII-Codes entsprechen (bis 127), und für `short` und `int` von -128 bis $+127$.

Automatische Typ-Anpassungen (1)

- Die Liste der automatischen Typ-Anpassungen bei Zuweisungen muss nun erweitert werden:
  - "Widening primitive conversions": Von einem kleineren primitiven Typ zu einem größeren, z.B. `int` $\longrightarrow$ `double`.
  - "Widening reference conversions": Von einer Unterklasse zu einer Oberklasse (oder implementierten Interface, allgemein Obertyp), z.B. `Student` $\longrightarrow$ `Person`.
  - "Boxing conversion, optionally followed by a widening reference conversion":
    Z.B. `int` $\longrightarrow$ `Integer` $\longrightarrow$ `Object`.
  - "Unboxing conversion, optionally followed by a widing primitive conversion":
    Z.B. `Integer` $\longrightarrow$ `int` $\longrightarrow$ `double`.

Automatische Typ-Anpassungen (2)

- Automatische Typ-Anpassungen bei Zuweisungen, Forts.:
  - Falls rechts ein konstanter Ausdruck steht, und er passt in den kleinen ganzzahligen Typ links, sind auch verkleinernde Umwandlungen ("narrowing primitive conversions") möglich, z.B. von `5` (formal ein `int`) nach `byte` oder `Byte`.
  - Nach den obigen Regeln geht z.B. Boxing nicht mehr nach einer Erweiterung des primitiven Typs, und tatsächlich liefert folgende Zuweisung einen "incompatible types" Fehler:

```
Double d = 1; // Falsch!
```

    Die Umwandlungs-Kette `int` $\longrightarrow$ `double` $\longrightarrow$ `Double` geht also nicht.
    Bei Bedarf muss man `1.0` schreiben, oder auch "`(double) 1`".

- Die Regeln gelten (bis auf die Ausnahme für konstante Ausdrücke) auch für die Parameter-Übergabe.

Automatische Typ-Anpassungen (3)

- Bei Operatoren wie den vier Grundrechenarten, numerischen Vergleichen wie < etc. wird die "Binary Numeric Promotion" angewendet:
  - Ist einer der beiden Operanden von einem Referenztyp, wird Unboxing durchgeführt.

    Das geht nur für die 8 Wrapper-Klassen, sonst hat man einen Typfehler.
  - Ist dann einer der Operanden von Typ `double`, wird der andere nach `double` konvertiert.
  - Ansonsten, ist einer vom Typ `float`, wird der andere nach `float` konvertiert.
  - Sonst: Ist einer `long`, wird der andere nach `long` konvertiert.
  - Sonst werden beide nach `int` konvertiert.

    Auch bei der Addition von zwei `byte`-Werten wird also mit `int` gerechnet.

Automatische Typ-Anpassungen (4)

- Ist bei `+` einer der Operanden (egal ob links oder rechts) vom Typ `String`, wird auf den anderen die "String Conversion" angewendet:
  - Formal wird ein Wert eines primitiven Typs zuerst in eine Wrapper-Klasse umgewandelt, z.B. mit `Integer(x)`.

    Die Typen `byte` und `short` werden auch nach `Integer` umgewandelt, ansonsten jeder primitive Typ in seine zugehörige Wrapper-Klasse.
  - Danach gibt es also nur noch Referenz-Typen. Ist der Wert `null`, wird dieser in den String `"null"` umgewandet.

    Hier gibt es also keine `NullPointerException`.
  - Sonst geschieht die Umwandlung durch Aufruf der Methode `toString()`, die schon in `Object` definiert ist, aber in Subklassen überschrieben sein kann.

    Insbesondere ist sie in den Wrapper-Klassen überschrieben.