Script generated by TTT

Title: Seidl: EOE1 (26.01.2012)

Date: Thu Jan 26 12:31:57 CET 2012

Duration: 86:25 min

Pages: 39

Fakt:

- Die Klasse `Object` ist eine gemeinsame Oberklasse für alle Klassen.
- Eine Klasse ohne angegebene Oberklasse ist eine direkte Unterklasse von `Object`.
- Einige nützliche Methoden der Klasse `Object` :
  - `String toString()` liefert (irgendeine) Darstellung als `String`;
  - `boolean equals(Object obj)` testet auf Objekt-Identität oder Referenz-Gleichheit:

```
public boolean equals(Object obj) {
   return this==obj;
   }
```

...

Beispiel:

```
public class Poly {
   public String toString() { return "Hello"; }
}
public class PolyTest {
   public static String addWorld(Object x) {
      return x.toString()+" World!";
   }
   public static void main(String[] args) {
      Object x = new Poly();
      System.out.print(addWorld(x)+"\n");
   }
}
```


- `int hashCode()` liefert eine einelonge Nummer für das Objekt.
- ... viele weitere geheimnisvolle Methoden, die u.a. mit paralleler Programm-Ausführung zu tun haben.

Achtung:

`Object`-Methoden können aber (und sollten evt.) in Unterklassen durch geeignetere Methoden überschrieben werden.

Beispiel:

```
public class Poly {
   public String toString() { return "Hello"; }
}
public class PolyTest {
   public static String addWorld(Object x) {
      return x.toString()+" World!";
   }
   public static void main(String[] args) {
      Object x = new Poly();
      System.out.print(addWorld(x)+"\n");
   }
}
```


... liefert:

```
Hello World!
```

- Die Klassen-Methode `addWorld()` kann auf jedes Objekt angewendet werden.
- Die Klasse `Poly` ist eine Unterklasse von `Object`.
- Einer Variable der Klasse A kann ein Objekt jeder Unterklasse von A zugewiesen werden.
- Darum kann `x` das neue `Poly`-Objekt aufnehmen.

Bemerkung:

- Die Klasse `Poly` enthält keinen explizit definierten Konstruktor.
- Eine Klasse A, die keinen anderen Konstruktor besitzt, enthält implizit den trivialen Konstruktor `public A () {}`.

Achtung:

```
public class Poly {
   public String greeting() {
      return "Hello";
   }
}
public class PolyTest {
   public static void main(String[] args) {
      Object x = new Poly();
      System.out.print(x.greeting()+" World!\n");
   }
}
```

... liefert ...

Beispiel:

```
public class Poly {
   public String toString() { return "Hello"; }
}
public class PolyTest {
   public static String addWorld(Object x) {
      return x.toString()+" World!";
   }
   public static void main(String[] args) {
      Object x = new Poly();
      System.out.print(addWorld(x)+"\n");
   }
}
```

Achtung:

```
public class Poly {
   public String greeting() {
      return "Hello";
   }
}
public class PolyTest {
   public static void main(String[] args) {
      Object x = new Poly();
      System.out.print(x.greeting()+" World!\n");
   }
}
```

... liefert ...

... einen Compiler-Fehler:

```
Method greeting() not found in class java.lang.Object.
    System.out.print(x.greeting()+" World!\n");
                                 ^
1 error
```

- Die Variable `x` ist als `Object` deklariert.
- Der Compiler weiss nicht, ob der aktuelle Wert von `x` ein Objekt aus einer Unterklasse ist, in welcher die Objekt-Methode `greeting()` definiert ist.
- Darum lehnt er dieses Programm ab.

Ausweg:

- Benutze einen expliziten cast in die entsprechende Unterklasse!

```
public class Poly {
   public String greeting() { return "Hello"; }
}
public class PolyTest {
   public void main(String[] args) {
      Object x = new Poly();
      if (x instanceof Poly)
         System.out.print(((Poly) x).greeting()+" World!\n");
      else
         System.out.print("Sorry: no cast possible!\n");
   }
}
```


Fazit:

- Eine Variable `x` einer Klasse A kann Objekte `b` aus sämtlichen Unterklassen B von A aufnehmen.
- Durch diese Zuweisung vergisst Java die Zugehörigkeit zu B, da Java alle Werte von `x` als Objekte der Klasse `A` behandelt.
- Mit dem Ausdruck `x instanceof B` können wir zur Laufzeit die Klassenzugehörigkeit von `x` testen ;-)
- Sind wir uns sicher, dass `x` aus der Klasse B ist, können wir in diesen Typ casten.
- Ist der aktuelle Wert der Variablen `x` bei der Überprüfung tatsächlich ein Objekt (einer Unterklasse) der Klasse B, liefert der Ausdruck genau dieses Objekt zurück. Andernfalls wird eine ↑Exception ausgelöst.

Beispiel: Unsere Listen

```
public class List {
   public Object info;
   public List next;
   public List(Object x, List l) {
      info=x; next=l;
   }
   public void insert(Object x) {
      next = new List(x,next);
   }
   public void delete() {
      if (next!=null) next=next.next;
   }
   ...
```



```
   public String toString() {
      String result = "["+info;
      for (List t=next; t!=null; t=t.next)
         result=result+", "+t.info;
      return result+"]";
   }
   ...
} // end of class List
```

- Die Implementierung funktioniert ganz analog zur Implementierung für `int`.
- Die `toString()`-Methode ruft implizit die (stets vorhandene) `toString()`-Methode für die Listen-Elemente auf.

... aber Achtung:

```
...
Poly x = new Poly();
List list = new List (x);
x = list.info;
System.out.print(x+"\n");
...
```

liefert ...

... einen Compiler-Fehler, da der Variablen `x` nur Objekte einer Unterklasse von `Poly` zugewiesen werden dürfen.

Stattdessen müssen wir schreiben:

```
...
Poly x = new Poly();
List list = new List (x);
x = (Poly) list.info;
System.out.print(x+"\n");
...
```

Das ist hässlich !!! Geht das nicht besser ???

13.2 Generische Klassen

Idee:

- Seit Version 1.5 verfügt Java über generische Klassen ...
- Anstatt das Attribut `info` als `Object` zu deklarieren, geben wir der Klasse einen Typ-Parameter `T` für `info` mit !!!
- Bei Anlegen eines Objekts der Klasse `List` bestimmen wir, welchen Typ `T` und damit `info` haben soll ...

Beispiel: Unsere Listen

```
public class List<T> {
   public T info;
   public List<T> next;
   public List (T x, List<T> l) {
      info=x; next=l;
   }
   public void insert(T x) {
      next = new List<T> (x,next);
   }
   public void delete() {
      if (next!=null) next=next.next;
   }
   ...
```



```
   public static void main (String [] args) {
      List<Poly> list = new List<Poly> (new Poly(),null);
      System.out.print (list.info.greeting()+"\n");
   }
} // end of class List
```



```
   public static void main (String [] args) {
      List<Poly> list = new List<Poly> (new Poly(),null);
      System.out.print (list.info.greeting()+"\n");
   }
} // end of class List
```


- Die Implementierung funktioniert ganz analog zur Implementierung für `Object`.
- Der Compiler weiß aber nun in `main`, dass `list` vom Typ `List` ist mit Typ-Parameter `T` = `Poly`.
- Deshalb ist `list.info` vom Typ `Poly`.
- Folglich ruft `list.info.greeting()` die entsprechende Methode der Klasse `Poly` auf.

Bemerkungen:

- Typ-Parameter dürfen nur in den Typen von Objekt-Attributen und Objekt-Methoden verwendet werden, weil ihre Werte durch den Aufruf eines Konstruktors festgelet werden !!!
- Jede Unterklasse einer parametrisierten Klasse muss mindestens die gleichen Parameter besitzen:

`A<S,T> extends B<T>` ist erlaubt.

`A<S> extends B<S,T>` ist verboten.

- `Poly` ist eine Unterklasse von `Object` ; aber `List<Poly>` ist keine Unterklasse von `List<Object>` !!!

Bemerkungen (Forts.):

- Für einen Typ-Parameter T kann man auch eine Oberklasse oder ein Interface angeben, das T auf jeden Fall erfüllen soll ...

```
public interface Executable {
   void execute ();
}
public class ExecutableList<E extends Executable> {
   E element;
   ExecutableList<E> next;
   void executeAll () {
        element.execute ();
        if (next == null) return;
        else next.executeAll ();
} }
```


376

Bemerkungen (Forts.):

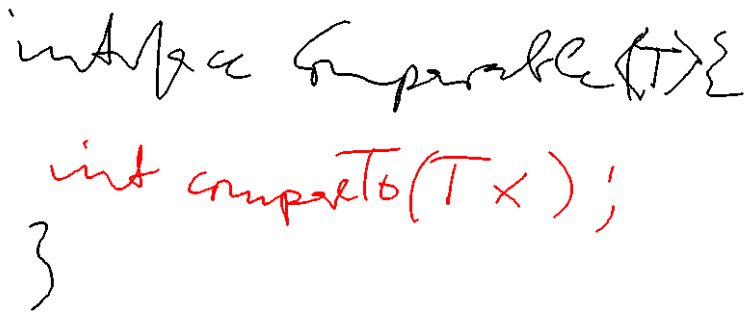
- Für einen Typ-Parameter T kann man auch eine Oberklasse oder ein Interface angeben, das T auf jeden Fall erfüllen soll ...

```
public interface Executable {
   void execute ();
}
public class ExecutableList<E extends Executable> {
   E element;
   ExecutableList<E> next;
   void executeAll () {
        element.execute ();
        if (next == null) return;
        else next.executeAll ();
} }
```


Bemerkungen (Forts.):

- Beachten Sie, dass hier ebenfalls das Schlüsselwort extends benutzt wird!
- Auch gelten hier weitere Beschränkungen, wie eine parametrisierte Klasse eine Oberklasse sein kann.
- Auch Interfaces können parametrisiert werden.
- Insbesondere kann Comparable parametrisiert werden – und zwar mit der Klasse, mit deren Objekten man vergleichen möchte

```
public class Test implements Comparable<Test> {
   public int compareTo (Test x) { return 0; }
}
```



Bemerkungen (Forts.):

- Typparameter können auch lokal für eine Methode eingesetzt werden. Eine Deklaration:

```
public static <T> List<T> create () {
   return new List<T>();
}
```

könnte in jeder anderen Klasse stehen. Der Typparameter `T` wird dann an der Aufrufstelle festgelegt. Der Aufruf

```
List<String> list = create ();
```

instantiiert z.B. den Parameter `T` mit `String`.

13.3 Wrapper-Klassen

... bleibt ein Problem:

- Der Datentyp `String` ist eine Klasse;
- Felder sind Klassen; aber
- Basistypen wie `int`, `boolean`, `double` sind keine Klassen!
  (Eine Zahl ist eine Zahl und kein Verweis auf eine Zahl.

13.3 Wrapper-Klassen

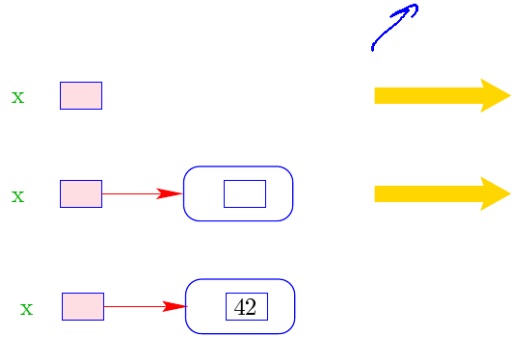
... bleibt ein Problem:

- Der Datentyp `String` ist eine Klasse;
- Felder sind Klassen; aber
- Basistypen wie `int`, `boolean`, `double` sind keine Klassen!
  (Eine Zahl ist eine Zahl und kein Verweis auf eine Zahl.

Ausweg:

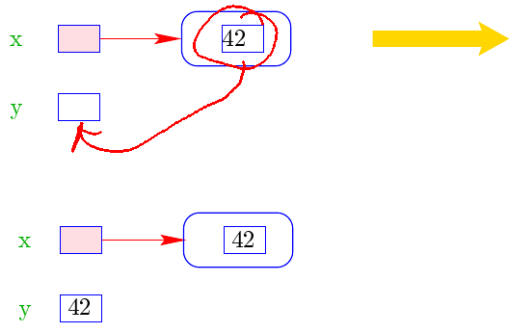
- Wickle die Werte eines Basis-Typs in ein Objekt ein!
  $\Longrightarrow$ Wrapper-Objekte aus Wrapper-Klassen.

Die Zuweisung `Integer x = new Integer(42);` bewirkt:

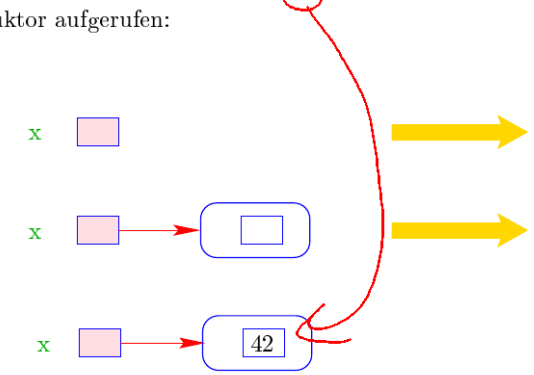


Eingewickelte Werte können auch wieder ausgewickelt werden.

Seit Java 1.5 erfolgt bei einer Zuweisung `int y = x;` eine automatische Konvertierung:



Umgekehrt wird bei Zuweisung eines `int`-Werts an eine `Integer`-Variable: `Integer x = 42;` automatisch der Konstruktor aufgerufen:



Gibt es erst einmal die Klasse `Integer`, lassen sich dort auch viele andere nützliche Dinge ablegen.

Zum Beispiel:

- `public static int MIN_VALUE = -2147483648;` liefert den kleinsten `int`-Wert;
- `public static int MAX_VALUE` = 2147483647; liefert den größten `int`-Wert;
- `public static int parseInt(String s) throws NumberFormatException;` berechnet aus dem `String`-Objekt `s` die dargestellte Zahl — sofern `s` einen `int`-Wert darstellt.

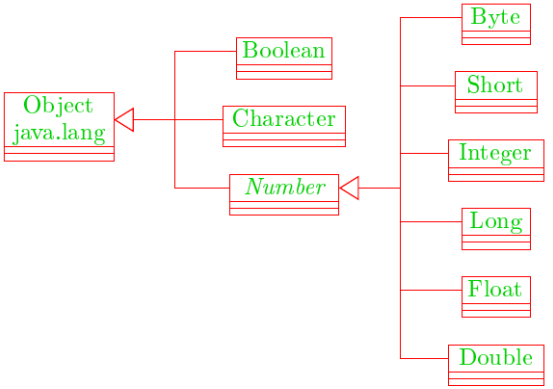
  Andernfalls wird eine ↑exception geworfen.

Bemerkungen:

- Außer dem Konstruktor: `public Integer(int value);` gibt es u.a. `public Integer(String s) throws NumberFormatException;`
- Dieser Konstruktor liefert zu einem `String`-Objekt `s` ein `Integer`-Objekt, dessen Wert `s` darstellt.
- `public boolean equals(Object obj);` liefert `true` genau dann wenn `obj` den gleichen `int`-Wert enthält.

- Sämtliche Wrapper-Klassen für Typen type (außer `char`) verfügen über
  - Konstruktoren aus Basiswerten bzw. `String`-Objekten;
  - eine statische Methode `type parseType(String s);`
  - eine Methode `boolean equals(Object obj)` (auch `Character`).
- Bis auf `Boolean` verfügen alle über Konstanten `MIN_VALUE` und `MAX_VALUE`.
- `Character` enthält weitere Hilfsfunktionen, z.B. um Ziffern zu erkennen, Klein- in Großbuchstaben umzuwandeln ...
- Die numerischen Wrapper-Klassen sind in der gemeinsamen Oberklasse `Number` zusammengefasst.
- Diese Klasse ist ↑abstrakt d.h. man kann keine `Number`-Objekte anlegen.

Wrapper-Klassen:



Spezialitäten:

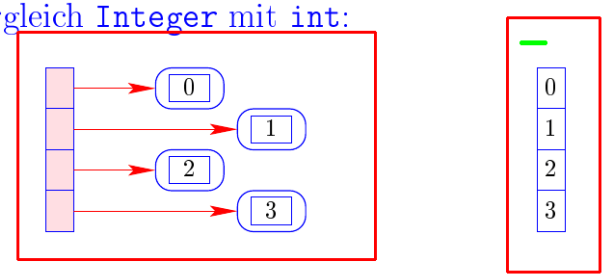
- `Double` und `Float` enthalten zusätzlich die Konstanten

`NEGATIVE_INFINITY`	=	$-1.0/0$
`POSITIVE_INFINITY`	=	$+1.0/0$
`NaN`	=	$0.0/0$

- Zusätzlich gibt es die Tests
  - `public static boolean isInfinite(double v);`
    `public static boolean isNaN(double v);`
    (analog für `float`)
  - `public boolean isInfinite();`
    `public boolean isNaN();`

  mittels derer man auf (Un)Endlichkeit der Werte testen kann.

Vergleich `Integer` mit `int`:

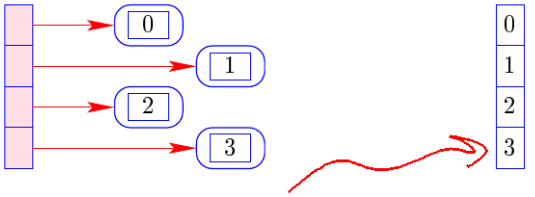


`Integer []` `int []`

- \+ `Integer`s können in polymorphen Datenstrukturen hausen.
- – Sie benötigen mehr als doppelt so viel Platz.
- – Sie führen zu vielen kleinen (evt.) über den gesamten Speicher verteilten Objekten $\Longrightarrow$ schlechteres Cache-Verhalten.

390

Vergleich `Integer` mit `int`:



`Integer []` `int []`

- \+ `Integer`s können in polymorphen Datenstrukturen hausen.
- – Sie benötigen mehr als doppelt so viel Platz.
- – Sie führen zu vielen kleinen (evt.) über den gesamten Speicher verteilten Objekten $\Longrightarrow$ schlechteres Cache-Verhalten.

14 Fehler-Objekte: Werfen, Fangen, Behandeln

- Tritt während der Programm-Ausführung ein Fehler auf, wird die normale Programm-ausführung abgebrochen und ein Fehler-Objekt erzeugt (geworfen).
- Die Klasse `Throwable` fasst alle Arten von Fehlern zusammen.
- Ein Fehler-Objekt kann gefangen und geeignet behandelt werden.

Idee: Explizite Trennung von

- normalem Programm-Ablauf (der effizient und übersichtlich sein sollte); und
- Behandlung von Sonderfällen (wie illegalen Eingaben, falscher Benutzung, Sicherheitsattacken, ...)