Script generated by TTT

Title: Westermann: Informatik_1 (09.01.2012)

Date: Mon Jan 09 17:15:52 CET 2012

Duration: 50:01 min

Pages: 29

2 Polymorphie (Vielgestaltigkeit)

Problem:

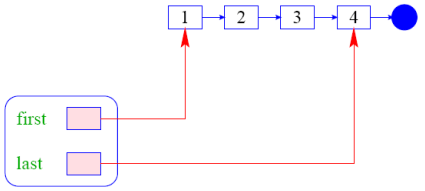
- Unsere Datenstrukturen und ADTs `List`, `Stack` und `Queue` können einzig und allein `int`-Werte aufnehmen.
- Wollen wir `String`-Objekte, andere Arten von Zahlen oder andere Objekttypen ablegen, müssen wir die jeweilige Datenstruktur nochmal für jeden speziellen Typ definieren

Implementierung von `dequeue()`:

- Falls nach Entfernen von `a[first]` die Schlange leer ist, werden `first` und `last` auf `-1` gesetzt.
- Andernfalls wird `first` um 1 (modulo der Länge von `a`) inkrementiert ...

Erste Idee:

- Realisiere Schlange mithilfe einer Liste :



- `first` zeigt auf das nächste zu entnehmende Element;
- `last` zeigt auf das Element, hinter dem eingefügt wird.

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();
      write(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 () {}`.

... 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 () {}`.

... 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();
      write(x.greeting()+" World!\n");
   }
}
```

... liefert ...

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)
         write(((Poly) x).greeting()+" World!\n");
      else
         write("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

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

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)
         write(((Poly) x).greeting()+" World!\n");
      else
         write("Sorry: no cast possible!\n");
   }
}
```


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)
         write(((Poly) x).greeting()+" World!\n");
      else
         write("Sorry: no cast possible!\n");
   }
}
```

65

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

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

66

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;
   }
   ...
```


Bemerkungen:

- 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 ();
}  }
```


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 ();
}  }
```

76

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; }
}
```



```
   public static void main (String [] args) {
      List<Poly> list = new List<Poly> (new Poly(),null);
      write (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 (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; }
}
```

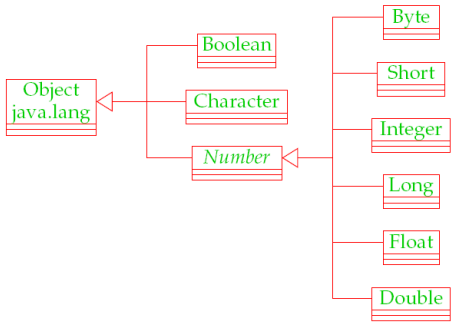
Gibt es erst einmal die Klasse `Integer`, lassen sich dort auch viele andere nützliche Dinge ablegen.

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

Wrapper-Klassen:



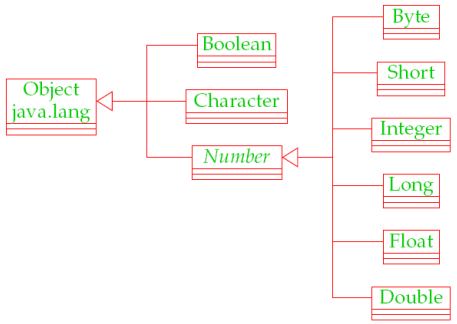
Bemerkungen:

- 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.

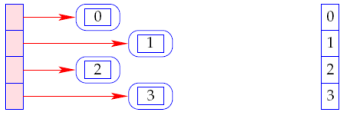
Ähnliche Wrapper-Klassen gibt es auch für die übrigen Basistypen ...

- 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:



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.