Script generated by TTT

Title: Seidl: EOE1 (19.01.2012)

Date: Thu Jan 19 12:30:14 CET 2012

Duration: 87:32 min

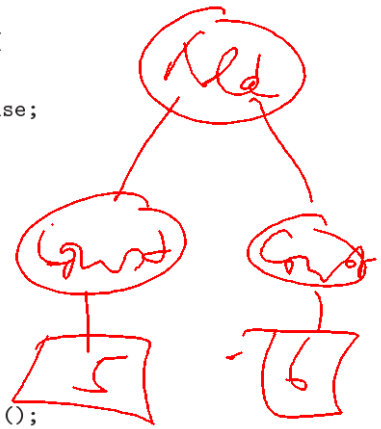
Pages: 39

12 Abstrakte Klassen, finale Klassen und Interfaces

- Eine abstrakte Objekt-Methode ist eine Methode, für die keine Implementierung bereit gestellt wird.
- Eine Klasse, die abstrakte Objekt-Methoden enthält, heißt ebenfalls abstrakt.
- Für eine abstrakte Klasse können offenbar keine Objekte angelegt werden.
- Mit abstrakten können wir Unterklassen mit verschiedenen Implementierungen der gleichen Objekt-Methoden zusammenfassen.

Beispiel: Auswertung von Ausdrücken

```
public abstract class Expression {
   private int value;
   private boolean evaluated = false;
   public int getValue() {
      if (evaluated) return value;
      else {
         value = evaluate();
         evaluated = true;
         return value;
      }
   }
   abstract protected int evaluate();
} // end of class Expression
```



- Die Unterklassen von `Expression` repräsentieren die verschiedenen Arten von Ausdrücken.
- Allen Unterklassen gemeinsam ist eine Objekt-Methode `evaluate()` – immer mit einer anderen Implementierung.

- Eine abstrakte Objekt-Methode wird durch das Schlüsselwort `abstract` gekennzeichnet.
- Eine Klasse, die eine abstrakte Methode enthält, muss selbst ebenfalls als `abstract` gekennzeichnet sein.
- Für die abstrakte Methode muss der vollständige Kopf angegeben werden – inklusive den Parameter-Typen und den (möglicherweise) geworfenen Exceptions.
- Eine abstrakte Klasse kann konkrete Methoden enthalten, hier: `int getValue()`.

Beispiel: Auswertung von Ausdrücken

```
public abstract class Expression {
   private int value;
   private boolean evaluated = false;
   public int getValue() {
      if (evaluated) return value;
      else {
         value = evaluate();
         evaluated = true;
         return value;
      }
   }
   abstract protected int evaluate();
} // end of class Expression
```


- Eine abstrakte Objekt-Methode wird durch das Schlüsselwort `abstract` gekennzeichnet.
- Eine Klasse, die eine abstrakte Methode enthält, muss selbst ebenfalls als `abstract` gekennzeichnet sein.
- Für die abstrakte Methode muss der vollständige Kopf angegeben werden – inklusive den Parameter-Typen und den (möglicherweise) geworfenen Exceptions.
- Eine abstrakte Klasse kann konkrete Methoden enthalten, hier: `int getValue()`.

- Die Methode `evaluate()` soll den Ausdruck auswerten.
- Die Methode `getValue()` speichert das Ergebnis in dem Attribut `value` ab und vermerkt, dass der Ausdruck bereits ausgewertet wurde.

Beispiel für einen Ausdruck:

```
public final class Const extends Expression {
   private int n;
   public Const(int x) { n=x; }
   protected int evaluate() {
      return n;
   }  // end of evaluate()
}     // end of class Const
```

- Der Ausdruck Const benötigt ein Argument. Dieses wird dem Konstruktor mitgegeben und in einer privaten Variable gespeichert.
- Die Klasse ist als `final` deklariert.
- Zu als final deklarierten Klassen dürfen keine Unterklassen deklariert werden !!!
- Aus Sicherheits- wie Effizienz-Gründen sollten so viele Klassen wie möglich als final deklariert werden ...
- Statt ganzer Klassen können auch einzelne Variablen oder Methoden als `final` deklariert werden.
- Finale Members dürfen nicht in Unterklassen umdefiniert werden.
- Finale Variablen dürfen zusätzlich nur initialisiert, aber nicht modifiziert werden $\Longrightarrow$ Konstanten.

... andere Ausdrücke:

```
public final class Add extends Expression {
   private Expression left, right;
   public Add(Expression l, Expression r) {
      left = l; right = r;
   }
   protected int evaluate() {
      return left.getValue() + right.getValue();
   }  // end of evaluate()
}     // end of class Add
public final class Neg extends Expression {
   private Expression arg;
   public Neg(Expression a) { arg = a; }
   protected int evaluate() { return -arg.getValue(); }
}     // end of class Neg
```


Beispiel: Auswertung von Ausdrücken

```
public abstract class Expression {
   private int value;
   private boolean evaluated = false;
   public int getValue() {
      if (evaluated) return value;
      else {
         value = evaluate();
         evaluated = true;
         return value;
      }
   }
   abstract protected int evaluate();
} // end of class Expression
```


... andere Ausdrücke:

```
public final class Add extends Expression {
   private Expression left, right;
   public Add(Expression l, Expression r) {
      left = l; right = r;
   }
   protected int evaluate() {
      return left.getValue() + right.getValue();
   }  // end of evaluate()
}     // end of class Add
public final class Neg extends Expression {
   private Expression arg;
   public Neg(Expression a) { arg = a; }
   protected int evaluate() { return -arg.getValue(); }
}     // end of class Neg
```

... die Funktion `main()` einer Klasse `TestExp`:

```
public static void main(String[] args) {
   Expression e = new Add (
      new Neg (new Const(8)),
      new Const(16));
   System.out.println(e.getValue())
   }
```

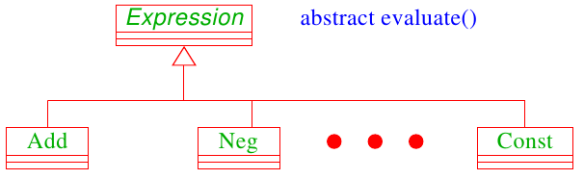
- Die Methode `getValue()` ruft eine Methode `evaluate()` sukzessive für jeden Teilausdruck von `e` auf.
- Welche konkrete Implementierung dieser Methode dabei jeweils gewählt wird, hängt von der konkreten Klasse des jeweiligen Teilausdrucks ab, d.h. entscheidet sich erst zur Laufzeit.
- Das nennt man auch dynamische Bindung.

... andere Ausdrücke:

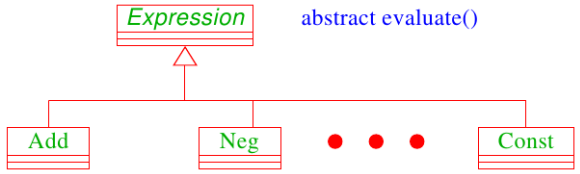
```
public final class Add extends Expression {
   private Expression left, right;
   public Add(Expression l, Expression r) {
      left = l; right = r;
   }
   protected int evaluate() {
      return left.getValue() + right.getValue();
   }  // end of evaluate()
}     // end of class Add
public final class Neg extends Expression {
   private Expression arg;
   public Neg(Expression a) { arg = a; }
   protected int evaluate() { return -arg.getValue(); }
}     // end of class Neg
```


- Eine abstrakte Objekt-Methode wird durch das Schlüsselwort `abstract` gekennzeichnet.
- Eine Klasse, die eine abstrakte Methode enthält, muss selbst ebenfalls als `abstract` gekennzeichnet sein.
- Für die abstrakte Methode muss der vollständige Kopf angegeben werden – inklusive den Parameter-Typen und den (möglicherweise) geworfenen Exceptions.
- Eine abstrakte Klasse kann konkrete Methoden enthalten, hier: `int getValue()`.

Die abstrakte Klasse *Expression*:

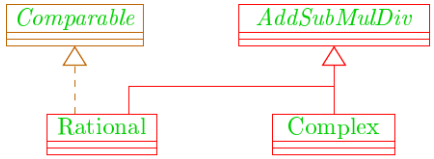


Die abstrakte Klasse *Expression*:



Leider (zum Glück?) lässt sich nicht die ganze Welt hierarchisch organisieren ...

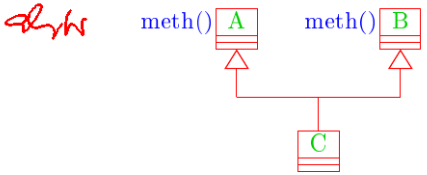
Beispiel:



AddSubMulDiv = Objekte mit Operationen `add()`, `sub()`, `mul()`, und `div()`

Comparable = Objekte, die eine `compareTo()`-Operation besitzen.

- Mehrere direkte Oberklassen einer Klasse führen zu konzeptuellen Problemen:
  - Auf welche Klasse bezieht sich `super` ?
  - Welche Objekt-Methode `meth()` ist gemeint, wenn wenn mehrere Oberklassen `meth()` implementieren ?



- Kein Problem entsteht, wenn die Objekt-Methode `meth()` in allen Oberklassen abstrakt ist.
- oder zumindest nur in maximal einer Oberklasse eine Implementierung besitzt.

Ein Interface kann aufgefasst werden als eine abstrakte Klasse, wobei:

- alle Objekt-Methoden abstrakt sind;
- es keine Klassen-Methoden gibt;
- alle Variablen Konstanten sind.

Beispiel:

```
public interface Comparable {
   int compareTo(Object x);
   }
```

- Object ist die gemeinsame Oberklasse aller Klassen.
- Methoden in Interfaces sind automatisch Objekt-Methoden und `public`.
- Es muss eine Obermenge der in Implementierungen geworfenen Exceptions angegeben werden.
- Evt. vorkommende Konstanten sind automatisch `public static`.

Beispiel (Forts.):

```
public class Rational extends AddSubMulDiv
                      implements Comparable {
private int zaehler, nenner;
public int compareTo(Object cmp) {
   Rational fraction = (Rational) cmp;
   long left = zaehler * fraction.nenner;
   long right = nenner * fraction.zaehler;
   if (left == right) return 0;
   else if (left < right) return -1;
   else return 1;
   } // end of compareTo
   ...
}    // end of class Rational
```


- `class` A `extends` B `implements` B1, B2,...,Bk `{...}` gibt an, dass die Klasse A als Oberklasse B hat und zusätzlich die Interfaces B1, B2,...,Bk unterstützt, d.h. passende Objekt-Methoden zur Verfügung stellt.
- Java gestattet maximal eine Oberklasse, aber beliebig viele implementierte Interfaces.
- Die Konstanten des Interface können in implementierenden Klassen direkt benutzt werden.
- Interfaces können als Typen für formale Parameter, Variablen oder Rückgabewerte benutzt werden.
- Darin abgelegte Objekte sind dann stets aus einer implementierenden Klasse.
- Expliziter Cast in eine solche Klasse ist möglich (und leider auch oft nötig).

Beispiel:

```
public interface Comparable {
   int compareTo(Object x);
   }
```

- Object ist die gemeinsame Oberklasse aller Klassen.
- Methoden in Interfaces sind automatisch Objekt-Methoden und `public`.
- Es muss eine Obermenge der in Implementierungen geworfenen Exceptions angegeben werden.
- Evt. vorkommende Konstanten sind automatisch `public static`.

Beispiel (Forts.):

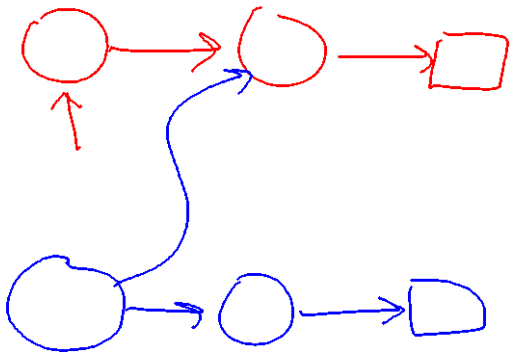
```
public class Rational extends AddSubMulDiv
                      implements Comparable {
private int zaehler, nenner;
public int compareTo(Object cmp) {
   Rational fraction = (Rational) cmp;
   long left = zaehler * fraction.nenner;
   long right = nenner * fraction.zaehler;
   if (left == right) return 0;
   else if (left < right) return -1;
   else return 1;
   } // end of compareTo
   ...
}    // end of class Rational
```


- `class` A `extends` B `implements` B1, B2,...,Bk `{...}` gibt an, dass die Klasse A als Oberklasse B hat und zusätzlich die Interfaces B1, B2,...,Bk unterstützt, d.h. passende Objekt-Methoden zur Verfügung stellt.
- Java gestattet maximal eine Oberklasse, aber beliebig viele implementierte Interfaces.
- Die Konstanten des Interface können in implementierenden Klassen direkt benutzt werden.
- Interfaces können als Typen für formale Parameter, Variablen oder Rückgabewerte benutzt werden.
- Darin abgelegte Objekte sind dann stets aus einer implementierenden Klasse.
- Expliziter Cast in eine solche Klasse ist möglich (und leider auch oft nötig).

- Interfaces können andere Interfaces erweitern oder gar mehrere andere Interfaces zusammenfassen.
- Erweiternde Interfaces können Konstanten umdefinieren...
- Kommt eine Konstante gleichen Namens const in verschiedenen implementierten Interfaces A und B vor, kann man sie durch A.const und B.const unterscheiden.

Beispiel (Forts.):

```
public interface Countable extends Comparable, Cloneable {
   Countable next();
   Countable prev();
   int number();
   }
```

- Interfaces können andere Interfaces erweitern oder gar mehrere andere Interfaces zusammenfassen.
- Erweiternde Interfaces können Konstanten umdefinieren...
- Kommt eine Konstante gleichen Namens const in verschiedenen implementierten Interfaces A und B vor, kann man sie durch A.const und B.const unterscheiden.

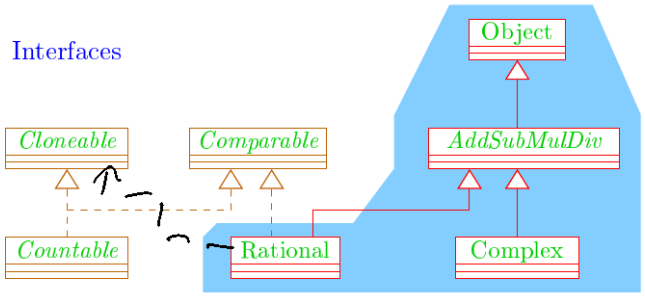
Beispiel (Forts.):

```
public interface Countable extends Comparable, Cloneable {
   Countable next();
   Countable prev();
   int number();
   }
```


Rational x = (Rational) obj.clone();

- Das Interface `Countable` umfasst die (beide vordefinierten) Interfaces `Comparable` und `Cloneable`.
- Das vordefinierte Interface `Cloneable` verlangt eine Objekt-Methode `public Object clone()` die eine Kopie des Objekts anlegt.
- Eine Klasse, die `Countable` implementiert, muss über die Objekt-Methoden `compareTo()`, `clone()`, `next()`, `prev()` und `number()` verfügen.

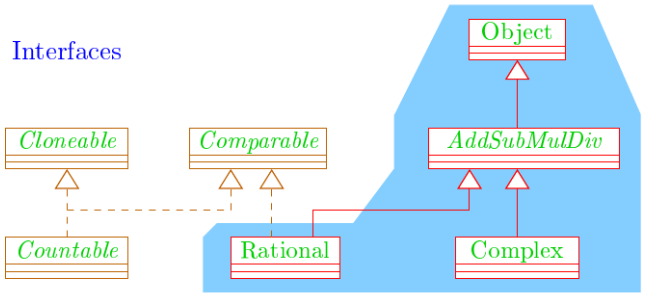
Übersicht:



Klassen-Hierarchie

354

Übersicht:



Klassen-Hierarchie

354

13 Polymorphie

Problem:

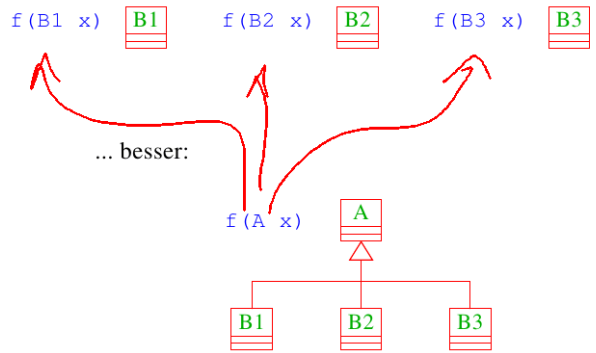
- Unsere Datenstrukturen `List`, `Stack` und `Queue` können einzig und allein `int`-Werte aufnehmen.
- Wollen wir `String`-Objekte oder andere Arten von Zahlen ablegen, müssen wir die jeweilige Datenstruktur grade nochmal definieren.

13.1 Unterklassen-Polymorphie

Idee:

- Eine Operation meth `( A x)` lässt sich auch mit einem Objekt aus einer Unterklasse von A aufrufen !!!
- Kennen wir eine gemeinsame Oberklasse Base für alle möglichen aktuellen Parameter unserer Operation, dann definieren wir meth einfach für Base ...
- Eine Funktion, die für mehrere Argument-Typen definiert ist, heißt auch polymorph.

Statt:



Betrachte einen Aufruf $e_0.f(e_1, \ldots, e_k)$.

Ermittlung der aufgerufenen Methode:

- Bestimme die statischen Typen $T_0, \ldots, T_k$ der Ausdrücke $e_0, \ldots, e_k$.
- Suche in einer Oberklasse von T_0 nach einer Methode mit Namen f, deren Liste von Argumenttypen bestmöglich zu der Liste $T_1, \ldots, T_k$ passt.

  Der Typ I dieser rein statisch gefundenen Methode ist die Signatur der Methode f an dieser Aufrufstelle im Programm.
- Der dynamische Typ D des Objekts, zu dem sich e_0 auswertet, gehört zu einer Unterklasse von T_0.
- Die Methode f wird nun aufgerufen, deren Typ I ist und die in der nächsten Oberklasse von D implementiert wird.

Betrachte einen Aufruf $e_0.f(e_1, \ldots, e_k)$.

Ermittlung der aufgerufenen Methode:

- Bestimme die statischen Typen $T_0, \ldots, T_k$ der Ausdrücke $e_0, \ldots, e_k$.
- Suche in einer Oberklasse von T_0 nach einer Methode mit Namen f, deren Liste von Argumenttypen bestmöglich zu der Liste $T_1, \ldots, T_k$ passt.

  Der Typ I dieser rein statisch gefundenen Methode ist die Signatur der Methode f an dieser Aufrufstelle im Programm.
- Der dynamische Typ D des Objekts, zu dem sich e_0 auswertet, gehört zu einer Unterklasse von T_0.
- Die Methode f wird nun aufgerufen, deren Typ I ist und die in der nächsten Oberklasse von D implementiert wird.

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;
   }
...
```


- `int hashCode()` liefert eine eindeutige 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.

Betrachte einen Aufruf $e_0.f(e_1, \ldots, e_k)$.

Ermittlung der aufgerufenen Methode:

- Bestimme die statischen Typen $T_0, \ldots, T_k$ der Ausdrücke $e_0, \ldots, e_k$.
- Suche in einer Oberklasse von T_0 nach einer Methode mit Namen f, deren Liste von Argumenttypen bestmöglich zu der Liste $T_1, \ldots, T_k$ passt.

  Der Typ I dieser rein statisch gefundenen Methode ist die Signatur der Methode f an dieser Aufrufstelle im Programm.
- Der dynamische Typ D des Objekts, zu dem sich e_0 auswertet, gehört zu einer Unterklasse von T_0.
- Die Methode f wird nun aufgerufen, deren Typ I ist und die in der nächsten Oberklasse von D implementiert wird.