Programmierzertifikat Objekt-Orientierte Programmierung mit Java

Vorlesung 05: Vergleichen von Objekten

Peter Thiemann

Universität Freiburg, Germany

SS 2009

Vergleichen von Objekten

Statischer Typ vs dynamischer Typ

- Der *statische Typ* (kurz: Typ) eines Ausdrucks ist der Typ, den Java für den Ausdruck aus dem Programmtext ausrechnet.
- Der *dynamische Typ* (*Laufzeittyp*) ist eine Eigenschaft eines Objekts. Es ist der Klassenname, mit dem das Objekt erzeugt worden ist.

Statischer Typ vs dynamischer Typ

- Der *statische Typ* (kurz: Typ) eines Ausdrucks ist der Typ, den Java für den Ausdruck aus dem Programmtext ausrechnet.
- Der *dynamische Typ* (*Laufzeittyp*) ist eine Eigenschaft eines Objekts. Es ist der Klassenname, mit dem das Objekt erzeugt worden ist.

Beispiele

- Angenommen A **extends** B (Klassentypen).

```
A a = new A (); // rhs: Typ A, dynamischer Typ A
B b = new B (); // rhs: Typ B, dynamischer Typ B
B x = new A (); // rhs: Typ A, dynamischer Typ A
// für x gilt: Typ B, dynamischer Typ A
```

- Bei einem Interfacetyp ist der dynamische Typ **immer** ein Subtyp.
- Im Rumpf einer Methode definiert in der Klasse C hat `this` den statischen Typ C. Der dynamische Typ kann ein Subtyp von C sein, falls die Methode vererbt worden ist.

Regeln für die Bestimmung des statischen Typs

- Falls Variable (Feld, Parameter) `x` durch ttt x deklariert ist, so ist der Typ von `x` genau ttt.
- Der Ausdruck `new C(...)` hat den Typ C.
- Wenn e ein Ausdruck vom Typ C ist und C eine Klasse mit Feld f vom Typ ttt ist, dann hat $e.f$ den Typ ttt.
- Wenn e ein Ausdruck vom Typ C ist und C eine Klasse oder Interface mit Methode m vom Rückgabetyp ttt ist, dann hat $e.m(\dots)$ den Typ ttt.
- Beim Aufruf eines Konstruktors oder einer Funktion müssen die Typen der Argumente jeweils Subtypen der Parametertypen sein.
- Bei einer Zuweisung muss der Typ des Audrucks auf der rechten Seiten ein Subtyp des Typs der Variable (Feld) sein.

Vergleichen von Objekten

Beispiel: Daten

```
class DateComparison {
  Date d1 = new Date(27,3,1941);
  Date d2 = new Date(8,5,1945);
  Date d3 = new Date(8,5,1945);

  boolean testD1D2 = d1 == d2; // Operator == auf Objekten
  boolean testD2D3 = d2 == d3;
  boolean testD3D3 = d3 == d3;
}
```

Vergleichen von Objekten

Beispiel: Daten

```
class DateComparison {
  Date d1 = new Date(27,3,1941);
  Date d2 = new Date(8,5,1945);
  Date d3 = new Date(8,5,1945);

  boolean testD1D2 = d1 == d2; // Operator == auf Objekten
  boolean testD2D3 = d2 == d3;
  boolean testD3D3 = d3 == d3;
}
```

Ergebnis

```
DateComparison(
   d1 = Date(...), d2 = Date(...), d3 = Date(...),
   testD1D2 = false,
   testD2D3 = false,
   testD3D3 = true)
```

Verschiedene Gleichheitsoperationen

- Der Gleichheitsoperator == ist auch auf Objekte anwendbar, aber liefert nicht das erwartete(?) Ergebnis!
- Er testet, ob beide Argument *dasselbe* Objekt bezeichnen.
- Oft ist **komponentenweise** Gleichheit gewünscht (*extensionale Gleichheit*).
- Muss als `equals()` Methode programmiert werden.
- `equals()` in Object vordefiniert; *muss* überschrieben werden!

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

- NB: Wenn `equals()` überschrieben wird, dann muss auch `hashCode()` überschrieben werden!
- Invariante: Wenn x.equals(y), dann gilt x.hashCode() == y.hashCode().

Gleichheit für einfache Klassen

- Einfache Klassen enthalten Felder von primitivem Typ.
- Ihre Werte können mit == verglichen werden (bzw. mit `equals()` für String).
- Beispiel: Vergleichsmethode `boolean same(Date that)` für die Date-Klasse.

```
// is this date the same as that date?
boolean same (Date that) {
  return (this.day == that.day) &&
         (this.month == that.month) &&
         (this.year == that.year);
}
```

Gleichheit für Mengen

Betrachte eine Klasse Set2, bei der jedes Objekt eine Menge von int mit *genau zwei Elementen* repräsentiert.

```
// Mengen von genau zwei Zahlen
class Set2 {
    private int one;
    private int two;

    public Set2 (int one, int two) { ... }

    // Elementtest
    public boolean contains (int x) {
        return (x == this.one) || (x == this.two);
    }
}
```

Gleichheit für Mengen

Implementierung

- Komponentenweise Gleichheit nicht angemessen.
- Zwei Mengen sind gleich, wenn sie sich gegenseitig enthalten

$$\{16, 42\} = \{42, 16\}$$

- `same()` Methode in Set2:

```
// Gleichheitstest
public boolean same (Set2 that) {
    return (this.contains (that.one))
        && (this.contains (that.two))
        && (that.contains (this.one))
        && (that.contains (this.two));
}
```

Gleichheit und Vererbung

```
class SpecialDate extends Date {
    private int rating;

    SpecialDate (int day, int month, int year, int rating) {
        super (day, month, year);
        this.rating = rating;
    }
}
```

- Spezielle Daten könnten mit einer Bewertung versehen sein.
- Die `same()` Methode aus der Klasse Date ist anwendbar, allerdings liefert sie nicht die erwarteten Ergebnisse.

```
class DateTest {
    SpecialDate s1 = new SpecialDate (12,8,2001,4000);
    SpecialDate s2 = new SpecialDate (12,8,2001,5000);

    boolean testss1 = s1.same (s1); // ==> true !!!
    boolean testss2 = s1.same (s2); // ==> true ???
}
```

Gleichheit und Vererbung

same-Methode in der Subklasse

- Eine spezialisierte Version der `same()` Methoden in der Subklasse ist erforderlich

```
boolean same (SpecialDate that) {
    return super.same (that) && (this.rating == that.rating);
}
```

- Damit funktioniert das Beispiel zunächst

```
class DateTest { // mit same(SpecialDate) in Klasse SpecialDate
    SpecialDate s1 = new SpecialDate (12,8,2001,4000);
    SpecialDate s2 = new SpecialDate (12,8,2001,5000);

    boolean testss1 = s1.same (s1); // ==> true !!!
    boolean testss2 = s1.same (s2); // ==> false !!!
}
```

Gleichheit und Vererbung

Weitere Probleme

- Andere Beispiele funktionieren nicht wie erwartet.
- same-Gleichheit ist nicht transitiv!

```
class DateTest { // mit same(SpecialDate) in Klasse SpecialDate
    SpecialDate s1 = new SpecialDate (12,8,2001,4000);
    SpecialDate s2 = new SpecialDate (12,8,2001,5000);
    Date d2 = new Date (12,8,2001);

    boolean testsd = s1.same (d2); // ==> true ???
    boolean testds = d2.same (s2); // ==> true ???
    boolean testss = s1.same (s2); // ==> false !!!
}
```

Gleichheit und Vererbung

Weitere Probleme

- Andere Beispiele funktionieren nicht wie erwartet.
- same-Gleichheit ist nicht transitiv!

```
class DateTest { // mit same(SpecialDate) in Klasse SpecialDate
    SpecialDate s1 = new SpecialDate (12,8,2001,4000);
    SpecialDate s2 = new SpecialDate (12,8,2001,5000);
    Date d2 = new Date (12,8,2001);

    boolean testsd = s1.same (d2); // ==> true ???
    boolean testds = d2.same (s2); // ==> true ???
    boolean testss = s1.same (s2); // ==> false !!!
}
```

Überraschung

- In den ersten beiden Fällen wird die Methode same der Klasse Date aufgerufen!
- Ursache: *Überladung* von Methoden.

Überladung von Methoden

- Überladung: in einer Klasse gibt es mehrere Methoden mit gleichem Namen, die sich nur in Anzahl oder Typ der Parameter unterscheiden.
- Die Auswahl der tatsächlich aufgerufenen Methode erfolgt durch Java aufgrund des ermittelten Argumenttyps.

Überladung von Methoden

- Überladung: in einer Klasse gibt es mehrere Methoden mit gleichem Namen, die sich nur in Anzahl oder Typ der Parameter unterscheiden.
- Die Auswahl der tatsächlich aufgerufenen Methode erfolgt durch Java aufgrund des ermittelten Argumenttyps.

Beispiel

- In der Klasse SpecialDate gibt es **zwei** Methoden mit Namen `same`, **die sich nur im Parametertyp unterscheiden**:
  1. `boolean same (Date that)` (geerbt von Date)
  2. `boolean same (SpecialDate that)` (selbst definiert)
- In `testsd` wird #1, die geerbte Methode, aufgerufen, da `d2` den Typ Date hat.
- In `testds` wird auch #1 aufgerufen, da das Empfängerobjekt den Typ Date hat.

Transitive Gleichheit

- Zufriedenstellende Implementierung benötigt **zwei** Methoden!
- Schwierigkeit: Feststellen, ob das Argumentobjekt den gleichen *dynamischen Typ* wie das Empfängerobjekt hat.
- Die Methode `same (Date that)` muss in Date definiert sein und in allen Subklassen von Date überschrieben werden.
- Sie stellt lediglich fest, welchen *dynamischen Typ* das Empfängerobjekt zur Laufzeit hat.
- Dann testet sie mit dem **instanceof**-Operator, ob das Argumentobjekt zu einer Subklasse dieses dynamischen Typs gehört.
- Die Hilfsmethode `reallysame (Date that)` führt denselben Test in **umgekehrter** Richtung aus, wobei schon sichergestellt ist, dass das Argumentobjekt zu einer Superklasse des Empfängertyps gehört.
- Nun sind die dynamischen Typen gleich und die Felder können verglichen werden. Die Felder von `that` müssen zunächst durch einen *Typcast* sichtbar gemacht werden.

Transitive Gleichheit (Implementierung)

Basisfall

```
class Date {
  private int day;
  private int month;
  private int year;

  protected boolean reallysame (Date that) {
    return (this.day == that.day) &&
           (this.month == that.month) &&
           (this.year == that.year);
  }

  public boolean same (Date that) {
    return that.reallysame (this);
  }
}
```

Code für Subklassen

```
class SpecialDate extends Date {
  private int rating;

  SpecialDate (int day, int month, int year, int rating) {
    super (day, month, year);
    this.rating = rating;
  }
  // dynamic type of that is a supertype of type of this
  protected boolean reallysame (Date that) {
    return (that instanceof SpecialDate)
        && super.reallysame (that)
        && (this.rating == ((SpecialDate)that).rating);
  }

  public boolean same (Date that) {
    return (that instanceof SpecialDate)
        && that.reallysame (this);
  }
}
```

Der **instanceof**-Operator

- Der boolesche Ausdruck

  ausdruck **instanceof** *objekttyp*

  testet ob der dynamische Typ des Werts von *ausdruck* ein Subtyp von *objekttyp* ist.
- Angenommen A **extends** B (Klassentypen):

```
A a = new A();
B b = new B();
B c = new A(); // statischer Typ B, dynamischer Typ A

a instanceof A // ==> true
a instanceof B // ==> true
b instanceof A // ==> false
b instanceof B // ==> true
c instanceof A // ==> true (testet den dynamischen Typ)
c instanceof B // ==> true
```

Der Typcast-Operator

- Der Ausdruck (*Typcast*)

  (*objekttyp*) *ausdruck*

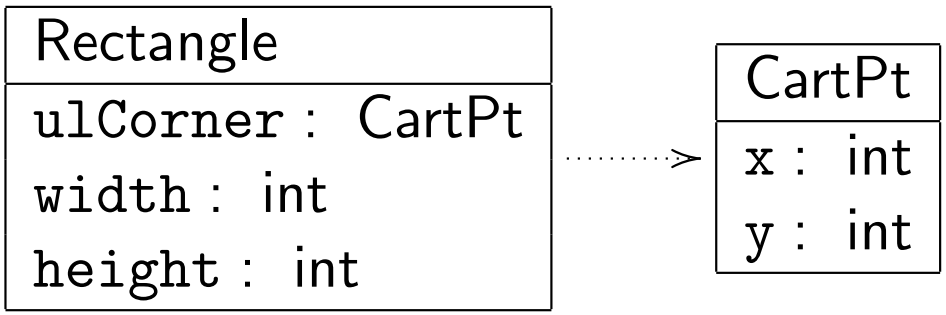
  hat den statischen Typ *objekttyp*, falls der statische Typ von *ausdruck* entweder ein Supertyp oder ein Subtyp von *objekttyp* ist.
- Zur Laufzeit testet der Typcast, ob der **dynamische Typ** des Werts von *ausdruck* ein Subtyp von *objekttyp* ist und bricht das Programm ab, falls das nicht zutrifft. (Vorher sicherstellen!)
- Angenommen A **extends** C und B **extends** C (Klassentypen), aber A und B stehen in keiner Beziehung zueinander:

```
A a = new A(); B b = new B(); C c = new C(); C d = new A();

(A)a // statisch ok, dynamisch ok
(B)a // Typfehler
(C)a // statisch ok, dynamisch ok
(B)d // statisch ok, dynamischer Fehler
(A)d // statisch ok, dynamisch ok
```

Gleichheit für zusammengesetzte Objekte

Beispiel




```
boolean same (Rectangle that) {
  return (this.x == that.x) && (this.y == that.y)
      && (this.ulCorner.same (that.ulCorner));
}
```

- Rufe die Gleichheit auf den untergeordneten Objekten auf.

Gleichheit für Vereinigungen von Klassen

- Definiere die `same()` Methode im Interface.
- Verwende die Vorgehensweise für Vererbung.
- Für abstrakte Klassen reicht es, die Methoden `same` und `reallysame` abstrakt zu belassen (da niemals Objekte existieren können, die diesen Klassentyp als Laufzeittyp besitzen).

Alternative Lösung

Ohne Verwendung von **instanceof** und Typcast

Am Beispiel von IShape:

- Voraussetzung: alle Varianten sind bekannt.
- Erweitere das Interface um Methoden, die die jeweilige Variante erkennen und ggf. ein IShape Objekt in ein Objekt vom spezifischen Typ umwandeln. Die Methoden liefern `null`, falls die Umwandlung nicht möglich ist.

```
interface IShape {
  Dot toDot();
  Square toSquare();
  Circle toCircle();

  boolean same (IShape that);
}
```

Alternative Lösung

Die abstrakte Klasse

```
abstract class AShape implements IShape {
  Dot toDot () { return null; }
  Square toSquare () { return null; }
  Circle toCircle () { return null; }

  abstract boolean same (IShape that);
}
```

Alternative Lösung

Implementierung für Dot

```
class Dot implements IShape {
  Dot toDot () { return this; }

  boolean same (IShape that) {
    Dot thatDot = that.toDot();
    return (thatDot != null)
        && (this.loc.same (thatDot.loc));
  }
}
```

Intensionale Gleichheit

- **Extensionale Gleichheit** testet ob zwei Objekte gleich sind und sich gleich verhalten.
- Diese Aufgabe hat in Java die `equals` Methode.
- **Intensionale Gleichheit** testet ob ihre Argumente dasselbe Objekt bezeichnen, in dem Sinn, dass eine Änderung am einen Argument immer die selbe Änderung am anderen Argument bewirkt.
- Diese Aufgabe hat in Java der == Operator. (Er testet die Gleichheit von Referenzen.)
- In Java vordefiniert:

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

- Jede Klasse ist automatisch Subklasse von Object und erbt diese (meist so nicht gewünschte) Implementierung.

Überschreiben von equals()

- Muss Parametertyp Object haben.
- Muss reflexiv, transitiv und symmetrisch sein, d.h.
  - Falls `x != null`, dann `x.equals(x)`.
  - Falls `x != null` und `y != null`, dann gilt `x.equals(y)` genau dann wenn `y.equals(x)`.
  - Falls `x != null`, `y != null` und `x.equals(y)` sowie `y.equals(z)` gelten, dann auch `y.equals(z)`.
- Die Java-Standardbibliothek verlässt sich auf diese Eigenschaften!
- Vgl. *Effective Java* von Joshua Bloch.

equals für Date und SpecialDate

```
// in class Date: is this date equal to that date?
public boolean equals (Object that0) {
  if ((!that0 instanceof Date) || (that0 == null)) return false;
  Date that = (Date)that0;
  return (this.getClass().equals (that.getClass())) &&
         (this.day == that.day) && (this.month == that.month) &&
         (this.year == that.year);
}
```

```
// in class SpecialDate: is this special date equal to that special date?
public boolean equals (Object that0) {
  if ((!that0 instanceof SpecialDate) || (that0 == null)) return false;
  SpecialDate that = (SpecialDate)that0;
  return (this.getClass().equals (that.getClass())) &&
         (this.day == that.day) && (this.month == that.month) &&
         (this.year == that.year) &&
         (this.rating == that.rating);
}
```

hashCode für Date und SpecialDate

Nicht optimal

```
// in class Date: hashCode consistent with the equals method
public int hashCode() {
  int hash = 1;
  hash = hash * 31 + this.day;
  hash = hash * 31 + this.month;
  hash = hash * 31 + this.year;
  return hash;
}
```

```
// in class SpecialDate: hashCode consistent with the equals method
public int hashCode() {
  int hash = super.hashCode();
  hash = hash * 31 + rating;
  return hash;
}
```

equals für Rectangle

```
// in class Rectangle: is this Rectangle equal to that Rectangle?
public boolean equals (Object that0) {
  if ((!that0 instanceof Rectangle) || (that0 == null)) return false;
  Rectangle that = (Rectangle)that0;
  return (this.getClass().equals (that.getClass())) &&
         (this.ulCorner.equals(that.ulCorner)) &&
         (this.width == that.width) &&
         (this.height == that.height);
}
```

- Verwende == für primitive Datentypen
- Verwende `equals` für Referenzdatentypen
- Implementierung für CartPt (einfaches Objekt) selbst

hashCode für Rectangle

```
// in class Rectangle: hashCode consistent with the equals method
public int hashCode() {
  int hash = this.ulCorner.hashCode();
  hash = hash * 31 + this.width;
  hash = hash * 31 + this.height;
  return hash;
}
```

- Implementierung für CartPt (einfaches Objekt) selbst

equals für Vereinigungen von Klassen

- Keine Besonderheiten