

Software-Architektur

— Idiome —

Prof. Dr. Oliver Braun

Letzte Änderung: 11.07.2017 15:12

- ▶ Programmiersprachenspezifische Muster
- ▶ z.B. in Joshua Bloch: *Effective Java*. Addison-Wesley, 2008.
- ▶ “Factory Methods” als Lesetext, Fragen?

Bloch: Item 8 equals

Redefinieren Sie `equals` **nicht**, wenn eine der folgenden Aussagen gilt:

- ▶ jede Instanz der Klasse eindeutig ist
- ▶ ein logischer Vergleich der Objekte ist unnötig, z.B.
`java.util.Random`
- ▶ in einer Oberklasse ist `equals` redefiniert und die Definition ist passend
- ▶ die Klasse ist `private` oder `package-private` und Sie sind sicher, dass `equals` nicht aufgerufen wird.

Sicherheitshalber könnten Sie im letzten Fall `equals` so redefinieren:

```
@Override public boolean equals(Object o) {  
    // Method is never called  
    throw new AssertionError();  
}
```

Wann equals redefinieren?

- ▶ wenn logische Gleichheit von Objektgleichheit abweicht
- ▶ **und** die Oberklasse(n) noch keine sinnvolle Redefinition bieten
- ▶ immer der Fall bei *value classes*

Wie equals redefinieren?

equals ist eine Äquivalenzrelation:

- ▶ *reflexiv*
- ▶ *symmetrisch*
- ▶ *transitiv*
- ▶ *widerspruchsfrei*, mehrfacher Aufruf führt zum selben Ergebnis
- ▶ Vergleich mit `null` immer `false`

Reflexivität und Symmetrie

- ▶ Reflexivität ist schwer aus Versehen nicht einzuhalten
- ▶ Symmetrie schon eher.
- ▶ Was ist hier das Problem?

```
public final class CaseInsensitiveString {
    private final String s;
    public CaseInsensitiveString(String s) {
        if (s == null)
            throw new NullPointerException();
        this.s = s;
    }
    @Override public boolean equals(Object o) {
        if (o instanceof CaseInsensitiveString)
            return s.equalsIgnoreCase(
                ((CaseInsensitiveString) o).s);
        if (o instanceof String)
            return
                s.equalsIgnoreCase((String) o);
    }
}
```

Interop-Problem

- ▶ gegeben sei

```
CaseInsensitiveString cis =  
    new CaseInsensitiveString("Polish");  
String s = "polish";
```

- ▶ dann gilt:

```
cis.equals(s) == true  
s.equals(cis) == false
```

- ▶ bei Code wie

```
List<CaseInsensitiveString> list =  
    new ArrayList<CaseInsensitiveString>();  
list.add(cis);
```

kann `list.contains(s)` z.B. in Abhängigkeit von der Implementierung `false`, `true` oder eine `Runtime-Exception` liefern

- ▶ Interoperabilität zwischen String und CaseInsensitiveString entfernen:

```
@Override public boolean equals(Object o) {  
    return o instanceof CaseInsensitiveString  
        && ((CaseInsensitiveString) o)  
            .s.equalsIgnoreCase(s);  
}
```


Transitivität: Beispiel Point

```
public class Point {
    private final int x;
    private final int y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    @Override public boolean equals(Object o) {
        if (!(o instanceof Point))
            return false;
        Point p = (Point)o;
        return p.x == x && p.y == y;
    }
    ...
}
```

Unterklasse ColorPoint

```
public class ColorPoint extends Point {  
    private final Color color;  
    public ColorPoint(int x, int y, Color color) {  
        super(x, y);  
        this.color = color;  
    }  
    ...  
}
```

Versuch 1: equals für ColorPoint

```
@Override public boolean equals(Object o) {  
    if (!(o instanceof ColorPoint))  
        return false;  
    return super.equals(o) &&  
        ((ColorPoint) o).color == color;  
}
```

Problem?

Versuch 1 verletzt die Symmetrieeigenschaft

```
Point p = new Point(1, 2);  
ColorPoint cp = new ColorPoint(1, 2, Color.RED);
```

Dann gilt:

```
p.equals(cp) == true  
cp.equals(p) == false
```

Versuch 2: equals für ColorPoint

Farbe ignorieren, wenn es kein ColorPoint ist.

```
@Override public boolean equals(Object o) {
    if (!(o instanceof Point))
        return false;
    // If o is a normal Point,
    // do a color-blind comparison
    if (!(o instanceof ColorPoint))
        return o.equals(this);
    // o is a ColorPoint; do a full comparison
    return super.equals(o)
        && ((ColorPoint)o).color == color;
}
```

Problem?

Versuch 2 verletzt die Transitivitätseigenschaft

```
ColorPoint p1 = new ColorPoint(1, 2, Color.RED);  
Point p2 = new Point(1, 2);  
ColorPoint p3 = new ColorPoint(1, 2, Color.BLUE);
```

Dann gilt:

```
p1.equals(p2) == true  
p2.equals(p3) == true
```

aber

```
p1.equals(p3) == false
```

Was ist die Lösung?

- ▶ Äquivalenzrelationen funktionieren nicht gut in OOPLs
- ▶ Bloch: **There is no way to extend an instantiable class and add a value component while preserving the equals contract**, unless you are willing to forgo the benefits of object-oriented abstraction.

Versuch 3: ein anderes equals für Point

```
@Override public boolean equals(Object o) {  
    if (o == null || o.getClass() != getClass())  
        return false;  
    Point p = (Point) o;  
    return p.x == x && p.y == y;  
}
```

Problem?

Versuch 3 verletzt das Liskovsche Substitutionsprinzip

- ▶ das Liskovsche Substitutionsprinzip (LSP) oder Ersetzbarkeitsprinzip besagt:

Let $\Phi(x)$ be a property provable about objects x of type T . Then $\Phi(y)$ should be true for objects y of type S where S is a subtype of T .

Konkretes Beispiel für Verletzung des LSP

- ▶ siehe `Code Point` und `CounterPoint`
- ▶ wenn `Point.onUnitCircle` mit einem `CounterPoint` aufgerufen wird, kommt immer `false` heraus.

Workaround

- ▶ keine zufriedenstellende Möglichkeit eine instantiierbare Klasse zu erweitern und ein Feld hinzuzufügen
- ▶ gemäß Bloch, Item 16: *Favor composition over inheritance.* gibt es einen Workaround
- ▶ siehe Code `ColorPoint`

Java Platform Libs

- ▶ Beispiel: `java.sql.Timestamp` erweitert `java.util.Date` um ein Feld `nanoseconds`
- ▶ `equals` in `Timestamp` ist nicht symmetrisch und erzeugt unerwartete Ergebnisse wenn `Date`- und `Timestamp`-Objekte z.B. in einer `Collection` genutzt werden
- ▶ die `Timestamp`-Klasse hat einen entsprechenden `Disclaimer`, aber die Probleme werden nicht verhindert
- ▶ dieses Verhalten der `Timestamp`-Klasse ist ein Fehler und sollte nicht wiederholt werden!

Konsistenz

- ▶ zwei Objekte die gleich sind, müssen so lange gleich bleiben, bis eines (oder beide) modifiziert werden
- ▶ schreiben Sie nie eine `equals`-Methode die auf unzuverlässige Ressourcen zurück greift
- ▶ Beispiel: die `equals`-Methode von `java.net.URL` überprüft ob zwei Hostnames in die gleiche IP aufgelöst werden können
- ▶ aus der Javadoc: *Note: The defined behavior for equals is known to be inconsistent with virtual hosting in HTTP.*
- ▶ das widerspricht der Konsistenz-Regel und führt zu praktischen Problemen
- ▶ auf Grund von Kompatibilitätsanforderungen wird dieses Verhalten aber nie geändert werden

“Non-nullity”

- ▶ alle Objekte müssen ungleich zu null sein
- ▶ Problem könnte sein, dass equals eine NullPointerException wirft
- ▶ oft gesehene Lösung:

```
@Override public boolean equals(Object o) {  
    if (o == null) return false;  
}
```

- ▶ das ist **unnötig**, da o instanceof MyType für o == null als Ergebnis sowieso false liefert:

```
@Override public boolean equals(Object o) {  
    if (!(o instanceof MyType)) return false;  
}
```

Das finale Rezept für eine qualitativ hochwertige equals-Implementierung

1. Mit `==` überprüfen ob das Argument eine Referenz zu diesem Objekt ist.
 - ▶ ist nur eine sehr sinnvolle Performanzmaßnahme
2. Mit `instanceof` überprüfen ob das Argument den korrekten Typ hat.
 - ▶ üblicherweise der Typ der Klasse, manchmal ein Interface das implementiert wird
3. Argument casten.
4. ...
5. ...

4. Vergleichen Sie alle “signifikanten” Felder

- ▶ für primitive Felder, außer float und double nutzen Sie ==
- ▶ für Objektreferenzen nutzen Sie equals rekursiv
- ▶ für float nutzen Sie Float.compare, für double Double.compare
- ▶ für Arrays die einzelnen Werte entsprechend vergleichen oder, wenn alle Elemente signifikant sind, eine Array.equals-Methode nutzen
- ▶ wenn ein Feld null sein kann, muss eine NullPointerException vermieden werden:

```
(field == null ? o.field == null  
           : field.equals(o.field))
```

wenn die beiden Felder oft gleich sind, kann folgendes effizienter sein:

```
(field == o.field  
 || (field != null && field.equals(o.field)))
```

- ▶ bei Klassen wie CaseInsensitiveString kann der

5. Symmetrisch? Transitiv? Konsistent?

- ▶ denken Sie darüber nach
- ▶ **und schreiben Sie Unittests**

Konkretes Beispiel

PhoneNumber.equals aus Item 9

```
public final class PhoneNumber {
    private final short areaCode;
    private final short prefix;
    private final short lineNumber;

    @Override public boolean equals(Object o) {
        if (o == this)
            return true;
        if (!(o instanceof PhoneNumber))
            return false;
        PhoneNumber pn = (PhoneNumber)o;
        return pn.lineNumber == lineNumber
            && pn.prefix == prefix
            && pn.areaCode == areaCode;
    }
}
```

Abschließende Anmerkungen zu equals

- ▶ immer wenn Sie equals redefinieren, redefinieren Sie auch hashCode
- ▶ versuchen Sie nicht so clever wie möglich zu sein
 - ▶ es ist eine gute Idee, dass in der File-Klasse verschiedene symbolische Links auf die gleiche Datei **verschieden** sind
- ▶ Redefinieren Sie equals und überladen Sie es nicht! Use @Override

Bloch: Item 9: Always override hashCode when you override equals

der hashCode-Kontrakt

- ▶ während der Ausführung einer Applikation muss hashCode immer den selben int liefern, solange sich nichts was in equals genutzt wird geändert hat
- ▶ wenn zwei Objekte gleich bzgl. equals sind, müssen sie bei hashCode das gleiche Ergebnis liefern
- ▶ wenn zwei Objekte nicht gleich bzgl. equals sind, muss hashCode keine verschiedenen Ergebnisse liefern. Es ist aber sinnvoll bzgl. der Performanz von HashMap, HashSet, ...

Beispiel für ein Problem

Code von PhoneNumber und

```
Map<PhoneNumber, String> m
    = new HashMap<PhoneNumber, String>();
m.put(new PhoneNumber(707, 867, 5309), "Jenny");
```

Was ergibt dann

```
m.get(new PhoneNumber(707, 867, 5309))
```

null

weil im falschen Hash-Bucket gesucht wird

Lösung?

```
@Override public int hashCode() { return 42; }
```

NEIN!

Die `hashCode`-Methode erfüllt zwar den Kontrakt, aber:
Damit landen **alle** Objekte im selben Hash-Bucket und
Hashtabellen degenerieren zu verketteten Listen!

Einfaches Rezept für eine gute Näherungslösung einer perfekten hashCode-Methode

1. Initialisieren Sie `int result` mit einem konstanten Wert ungleich 0, z.B. 17.
2. Für jedes signifikante Feld (wie bei `equals`) berechne einen `int`-Wert `c` und füge ihn folgendermaßen zu `result` hinzu:

```
result = 31 * result + c
```

- 2.1 für einen `boolean` berechne `(f ? 1 : 0)`
- 2.2 für ein `byte`, `char`, `short` oder `int` berechne `(int) f`
- 2.3 für einen `long` berechne `(int) (f^(f>>>32))`
- 2.4 für einen `float`, berechne `Float.floatToIntBits(f)`
- 2.5 für einen `double`, berechne `Double.doubleToLongBits(f)` und wandele den `long` wie oben beschrieben in einen `int` um
- 2.6 für eine Objektreferenz mit einer `equals`-Methode die für ihre Felder `equals` rekursiv aufruft, rufe `hashCode` rekursiv auf
Ansonsten berechne eine kanonische Repräsentation und rufe darauf `hashCode` auf

Für `null` gebe den Wert 0 zurück

- 2.7 für ein `Array` berechne jeden notwendigen Wert und füge ihn

Item 12: Consider implementing Comparable

- ▶ jede Java-Klasse die das Interface Comparable implementiert, kann auf eine Vielzahl praktischer generischer Algorithmen und Collection-Implementierungen zurück greifen

```
public interface Comparable<T> {  
    int compareTo(T o);  
}
```

Scala Traits

- ▶ Scala hat Traits
- ▶ Traits können beliebige abstrakte und konkrete Member enthalten
- ▶ im Gegensatz zu Interfaces können mehrere Traits “hineingemixt” werden
- ▶ analog zum Java-Interface Comparable gibt es den Scala-Trait Ordered

```
trait Ordered[A] extends Comparable[A] {  
  abstract def compare(that: A): Int  
  def <(that: A): Boolean =  
    (this compare that) < 0  
  def <=(that: A): Boolean =  
    (this compare that) <= 0  
  def >(that: A): Boolean =  
    (this compare that) > 0  
  def >=(that: A): Boolean =  
    (this compare that) >= 0
```

Beispiel

```
case class Person(name: String, age: Int)
  extends Ordered[Person] {
  def compare(other: Person): Int =
    age - other.age
}
```

```
scala> Person("Hans", 5) < Person("Helge", 7)
res0: Boolean = true
```

Noch ein paar andere Items kurz angerissen

Item 42: Use varargs judiciously

- ▶ Varargs nur dann verwenden, wenn es tatsächlich auch 0 Argumente sein können
- ▶ nicht grundsätzlich statt Array Parameter Varargs nehmen
- ▶ insbesondere sowas:

```
ReturnType1 suspect1(Object... args) { ... }  
<T> ReturnType2 suspect2(T... args) { ... }
```

akzeptiert **jede beliebige** Parameterliste und macht Compiletime Errors so zu Runtime Errors!

- ▶ gerade bei performanzkritischen Berechnungen ist Varargs schlecht (Array-Allocation und -initialisierung)
- ▶ wenn z.B. klar ist, dass 95% der Aufrufe von foo drei oder weniger Argumente haben, ist es sinnvoll so zu definieren:

```
public void foo() { }  
public void foo(int a1) { }  
public void foo(int a1, int a2) { }  
public void foo(int a1, int a2, int a3) { }
```

Item 57: Use exceptions only for exceptional conditions

- ▶ nutzen Sie Exceptions nie um normalen Kontrollfluß zu implementieren
- ▶ z.B. `IndexOutOfBoundsException` für den Abbruch einer Iteration
- ▶ oder mit `next()` iterieren bis eine `NoSuchElementException` fliegt

Item 69: Prefer concurrency utilities to wait and notify

- ▶ `wait` and `notify` korrekt zu nutzen ist schwierig und fehleranfällig
- ▶ es gibt z.B. das Executor Framework, Concurrent Collections, Synchronizers
- ▶ in Scala und in Zukunft noch viel mehr
- ▶ nutzen Sie diese Möglichkeiten!

Seit Java 8: Geben Sie kein `null` mehr zurück!

- ▶ seit Java 8 gibt es `Optional` in Java
- ▶ wenn etwas optional ist, wurde bisher `null` verwendet um Anzuzeigen, dass es gerade nicht da ist
- ▶ das hat einige Nachteile
 - ▶ ich muss Doku lesen um zu wissen, dass so etwas passieren kann
 - ▶ wenn ich auf den Null-Pointer nicht reagiere, fliegt er mir um die Ohren
 - ▶ der Compiler hilft mir nicht
 - ▶ vorsichtige Leute prüfen **immer** auf `null` und bremsen so die Performanz
- ▶ besser: `Optional<T>` verwenden, wo notwendig
 - ▶ ich sehe am Typ, dass so etwas passieren kann
 - ▶ ich muss darauf reagieren, den sonst bekomme ich den Wert nicht heraus
 - ▶ der Compiler sieht das am Typ und kann mir helfen
 - ▶ nur wenn der Typ es erfordert, muss ich den Aufwand betreiben

▶ mehr Infos zu Javas `Optional`: <http://www.oracle.com/>

Scala Option

- ▶ Scalas Option entspricht Javas Optional, ist aber leichter zu nutzen/zu verstehen
- ▶ Beispiel:

```
case class Person(name: String,  
    private var age: Option[Int]) {  
    def printAge(): Unit =  
        println(age getOrElse "unkown")  
    def birthday(): Unit = age = age map {_+1}  
}  
val helga = Person("Helga", None)  
val helge = Person("Helge", Some(34))
```

- ▶ Anmerkung: Natürlich kann so keine (totale) Ordnung mehr definiert werden