Wie man zwei Objekte vergleicht

Objektorientierung mit Java

Dominikus Herzberg, THM

Version 1.0, 14.4.2021: PiS/VW2

Lerninhalte

Wie man `equals()` implementiert

- Unser Selbstkonzept: Ich als Identität, Zwillinge als gleich aber nicht identisch
- Wann sind Dinge gleich? Ist "A == A"? Gleichheit als der Wille, nicht genauer hinzusehen.

Was Identität und Gleichheit unterscheidet: `==` `!=` `equals`

Gleichheitsoperator vergleicht Inhalte der Speicherzellen

```
// == ist Gleichheit bei elementaren Datentypen
// == ist Gleichheit der Referenzen, d.h. identische Objekte (=> Identität)
// d.h. Identität ist ein Merkmal von Referenzen: Zeige ich auf ein und dasselbe?
assert 7 == 7;
assert !(7 == 3);

class A {}
A a1 = new A();
A a2 = new A();
assert !(a1 == a2);
assert !a1.equals(a2);
```

Warum in einem Programm ohne (Un)Gleichheit nichts passiert

- Die bedingte Auswertung von Ausdrücken (`?:`)
- Die bedingte Ausführung von Anweisungen (`if`, `switch/case`, `for`, `while`, `do/while`)

Die 5 Bausteine einer `equals()`-Implementierung

Wichtig: Die `equals()`-Methode darf niemals eine Ausnahme auslösen.

Beispiel

```
class Point {

    private double x, y;

    Point(double x, double y) {
        this.x = x;
        this.y = y;
    }

    @Override
    public boolean equals(Object other) {                ①
        if (other == null) return false;                 // Null abwehren!
        if (other == this) return true;                  // Bin ich's selbst?
        if (other.getClass() != getClass()) return false; // Andere Klasse?
        Point that = (Point)other;                       // Casting
        return this.x == that.x && this.y == that.y;     // Was definiert Gleichheit?
    }
}
```

① Vorbild: Hier wird kein `if/else` eingesetzt!

> Merkhilfe: Null nein — Ich ja — Klasse gleich? — Casting — Vergleich

Demonstration

```
Point p1 = new Point(2,0);
Point p2 = new Point(0,2);
Point p3 = new Point(2,0);
assert !p1.equals(null);
assert p1 == p1 && p1.equals(p1);
assert p1 != p2 && !p1.equals(p2) && !p2.equals(p1);
assert p1 != p3 && p1.equals(p3) && p3.equals(p1);
assert p2 != p3 && !p2.equals(p3) && !p3.equals(p2);
```

Anforderungen an Gleichheit

- *reflexiv*: `x.equals(x) ⇒ true` ("Ich bin ich")
- *symmetrisch*: `x.equals(y) == y.equals(x)`
- *transitiv*: Wenn `x.equals(y) ⇒ true` und `y.equals(z) ⇒ true`, dann `x.equals(z) ⇒ true`
- *konsistent*: wiederholte Aufrufe von `x.equals(y)` führen zum gleichen Resultat
- *nullfalschig*: `x.equals(null) == false`

Merkhilfe: *reflexiv* bezieht sich auf ein Objekt, *symmetrisch* auf zwei, *transitiv* auf drei, *konsistent* auf alles, *nullfalschig* auf `null` als Argument.

Was eine Tiefengleichheit bedeutet? Beispiel: `Objects.deepEquals(…)`

- Begriff meint den "Tiefenvergleich" bei verschachtelten Arrays
- `deepEquals(Object a, Object b)` entspricht einem `a.equals(b)`, bei Arrays einem `Arrays.deepEquals(a, b)`
- `deepEquals(null, null) ⇒ true`

Wie man `hashCode()` implementiert

Meist geht die Implementierung zur Berechnung des HashCodes Hand in Hand mit der Implementierung der Gleichheit.

HashCode: Wenn Du das, was Dich im Vergleich ausmacht, als Zahl ausdrückst

- Jedes Objekt hat eine eindeutige Nummer (Beispiel: Personalausweis)
- Diese Nummer entspricht der Referenz des Objekts; ist in Java *by design* nicht zugänglich
- Der HashCode ist eine Zahl, die das, was ein Objekt in einer `equals`-Implementierung an Eigenschaften einbringt, in eine `int`-Zahl umsetzt
- Der HashCode könnte man als Pseudo-Identität bezeichnen; man muss damit rechnen, dass mehrere Objekte gleichen HashCode haben.

Standard-Ausgabe eines Objekts gibt den HashCode an

```
jshell> p1
p1 ==> Point@45ff54e6

jshell> Integer.toHexString(p1.hashCode())
$47 ==> "45ff54e6"
```

Wozu ein HashCode nützlich ist

- Wozu braucht's den HashCode? Für den Einsatz als Schlüssel in einer `Map`
- `Map` ist das Interface zu einer Datenstruktur, die Schlüssel/Wert-Paare verwaltet
- Effizienter Zugriff mit einem HashCode
- Siehe beispielsweise `HashMap` in Java-Dokumentation

Einfaches Map-Beispiel

```
assert Map.of("Jan", 31, "Feb", 29, "Mrz", 31, "Apr", 30).get("Mrz") == 31;
```

Ein schlecht gewählter HashCode (Stichwort: Kollision) macht Map nicht unbrauchbar, nur ineffizient.

Alle Objekte haben den gleichen HashCode: nicht sinnvoll, aber läuft

```
class H {
    @Override
    public int hashCode() { return 1; }
}

// a hash conflict doesn't render maps useless
H h1 = new H(); // h1 ==> H@1
H h2 = new H(); // h2 ==> H@1
assert Map.of(h1, "a", h2, "b").get(h1).equals("a");
assert Map.of(h1, "a", h2, "b").get(h2).equals("b");
```

Ein Helfer bei der `hashCode()`-Implementierung: `Objects.hash(···)`

Wozu selber ein Berechnungsverfahren für einen HashCode ausdenken?

Die einfachste Lösung: Verwende `Objects.hash(···)`

```
class Point {
    private double x, y;

    Point(double x, double y) {
        this.x = x;
        this.y = y;
    }

    @Override public int hashCode() {
        return Objects.hash(x, y);
    }
}
```

Beachte: Getauschte Argumente, unterschiedliche HashCodes

```
jshell> new Point(2,3).hashCode()
$135 ==> 525249

jshell> new Point(3,2).hashCode()
$136 ==> 16253889
```

Objekte, die gleich sind, sollen den gleichen HashCode haben

Wie man den Vergleich mit `compareTo()` und `compare()` umsetzt

Von den Grundtypen abgeschaut: `<`, `>`, `>=`, etc.

- Die Operatoren für die Größenvergleiche bei den elementaren Zahlentypen basieren auf der natürlichen Ordnung der Zahlen
- Das Interface, das man entsprechend bei Objekten verwendet heißt `Comparable<T>` mit der Methode `compareTo(T other)`.
- `a.compareTo(b) < 0` meint sozusagen `a < b`
- `a.compareTo(b) > 0` meint sozusagen `a > b`
- `a.compareTo(b) == 0` meint `a.equals(b)`

Interface `Comparable<T>` und die Methode `compareTo(T other)`

Ein Beispiel

```
class Student implements Comparable<Student> {
    String name;
    int id;

    Student(int id, String name) {
        assert Objects.nonNull(name) && !name.isBlank();
        this.id = id;
        this.name = name;
    }
    @Override public int compareTo(Student other) {
        assert this.id != other.id : "Everyone has a unique id"; ①
        return Integer.compare(this.id, other.id); ②
    }
    @Override public String toString() {
        return "Student[" + id + "](" + name + ")";
    }
}
```

① Die `id` (Matrikelnummer) der Studenten soll eindeutig sein; wird diese Bedingung verletzt, springt das `assert` an

② Naheliegend wäre `return this.id - other.id` gewesen; ist problematisch wegen Kreisarithmetik in Java. Beispiel: `-2_000_000_000 - 1_000_000_000 == 1_294_967_296`. Hier hilft die Methode `Integer.compare()`.

Ein typischer Anwendungsfall: Ein Array von Objekten, die sortiert werden

```
Student bob = new Student(10, "Bob");
Student alice = new Student(5, "Alice");
Student chris = new Student(6, "Chris");
Student[] students = { bob, alice, chris };
Arrays.sort(students);
assert students[0] == alice && students[1] == chris && students[2] == bob;
```

`Comparator`: Ein Helfer für Sortierungen jeglicher Art

- Man implementiert `Comparable`, um die natürliche Ordnung der Objekte abzubilden
- Man kann, wie bei `Student`, aber auch nach anderen Kriterien sortieren wollen
- Verwende `Arrays.sort`, das als zweites Argument eine `Comparator`-Instanz erwartet
- `Comparator` ist ein funktionales Interface mit `compare`-Methode; ein Lambda-Ausdruck kann zum Einsatz kommen

So kann man Studenten nach dem Namen sortieren

```
Arrays.sort(students, (s1, s2) -> s1.name.compareTo(s2.name))
assert students[0] == alice && students[1] == bob && students[2] == chris;
```