

# Kopieren und Vergleichen

Softwareentwicklung II (IB)

---

Prof. Dr. Oliver Braun

Letzte Änderung: 24.05.2018 13:24

- Erster Ansatz zum Duplizieren eines Objektes: Kopier-Konstruktor

```
class Rational {  
    Rational(Rational that) {  
        numer = that.numer;  
        denom = that.denom;  
    }  
    ...  
}
```

- Kopieren eines `Rational`-Objekts `original` mit Kopier-Konstruktor:

```
Rational original = ... ;  
Rational copy = new Rational(original);
```

## Kopie eines Objektes (2/4)

- Anschließend: `copy` ist ein anderes Objekt, aber mit dem gleichen Inhalt wie `original`:

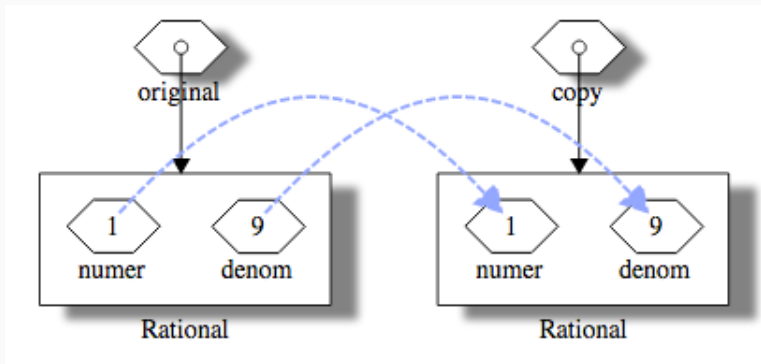
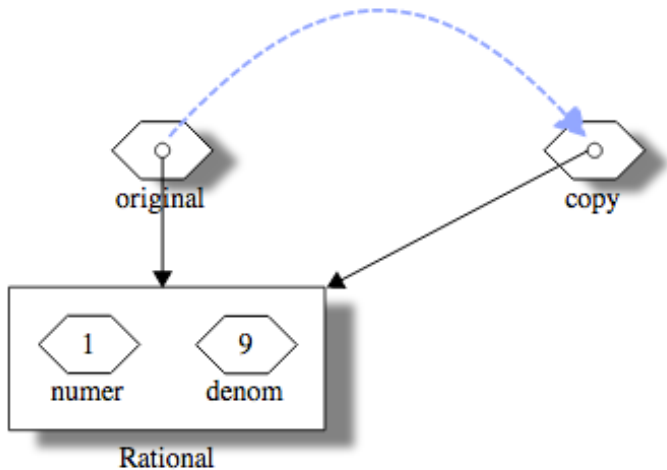


Abbildung: Copy

- Original und Kopie unabhängig: Änderungen eines Objektes ohne Einfluss auf das andere
- Zum Vergleich Wertzuweisung: `copy` referenziert das selbe Objekt wie `original`:

```
Rational original = ... ;  
Rational copy = original;
```

## Kopie eines Objektes (4/4)



- Problematisch: Klasse mit Objekten als Objektvariablen
- Beispiel: Intervalle mit Brüchen als Grenzen, zum Beispiel:

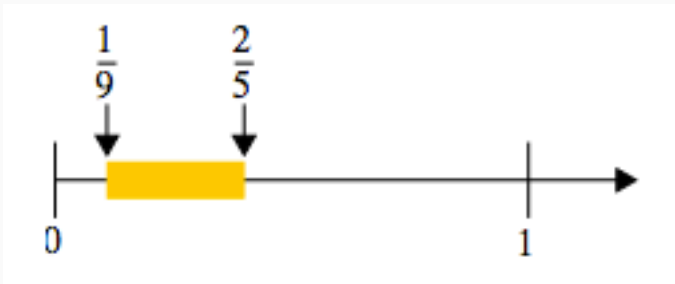


Abbildung: Intervall

- Klasse `RatRange` mit je einem `Rational`-Objekt als Unter- und Obergrenze des Intervalls:

```
class RatRange {  
    private Rational lower;  
    private Rational upper;  
  
    RatRange(Rational lower, Rational upper) ...  
}
```

Erzeugen eines Intervalles mit:

```
new RatRange(new Rational(1, 9), new Rational(2, 5))
```

- Simpler Kopier-Konstruktor mit Wertzuweisungen: dupliziert Referenzen, nicht Objekte (Aliasing)

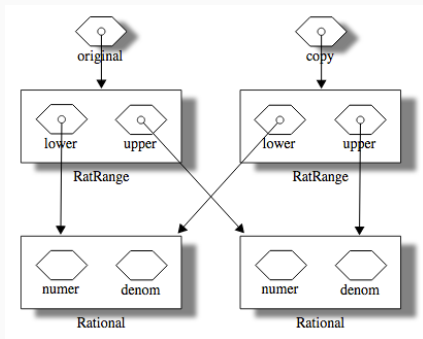
```
class RatRange {  
    RatRange(RatRange that) {  
        lower = that.lower;  
        upper = that.upper;  
    }  
    ...  
}
```

- Erzeugt eine **flache Kopie** (shallow copy): Original und Kopie referenzieren die selben Objektvariablen:



```
RatRange original = ... ;  
RatRange copy = new RatRange(original);
```

Skizze:



- Original und Kopie sind abhängig: Änderungen an einem Objekt werden im anderen sichtbar

- Objektvariablen mit Kopier-Konstruktor statt Wertzuweisung kopieren:

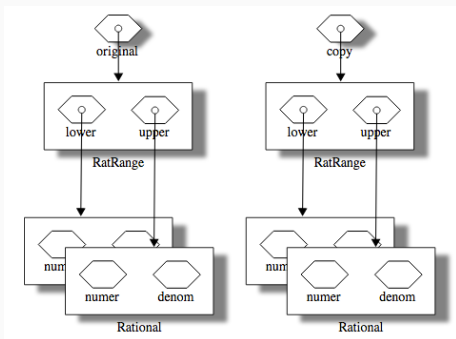
```
class RatRange {  
    RatRange(RatRange that) {  
        lower = new Rational(that.lower);  
        upper = new Rational(that.upper);  
    }  
    ...  
}
```

- Äußerlich kein Unterschied:

```
RatRange original = ... ;  
RatRange copy = new RatRange(original);
```

## Tiefe Kopie (2/2)

- Erzeugt eine **tiefe Kopie** (deep copy): Original und Kopie enthalten logisch gleiche, isolierte Objektvariablen:



- Änderungen eines Objektes ohne Einfluss auf das andere

- Regeln zum Kopieren von Objektvariablen im Kopier-Konstruktor abhängig vom Typ:

---

primitiver Typ	Wertzuweisung (Beispiel <b>Rational</b> )
Referenztyp	Kopier-Konstruktor
Unveränderliche Klassen	Wertzuweisung (wie primitive Objektvariablen)

---

- Aber: Kopier-Konstruktor zu schwach bei Vererbung (dort mächtigerer Mechanismus nötig)

- Methode **equals** zum Vergleich von Objekten
- `equals` erwartet anderes Objekt `x` als Parameter, liefert ...

---

<code>true</code>	wenn dieses Objekt und <code>x</code> logisch gleichwertig sind;
<code>false</code>	wenn dieses Objekt und <code>x</code> logisch verschieden sind.

---

## Vergleich von Objekten (2/4)

- Was heißt „logisch gleich“?
  - Benutzer kann Objekte nicht unterscheiden
  - ... so lange keines von beiden verändert wird
- Begriffe

---

Objekte sind ...	wenn ...	Test mit ...
<b>identisch</b>	sie ein und das selbe Objekt sind	<b>=</b>
<b>gleich</b>	sie nicht unterscheidbar sind	<b>equals</b>

---

## Vergleich von Objekten (3/4)

- Gleichheit ist schwächer als Identität:
  - identische Objekte sind immer gleich,
  - gleiche Objekte müssen nicht identisch sein
- Beispiel:

```
Rational x = ...;  
Rational y = ...;  
if (x == y)  
    System.out.println("identisch");  
if (x.equals(y))  
    System.out.println("gleich");
```



- Anwender interessiert sind i.d.R. für Gleichheit, kaum für Identität  $\Rightarrow$  Objektvergleich mit  $=$  selten sinnvoll

- Schema zur Implementierung von `equals`:

```
@Override  
public boolean equals(Object x) {  
    ...  
}
```

- Vergleich in mehreren Schritten:

1. Existiert x überhaupt?

```
if (x == null)  
    return false;
```

Ein existierendes Objekt kann niemals gleich mit einem nicht existierenden sein

## Implementierungsschema `equals` (2/3)

2. Hat `x` den gleichen Typ? (Begründung siehe später)

```
if (getClass()  $\neq$  x.getClass())  
    return false;
```

Falls nein: Wenn `x` etwas anderes als ein `Rational` ist, kann es nicht gleichwertig sein.

Falls ja: Typecast auf eigene Klasse, in diesem Beispiel `Rational`:

```
Rational that = (Rational) x;
```

3. Sind die Objektvariablen paarweise gleich?

```
if (getNumer()  $\neq$  that.getNumer())  
    return false;  
if (getDenom()  $\neq$  that.getDenom())  
    return false;
```

Wenn nicht, ist `that` ein `Rational`-Objekt mit anderem Inhalt.

4. Wenn alle vorhergehenden Tests bestanden wurden, sind die Objekte gleich:

```
return true;
```

- Gleichheit muss symmetrisch sein:

`x.equals(y)`

`y.equals(x)`

müssen beide gelten oder beide nicht, also:

`x.equals(y) = y.equals(x)`

## Vergleich von geschachtelten Objekten (1/3)

- Primitive Objektvariablen: Vergleich mit `=` reicht aus
- Objektvariablen von Referenztypen: Operator `=` ungeeignet
- Statt dessen: Objektvariablen paarweise mit deren `equals`-Methode vergleichen
- Beispiel `RatRange`:

## Vergleich von geschachtelten Objekten (2/3)

```
class RatRange {  
    @Override  
    public boolean equals(Object that) {  
        ...  
        if (!getLower().equals(that.getLower()))  
            return false;  
        if (!getUpper().equals(that.getUpper()))  
            return false;  
        return true;  
    }  
    ...  
}
```

Name „equals“ steht für zwei verschiedene Methoden:

1. Kopf: Definition der Methode `equals` der Klasse `RatRange`
2. Rumpf: Aufruf der Methode `equals` der Klasse `Rational`

- eine weitere Standardmethode neben `equals` ist `hashCode`

```
public int hashCode()
```

- `hashCode` berechnet einen sog. Hashwert für ein Objekt
- sobald Objekte beispielsweise in einer `HashMap` oder einem `HashSet` gespeichert werden sollen, sollte für `hashCode` eine eigene Implementierung angegeben werden
- Joshua Bloch sagt in "Effective Java" sogar:

*Always override `hashCode` when you override `equals`*



## Sinnvolle Implementierung von `hashCode`

- beginne mit einem beliebigen Wert ungleich 0, z.B. 17
- berechne einen `int c` für jede Objektvariable `f` auf folgende Weise
  - `boolean`: berechne `(f ? 1 : 0)`
  - ganzzahlig, außer `long`: berechne `(int) f`
  - `long`: berechne `(int) (f ^ (f >>> 32))`
  - `float`: berechne `Float.floatToIntBits(f)`
  - `double`: berechne `Double.doubleToLongBits(f)` und wandele den `long` wie oben beschrieben in einen `int` um
  - Objekt: berechne `hashCode` von dem Objekt bzw. 0 für `null`

Addiere dann den neuen Wert zum bisherigen Hashwert `result` mit folgender Formel: `result = 31 * result + c`

- gib `result` zurück

## Beispiel für hashCode

```
class Person {  
    private String name;  
    private int age;  
  
    @Override  
    public int hashCode() {  
        int result = 17;  
        result = 31 * result + name.hashCode();  
        result = 31 * result + age;  
        return result;  
    }  
}
```