OOP 1 Zusammenfassung

Syntax und Semantik

Syntax: Grammatik
Semantik: Bedeutung der Sprachelemente (Was macht eine While-Schlaufe?)

Unäre Operatoren

x++ ⟺ Gib x zurück; x = x + 1.
++x ⟺ x = x + 1; Gib x zurück.

Strings

String Pooling

Eine reine Compiler-Optimisation. Gleiche Strings können als einziges Objekt alloziiert werden. Beispiel:

```
String a = "Hello";
String b = "Hello";
a == b // true

// aber:

String a = "Hello";
String b = "H";
b += "ello";

a == b // false
```

Textblocks (Multiline Strings)

```
String a = """
Multiline
String with "(unescaped) double quotes
inside".""";
```

Methoden

Overloading

Merke

f spezifischer als *g* ⟺ Alle möglichen Aufrufe von *f* passen auch für *g* (aber nicht umgekehrt).

Bei Overlaoding gibt es **keine** Priorisierung von links nach rechts (oder umgekehrt):

```
print(int a, double b) {}
print(double a, int b) {}

print(1, 1) // ambiguous method call
```

Dynamische vs Statische Bindung

Alle nicht privaten Methoden verwenden Dynamic Dispatch.
Static Dispatch wird verwendet bei:

- Konstruktoren
- Privaten Methoden
- Statischen Methoden

Covarianz

Der Rückgabe-Typ einer überschriebenen Methode kann Subtyp sein:

```
class Vehicle {
  Vehicle getClone() {}
}
class Car extends Vehicle {
  @Override
  Car getClone() {}
}
```

Wichtige Spezialfälle der Gleichheit

```
double a = Double.POSITIVE_INFINITY;
a + 1 == a + 2; // true

double a = Double.NaN;
a != a; // true

null == null; // true
```

Hiding

```
class Vehicle {
  String description = "Any vehicle";
}
class Car extends Vehicle {
  String description = "This is a car";
}
```

Statische Bindung:

- Zugriff auf das Feld der eigenen Klasse mit `description` oder `this.description`
- Zugriff auf das Feld der Basisklasse mit `super.description` oder `((Vehicle)this).description`
- Zugriff auf das Feld irgendeiner Klasse in der Vererbungshierarchie mit `((SuperSuperClass)this).description` (Es existiert kein `super.super`).

Final

Finale Methode

```
class Vehicle {
  public final void stop() {}
}
class Car extends Vehicle {
  public void stop() {} // Compiler-Error:
Cannot override the final method...
}
```

Finale Klasse

```
final class Vehicle {}
class Car extends Vehicle {} // Compiler-
Error: Cannot inherit from final class...
```

Equals-Overriding

Warnung

Bei equals stets `getClass() != obj.getClass()` verwenden, anstelle `instanceof`, da `instanceof` die Vererbungshierarchie berücksichtigt.

Regeln

- Reflexivität:
  - `x.equals(x)` ⟶ `true`
- Symmetrie:
  - `x.equals(y) == y.equals(x)`
- Transitivität:
  - `x.equals(y) && y.equals(z)` ⟶ `x.equals(z)`
- Konsistenz:
  - Determinismus: Immer dasselbe Resultat für dieselben Argumente.
- Null
  - `x.equals(null)` ⟶ `false`

Hash-Code

```
@Override
public int hashCode() {
  return Objects.hash(firstName, lastName,
age);
}
```

Collections

Methode	Effizienz
`get()`, `set()`	Sehr schnell
`add()`	Sehr schnell (amortisiert)
`remove(int)`	Langsam (meist umkopieren)
`contains()`	Langsam (durchsuchen)

Amortisierung von `add()`

Jedes neue Array, welches erstellt wird, wird um 1.5 grösser, muss aber nicht bei jedem `add()` vergrössert werden.

Amortisierte Kostenanalyse: Einfügen im Worst Case langsam, im Durchschnitt aber sehr schnell.

Max. Anzahl Umkopieren bei n+1 Einfügen:

$$n + n\left(\frac{2}{3}\right) + n\left(\frac{2}{3}\right)^2 + n\left(\frac{2}{3}\right)^3 + \ldots = 3n$$

Die amortisierte Kostenanalyse beträgt somit <= 3 pro Einfügen.