OOP 1 Zusammenfassung

Unäre Operatoren

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

Datentypen

Double

Mantisse: 52 Bit Exponent: 11 Bit Vorzeichen: 1 Bit

Float

Mantisse: 23 Bit Exponent: 8 Bit Vorzeichen 1 Bit

Ordnung von Primitives

1. long, double (64 Bit)
2. int, float (32 Bit)
3. short (16 bit)
4. byte (8 bit)

Iterator

```
Iterator<String> iter = stringList.iterator();
while(it.hasNext()) {
  String elem = it.next();

  it.remove();
}
```

Wrapper-Klassen

```
Integer boxed = Integer.valueOf(5);
int unboxed = boxed.intValue();
```

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.""";
```

Enums

```
public enum Weekday {
  MONDAY(true), TUESDAY(true),
WEDNESDAY(true), THURSDAY(true), FRIDAY(true),
SATURDAY(false), SUNDAY(false);

  private boolean isWeekday;

  public Weekday(boolean isWeekday) {
    this.isWeekday = isWeekday;
  }
}
```

Der ==-Operator funktioniert für Enums by default.

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`).

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.

Exceptions

Bei einem Rethrow in einer try-Methode, werden alle nachfolgenden catches nicht behandelt.

`` `Exception(String message, Throwable cause) `` (cause kann Exception sein, damit kann man den Stack Trace selbst aufbauen)

Finally

Wird immer ausgeführt, auch wenn Exception nicht geprüft wurde oder nach einem Rethrow. Wird auch ausgeführt nach einem early return.

Die zweite Exception im Finally-Block überschreibt erste Exception im catch Block.

```
try {
  // ...
} catch(RuntimeException ex) {
  throw ex; // wird ignoriert
} finally {
  throw new Exception();
}
```

Comparable-Interface

```
class Person implements Comparable<Person> {
  private int age;
  public int compareTo(Person other) {
    return Integer.compare(age, other.age);
  }
}
```

Comparator-Interface

```
interface Comparator<T> {
  int compare(T a, T b);
}
```

```
Collections.sort(people, new MyComparator());
people.sort(this::compareByAge); //
Methodenreferenz (Higher order function)
```

```
  @FunctionalInterface
  interface Comparable<T> {
    int compare(T first, T second);
  }
```

Comparator-Bausteine

```
people.sort(Comparator.comparing(Person::getAge));
people.sort(Comparator.comparing(
  Person::getLastName).reversed()); //
reversed ist eine default methode auf dem
Comparator-Interface
people.sort(Comparator
  .comparing(Person::getLastName)
  .thenComparing(Person::getFirstName));
```

Stream-API

```
list.forEach(System.out::println);
```

von Package `java.util.function`:

```
filter(Predicate<T> p)
map(Function<T, U> f)
foreach(Consumer<T> c)
```

```
var random = new Random();
Stream.generate(random::nextInt)
  .forEach(System.out::println);
```

```
Map<Integer, List<Person>> peopleByAge =
  people.stream()
 .collect(Collectors.groupingBy(Person::Age));
```