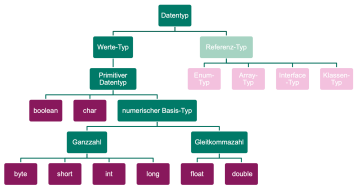
Unäre Operatoren

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

Implizites Casting von unten nach oben:

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

Long ist spezifischer als float:

```
long l = 1;
float f = l; // ok
```

Spezialfall char (unsigned 16 Bit):

1. explizites Casting von/zu byte, short.
2. implizites Casting zu int, float (und grösser).

Iterator

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

  it.remove();
}
```

List

```
list.removeAll(List.of("Bsys1",
"CN1"));
```

Wrapper-Klassen

```
Integer boxed =
Integer.valueOf(5);
int unboxed = boxed.intValue();
// auch atomatisches Boxing:
Integer wrapper = 123; //
Integer.valueOf(123);
int value = wrapper; //
wrapper.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

// oder:

String a = "Hello";
String b = new String(a);
a == b // false
```

Info: Compiler erkennt konstanten Ausdruck „OO“ + „Prog“ als „OOProg“. Somit ist `"OO" + "Prog" == "OOProg"`.

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.

Arrays

```
Arrays.equals(a, b); // Vergleicht
die Inhalte der Arrays
Arrays.deepEquals(a, b); //
Vergleicht verschachtelte Arrays
```

Mehrdimensionale Arrays

```
int[][] matrix = new int[2][3];
```

Methoden

Overloading

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

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`

Implementierung

```
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() !=
o.getClass()) return false;
    if (!super.equals(o)) return
false; // Wichtig bei Vererbung
    Person person = (Person) o;
    return
Objects.equals(firstName,
person.firstName) &&
Objects.equals(lastName,
person.lastName);
}
```

Hash-Code

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

Collections

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

Performance-Übersicht

	Finden	Einfügen	Löschen
ArrayList	Langsam	Sehr schnell am Ende	Langsam
LinkedList	Langsam	Sehr schnell an Enden	Sehr schnell an Enden
HashSet	Sehr schnell	Sehr schnell	Sehr schnell
HashMap	Sehr schnell	Sehr schnell	Sehr schnell
TreeSet	Schnell	Schnell	Schnell
TreeMap	Schnell	Schnell	Schnell

Feature-Übersicht

	Indexiert	Sortiert	Duplikate	null-Elem
ArrayList	Ja	Nein	Ja	Ja
LinkedList	Ja	Nein	Ja	Ja
HashSet	Nein	Nein	Nein	Ja
HashMap	Nein	Nein	Key: Nein	Ja
TreeSet	Nein	Ja	Nein	Nein
TreeMap	Nein	Ja	Key: Nein	Key: Nein

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:

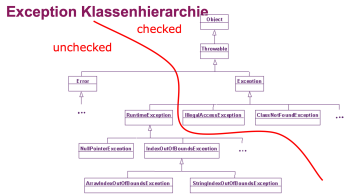
$$n + n\left(\frac{2}{3}\right) + n\left(\frac{2}{3}\right)^2 + n\left(\frac{2}{3}\right)^3 + \ldots = 3n$$

Die amortisierte Kostenanalyse beträgt somit <= 3 pro Einfügen.

Exceptions

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

Übersicht



`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();
}
```

Multicatch

```
try {
  // ...
} catch(NoStringException |
ShortStringException ex) {
  System.out.println("clip error:
" + ex.getMessage());
}
```

Try-With-Resources

```
try(var scanner = new
Scanner(System.in)) {}
// äquivalent zu:
{
  Scanner scanner = new
Scanner(System.in);
  try {

  } finally {
    if (scanner != null) {
      scanner.close();
    }
  }
}
```

Liskov Substitution Principle

„Objekte einer Klasse soll man durch Objekte einer Subklasse ersetzen können, ohne die Programm-Korrektheit zu verletzen."

Downcasting von `null`

Downcasting von `null` geschieht ohne Fehler:

```
Vehicle v = null;
Car c = (Car)v; // c == null
```

Packages

```
package p1;
public class A {}

package p2;
import p1.A;
public class C {}
```

Exakte Imports haben Priorität, Wildcards nicht:

```
package p1;
public class A {}

package p2;
public class A {}
```

Wildcard:

```
package p1;
import p2.*;

class Test {
  A a1;
  p2.A a2;
}
```

Exakter Import:

```
package p1;
import p2.A;

class Test {
  p1.A a1;
  A a2;
}
```

Static Imports

```
import static java.lang.Math.*;
```

`import java.lang.*` ist implizit immer vorhanden.

Interfaces

Gleiche Signatur, aber unterschiedliche Rückgabetypen geht nicht:

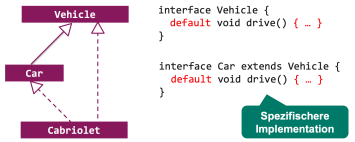
```
interface RoadVehicle {
  String getModel();
}
interface WaterVehicle {
  int getModel();
}
```

```
class AmphibianMobile implements
RoadVehicle, WaterVehicle {
  // Compilerfehler
  public int getModel() {
    return 1;
  }
}
```

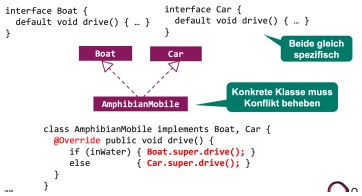
Mit Subtypen funktioniert es aber:

```
interface RoadVehicle {
  RoadVehicle clone();
}
interface WaterVehicle {
  WaterVehicle clone();
}
class AmphibianMobile implements
RoadVehicle, WaterVehicle {
  @Override
  public AmphibianMobile clone()
{ // Covarianz
    return new AmphibianMobile();
  }
}
```

Mehrere Interfaces mit spezifischerem Interface (ok)



Mehrere Interfaces mit derselben Priorisierung (nicht ok, Klasse muss den Konflikt beheben)



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);
}
```

```
class AgeComparator implements
Comparator<Person> {
  public int compare(Person p1,
Person p2) {
    return
Integer.compare(p1.getAge(),
p2.getAge());
  }
}

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

Stream-API

Wichtige Stream-Methoden

- filter
- map
- flatMap
- mapToInt/mapToDouble/mapToLong
- sorted
- distinct
- limit
- skip

Terminaloperationen

- forEach
- forEachOrdered (erhält Reihenfolge, besonders wichtig bei Parallelisierung)
- count
- min, max
- average, sum
- findAny, findFirst
- allMatch, anyMatch, noneMatch
- reduce

Parallelisierung

```
people.parallelStream()
  .filter(p -> p.getAge() > 16)
  .forEach(System.out::println);
```

Vordefinierte Funktionsschnittstellen

```
interface Predicate<T> {
  boolean test(T input);
}
interface Function<T, R> {
  R apply(T input);
}
interface Consumer<T> {
  void accept(T input);
}
```

Von `java.util.function`.

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

Methodenreferenz-Syntax

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

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

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

Optional

```
OptionalDouble averageAge =
people.stream()
  .mapToInt(Person::getAge)
  .average();

if (averageAge.isPresent()) {
  System.out.println(averageAge
    .getAsDouble());
}
```

Methoden: `empty()`, `of(double value)`, `ifPresent(Consumer)`, `orElse(double value)`

Groupings

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