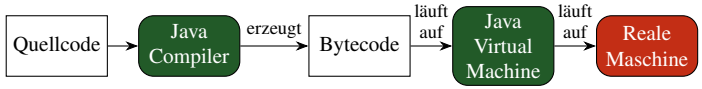
Java für C++ Programmierer — Frieder Loch

Laurin Heitzer
HS23

Java Virtual Machine

Java Runtime-Architektur



Bytecode

```
1 int n = 12;
2 int p = 1;
3 int i = 1;
4 while (i <= n) {
5     p = p * i;
6     i = i + 1;
7 }
8 System.out.println(p);
```

```
BIPUSH 12
ISTORE 1

ICONST_1
ISTORE 2

ICONST_1
ISTORE 3

ILOAD 3
ILOAD 1
IF_ICMPLE L4
//...
```

Wert 12 auf Stack legen
Wert von Stack an Index 1 speichern
Wert 1 auf Stack legen
Wert von Stack an Index 2 speichern
Wert 1 auf Stack legen
Wert von Stack an Index 3 speichern
Wert Index 3 auf Stack legen
Wert Index 1 auf Stack legen
Wenn 3 > 1

Java Bytecode

- Standardisierte Zwischensprache
- Für hypot. Stackmaschine

Java Virtual Machine

- Interpretiert Bytecode
- Implementiert für jede Zielplattform
- **J**ust-**I**n-**T**ime Compiler in realen Maschinencode

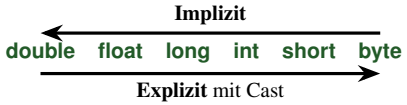
Datentypen

Primitive Datentypen

- Im Gegensatz zu C++ sind Wertebereiche auf jeder Plattform gleich
- Keine `unsigned` Typen

boolean	Boolscher Wert	true, false
char	Textzeichen (UTF16)	'a', 'b', 'c'
byte	Ganzzahl (8 Bit)	-128 bis 127
short	Ganzzahl (16 Bit)	-32'768 bis 32'767
int	Ganzzahl (32 Bit)	-2^{31} bis $2^{31}-1$
long	Ganzzahl (64 Bit)	-2^{63} bis $2^{63}-1$, 1L (L Suffix)
float	Gleitkommazahl (32 Bit)	0.1f, 2e4f ($2*10^4$)
double	Gleitkommazahl (64 Bit)	0.1, 2e4 ($2*10^4$)

Typumwandlung



boolean kann nicht implizit in andere Typen umgewandelt werden.

Explizit

Beispiel:
`int i = (int)3.14;`

Informationsverlust:

- Ganzzahl ⇒ Ganzzahl: Nur untere Bits werden übernommen
- Gleitkommazahl ⇒ Ganzzahl: Nachkommastellen werden abgeschnitten

Arrays

- Speichern Referenzen auf gleichartige Objekte
- Zugriff über Index

Beispiel: `int[] a = new int[5];`

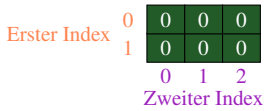
a[0]	a[1]	a[2]	a[3]	a[4]
0	0	0	0	0

Wenn Arrays vergrössert werden, wird deren Inhalt in einen neuen, grösseren Speicherbereich kopiert.

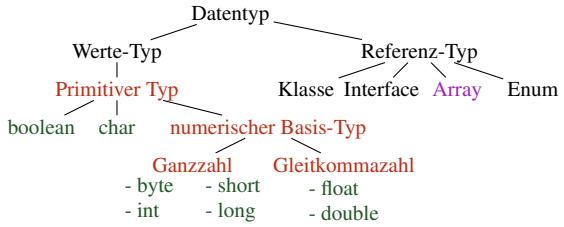
Mehrdimensionale Arrays

Beispiel: `int[][] m = new int[2][3];`

`m.length` ⇒ 2 `m[0].length` ⇒ 3



Einordnung



Enums

Enums sind ein eigener Datentyp, mit endlichem Wertebereich.

Syntax

Definition

```
1 public enum Weekday {
2     MON, TUE, WED, THU,
3     FRI, SAT, SUN;
4     public boolean isWeekend() {
5         return (this == SAT ||
6                 this == SUN);
7     }
8 }
```

Verwendung

```
1 // Deklaration einer Variable
2 Weekday currentDay; // default: null
3 // Zuweisung eines Wertes
4 currentDay = Weekday.MON;
5 // Methodenaufruf
6 currentDay.isWeekend(); // false
```

Enums kann man auch als eine Art Klasse interpretieren. Sie können auch Methoden, Variablen und Konstruktoren enthalten. Zusätzlich sind sie auch *type-safe*.
Es ist auch möglich einzelnen Enum-Konstanten Werte zuzuweisen:

```
1 public enum states {
2     ONE(1), TWO(2), THREE(3), FOUR(4), FIVE(5);
3 }
```

ArrayList

Eigenschaften

- ArrayList enthält Referenzen auf Objekte
- Kann dynamisch vergrössert/verkleinert werden
- Kann nicht direkt mit primitiven Datentypen (**int**, **float**, etc.) verwendet werden

Syntax

```
1 var stringList = new ArrayList<String>();
2 stringList.add("one"); // adds "one" to end of list
3 stringList.get(0); // returns String at index 0
4 stringList.set(0, "two"); // replaces element at index 0
5 stringList.remove(0); // removes element at index 0
```

Wrapping

Um einen Primitiven Datentyp zu referenzieren, muss dieser zuerst in ein Objekt gewrappt werden. Boxing/Unboxing implizit möglich.

Primitiver Typ	Wrapper-Klasse	Primitiver Typ	Wrapper-Klasse
boolean	Boolean	**int**	Integer
char	Character	**long**	Long
byte	Byte	**float**	Float
short	Short	**double**	Double

Weitere Collections

List	Folge von Elementen
Set	Menge von Elementen
Map	Abbildung von Schlüssel auf Wert

Methoden

Aufruf

Aufruf ohne Argumente

```
starLine();
```

Aufruf mit Argumenten

```
symbolLine('*', 5);
```

Call by Value

- Wert des Arguments wird in Parameter kopiert
- Änderung der Parameter ausserhalb Funktion nicht sichtbar

Mit Referenztypen

Referenz des Arguments wird in Parameter kopiert.

- Änderung am Objekt sichtbar
- Änderung an der Referenz nicht sichtbar

```
1 int[] v = new int[] {2,3};
2 square(v);
3
4 // v[0] == 4, v[1] == 9
5 // v != null
```

```
1 static void square(int[] p) {
2     p[0] *= p[0];
3     p[1] *= p[1];
4     p = null;
5 }
```

Deklaration

Deklaration ohne Parameter

```
1 static void starLine(){
2     for(int i = 1; i < 17; i++){
3     System.out.print('*');
4     }
5     System.out.println();
6 }
```

Deklaration mit Parameter

```
1 static void symbolLine(char symbol,
2         int length){
3     // ...
4 }
```

Rückgabewert

return-Anweisung zwingend, ausnahme: **void**-Methoden

Modifiers (gelten auch für Klassen und Variablen)

- **public**: von überall aus sichtbar
- **private**: nur innerhalb der Klasse sichtbar
- **protected**: nur innerhalb der Klasse und von Subklassen sichtbar
- **static**: Methode gehört zur Klasse, nicht zu einem Objekt (Instanziierung der Klasse nicht nötig)
- **final**: Methode kann nicht überschrieben werden
- **abstract**: Methode muss in Subklasse überschrieben werden

Variadische Methoden

Variadische Methoden sind Methoden, die eine variable Anzahl an Argumenten akzeptieren.

Syntax

Definition

```
1 static int sum(int... values) {
2     int result = 0;
3     for (int value : values) {
4         result += value;
5     }
6     return result;
7 }
```

Aufruf

```
1 s = sum(); // 0
2 s = sum(1); // 1
3 s = sum(1, 2); // 3
4 s = sum(1, 2, 3); // 6
5 s = sum(1, 2, 3, 4); // 10
6 //...
```

- Compiler erzeugt für variable Parameter ein Array, welches Argumente enthält.
- Argumente können jeweils nur von **einem** Typ sein.
- Parameter in der Variadischen Funktion muss **immer** der letzte Parameter sein.
- Parameter kann auch als Array übergeben werden.

Klassen

Definition

Eine Klasse spezifiziert die Gemeinsamkeiten einer Menge von Objekten mit denselben Eigenschaften, demselben Verhalten und denselben Beziehungen. [Balzert]

Aufbau

```
class Rectangle {
  //Variablen (Zustand)
  private int width;
  private int height;
  //Methoden (Verhalten)
    //Konstruktor
  public Rectangle(int h, int w) {
    this.height = h;
    this.width = w;
  }
}
```

Konstruktor

- Initialisiert ein Objekt/eine Klasse
- Hat keinen Rückgabewert
- Hat gleichen Namen wie Klasse
- Kann überladen werden
- Compiler erzeugt einen Standardkonstruktor, wenn kein Konstruktor definiert ist

null-Referenz

- Referenz auf "kein Objekt"
- Meist zur Initialisierung von Referenzen verwendet
- Gültig für alle Referenztypen
- Dereferenzierung führt zu NullPointerException

Selbstreferenz

Zur Selbstreferenz wird das Schlüsselwort **this** verwendet.

Instanziierung

Objekte werden mit dem **new**-Operator erzeugt (instanziiert): `Rectangle r = new Rectangle();`

Binding

Bei Referenzen wird zwischen statischem und dynamischem Binding unterschieden.

```
Vehicle vehicle = new Car();
```

Dynamischer Typ — Statischer Typ

Statische Bindung

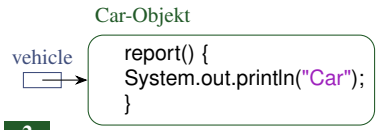
```
public class Vehicle {
  //...
  public static void test() {
    System.out.println("Vehicle");
  }
}
public class Car extends Vehicle {
  //...
  public static void test() {
    System.out.println("Car");
  }
}
```

```
Car c = new Car();
c.test(); // Car
Vehicle v = new Car();
v.test(); // Vehicle
```

Dynamische Bindung

Bei Aufruf nicht-privater Instanzmethoden wird Methode gemäss dynamischem Typ des Objekts aufgerufen.

```
Vehicle v = new Car();
v.report(); //dyn. Typ: Car
```



Details

Statische Bindung bei…

- …Konstruktoren
- …privaten Methoden
- …statischen Methoden

Dynamische Bindung bei…

- …nicht-privaten Instanzmethoden

Vererbung

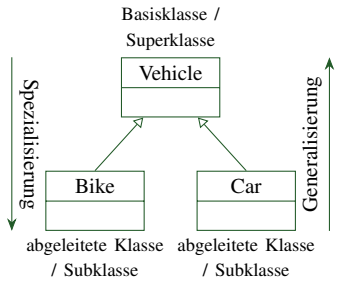
Syntax

```
public class Sub extends Super {
  public Sub() {
    //Konstruktor von Super
    //aufrufen
    super();
  }
}
```

Vererbung in Java

- Subklassen erben alle Attribute und Methoden aller Superklassen, die nicht **private** sind.

Vererbungshierarchie



Object

Alle Klassen erben implizit von der Klasse **Object**. Diese stellt folgende Methoden zur Verfügung:

- **public boolean** equals(**Object** obj): Vergleicht zwei Objekte auf Gleichheit
- **public String** toString(): Gibt eine String-Repräsentation des Objekts zurück
- **public int** hashCode(): Gibt einen Hashcode für das Objekt zurück

Diese Methoden können in jeder Klasse überschrieben werden, um sie an die jeweiligen Bedürfnisse anzupassen.

Polymorphie

Ein Objekt ist mit seinem Typ, sowie Typen aller Superklassen kompatibel. Jedoch nicht mit Typen von Subklassen.

```
Car c = new Car();
Vehicle v = c;
Object o = v;
```

Overriding

```
class Vehicle {
  public void move() {
    // do something
  }
}
```

```
class Car extends Vehicle {
  @Override
  public void move() {
    // do something else
  }
}
```

- Methoden können in Subklassen überschrieben werden
- Signatur **muss** gleich sein
- @Override Annotation ist optional, aber empfohlen
- Falls eine Klasse eine Superklasse, sowie ein Interface implementieren/erweitern soll, so muss zuerst **extends** *SuperKlasse* und dann **implements** *Schnittstelle* stehen (mit Leerzeichen dazwischen, kein Komma).

Schnittstellen (Interfaces)

Syntax

- Implizit **public** und **abstract**. (Andere Modifikatoren **nicht** erlaubt)
- Alle Methoden müssen von der implementierenden Klasse implementiert werden, sofern diese nicht **abstract** ist.
- Methoden dürfen **keine** Implementierung im Interface haben.
- Interfaces können **nicht** instanziiert werden.
- Eine Implementierung kann mehrer Interfaces implementieren.
- Falls mehrere Methoden mit gleichem Namen existieren, wird nur eine Methode implementiert.
- Konstanten werden automatisch als **public static final** deklariert.

```
interface Vehicle {
  void drive();
  int maxSpeed();
}
class Car implements Vehicle {
  @Override
  public void drive() {
    System.out.println("Driving");
  }
  @Override
  public int maxSpeed() {
    return 180;
  }
}
```

- Falls mehrere Konstanten mit demselben Namen existieren, muss der Name des Interfaces vorangestellt werden.
- Schnittstellen können andere Schnittstellen erweitern mit **extends**.

Lose Kopplung

Mittels loser Kopplung können mehrere Teams einfach miteinander arbeiten. Die Teams müssen sich nur auf die Schnittstelle einigen. Die Implementierung kann unabhängig voneinander erfolgen.

Abstrakte Klassen

Klasse, die nicht instanziiert werden kann und mindestens eine abstrakte Methode enthält.

Abstrakte Methoden

Deklaration einer Methode ohne Implementierung. Geht nur in abstrakten Klassen und Interfaces.

```
abstract void foo(int a);
```

Set-Interface

Menge von Elementen ohne Duplikate.

Methoden

- **boolean** add(E e): Fügt Element hinzu, wenn nicht bereits vorhanden
- Gleichheitsprüfung mit equals()

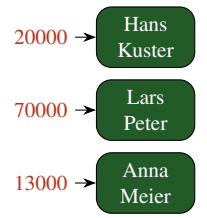
Beispiel

```
Set<String> carModels = new HashSet<>();
carModels.add("Mercedes");
carModels.add("Ferrari");
carModels.add("Ferrari"); // Returns false: bereits vorhanden
```

Map-Interface

- Objekt, das Schlüssel auf Werte abbildet.
- Kann keine Duplikate von Schlüsseln enthalten.
- Jeder Schlüssel kann höchstens einem Wert zugeordnet werden.

Visualisierung



Beispiel

```
Map<Integer, Student> map = new HashMap<>();
// Schlüssel: Matrikelnummer, Wert: Student
map.put(20000, new Student("Hans", "Kuster"));
map.put(70000, new Student("Lars", "Peter"));
map.put(13000, new Student("Anna", "Meier"));
// Schlüssel finden:
Student s = map.get(70000);
// Collection aller Werte
for (Student s : map.values()) { //...
}
```

Comparator-Interface

```
interface Comparator<T> {
  int compare(T o1, T o2);
  boolean equals(Object obj);
}
```

Returnwert compare

- **< 0**: o1 < o2
- **0**: o1 = o2
- **> 0**: o1 > o2

Returnwert equals

- **true**: obj = **this**
- **false**: obj ≠ **this**

Functional Interface

- Interface mit genau einer abstrakten Methode
- Kann mit Lambda-Ausdrücken verwendet werden
- Annotation @FunctionalInterface ist optional aber empfohlen
- Methodenreferenzen passen auf funktionale Interfaces

Beispiel

Funktionales Interface:

```
@FunctionalInterface
interface Comparator<T> {
  int compare(T o1, T o2);
}
```

Implementierung:

```
int compareByAge(Person a, Person b) {
  return Integer.compare(
    a.getAge(), b.getAge();)
}
```

Methodenreferenz:

```
Comparator<Person> myComp = this::compareByAge;
```

Equals-Methode

- Standardmässig vergleicht equals() auf Referenzgleichheit.
- Für Inhaltsvergleich muss equals() überschrieben werden.
  - Nur bei **String** bereits implementiert!
- Gibt **true** zurück, wenn die Objekte gleich sind.
- Wird equals() überschrieben, muss auch hashCode() überschrieben werden.

Syntax

```
class Person {
    private String name;
    @Override
    public boolean equals(Object o) {
        //...
    }
}
```

Vergleiche

- == vergleicht Referenzen.
- equals() vergleicht Inhalte.
- **instanceof** vergleicht Typen.
- Objects.equals() vergleicht Inhalte und behandelt **null** richtig.

HashCode-Methode / Hashing

HashCode-Methode

- Muss überschrieben werden, wenn equals() überschrieben wird.
- Gibt einen Hashcode zurück, der für gleiche Objekte gleich ist.
- Sollte möglichst effizient sein.

Syntax

```
@Override
public int hashCode() {
    return firstname.hashCode() + 31 * lastname.hashCode();
}
```

Bei mehreren Instanzvariablen: Hashcodes der Instanzvariablen mit verschiedenen Primzahlen multiplizieren und addieren.

Hashing

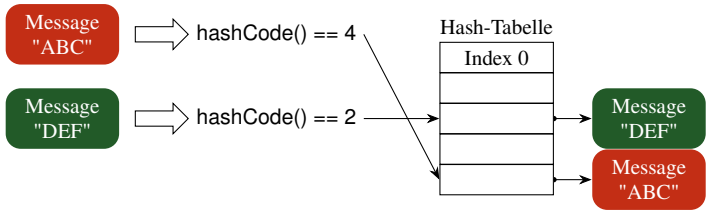
- Elemente werden auf Array (Hash-Tabelle) verstreut.
- Hash-Code wird aus Objekt berechnet und bestimmt den Index.

Regeln

x.equals(y) ⇒ x.hashCode() == y.hashCode()

- Umkehrung gilt nicht notwendigerweise.
- Überschreibung von equals() erfordert Überschreibung von hashCode().

Beispiel



Generics

Generics werden verwendet um einen einheitlichen Elementtypen zu forcieren. Beispielsweise in einer ArrayList.

Benennungskonventionen

- E – Element
- N – Number
- V – Value
- K – Key
- T – Type
- S, U, V, . . . – 2nd, 3rd, 4th types

Generische Typen

Verschiedene generische Typen

```
var strStack = new Stack<String>();
var intStack = new Stack<Integer>();
```

Kein Type-Cast

```
String value = stringStack.pop();
```

Statische Typ-Prüfung

```
// compile-time error bei:
strStack.push(23);
intStack.push("A");
```

Generische Klasse

Typ-Parameter

Platzhalter für generischen Typ.

```
class Stack<T> {
    //...   Typ-Parameter
}
```

- Typ-Parameter dient als "Typ-Variable" innerhalb generischer Klassen.
- Wie normaler Typ verwendbar.

Typ-Argument

Typ bei Einsatz angeben.

```
Stack<String> stack1;
Stack<Integer> stack2;
```

Typ-Argument

Generische Methode

```
static <T> void disp(T elem) {
    System.out.println(elem);
}
```

Beim Aufruf generischer Methoden muss der Typ nicht angegeben werden.

Generische Interfaces

Beispiel mit Iterator

```
interface Iterable<T> {
    Iterator<T> iterator();// ...
}
interface Iterator<E> {
    boolean hasNext();
    E next(); // ...
}
```

Iteration

For-Schleife:

```
for(String s : sList){
    System.out.println(s);
}
```

Tatsächlich generierter Code:

```
for(Iterator<String> i=sList.iterator();i.hasNext();){
    String s = i.next();
    System.out.println(s);
}
```

Type Bounds

Beispiel

```
class GfxStack<T extends Graphic>...
```

Mehrere Bounds sind mit & zu verknüpfen.

Nutzen

- Typ-Argument **muss** Subtyp von Graphic sein
- Bei Verwendung von spez. Methoden in gen. Methode/Klasse

Exceptions

Checked Exceptions

- Bei Methodendeklaration müssen Exceptions angegeben werden, welche gechecked werden müssen.

```
String clip(String s) throws Exception {
    if (s == null) {
        throw new Exception("s is null");
    }//...
}
```

- Nach **throws** können mehrere Exceptions mit "," separiert angegeben werden.

Unchecked Exceptions

- Keine **throws**-Deklaration und keine Behandlung nötig
- RuntimeException und Error sowie Subklassen
- Behebung nur durch Änderung des Codes

```
String clip(String s) {
    if (s == null) {
        throw new
            NullPointerException("no string");
    }// ...
}
```

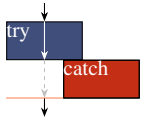
Exceptions behandeln

Werden mit **try–catch**-Blöcken behandelt.

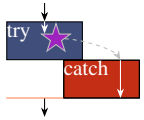
```
try {
    clip(null);
} catch (Exception e) {
    System.out.println("Exception: " + e);
}
```

Ablauf

Normalfall



Ausnahmefall



Stack Unwinding

- Baue solange Methoden-Frames vom Stack ab, bis Exception behandelt wird
- Falls main() mit Exception returnt, behandelt JVM diese mit Programmabbruch

Error vs. Exception

Error

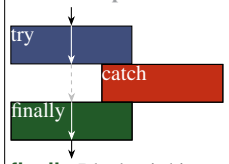
- Schwerwiegende Fehler, die **nicht behandelt** werden sollen
- Fehler in JVM: OutOfMemoryError, StackOverflowError
- Programmierfehler: AssertionError

Exception

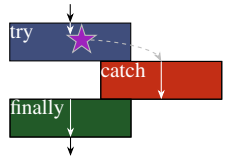
- Laufzeitfehler, die **behandelbar** sind
- Fehler in Eingabe, Parameter, Bedienung, . . . z.B. IOException ⇒ grundsätzlich behandeln
- Andere Laufzeitfehler ⇒ lieber nicht behandeln

finally-Ablauf

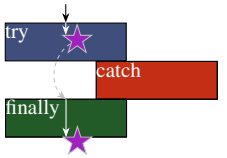
Keine Exception



Behandelte Exception



Unbehandelte Exception



finally-Block wird immer ausgeführt, auch wenn **catch**-Block eine Exception wirft.

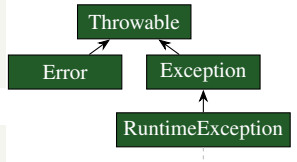
try-with-resources

- Objekte, die geschlossen werden müssen
- AutoCloseable-Interface

```
try(Scanner s=new Scanner(System.in)) {
    // mit s arbeiten
} catch (Exception e) {
    //...
}
```

Throwable

- Klasse oder Unterklasse von Throwable
- Klassifizierung des Fehlers



Benutzerdefinierte Exceptions

```
class MyException extends Exception{
    MyException() {}
    MyException(String msg) {super(msg);}
}
```

Lambdas

Syntax

```
(Parameter) -> {Methodenkörper}
```

- Parameter als Parameterlist übergeben (p1,p2,...)
- **return** benötigt in Methodenkörper, wenn {} verwendet
- **return**-typ implizit
- Wenn nur ein Statement, dann (), {} und **return** optional

Beispiel

```
people.sort((p1, p2) -> {
    return Integer.compare(p1.age(), p2.age());
});
```

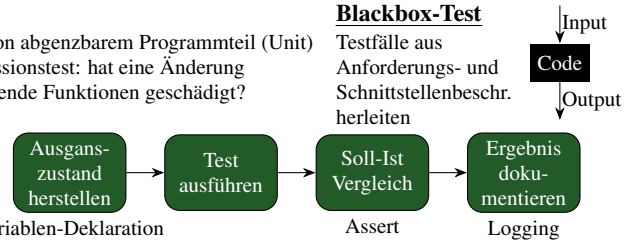
Unit Tests

Konzept

- Test von abgenzbarem Programmteil (Unit)
- Regressionstest: hat eine Änderung bestehende Funktionen geschädigt?

Blackbox-Test

Testfälle aus Anforderungs- und Schnittstellenbeschr. herleiten



Äquivalenzklassenbildung

1. Klassen bilden

Wertebereich der Parameter in Bereiche zerlegen, die von der Funktion wahrscheinlich gleich behandelt werden.

2. Tests erstellen

Pro Äquivalenzklasse Belegung der Eingangsvariablen wählen und Testfall schreiben.

Aufbau Testmethode

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;
//...
@Test
void testAbs() {
    int negativeValue = -1;
    assertEquals(1, abs(negativeValue));
}//...
```

- Keine Parameter
- Rückgabetyp **void**
- @Test-Annotation
- Asserts um Werte zu prüfen
- Testmethoden isoliert von anderen Testmethoden

Asserts

Prüfen von Gleichheit (inhaltlich)

- assertEquals(expected, actual)
- assertArrayEquals(expected, actual)
- ...

Prüfen von boolschen Ausdrücken

- assertTrue(actual)
- assertFalse(actual)
- ...

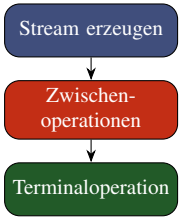
FIRST-Prinzip

- **F**ast: Ausführung ist schnell
- **I**ndependent: Reihenfolge der Tests ist nicht relevant.
- **R**epeatable: Ergebnis ändert sich nur wenn sich Implementierung ändert.
- **S**elf-validating: Testergebnis benötigt keine Interpretation.
- **T**imely: Tests werden früh geschrieben.

Stream-API

- Für deklarative Verarbeitung von Datenströmen
- Definiere **was** gemacht werden soll, nicht **wie**

Idee



Beispiel

```
people
    .stream()
    .filter(p -> p.getAge() >= 18)
    .map(p -> p.getName())
    .sorted()
    .forEach(System.out::println);
```

Endliche Quellen

- IntStream.range(0, 10): Zahlen von 0 bis 9
- Stream.of(2, 3, 4): Eigene Aufzählung
- Stream.empty(): Leerer Stream
- Collection.stream(): Stream aus Collection
- Stream.concat(s1, s2): Verkettung zweier Streams

Unendliche Quellen

generate()

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

iterate()

```
IntStream.iterate(0, i -> i + 1)
    .forEach(System.out::println);
```

Zwischenoperationen

Bei Collections ist es **nicht** erlaubt, diese mit Zwischenoperationen zu ändern. Auch nicht erlaubt sind Abhängigkeiten zu äusseren, änderbaren Variablen.

- filter(Predicate): Filtern mit Predicate-Funktionsobjekt/Lambda
- map(Function): Projizieren mit Funktionsobjekt/Lamda
- mapToInt...(Function): Projizieren auf **int**, **long**, **double**
- sorted(): Sortieren mit/ohne Comparator
- distinct(): Duplikate entfernen (equals())
- limit(**long** n): n-Elemente liefern
- skip(**long** n): n-Elemente überspringen

Terminaloperationen

- forEach(Consumer): Pro Element Operation anwenden, meist mit Seiteneffekt
- count(): Anzahl Elemente
- min(), max(): Mit Comparator-Argument
- average(), sum(): Nur für numerische Streams
- findAny(), findFirst(): Gibt irgendein / erstes Element zurück

Optional-Wrapper

- Wert exisitiert oder nicht
- average(), min(), max() geben Optional zurück
- Überprüfung mit isPresent()

Matching

- allMatch(), anyMatch(), noneMatch()
- Prüfen ob Prädikat auf alle/irgendein/kein Element zutrifft

Lazy Evaluation

- Element wird erst bereitgestellt, wenn Nachfolger Element anfordert
- Unendliche Streams sind meist Lazy

Collectors

- Collectors.toList(): Liste aller Elemente
- Collectors.toCollection(TreeSet::**new**): In beliebige Collection abbilden
- Collectors.groupingBy(key, aggregator): Gruppierung, Aggregator optional

  Aggregator: (averaging, summing, counting)

Gruppierungen

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

Input/Output

Streams allgemein

Input Stream

- Daten **von aussen** lesen
- Tastatur, Netzwerk, Dateien, ...

Output Stream

- Daten **nach aussen** schreiben
- Bildschirm, Netzwerk, Dateien, ...

Byte Streams

- Byteweises lesen (8-Bit Daten)
- Erben von: InputStream, OutputStream
- Beispiel: FileInputStream, FileOutputStream
- Close bei Exceptions: in.close() in **finally**-Block

Character Streams

- Unicode mit 16-Bit (UTF-16) codiert
- Erben von: Reader, Writer
- Zeichen-/Zeilenweise Ein- & Ausgabe

InputStream

```
int read(byte[] b, int offset, int length)
```

- Lese length Bytes in Array b ab Index offset
- Rückgabe = gelesene Bytes (-1 ↔ end of stream)

OutputStream

```
void write(byte[] b, int offset, int length)
void flush()
```

- Schreibt eventuell noch im Cache zwischengespeicherte Ausgaben
- Implizit bei close()

File-Input

```
var in = new FileInputStream("in.txt");
int value = in.read();
while(value >= 0) {
    byte b = (byte)value;
    // Mit b arbeiten
    value = in.read();
}
in.close();
```

Bestehende Datei zum Lesen öffnen

-1: end of file

Gelesenes Byte (wenn positiv)

File-Output

```
var out = new FileOutputStream("out.txt");
while(...){
    byte b = ...;
    out.write(b);
}
out.close();

new FileOutputStream("out.txt", true);
```

Datei neu anlegen bzw. überschreiben

Schreiben des Rests beim Schliessen ("Flush")

An Datei anhängen, falls existiert

Einfachster Dateizugriff

Ganze Datei binär einlesen:

```
byte[] data = Files.readAllBytes(Path.of("in.txt"));
```

Speicherintensiv bei grossen Dateien

Ganze Datei binär schreiben:

```
Files.write(Path.of("out.txt"), data);
```

Standard I/O

System.in

- InputStream

System.out und System.err

- PrintStream (Subklasse von OutputStream)

File-Reader

```
try(var rd = new FileReader("a.txt")){
    int val = rd.read();
    while (val >= 0) { //...
    }
}
```

- Systemabhängiges Character Set
- val = -1 ⇒ end of file
- value ist 16-Bit Unicode char

File-Writer

```
try(var wr = new FileWriter("b.txt", true)){
    wr.write("Hello");
    wr.write('\n');
}
```

- **true** ⇒ append
- .write("") String schreiben
- .write('') Einzelnen char schreiben

Zeilenweise lesen (InputStream)