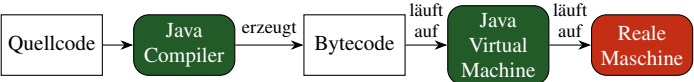




Java Virtual Machine

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 11 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 hypothetische Stack-Maschine

Java Virtual Machine

- Interpretiert Bytecode
- Implementiert für jede Zielformat
- Just-In-Time (JIT) Compiler in realen Maschinencode

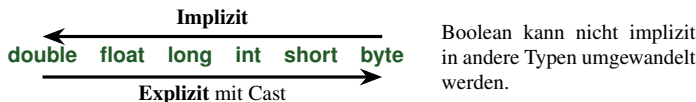
Datentypen

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



Explizite Typumwandlung

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

Informationsverlust:

- Ganzzahl \Rightarrow Ganzzahl: Nur untere Bits werden übernommen
- Gleitkommazahl \Rightarrow Ganzzahl: Nachkommastellen werden abgeschnitten

Arrays

- Speichern Referenzen auf gleichartige Objekte
- Zugriff über Index

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

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

Mehrdimensionale Arrays

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

`m.length \Rightarrow 2` `m[0].length \Rightarrow 3`

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

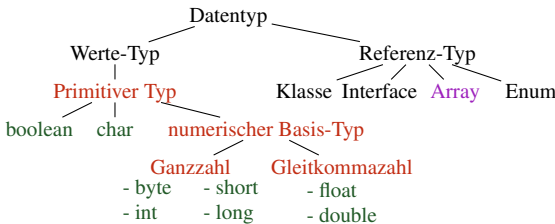
0 0 0 0 0

Erster Index

0 0 0 0
1 0 0 0

0 1 2
Zweiter Index

Einordnung



Enums

Enums sind ein eigener Datentyp, mit endlichem Wertebereich.

Syntax

Definition

```
1 public enum Weekday {
2     MONDAY,
3     TUESDAY,
4     WEDNESDAY,
5     THURSDAY,
6     FRIDAY,
7     SATURDAY,
8     SUNDAY;
9 }
```

Verwendung

```
1 //Deklaration einer Variable
2 Weekday currentDay;
3 //default-Wert => null
4 //Zuweisung eines Wertes
5 currentDay = Weekday.MONDAY;
```

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

`var stringList = new ArrayList<String>();`

Wrapping

Um einen Primitiven Datentyp zu referenzieren, muss dieser zuerst in ein Objekt gewrappert 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, int length){
2     // ...
3 }
```

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

```
1 class Rectangle {  
2     //Variablen (Zustand)  
3     private int width;  
4     private int height;  
5     //Methoden (Verhalten)  
6     //Konstruktor  
7     public Rectangle(int h, int w) {  
8         this.height = h;  
9         this.width = w;  
10    }  
11 }
```

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 (instanziert): `Rectangle r = new Rectangle();`

Binding

Bei Referenzen wird zwischen statischem und dynamischem Binding unterschieden.

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

Dynamischer Typ Statischer Typ

Statische Bindung

```
1 public class Vehicle {  
2     //...  
3     public static void test() {  
4         System.out.println("I'm a vehicle!");  
5     }  
6 }  
7 public class Car extends Vehicle {  
8     //...  
9     public static void test() {  
10        System.out.println("I'm a car!");  
11    }  
12 }
```

```
1 Car c = new Car();  
2 c.test(); // Test  
3 Vehicle v = new Car();  
4 v.test(); // Vehicle
```

Dynamische Bindung

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

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

Car-Objekt

```
vehicle  
└─> report() {  
      System.out.println("Car");  
    }
```

Details

Statische Bindung bei...

- ... Konstruktoren
- ... privaten Methoden
- ... statischen Methoden

Dynamische Bindung bei...

- ... nicht-privaten Instanzmethoden

Vererbung

Syntax

```
1 public class Sub extends Super {  
2     public Sub() {  
3         //Konstruktor von Super  
4         //aufrufen  
5         super();  
6     }  
7 }
```

Vererbung in Java

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

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.

Overriding

```
1 class Vehicle {  
2     public void move() {  
3         // do something  
4     }  
5 }
```

```
1 class Car extends Vehicle {  
2     @Override  
3     public void move() {  
4         // do something else  
5     }  
6 }
```

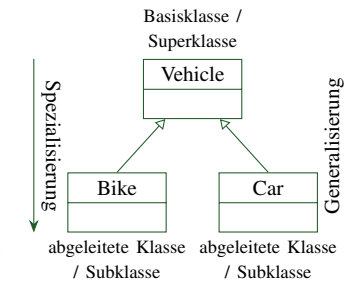
- Methoden können in Subklassen überschrieben werden
- Signatur **muss** gleich sein
- **@Override** Annotation ist optional, aber empfohlen
- Falls eine Klasse eine Superklasse, sowie ein Interface implementieren/erweitern sollte, so muss zuerst **extends SuperKlasse** und dann **implements Schnittstelle**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

Schnittstellen

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 mehrerer Interfaces implementieren.
- Falls mehrere Methoden mit gleichem Namen existieren, wird nur eine Methode implementiert.
- Konstanten werden automatisch als **public static final** deklariert.

Vererbungshierarchie



- 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.

Set-Interface

Menge von Elementen ohne Duplikate.

Methoden

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

Beispiel

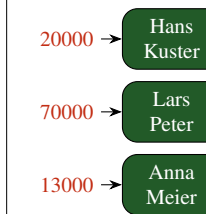
```
1 Set<String> carModels = new HashSet<>();  
2 carModels.add("Mercedes");  
3 carModels.add("Ferrari");  
4 carModels.add("Ferrari"); // Returns false: bereits vorhanden
```

Map-Interface

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

Visualisierung

Matrikelnr. Student



Beispiel

```
1 Map<Integer, Student> map = new HashMap<>();  
2 // Schlüssel: Matrikelnummer, Wert: Student  
3 map.put(20000, new Student("Hans", "Kuster"));  
4 map.put(70000, new Student("Lars", "Peter"));  
5 map.put(13000, new Student("Anna", "Meier"));  
6 // Schlüssel finden:  
7 Student s = map.get(70000);  
8 // Collection aller Werte  
9 for (Student s : map.values()) { ...  
10 }
```

Comparator-Interface

```
1 interface Comparator<T> {  
2     int compare(T o1, T o2);  
3     boolean equals(Object obj);  
4 }
```

Returnwert compare Returnwert equals

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

```
1 @FunctionalInterface  
2 interface Comparator<T> {  
3     int compare(T o1, T o2);  
4 }
```

Implementierung:

```
1 int compareByAge(Person a, Person b) {  
2     return Integer.compare(  
3         a.getAge(), b.getAge());  
4 }
```

Methodenreferenz:

```
1 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

```
1 class Person {
2     private String name;
3     @Override
4     public boolean equals(Object o) {
5         //...
6     }
7 }
```

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

```
1 @Override
2 public int hashCode() {
3     return firstname.hashCode() + 31 * lastname.hashCode();
4 }
```

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

Hashing

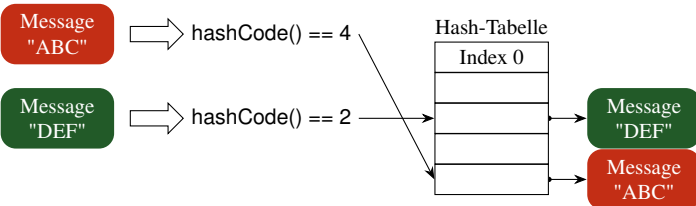
- Elemente werden auf Array (Hash-Tabelle) verstreut.
- Hash-Code wird aus Objekt berechnet und bestimmt den Index.

Regeln

$x.equals(y) \Rightarrow x.hashCode() == y.hashCode()$

- 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

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

Statische Typ-Prüfung

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

Kein Type-Cast

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

Generische Klasse

Typ-Parameter

Platzhalter für generischen Typ.

```
1 class Stack<T> {
2     //...
3 }
```

Typ-Argument

Typ bei Einsatz angeben.

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

- Typ-Parameter dient als "Typ-Variable" innerhalb der generischen Klasse.
- Wie normaler Typ verwendbar.

Generische Methode

```
1 static <T> void disp(T elem) {
2     System.out.println(elem);
3 }
```

Beim Aufruf generischer Methoden muss der Typ nicht angegeben werden.

Generische Interfaces

Beispiel mit Iterator

```
1 interface Iterable<T> {
2     Iterator<T> iterator();
3     // ...
4 }
```

```
1 interface Iterator<E> {
2     boolean hasNext();
3     E next(); // ...
4 }
```

Iteration

For-Schleife:

```
1 for(String s : sList){
2     System.out.println(s);
3 }
```

Tatsächlich generierter Code:

```
1 for(Iterator<String> i=sList.iterator();i.hasNext();){
2     String s = i.next();
3     System.out.println(s);
4 }
```

Type Bounds

Nutzen

- Typ-Argument muss Subtyp von Graphic sein
- Bei Verwendung von spez. Funktionen innerh. generischer Funktion/Klasse

Beispiel

```
1 class GraphicStack<T extends Graphic>...
// Mehrere Bounds können mit & verknüpft werden.
```

Exceptions

Checked Exceptions

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

```
1 String clip(String s) throws Exception {
2     if (s == null) {
3         throw new Exception("s is null");
4     }
5 }
```

Es können nach **throws** auch mehrere Exceptions mit Komma separiert angegeben werden.

Exceptions behandeln

Exceptions werden mit **try-catch**-Blöcken behandelt.

```
1 try {
2     clip(null);
3 } catch (Exception e) {
4     System.out.println("Exception: " + e);
5 }
```

Stack Unwinding

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

Error vs. Exception

Error

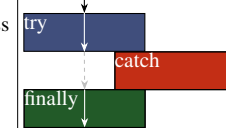
- Schwerwiegende Fehler, die **nicht behandelt** werden sollen
- Fehler in JVM: OutOfMemoryError, StackOverflowError
- Programmierfehler: AssertionError

Exception

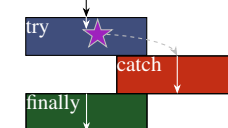
- Laufzeitfehler, die **behandelbar** sind
- Fehler in Eingabe, Parameter, Bedienung, ... z.B. IOException => grundsätzlich behandeln
- Andere Laufzeitfehler => lieber nicht behandeln

finally-Ablauf

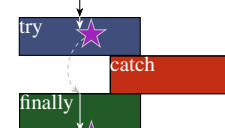
Normalfall



Behandelte Exception



Unbehandelte Exception



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

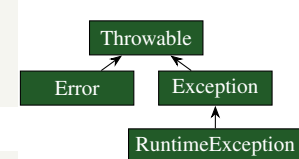
try-with-resources

- Objekte, die geschlossen werden müssen
- AutoCloseable-Interface

```
1 try(Scanner s=new Scanner(System.in)) {
2     // mit s arbeiten
3 } catch (Exception e) {
4     //...
5 }
```

Throwable

- Klasse oder Unterklasse von Throwable
- Klassifizierung des Fehlers



Benutzerdefinierte Exceptions

```
1 class MyException extends Exception{
2     MyException() {}
3     MyException(String msg) {super(msg);}
4 }
```

Lambdas

Syntax

- 1 (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

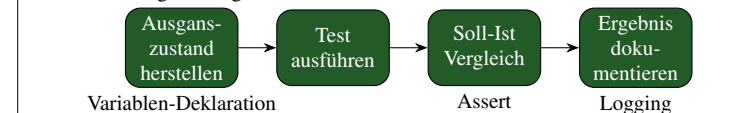
Beispiel

```
1 people.sort((p1, p2) -> {
2     return Integer.compare(p1.age(), p2.age());
3 });
```

Unit Tests

Konzept

- Test von abgrenzbarem Programmteil (Unit)
- Regressionstest: hat eine Änderung die bestehenden Funktionen geschädigt?



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

```
1 import static org.junit.jupiter.api.Assertions.*;
2 import org.junit.jupiter.api.Test;
3 //...
4 @Test
5 void testAbs() {
6     int negativeValue = -1;
7     assertEquals(1, abs(negativeValue));
8 }//...
```

- Keine Parameter
- Rückgabety **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

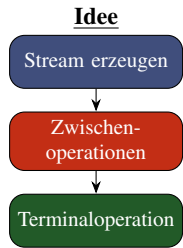
- **Fast**: Ausführung ist schnell
- **Independent**: Reihenfolge der Tests ist nicht relevant.
- **Repeatable**: Ergebnis ändert sich nur wenn sich Implementierung ändert.
- **Self-validating**: Testergebnis benötigt keine Interpretation.
- **Timely**: Tests werden früh geschrieben.

Stream-API

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

Beispiel

```
1 People
2 .stream()
3 .filter(p -> p.getAge() >= 18)
4 .map(p -> p.getName())
5 .sorted()
6 .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()
1 Random random = new Random();
2 Stream.generate(random::nextInt)
3 .forEach(System.out::println);

iterate()
1 IntStream.iterate(0, i -> i + 1)
2 .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/Lambda
- 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 existiert 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

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

1 Map<String, Integer> totalAgeByCity =
2     people.stream()
3     .collect(Collectors.groupingBy(Person::getCity,
4         Collectors.summingInt(Person::getAge)));
```

Input/Output

Streams allgemein

Input Stream

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

Byte Streams

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

InputStream

```
1 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

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

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

Output Stream

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

Character Streams

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

File-Input

```
1 var in = new FileInputStream("in.txt");
2 int value = in.read();
3 while(value >= 0) {
4     byte b = (byte)value;
5     // Mit b arbeiten
6     value = in.read();
7 }
8 in.close();
```

Bestehende Datei zum Lesen öffnen

-1: end of file

Gelesenes Byte (wenn positiv)

File-Output

```
1 var out = new FileOutputStream("out.txt");
2 while(...) {
3     byte b = ...;
4     out.write(b);
5 }
6 out.close();
7
8 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:

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

Speicherintensiv bei grossen Dateien

Ganze Datei binär schreiben:

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

Standard I/O

System.in

- InputStream

System.out und System.err

- PrintStream (Subklasse von OutputStream)

File-Reader

```
1 try(var rd = new FileReader("a.txt")) {
2     int val = rd.read();
3     while (val >= 0) { //...
4     }
5 }
```

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

File-Writer

```
1 try(var wr = new FileWriter("b.txt", true)) {
2     wr.write("Hello");
3     wr.write("\n");
4 }
```

- **true** => append
- .write("") String schreiben
- .write("") Einzelnen char schreiben

Zeilenweise lesen (InputStream)

