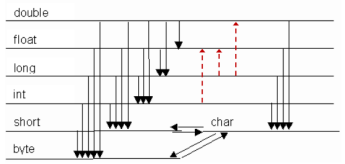
Primitive Datentypen

Alle signed

boolean	Boolescher Wert	true,false
char	UTF16, 16Bit	'a', 'B', '0', 'ë' etc.
byte	8 Bit	-128 bis 127
short	16 Bit	32'768 bis 32'767
int	32 Bit	-2^{31} bis 2^{31} -1
long	64 Bit	-2^{63} bis 2^{63} -1 , 1l (l Suffix, zwingend)
float	32 Bit	0.1f, 2e4f ($2 * 10^4$) (f Suffix, zwingend)
double	64 Bit	0.1, 2e4 (d Suffix, nicht zwingend)

Float comparison: `Math.abs(a - b) < 1e-8` sonst nummerische Fehler, 1/0: `ArithmeticException`, 0/0.0: `NaN`, 1/0.0: `Double.POSITIVE_INFINITY`, -1/0.0: `Double.NEGATIVE_INFINITY`

Casts



schwarz: explizit, implizit (ggf. Genauigkeitsverlust), andere implizit
Ganzzahl zu Ganzzahl(nur untere Bits): `(byte)0x1234`→0x34
Gleitkommazahl zu Ganzzahl→wird abgeschnitten 3.7→3

Referenzielle Datentypen

Array

`Arrays.equals(array1, array2)`
`Arrays.deepEquals(array1, array2)` für mehrdimensionale

Enum

```
public enum Weekday{
Monday(true), Tuesday(true), ...
private boolean workday;
/*Nur privater Konstruktor, implizit*/
Weekday(boolean workDay){this.workDay = workDay;}
public boolean isWorkDay(){return WorkDay;}}

enum Snafu{DISCORDIA, ERIS}
Snafu goettin = Snafu.ERIS;
```

Selektionen/Schlaufen

```
for (String s : array|arrayList){...}
switch (int|char|String|...){case wert1: break;}default:
```

Overloading/Overriding

`super.methodenname();` → überschriebene Methode aufrufen, Verweise innerhalb superMethode, wenn überschrieben, automatisch Aufruf der überschriebenen
Test ob dynamischer Typ auf statischer passt: reference `instanceof` TypeName
Variablenwerte werden nicht überschrieben!

Klassen

```
abstract class Vehicle {
abstract void report();
void print() {...}}
```

von `final` Klassen kann nicht geerbt werden, Alle Methoden auch automatisch `final`

Interface

Methoden implizit public & abstract
nur Konstanten erlaubt (keine Variablen)
`default`-Methoden: Können überschrieben werden, sonst Default-Implementierung, `static` unterstützt, Member von `Object`
Mehrere Default-Methoden mit gleicher Signatur, Adressierung: `nameInterface.super.methodenName();`

Varia

Zugriffskontrolle

public: Alle Klassen, **protected:** Package + Sub-Klassen, **(keines):** selbes Package, **private:** eigene Klasse

Packages

```
package p1; package p1.sub;
import p1.*; //Alle public Klassen, p1.sub nicht
```

Klassenvariablen direkt benutzen: `import static java.lang.System.out;`

Scanner

```
Scanner scanner = new Scanner(System.in);
```

Primzahlen

```
public boolean isPrime(long number) {
if (number <= 1) {
throw new IllegalArgumentException("number must be > 1");}
for (long x = 2; x <= Math.sqrt(number); x++) {
if (number % x == 0) { return false; }}
return true;}
```

Kleinster Wert

```
 double min = Double.MAX_VALUE;
 TreeNode smallest = null;
 for (TreeNode node: trees) {
 if (node.getFrequency() < min) {
 min = node.getFrequency();
 smallest = node;}
```

Permutation

```
 public static <T> void generate
 (List<T> permutation, Set<T> rest, PermutationEater<T> eater) {
  if(rest.isEmpty())
    eater.eat(permutation);
  else
   for(T element: rest) {
   permutation.add(element);
   Set<T> restCopy = new HashSet<>(rest);
   restCopy.remove(element);
   generate(permutation, restCopy, eater);
   permutation.remove(element);}}}
```

Potenzmenge generieren

```
static <T> Set<Set<T>> powerSet(List<T> elements) {
if (elements.size() == 0) {
        Set<Set<T>> empty = new HashSet<>();
        empty.add(new HashSet<>());
        return empty;}
List<T> remaining = new ArrayList<>(elements);
T current = remaining.remove(0);
Set<Set<T>> result = new HashSet<>();
for (Set<T> set : powerSet(remaining)) {
    result.add(set);
        Set<T> augmentedSet = new HashSet<>(set);
        augmentedSet.add(current);
        result.add(augmentedSet);}
return result;}
```

Verknüpfungen

```
boolean isLinked(Person from, Person to) {
return isLinked(from, to, new HashSet<>()); }

boolean isLinked(Person from, Person to, Set visited) {
if (from == to) { return true; }
if (visited.contains(from)) { return false; }
visited.add(from);
for (Person other : from.getKnownPeople())
{ if (isLinked(other, to, visited)) {
return true; }
} return false; }
```

Objektmethoden

Equals

```
@Override
public boolean equals(Object obj) {
if (obj == null) {
return false;}
if (getClass() != obj.getClass()) {
return false;} //von gleicher Klasse produziert?
Person other = (Person)obj;
return Objects.equals(firstName, other.firstName) &&
Objects.equals(lastName, other.lastName);
}}//Objects: null=null -> true, sonst equals-Methode
```

CompareTo

< 0: this ist kleiner als other , > 0: this ist grösser als other , 0: this ist gleich other

```
class Person implements Comparable<Person> {
private String firstName, lastName;
@Override
public int compareTo(Person other) {
int result = lastName.compareTo(other.lastName);
if (result == 0) {
result = firstName.compareTo(other.firstName);
} return result; }}
```

clone

```
class Person implements Cloneable {
@Override
public Person clone() {
return new Person(firstName, lastName); }}
// (Person)super.clone () CloneNotSupportedException
```

Java Collections

	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

	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

Methoden

Collection:`size()`, `toArray()`, `clear()`, `isEmpty()`
→**List**: `set(index)`, `get(index)`
→**Set**: `implements Comparable<T>`, `add(obj)`
Maps: wie Sets, `containsKey(Obj key)`, `containsValue(Obj value)`, `put(K key, V value)`, `Collection<V> values()`

Iteration

```
Iterator<String> it = stringList.iterator();
while (it.hasNext()) {
String elem = it.next();
if (elem.equals("...")) {
it.remove();} //geht nur bei richtigem Iterator
```

Iterator geht nur mit Sets, Lists — Alternativen für `HashMap`

```
for (int i : hashMap.keySet()){
hashMap.get(i);}
```

```
for (Map.Entry<Character, Double> entry: map.entrySet()) {
 entry.getValue(), entry.getKey()}
```

Comparator

```
Collections.sort(List<T> list);
```
aufsteigend, Alle Elemente: Comparable
```
Collections.sort(List<T> list, Comparator<T> comparator);
```

```
class MyComparator implements Comparator<Person>{
@Override
public int compare(Person p1, Person p2){}}
```

Generics

Klasse

```
Node<String> stringNode = new Node<>();
```

```
class Pair <T, U> {
private final T first;
private final U second;}
```

T muss von Klasse Person und weiterer erben/implementieren:
```
class Node<T extends Person & weitere> {...}
```
Nodes müssen sortiert werden können:
```
class Node<T extends Comparable<T>>
```

Methode

```
public <T> T majority(T x, T y, T z){...}
```
Bei generischen Klassen→`<T>` bei Methode nicht angeben!

Gewisse Klasse oder Sub-Klasse:
```
Node<? extends Person>node;
```
Gewisse Klasse oder Super-Klasse:
```
Node <? super Person>node;
```

Methodenreferenzen

Passen auf Funktionsschnittstellen `@FunctionalInterface` (genau 1 abstrakte Methode)
`this::compare`: im selben Objekt
`other::compare`: in Objekt other
`MyClass::compare`: statische in Klasse MyClass
`MyClass::new`: Konstruktor der Klasse MyClass

Lambda

weglassen (wird berechnet): Rückgabetyp, Typen der Parameter, {} des Body
```
(p1, p2) -> Integer.compare(p1.getAge(), p2.getAge())
```
() weglassen, wenn nur ein Parameter
```
p -> p.getAge() >= 18
```
Zugriff in Lambdas auf lokale Variablen?→Variablen automatisch `final` (auch ohne Deklaration)

Stream API

Zwischenoperationen

```
people
 .stream()
 //Filtern mit Lamda/Predicate-Fktobj
 .filter(p -> p.getAge() >=18)
 //Projizieren nur Lambda oder prim. Datentyp
 .map(p ->  p.getLastName())
 .mapToInt/mapToLong/mapToDouble(Function)
 //weitere
 .sorted([Comparator])
 .distinct()//gemäss equals
 .limit(long n)
 .skip(long n)
```

Terminaloptionen

```
 .forEach(System.out::print);
 .forEachOrdered(Sys...);//erhält Reihenfolge
 .count();
 .min([Comp.]);|.max([Comp.]);
 .average();, sum(); //Int, Long, Double
 .findAny();, findFirst(); //irgend, 1.
 .reduce();
 .toArray();
```

`sum()`:

```
people.stream()
 .mapToInt(p -> p.getAge())
 .reduce((age1, age2) -> age1 + age2);
```

`average()`, `min()`, `max()` lieferen Optional-Werte →`OptionalDouble`, `OptionalLong`, `OptionalInt`
```
if (averageAge.isPresent())
averageAge.ifPresent(System.out::println)
```
`allMatch()`, `anyMatch()`, `noneMatch()`:

```
boolean adultsOnly =
 people.stream()
  .allMatch(p -> p.getAge() >= 18);
```

unendliche Quellen:

```
Random random = new Random();
Stream.generate(random::nextInt);
```

```
Intstream.iterate(0, i -> i + 1)
```
Rückumwandlung in Collection:

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

Input/Output

Byte Streams: 8-Bit Daten, `InputStream`,`OutputStream`

```
try (FileInputStream in = new FileInputStream("...")){
} catch (FileNotFoundException e){
} catch (IOException e) {} //Fehler beim Lesen
```

Character Stream: 16-Bit (UTF-16), `Reader`, `Writer`

```
try (FileReader reader = new FileReader("quotes.txt")){
int value = reader.read(); //zwingend int, da -1 bei Ende
while (value >= 0) {
char c = (char)value;
//use character
value = reader.read();}}
```

```
try (FileWriter writer = new FileWriter("test.txt", true)){
writer.write("Hello!");
writer.write('\n');}
```

```
BufferedReader bufferedReader = new BufferedReader(fileReader);
String line = bufferedReader.readLine();
```

Serialisieren

```
Person implements Serializable
```
ausschliessen von Serialisierung: `private transient int age;`

```
try (ObjectOutputStream stream = new ObjectOutputStream(
new FileOutputStream("serial.bin"))) {
stream.writeObject(person);}
```

Regex

Uhrzeit: `if(Pattern.matches("[0-2][0-9]:[0-5][0-9]", myText))`

```
(?<HOURS>[0-2]?[0-9]):(?<MINUTES>[0-5][0-9])
String hoursPart = matcher.group("HOURS"|1);
.*\n[A-Z]*-([0-9]*).*//CH-8640
```

```
Integer.parseInt(line.substring(0, 3)//3 exkl.
```

JUnit

`assertEquals(expected, actual)`	actual equals expected
`ssertNotEquals(expected, actual)`	!(actual equals expected)
`assertSame(expected, actual)`	Referenzvergleich
`assertNotSame(expected, actual)`	!Referenzvergleich
`assertTrue(condition)`	condition
`assertFalse(condition)`	!condition
`assertNull(value)`	value == null
`assertNotNull(value)`	value != null
`fail()`	Immer verletzt

```
@Test
void testUnderflow() {
assertThrows(StackException.class, () -> {
Stack s = new Stack<String>(10);
s.pop();});}
```