Primitive Datentypen

boolean	Boolescher Wert	true, false
char	Textzeichen (UTF16)	'a', 'B', '0', 'ë' etc.
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

short * int = int
int / double = double
i++ = i
++i = i+1
x / 0 => ArithmeticException
float + int = float
true and false or not true and not false => false
(long)int == int => true
int + long * float => float
int < long < float => Compiler-Fehler (falscher Type beim zweiten Vergleich)
string + int + long = String («» + 1 + 2 = 12)
string + int == int + string => false, kein String Pooling
1 / 0.0 => Infinity (Double.POSITIVE_INFINITY)
0.0 / 0.f => NaN (Double.NaN)
1 == 2 && 0 / 0 == 0 => false
1 == 2 || 0 / 0 => Fehler / by zero (ArithmeticException)
«AB» == new String («AB») => false
1 + 3 / 2 => 2 (3/2 entspricht 1 da int - abgerundet)
1.0 + 3 / 2 => 2.0 (double)
1 + 3.0 / 2 => 2.5 (double)
1 + 3.0 / 0 => Double.POSITIVE_INFINITY
1 + 3.0 / 0.0 => Double.POSITIVE_INFINITY
A + 1 < a ergibt True, wenn Overflow, a ist Integer.MAX_VALUE
x * y / x != y, wegen numerischem Fehler bei Gleitkommazahlen

---

Typkonversion

Implizite Typkonversion: Zu breiterem Datentyp, kein Informationsverlust, aber zT Genauigkeitsverlus, zB int -> float. Compiler macht automatisch.
Explizite Typkonversion: Zu schmalerem Datentyp, Informationsverlust möglich, nicht automatisch.
b = (int)c;

Null-Referenz

Spezielle Referenz auf "kein Objekt". Vordefinierte Konstante, gültig für alle Referenztypen.
NullPointerException

Objekt-Initialisierung

Lokale Variablen: Wert nach Deklaration: undefiniert, Initialisierung: explizit
Instanz-Variablen: Wert nach Deklaration: Default, Initialisierung: Default, danach explizit

---

Overloading vs. Overriding

Overloading: Methode mit gleichem Namen, aber unterschiedlicher Parameterliste. Auflösung statisch.

```
1. int round(int x); 2. double round(double x);
3. Integer round(Integer x);

double r = round(2); => 1
Integer r = round(1); => 1
round(null); => 3
```

Bei mehreren Kandidaten wird die spezifischere gewählt, Rückgabetyp ist nicht entscheidend.

Overriding: Methode mit gleicher Signatur wird in Subklasse erneut implementiert. Auflösung dynamisch - ***dynamic Dispatch*** => Es wird die Methode in Subklasse verwendet. Zugriff auf Methode der Basisklasse mit `super.methode()`. Mit `@Override` lässt sich überprüfen, ob Methode in Basisklasse vorhanden ist.

```
class Base {
 void copyTo(Base other) {…}
}
class Sub extends Base {
 void copyTo(Base other) {…}
}
```

Hiding / Shadowing: Subklasse definiert Instanzvariable mit gleichem Namen wie die Superklasse neu. Zugriff auf Variable in Superklasse mit `super.` `((SuperSuperClass)this).variable`

Regeln bei equals()

```
x.equals(x) => true
x.equals(y) == y.equals(x)
x.equals(y) && y.equals(z) => x.equals(z)
x.equals(null) => false
```

Wenn eine ***equals()-Methode*** implementiert wird, muss auch eine hashCode Methode implementiert werden.

```
var point = new Point(1,2);
var set = new HashSet<Point>();
set.add(point);
assertTrue(set.contains(new Point(1,2)));
-----
public int hashCode() {
 return Objects.hash(firstName, lastName);
}
```

Exceptions

```
String clip() throws Exception {
 if (s == null) {throw new Exception(«xy»)}
 return s;
}

void main() {
 try {
  clip("stringToClip");
 } catch (Exception e) {
  // error handling
 } finally {
  // executed after try or catch
 }
}
```

Error: Schwerwiegende Fehler, die nicht behandelt werden sollen. Können sowohl zur Laufzeit als auch zur Kompilierzeit auftreten. (Fehler in JVM: OutOfMemoryError, VirtualMachineError, StackOverflowError / Programmierfehler: AssertionError)

Checked Exceptions: Exception wird zu Laufzeit geprüft und muss behandelt werden. (ClassNotFoundEx, IllegalAccessEx, IOException).
Unchecked Exceptions: Wird vom Compiler nicht geprüft, tritt auf wenn fehlerhafter Code ausgeführt wird. Alle unchecked Exceptions erben von RuntimeExeption. (NullPointerEx, IndexOutOfBoundsEx, ClassCastEx)
finally wird immer ausgeführt. Elemente nach dem try & catch werden nur ausgeführt, wenn das Programm vorher nicht mit einem Error abbricht. **Mehrere catch-klauseln:** passender catch wird von oben nach unten gesucht. Wenn eine neue Exception in einer catch-Klausel geworfen wird, wird diese nicht mehr im gleichen try-catch-Block behandelt. **Geschachtelte try-Blöcke:** Zuerst Behandlung des innersten try-Block, wenn unbehandelt, äusserer try-Block.
Try-with-resources: Objekte, die geschlossen werden müssen, können statt im finally auch direkt ins try(hier) geschrieben werden.

Kurzschreibweise if

```
x = a ? b : c <=> if (a) {x = b} else {x = c}
```

Switch-Statement

```
switch(ausdruck) {
 case Wert1:
  anweisungen;
  break;
 …
 default:
  anweisungen
}
```

Switch-Expression

```
Int k = …
String howMany = switch(k) {
 case 1 -> "one";
 case 2 -> "two";
 default -> "many";
};
```

Repetitions-Statements

```
while (Bedingung) {Anweisungen;}
```

Mindestens einmal ausgeführt:

```
do {Anweisungen;} while (Bedingung);

for (init; Bedingung; Update) {Anweisungen;}
```

For-Each

```
for (String s: stringList) {Anweisung;}
for (int i=0; i < array.length, i++) {Anweisung;}
```

Rekursion

```
Static long factorial(long number) {
 if(number == 0) {return 1;}
 else {return number * factorial(number - 1);}
}
```

Strings

Gleiche Strings können verschiedene Objekte sein. Deshalb funktioniert `==` bei String nicht (Referenzvergleich), ausser bei String Pooling. Deshalb besser `equals()` verwenden.

Arrays

```
int[] a = new int [5];
int[][] m = new int[2][3];
```

`a == b` vergleicht nur Referenzen zu Array-Objekt
`a.equals(b)` prüft, ob es dasselbe Array-Objekt ist
`Arrays.equals(a,b)` vergleicht Werte in Arrays
`Arrays.deepEquals(a,b)` vergleicht Werte in verschachtelten Arrays

Collections

List Folge von Elem, **Set** Menge von Elem, **Queue** Warteschlange, **Map** Abbildung Schlüssel -> Werte

Iterator

```
interface Iterator<T> {
  Boolean hasNext(); T next();
}
interface Iterable<T> {
 Iterator<T> iterator();
}
var stringList = new ArrayList<String>();
Iterator<String> it = stringList.iterator();
while (it.hasNext()) {String s = it.next(); …}
```

Array-List

```
ArrayList<String> stringList = new ArrayList<>();
stringList.add("OOP1");
stringList.add(0,"OOP1"); //insert at pos 0
String x = stringList.get(1); //get at pos 1
stringList.set(0, "OOP2"); //replace at pos 0
stringList.remove("OOP1");
stringList.remove(1); //remove at pos 1
stringList.contains("OOP1"); //Boolean
int size = stringList.size(); //list length
```

LinkedList

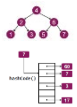
Verkettete Liste von Elementen: Dynamisch hinzufügbar und entfernbar, kein Umkopieren beim Einfügen und Löschen.

```
Deque<String> queue = new LinkedList<>();
```

FIFO-Queue (First-in-first-out)
Einfügen: `queue.addLast(elem);`
Entfernen: `queue.removeFirst(elem);`

LIFO-Queue (Last-in-first-out)
Einfügen: `queue.addLast(elem);`
Entfernen: `queue.removeLast(elem);`

Set



Keine Duplikate. Es gibt TreeSets und HashSets.
TreeSet: Elemente implementieren Comparable und equals()
HashSet: Elemente geben hashCode konsistent zu equals()

```
Set<String> firstSet = new TreeSet<>();
Set<String> otherSet = new HashSet<>();
```

Map

Es gibt TreeMaps und HashMaps.

```
Map<Integer, Student> map = new HashMap<>();
Student a = new Student("Andrea", "Meier);
map.put(2000, a);
Student x = map.get(2000);
For (int number : map.keySet()) {
 System.out.println(number);
}
```

Feature-Übersicht

	Indexiert	Sortiert	Duplikate	Null-Elem
ArrayList	x		x	x
LinkedList	x		x	x
HashSet				x
HashMap				x
TreeSet		x		
TreeMap		x		

Argumenttypen für Collection müssen immer Referenztypen sein. Deshalb verwendet man **Wrapper-Objekte**:
Char -> Character, int -> Integer, sonst gleich einfach gross

---

Zugriffskontrolle

Public: Sichtbar für alle Klassen, ***Protected:*** Package und alle Sub-Klassen, ***private:*** Nur eigene und äussere Klasse, ***keines:*** Alle Klassen im selben Package

---

Vererbung

```
Class Car extends Vehicle {}
```

Car erbt alle Instanzvariablen und Methoden der Basisklasse. Mit `super()` -> Zugriff auf Inhalte der Basisklasse

---

Typ-Polymorphismus

`car c = new Car();` sieht Variablen & Methoden von Car
`Vehicle v = c;` sieht Variablen & Methoden von Vehicle
`Object o = v;` sieht Variablen & Methoden von Object

Statischer Typ: gemäss Variablendeklaration zur Compile-Zeit (car, vehicle & object). ***Dynamischer Typ:*** effektiver Typ der Instanz zur Laufzeit (car bei allen). Der statische Typ bestimmt, welche Methoden aufrufbar sind.

Dynamische Typ-Prüfung (explizites Cast)

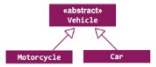
```
Vehicle v = …;
If (v instanceof Car car) {
 car.lock();
}
```

Abstract

Unvollständig implementierte Klasse. Dient als Basistyp für Sub-Klassen.

```
abstract class Vehicle {
 int speed;
 void accelerate() {…}
}
class Motorcycle extends Vehicle {…}

Vehicle vehicle1 = new Car();
New Vehicle(); //unmöglich
```



Abstrakte Methode: Methode ohne Rumpf. Muss in konkreten Subklassen implementiert werden.

Interfaces

Kann anstelle von Klasse verwendet werden. Eine Unterklasse kann nicht zwei Basisklassen implementieren, zwei Interfaces aber schon. ***Trennung Spezifikation und Implementation.***

```
Interface Vehicle { void drive(); int maxSpeed();}

class RegularCar implements Vehicle {
 @Override
 public void drive() {…}
 @Override
 Public int maxSpeed() {…}
}

Vehicle v = new RegularCar();
```

Abstrakte Klassen vs. Interfaces

Abstrakte Klassen können Attribute beinhalten und werden einfach vererbt. Interfaces können keine Attribute beinhalten und beliebig oft vererbt werden.

ENUMS

Eigener Datentyp mit endlichem Wertebereich. Auflistung von Werten. Enum-Werte werden in Grossbuchstaben geschrieben. Kann auch einen Konstruktor beinhalten.

```
public enum Weekday {
 MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
 SATURDAY, SUNDAY
}
if(currentDay == Weekday.SUNDAY) {…}
//oder
public enum Weekday {
 MONDAY(true), TUESDAY(true), WEDNESDAY(true),
 THURSDAY(true), FRIDAY(true),
 SATURDAY(false), SUNDAY(false);

 private boolean workDay;
 Weekday(boolean workDay) { // Konstruktor
  this.workDay = workDay;
 }
 public boolean isWorkDay() { // check if workday
  return workDay;
 }
}
Weekday testIfWorkday = Weekday.FRIDAY;
System.out.println(testIfWorkday.isWorkDay());
```

Hashing (Streuspeicher)

Elemente können sehr schnell gefunden werden.

Eigene Hashfunktion:

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

Lambdas

Ein Lambda ist eine Referenz auf eine anonyme Methode.

Methodenreferenz ohne Lambda:

```
int compareByAge(Person p1, Person p2) {
 return Integer.compare(p1.getAge(), p2.getAge());
}
people.sort(this::compareByAge);
```

Mit Lambda:

```
people.sort((p1, p2) ->
Integer.compare(p1.getAge(), p2.getAge()));
```

Removeall mit Lambda

```
Class Utils {
 static void removeAll(Collection<Person>
 collection, Predicate criterion){
  var it = collection.iterator();
  while(it.hasNext()){
   if (criterion.test(it.next())){
    it.remove();
   }
  }
 }
}
Utils.removeAll(people, p -> p.getAge() < 18);
```

Stream API

Deklarative Abfrage von Collections. Inspiriert von SQL. Wird Lazy verarbeitet, startet bei der Terminaloperation

```
People
 .stream()
 .filter(p -> p.getAge() >= 18)
 .map(p -> p.getLastName())
 .sorted()
```

Rückumwandlung zu Array:

```
Person[] array = peopleStream.toArray(Person[]::new)
```

Rückumwandlung zu Collection:

```
List<Person> list =
peopleStream.collect(Collectors.toSet());
List<Person> list = peopleStream.toList();
```

Reduzieren

```
Optional<String> result =
 people.stream()
 .map(p -> p.getName())
 .reduce((name1, name2) -> name1 + name2);
```

Zwischenoperatoren:
`filter()`, `map()` //projezieren, **`mapToInt()`, `sorted()`, `distinct()`** //keine duplikate, **`limit()`** //erste elemente liefern , **`skip()`** //ignorieren

Terminaloperatoren:
`forEach()`, `forEachOrdered()`, `count()`, `min()`, `max()`, `average()`, `sum()`, `findAny()`, `findFirst()`

Optional-Operationen:
`OptionalDouble.empty()` //inexistenter Wert,
`OptionalDouble.of()` //existenter Wert,
`OptionalDouble.ifPresent()` //Falls existiert,
`OptionalDouble.orElse()` //liefert Wert falls vorhanden, sonst anderer

Stream vs. Collection

Stream: Kein direkter Zugriff auf Elemente, nur einmal verwendbar, Pipeline statt Änderungen, deklarativ «Was wird abgefragt?»

Collection: Direkter Zugriff auf Elemente, Häufig als Attribute in Klassen verwendet, Änderungen üblich, imperativ «Wie wird abgefragt»

Records

Für einfache Werte, automatische Implementierung. Zum Beispiel *PersonID*. Getter etc. werden automatisch gemacht

```
Record PersonID(Long id) {
}
```

Getter nicht `getPersonID`, sondern einfach `PersonID`.

Annotations

Codebeispiele

```
class X {}
class Y extends X {}
class Z extends Y {}

public class Main {
 public static void print (Y y) {println("y");}
 public static void print (Z z) {println("z");}
 public static void main(String[] args) {
  print((Y) new Y()); //y
  print((Z) new Z()); //z
  print((Y) new Z()); //y
  print((Z) new Y()); //Exception
 }
}
```

```
class Graphic {
 void moveTo(Graphic other) {
  System.out.println("1");
 }
}
Class Circle extends Graphic {
 void moveTo(Graphic other) {
  System.out.println("2");
 }
 void moveTo(Circle other) {
  System.out-println("3");
 }
}
Circle c = new Circle();
Graphic g = new Circle();

c.moveTo(g); //2
c.moveTo(c); //3
g.moveTo(c); //2
g.moveTo(g); //2
((Graphic)c).moveTo(g); //2
((Circle)g).moveTo(c); //3
```

```
void append(Train tail) {} //3
void append(FastTrain tail) {} //4
Train a = new FastTrain();
FastTrain b = new FastTrain();

a.append(null); //4
a.append(b); //4
a.append((Train)b) //3
b.append(null); //4
b.append(a); //3
((Train)b).append(a); //3
```

=> Vorderer Wert ist nicht relevant

Unittests

```
@Test
Public void showInconsistency() {
 var point = new Point(1, 2);
 var set = new HashSet<Point>();
 set.add(point);
 assertTrue(set.contains(new Point(1, 2)));
}
```