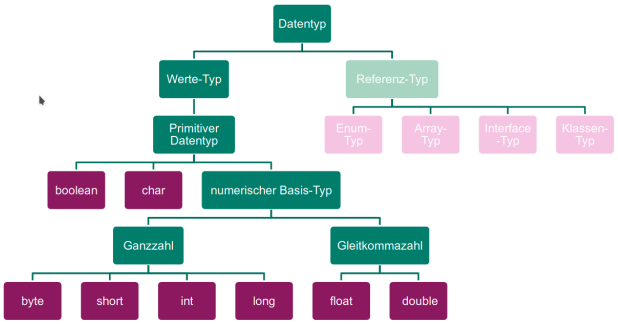
boolean	Boolescher Wert	true, false
char	Textzeichen (UTF16)	'a', 'B', '0', 'ë' etc.
byte	Ganzzahl (8 Bit)	-128 bis 127
short	Ganzzahl (16 Bit)	-32'768 bis 32'767
int	Ganzzahl (32 Bit)	-2^{31} bis $2^{31}-1$
long	Ganzzahl (64 Bit)	-2^{63} bis $2^{63}-1$, 1L (L Suffix)
float	Gleitkommazahl (32 Bit)	0.1f,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



	Lokale Variablen	Instanz-Variablen (und Klassen-Variablen)
Wert nach Deklaration	undefiniert	Default-Wert
Initialisierung	muss explizit geschehen	Default-Wert, danach explizit zuweisbar

Typ	Default-Wert
boolean	false
char	'\u0000'
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
Referenztyp	null

in sets etc with <> you can't use primitive types, use the others

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

import ranks: single import, your own, on demand import p2.A, A, p2

static import = using namespace

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

public class Demo {
  public static void main(String[] args) {
    out.println(sin(PI/2));
  }
}
```

Statt System.out

Statt Math.PI

Only the first class in a file can be public and therefore be used in other files. The rest of the classes are private to that file.

- Reihenfolge des Vorrangs
  1. Eigene Klasse (bzw. innere Klassen, später)
  2. Single Type Imports
     - `import p2.A;`
  3. Typen in eigenem Package
     - `package p1; class X {}`
  4. Import on Demand
     - `import p2.*;`
- Notwendige Konfliktauflösung
  - Mehrere Kandidaten mit gleichem Vorrang

Muss public sein

```
public class TopLevelClass {

}

abstract class AnotherTopLevelClass {
  abstract void method();
}
```

Kein public erlaubt, aber z.B. abstract

override can expand visibility just like it can change return type

```
class Vehicle {
  protected void print() { … }
}

class Car extends Vehicle {
  public void print() { … }
```

Typ-Polymorphismus

- Jede Schnittstelle definiert auch einen Typ
- Implementierende Objekte haben auch diesen Typ
- Liskov Substitution Principle:
  - Kann bei Schnittstellentyp Objekte verwenden, deren Klasse diese Schnittstelle implementiert

```
Vehicle v = new RegularCar();
```

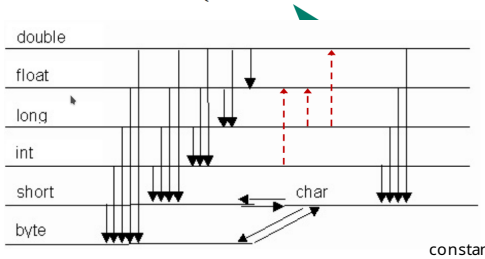
Bis jetzt: Iteration

$$factorial(n) = \prod_{i=1}^{n} i$$

Rekursion

Abbruchbedingung («Verankerung»)

$$factorial(n) = \begin{cases} 1 & (n = 0) \\ n \cdot factorial(n-1) & (n > 0) \end{cases}$$



constructors can be created this way:

```
public Rectangle() {
  this(10, 20);
}
```

Muss erste Anweisung sein

```
public Rectangle(int length, int width) {
  this.length = length; this.width = width;
```

the static type only dictates what funtions can be called not where the function is written. It will always use the function of the dynamic type if available!

- `double` >₁ `float`
- `float` >₁ `long`
- `long` >₁ `int`
- `int` >₁ `char`
- `int` >₁ `short`
- `short` >₁ `byte`

multiple line strings """

var = auto

var a = new Object();

not overridable function

```
public final void stop() {
  speed = 0;
}
```

final classes exist as well those can't be extended / inherited from

check if 2 classes are the SAME. not inherited like instance of

```
if (getClass() != obj.getClass()) {
  return false;
```

- Paket-Name aus (fiktivem) Domain-Namen bilden: in umgekehrter Reihenfolge
  - Aus `oop1.ost.ch` => Paketname `ch.ost.oop1`
- Weitere Differenzierung
  - `ch.ost.oop1.examples`

modules as a way to combine packages into usable groups

```
module mathlib {
  exports ch.ost.oo.math.core;
  exports ch.ost.oo.math.special;
}
```

Explizite Sichtbarkeit ausserhalb Moduls

essentially lambda version of classes

Anonyme Innere Klassen

- Noch knapper wird der Code mit einer anonymen inneren Klasse:

```
public static void main(String[] args) {

  var reminder = new TimerTask() {
    @Override
    public void run() {
      System.out.println("OOP1 Übungen!");
    }
  };
```

TimerTask ist eine abstrakte Klasse

This is just awesome !

Enum mit Konstruktor

```
public enum Weekday {
   MONDAY(true), TUESDAY(true), WEDNESDAY(true),
   THURSDAY(true), FRIDAY(true),
   SATURDAY(false), SUNDAY(false);

   private boolean workDay;

   Weekday(boolean workDay) {
     this.workDay = workDay;
   }

   public boolean isWorkDay() {
     return workDay;
   }
}
```

Nur privater Konstruktor (implizit ohne Angabe)

Rundungsfehler

- `0.1 + 0.1 + 0.1` ist nicht exakt `0.3`
- Vergleich `x == y` von float/double problematisch

Überlauf / Unterlauf

- Ganzzahlen
  - Falsche Ergebnisse bei Bereichsüberschreitung
  - `2147483647 + 1` → `-2147483648`
- Gleitkomma-Zahlen
  - `2 * 1e308` → POSITIVE_INFINITY
  - `5e-324/2` → `0.0`

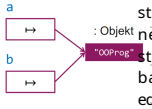
Undefinierte Operationen

- Ganzzahlen
  - Fehler/Exception bei Divison/Modulo durch 0
- Gleitkomma-Zahlen
  - `1.0 / 0.0` → POSITIVE_INFINITY
  - `-1.0 / 0.0` → NEGATIVE_INFINITY
  - Undefiniert → NaN (Not a Number)
    - `0.0/0.0`, `∞ - ∞`, etc.

constant = final in java : public final int number = 5;

static in java used to make functions work without objects.

- Compiler-Optimierung
- Gleiche String-Literale als dasselbe Objekt



string is imutable new string every time strings are reference basd in java use equals for content

```
String a = "OOProg";
String b = "OOProg";
```

- Achtung
  - `a == b` vergleicht nur Referenzen zu Array-Objekt => `false`
  - `a.equals(b)` prüft, ob es dasselbe Array-Objekt ist => `false`
- `Arrays.equals(a, b)` vergleicht Werte in Arrays => `true`
  - Geschachtelte Arrays mit `Arrays.deepEquals(a, b)` vergleichen

List	Folge von Elementen
Set	Menge von Elementen
Queue	Warteschlange
Map	Abbildung Schlüssel => Werte

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

contains,add, set, remove

Deepequals for regular arrays!

Expliziter Konstruktor

```
public class Vehicle {
   private int speed;

   public Vehicle(int speed) {
     this.speed = speed;
   }
}

public class Car extends Vehicle {
   private int doors;

   public Car(int speed, int doors) {
     super(speed);
     this.doors = doors;
   }
}
```

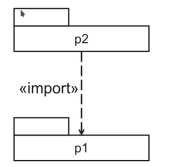
own constructor has to be explicitly called!!

```
if (v instanceof Car) {
  Car c = (Car)v;
  c.lock();
}
```

- Statische Bindung
  - Zugriff auf Field der eigenen Klasse mit `description` oder `this.description`
  - Zugriff auf Field der Basisklasse mit `super.description` oder `((Vehicle)this).description`
  - `super.super` gibt es nicht, Zugriff stattdessen mit `((SuperSuperClass)this).variable`

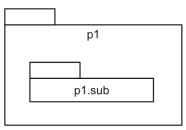
check references to ensure no nullpointer exception.

```
return Objects.equals(firstName, other.firstName) &&
       Objects.equals(lastName, other.lastName);
```




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

```
package p1;
public class A {}
…
```




```
package p1;
public class A {}
…
```

```
package p1.sub;
public class E {}
…
```

access control

Keyword	Sichtbar für
public	Alle Klassen
protected	Package und alle Sub-Klassen
(keines)	Alle Klassen im selben Package
private	Nur eigene und äussere Klasse

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

```
…
@Override
public void drive() {
  System.out.println("driving…");
}

@Override
public int maxSpeed() {
  return 120;
}
…
}
```

interfaces are always abtstract and can therfore now be used with new interface(). All Methods of interfaces are public. (implicit) if you define variables then they are implicitly final -> int a = 10

Auch Mehrfach-Spezialisierung

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

interface RoadVehicle extends Vehicle {
   String tireModel();
```

Erbt D

- Muss explizit auf Konstante zugreifen

```
interface RoadVehicle {
  int MAX_SPEED = 120;
}

interface WaterVehicle {
  int MAX_SPEED = 80;
}

class AmphibianMobile implements RoadVehicle, WaterVehicle {
  void test() {
    System.out.println(RoadVehicle.MAX_SPEED);
    System.out.println(WaterVehicle.MAX_SPEED);
  }
}
```

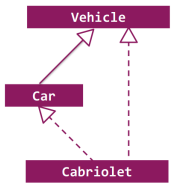
MAX_SPEED ist

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

interface WaterVehicle {
   int getModel();
}

class AmphibianMobile implements RoadVehicle, WaterVehicle {
}
```

nicht implementierbar

Default Methods: Resolution

- Wähle spezifischere Default-Implementation
  - Spezifischer = in spezialisierten Schnittstelle implementiert



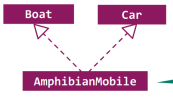

```
interface Vehicle {
  default void drive() { … }
}

interface Car extends Vehicle {
  default void drive() { … }
}
```

Spezifischere Implementation

Default Methods: Konflikte

```
interface Boat {
  default void drive() { … }
}

interface Car {
  default void drive() { … }
}
```

Beide gleich spezifisc



Konkrete Klasse muss Konflikt beheben

```
class AmphibianMobile implements Boat, Car {
  @Override public void drive() {
     if (inWater) { Boat.super.drive(); }
     else         { Car.super.drive(); }
  }
```

Ausnahme: Entfernen via Iterator

solves the concurrent modfication problem

```
var it = stringList.iterator();
while (it.hasNext()) {
  String elem = it.next();
  if (elem.equals("...")) {
    it.remove();
```

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

Einheitliche Schnittstelle

Sortiert & immer effizient

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

Unsortiert & oft sehr effizient

```
Map<Integer, Student> bachelors = new TreeMap<>();
```

Einheitliche Schnittstelle

Nach Schlüssel sortiert & immer effizient

```
Map<Integer, Student> masters = new HashMap<>();
```

Unsortiert & oft sehr effizient

use Objects.hash(variables) to include class variables!

```
List<String> firstList = new ArrayList<>();
```

Einheitliche Schnittstelle

Schnelles Zugre

```
List<String> otherList = new LinkedList<>();
```

	ArrayList	LinkedList
Index-Zugriff	Sehr schnell	Langsam
Einfügen	Langsam	Sehr schnell *
Suchen	Langsam	Langsam
Speicher	Array mit Reserveplatz	Overhead pro Element

* an Anfang und Ende
schnell = optimale Laufzeit (hier konstant)
langsam = nicht optimal (hier linear)

	TreeSet	HashSet
Finden	Schnell	Sehr schnell
Einfügen	Schnell	Sehr schnell
Löschen	Schnell	Sehr schnell
Sortierung	Ja	Nein

Hashing: Sehr schnelles Finden

- Set: Finden eines Elements (`contains`)
- Map: Finden via Schlüssel (`get`)
- Listen sind dafür ungeeignet
  - Durchsuchen der gesamten Menge: lineare Zeit

Map-Beispiel

Schlüssel: Matrikelnummer
Wert: Student

```
Map<Integer, Student> map = new HashMap<>();

Student a = new Student("Andrea", "Meier");
Student b = new Student("Bertha", "Müller");

map.put(20000, a);
map.put(70000, b);

Student x = map.get(20000);

for (int number : map.keySet()) {
  System.out.println(number);
}
```

Finden nach Schlüssel (sehr effizient)

- Konstruktoren
  - `Exception()`
  - `Exception(String message)`
  - `Exception(String message, Throwable cause)`
- Methoden
  - `String getMessage()`
  - `void printStackTrace()`

```
java.lang.NullPointerE
    at Test.h(Test
    at Test.g(Test
```

- `ArithmeticException`
  - Division durch Null etc.
- `NullPointerException`
  - Dereferenzieren von null
- `ArrayIndexOutOfBoundsException`
  - Array-Zugriff mit ungültigem Index
- `ClassCastException`
  - Objekt hat nicht den Typ des Down-Casts

…tion auslösen

Aufrufer soll wissen, dass es diese Exception geben kann

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

Exce

```
try {
  … regular code …
} catch (Exception e) {
  … error handling …
}
```

rethrow, catch, do something then throw further

```
…
} catch(StringClipException e) {
  …
  throw e;
```

Benutzerdefinierte Exceptions

```
class StringClipException extends Exception {
  public StringClipException() {}

  public StringClipException(String message) {
    super(message);
```

used with try catch block (finally is optional)
will always be executed, after catch clause

```
} finally {
  … statements …
}
```

Error und 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 etc. z.B. `IOException` → grundsätzlich behandeln
  - Andere Laufzeitfehler (`NullPointerException` etc.) → lieber nicht behandeln (Bugs)

Errors and runtime exceptions can't be handled. Those lead to a crash.
Runtime errors include out of bounds exceptions

```
try (Scanner s = new Scanner(System.in)) {
  … // work with s
```

`this::compare`	Methode `compare` im selben Objekt
`other::compare`	Methode `compare` in Objekt `other`
`MyClass::compare`	Statische Methode in Klasse `MyClass`
`MyClass::new`	Konstruktor der Klasse `MyClass`

äquivalent

Einsatz von Methoden

- Bis anhin: Aufrufe
  - `compareByAge(hans, susi)`
  - Argumente werden ausgewertet
  - Methode wird direkt aufgerufen
- Neu: Referenz
  - `this::compareByAge`
  - Methode wird wie ein Objekt behandelt
  - Referenz auf die Methode (noch kein Aufruf)

```
void print(List<Person> people) {
  people.sort(this::compareByAge);
  System.out.println(people);
}
```

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

infinite streams:
`Stream.generate(random::nextInt)`
`Intstream.iterate(0, i -> i + 1)`

Endliche Quellen

- `IntStream.range(1, 100)`
  - Zahlen von 1 bis 100 (analog für LongStream)
- `Stream.of(2, 3, 5, 7, 11, 13)`
  - Eigene Aufzählung
- `Stream.empty()`
  - Leerer Stream (Testzwecke)
- `Collection.stream()`
  - Wie gehabt
- `Stream.concat(stream1, stream2)`
  - Alle Elemente in stream 1, gefolgt von Elemente in stream2

Rückumwandlung

- Zu Array

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

- Zu Collection

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

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

ambda Ausdruck

Referenz auf anonyme Methode

- Syntax: **(Parameterliste) -> { Body }**
- Rückgabetyp wird inferriert (berechnet)

contractions

```
p -> p.getAge() >= 18

(p1, p2) -> {
  return Integer.compare(p1.getAge(), p2.getAge());
}

(p1, p2) -> Integer.compare(p1.getAge(), p2.ge
```

```
interface Comparable<T> {
  int compareTo(T other);
}
```

```
interface Comparable<T> {
  int compareTo(T other);
}
```

- Resultat
  - < 0: `this` ist kleiner als `other`
  - > 0: `this` ist grösser als `other`
  - 0: `this` ist gleich `other`

```
Collections.sort(List<T> list, Comparator<T> comparator
```

```
interface Comparator<T> {
  int compare(T first, T second);
}
```

- Passen auf Funktionsschnittstellen
  - Interface mit genau einer abstrakten Methode
  - Methode hat passende Signatur und Rückgabetyp

```
interface Comparator<T> {
  int compare(T first, T second);
}
```

Typ-kompatibel

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

Matching

- `allMatch()`, `anyMatch()`, `noneMatch()`
  - Prüfen ob Prädikate auf alle bzw. irgendein bzw kein element zutrifft

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

boolean hasChildren =
  people.stream()
  .anyMatch(p -> p.getAge() < 18);
```

Stream API

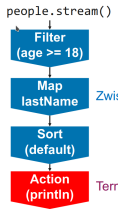
- Deklarative Abfragen von Collections
  - Definiere was gesucht wird, nicht wie!
  - Inspiriert von SQL

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

Terminaloperationen

- `forEach(Consumer)`
  - Pro Element Operation anwenden, meist mit Seiteneffekt
- `forEachOrdered(Consumer)`
  - Erhält die Reihenfolge der Elemente (v.a. bei parallelen Streams)
- `count()`
  - Anzahl Elemente
- `min(), max()`
  - Mit Comparator Argument
- `average(), sum()`
  - Nur bei Int/Long/Double Streams
- `findAny(), findFirst()`
  - Gibt irgendein bzw. erstes Element zurück

- `filter(Predicate)`
  - Rauspicken gemäss Predicate-Funktionsobjekt/Lambda
- `map(Function)`
  - Projizieren gemäss Funktionsobjekt/Lambda
- `mapToInt/mapToLong/mapToDouble(Function)`
  - Projizieren auf int, long, double (primitiver Datentyp)
- `sorted()`
  - Sortieren, mit oder ohne Comparator
- `distinct()`
  - Unterschiedliche Elemente gemäss equals()
- `limit(long n)`
  - Erste n Elemente liefern, danach ignorieren
- `skip(long n)`
  - Erste n Elemente ignorieren, danach weiterliefern



Zwischenoperationen

Übersicht Collectors

- `Collectors.toList()`
  - in Liste abbilden
- `Collectors.toCollection(HashSet::new)`
  - in beliebige Collection abbilden (Konstruktorreferenz)
- `Collectors.groupingBy(key, aggregator)`
  - Gruppierung mit optionalem Aggregator
  - Aggregator: averaging, summing, counting
- `Collectors.joining(" ")`
  - Konkatenieren der Elemente mit Separator " "

Optional-Operationen

- `OptionalDouble.empty()`
  - Inexistenter Wert
- `OptionalDouble.of(double value)`
  - Existenter Wert
- `optionalValue.ifPresent(Consumer)`
  - Falls existent, Consumer aufrufen
- `optionalValue.orElse(double other)`
  - liefert Wert oder, falls inexistent, dann other

Regeln für Lambdas bei Stre

- Keine Interferenz
  - Darf Collection nicht selbst abändern
  - Z.B. `filter(p -> people.add(p))`
- Keine Seiteneffekte
  - Keine Abhängigkeit zu äusseren änderbaren Variablen
  - Z.B. `map(p -> { globalCount++; return p; })`

reduce

```
reduce(
  0,
  (totalAge, person) -> totalAge + person.getAge(),
  (age1, age2) -> age1 + age2
```

automatic parallizing
.parallelStream() allows access to to mutable variables from other sources.