JavaC - CheatSheet

Allgemein

Primitive Datentypen

Typ	Beschreibung	Wertebereich	Wrapper-Klasse
boolean	Boolescher Wert	true, false	Boolean
char	Textzeichen (UTF16)	'a', 'B', '0', 'é' etc.	Character
byte	Ganzzahl (8 Bit)	-128 bis 127	Byte
short	Ganzzahl (16 Bit)	-32'768 bis 32'767	Short
int	Ganzzahl (32 Bit)	-2^{31} bis 2^{31}-1	Integer
long	Ganzzahl (64 Bit)	-2^{63} bis 2^{63}-1, 1L (L Suffix)	Long
float	Gleitkommazahl(32 Bit)	0.1f, 2e4f ($2*10^4$)	Float
double	Gleitkommazahl(64 Bit)	0.1, 2e4	Double

Überlauf bzw. Unterlauf ist in Java definiert. Bei einem Überlauf wird ganz unten weitergezählt, bei einem Unterlauf wird von ganz oben fortgesetzt. Bei Gleitkommazahlen führt ein Über-/Unterlauf zu POSITIVE_INFINITY bzw. NEGATIVE_INFINITY.

Explizite Typkonversation

Nur C-Style Cast: (int)3.5; → 3

Collections

Collections sind Datenstrukturen für Gruppen von Elementen und fordern einen Import aus dem Packet java.util

List

Eine Liste ist eine Folge von Elementen und kann wie folgt definiert werden:

```
ArrayList<Obj> name = new ArrayList<Obj>();
ArrayList<Obj> name = new ArrayList<>();
var name = new Arraylist<Obj>();
List<Obj> name = new ArrayList<Obj>();
List<Obj> name = new ArrayList<>();
```

Iteration mit Enhanced for

Besucht jedes Element in einer Collection:

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

Einige nützliche Operationen mit Listen:

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

stringList.add("one");          // add at the end
stringList.add(0, "two");       // insert at pos 0
//add -> wenn Array voll, umkopieren in doppelt so grosses Array (gibt leere plaetze)
String x = stringList.get(1);   // get at pos 1
stringList.set(0, "three");     // replace at pos 0
stringList.remove("two");       // removes the FIRST "two" in List
stringList.remove(1);           // remove at pos 1
// remove -> alles dahinter wird nach vorne geschoben
stringList.contains("three");   // true -> "three" is in List, else -> falselong
long size = stringList.size();  // get size (number of Elements)
```

Set

Ein Set ist eine Menge von Elementen, in welchem jedes Element genau einmal vorkommt und wird wie folgt verwendent:

```
Set<String> carModels = new HashSet<>();
carModels.add("Ferrari");
carModels.add("Maserati");
carModels.add("Lamborghini");
carModels.add("Ferrari"); // already present (no effect)
if(carModels.contains("Volkswagen")){...}
```


BY-NC-SA

Map

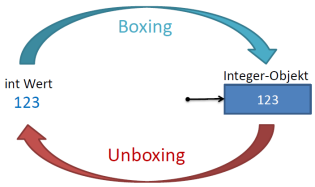
Abbildung Schlüssel → Werte

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

Student st1 = new Student("Andrea", "Meier");
Student st1 = new Student("Bertha", "Mueller");
Student st1 = new Student("Clara", "Schneider");

map.put(20000, st1);     // Bei gleichem Schluessel
map.put(70000, st2);     // wird der Wert ueberschrieben
map.put(13000, st3);     // -> nur ein Key pro Map

Student st = map.get(12345);     // Finden nach Schluessel (sehr effizient)

for(Student s : map.values()){  // .values() gibt Collection aller Werte
    System.out.println(s);
}
```

Wrapper-Klassen

Collections (bzw. alle Generics) nehmen nur Referenzen, welche primitive Datentypen nicht bringen. Um dennoch int's oder double's in Listen zu speichern, gibt es sogenannte Wrapper-Klassen.

```
// Implizites Boxing (Integer.valueOf(123))
Integer wrapper = 123;

// Implizites Unboxing (wrapper.intValue())
int value = wrapper;
```



Vererbung

In Java gibt es nur Einfachvererbung, sprich jede Klasse hat maximal eine Basisklasse. Die Subklasse bietet alles was Superklasse bietet und eventuell mehr.

Root Class Object

Jede Klasse erbt automatisch (direkt oder indirekt) von der obersten Basisklasse `Object`. Folgend einige der wichtigsten Methoden der Klasse `Object`:

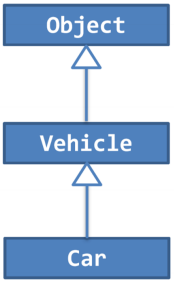
```
public String toString()
public boolean equals(Object obj)
public int hashCode()
```

Typ-Polymorphismus

Ein Objekt hat nicht nur den Typ seiner Klasse, sondern auch die Typen seiner Superklassen. Beispiel:

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

TO DO: evtl. @Override??



Konstruktor bei Vererbung

Das erste Statement in jedem Konstruktor ist der Aufruf des Basis-Konstruktors mittels `super()`. Dieser wird implizit vom Compiler eingefügt, wenn ein Default-Konstruktor(ohne Parameter) existiert, ansonsten muss er an **erster** Stelle im Konstruktor explizit aufgerufen werden.

```
public class Vehicle{
    private int speed;

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

public class Car extends Vehicle{
    private boolean[] isDoorOpen;

    public Car(int speed, int nofDoors){
        super(speed);   // expliziter Aufruf
        isDoorOpen = new boolean[nofDoors];
    }
}
```


BY-NC-SA

Schnittstellen(Interfaces)

Ein Interface beschreibt die öffentlich nutzbare Funktionalität einer Klasse. Während aber Klassen instanziierbar sind, sind Interfaces lediglich als deklarierbare Typen verwendbar. **TO DO: evtl beispiel???**

Spezifikation

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

Methoden einer Schnittstelle sind implizit **public** und **abstract**, sprich diese modifier können weggelassen werden. Andere Modifier sind ungültig.

Kontstanten in Schnittstellen

```
public interface Vehicle{
    int HIGHWAY_MIN_SPEED = 60;
    int HIGHWAY_MAX_SPEED = 120;
    ...
}
```

Vermeintliche Variablen sind in Schnittstellen Konstanten, welche bei gutem Stil in Grossbuchstaben geschrieben werden. Diese sind implizit **public**, **static** und **final**. **TO DO: default Methoden!!!**

Implementation

```
class RegularCar implements Vehicle{
    ...
    @Override
    public void drive(){
        System.out.println("driving...");
    }

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

Vererbungen und Mehrfachimplementation

```
// Vererbung:
interface I1 extends I2 {...}
// Mehrfachimplementation:
class C implements I1, I2 {...}
// Kombination:
class C1 extends C2 implements I1, I2 {...}
```

Kollisionen bei Mehrfach-Implementation

gleiche Methode: Methode wird nur einmal implementiert → kein Problem
gleichnamige Konstanten: Muss explizit auf Konstante zugegriffen werden → `Vehicle.HIGHWAY_MIN_SPEED`

Variadische Funktionen mit Varargs

- Erlaubt beliebige Anzahl Parameter
- Nur am Schluss der Parameterliste erlaubt
- Compiler generiert ein Array

```
s = sum();
s = sum(1);
s = sum(1, 2, 3);
```

```
static int sum(int... numbers){
    int sum = 0;
    for(int i = 0 ; i < numbers.length ; i++){
        sum += numbers[i];
    }
    return sum;
}
```


BY-NC-SA

Spezielle Grundfunktionen

Grundfunktionen sind Funktionen, welche in jedem `Object()` vorhanden sind und nach bedarf überschrieben werden können.

Gleichheit

`equals()` ist standardmässig nur **Referenzvergleich**. Für einen inhaltlichen Vergleich muss `equals()` überschrieben werden. **TO DO: sollte das immer der Fall sein?** Bei der `String` Klasse ist es bereits implementiert, bei Arrays aber nicht!

```
class Person {
    private String firstName, lastName;
    ...
    @Override
    public boolean equals(Object obj){
        if(obj == null){
            return false;
        }
        if(getClass() != obj.getClass()){   //Pruefe ob von untersch. Klassen erzeugt
            return false;
        }
        Person other = (Person)obj;
        return Objects.equals(firstName, other.firstName) && Objects.equals(lastName,
    other.lastName);
```

Hash-Code

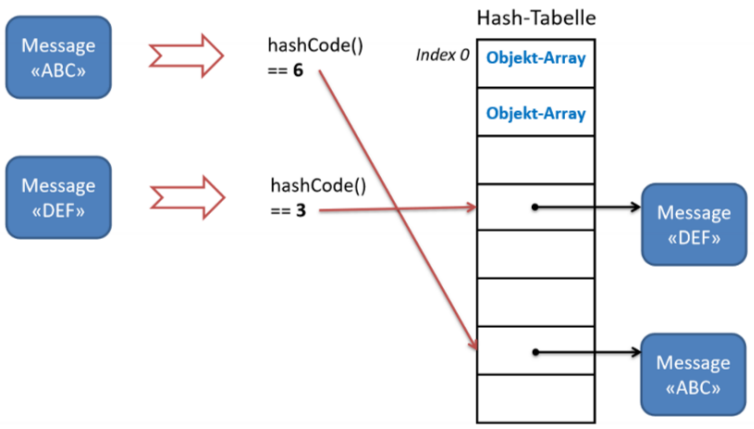
- Sobald `equals()` überschrieben wird, muss auch `hashCode()` überschrieben werden.
- `hash-Code()` berechnet aus Objektinhalt den Hash-Code.
- Hashcodes müssen immer gleich sein, wenn Objekte equals sind.
- Umkehrung muss nicht umbedingt gelten!
- Mittels **Hashing** kann ein Element effizient gefunden werden.
  - **Streuspeicher**: Elemente werden auf ein Array(Hash-Tabelle) verstreut, mit 'Hash-Code' als jeweiliger Index.

Eigene Hashfunktion:

```
@Override
public int hashCode(){
    return firstName.hashCode()
        + 31 * lastName.hashCode();
```

Die Erstellung von `hashCode()` und `equals()` wird üblicherweise der IDE überlassen.

Beispiel:



BY-NC-SA

Generics

Generics sind zu vergleichen mit Templates in C++. In Java könnte es auch gelöst werden, indem ein Element mit dem Typ `Object` entgegen genommen wird, dann ist aber ein Type-Cast notwendig wenn das Objekt, ohne Typinformationsverluste, wieder zurückgenommen werden will.

Generische Klasse

<Typ-Parameter>: Platzhalter für unbekannten Typ

```
class Stack<T> {
    ...
    ...
    ...
}
```

Einsatz mit Typ-Argument

<Typ-Argument>: muss angegeben werden

```
Stack<String> stack1;
Stack<Integer> stack2;
Stack<Person> stack3;
Stack<double[]> stack4;
Stack<Object> stack5;
```

Generische Interfaces

```
interface Iterator<E> {
    boolean hasNext();
    E next();
}

interface Iterable<T> {
    Iterator<T> iterator();
}
```

Generische Methoden

```
public <E> Stack<E> multiPush(E value, int times){
    var result = new Stack<E>();
    for(int i = 0; i < times; i++){
        result.push(value);
    }

    return result;
}
```

Type-Bound

- extends-Klausel bei Typ-Parameter
- Typ-Argument muss Subtyp von Graphic sein

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

Type Inference

Generische Methoden ohne Typ-Argument aufrufen → wird automatisch von Methodenargument erkannt.

```
var stack1 = multiPush("Hello", 100);

var stack2 = multiPush(3.141, 3);
```

Exceptions

Wenn eine Exception nirgendwo abgefangen wird und somit `main()` mit dieser Exception zurückkehrt, behandelt die JVM das mit einem Programmabbruch.

Exception auslösen

Jede Funktion die entweder eine Exception auslöst oder eine allfällige Exception nicht behandelt, muss alle potentiellen Exceptions deklarieren, die der Aufrufer erhalten könnte.(ausnahme sind unchecked Exceptions)

```
String clip(String s) throws Exception{
    if(s == null){
        throw new Exception("String null");
    }
    if(s.length() < 2){
        throw new Exception("String too short");
    }
    return s.substring(1, s.length()-1);
}
```

Exceptions behandeln

```
void test(){
    try{
        String s = null;
        String c = clip(s);
        System.out.println(c);
    } catch(Exception e){
        System.out.println("
   Error");
    }
}
```


BY-NC-SA

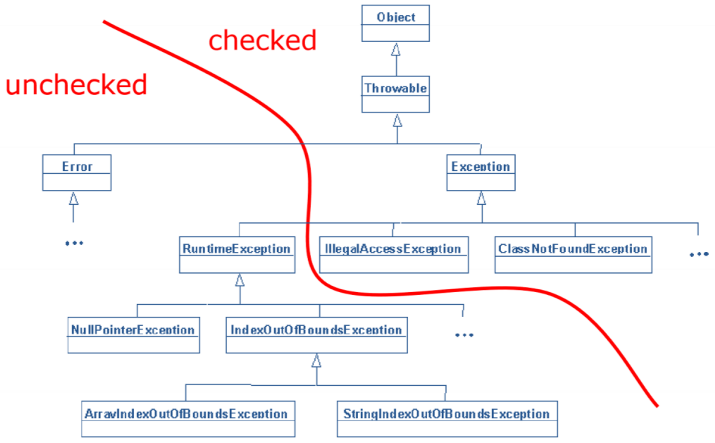
Exception Klassen

Checked Exceptions:

- Exception muss behandelt werden, ODER
- throws-Deklaration in Methodenkopf.
- Vom Compiler geprüft

Unchecked Exceptions:

- Keine throws-Deklaration und keine Behandlung nötig
- kann aber behandelt werden, falls gewünscht
- `RuntimeExceptions` und `Error` sowie Unterklasen davon
- Nicht vom Compiler geprüft



Benutzerdefinierte Exceptions

```
class StringClipException extends Exception{    // muss von einer Exception erben!!!
    StringClipException(){}

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

Mehrere Catch-Klauseln

- Passender Catch wird von oben nach unten gesucht
- **nur** der erste kompatible catch ausgeführt
- kein passender Catch → Exc. bleibt unbehandelt

```
try{
    ...regular code...
} catch (exceptionType1 e){
    ...error handling...
} catch (exception Type2 e){
    ...error handling...
}
```

- Wenn mehrere Exception Typen gleich behandelt werden sollen, ist ein sogenannter Multi-Catch möglich:

```
try{
    ...regular code...
} catch (exceptionType1 | exceptionType2 e){
    ...error handling...
}
```

finally-Block

Optionaler finally Block am Ende des try-Konstrukts, welcher in jedem Fall durchlaufen wird.

```
try{
    String s = ...;
    String c = clip(s);
}catch(StringClipException e){
    System.out.println("cannot clip");
}finally{
    System.out.println("finished");
}
```

oftmals auch ohne `catch`-Block:

```
try{
    ...
    ...// work with s
}finally{
    s.close();
}
```

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

→ Dieser Block ist äquivalent zum Block **unten**, wobei der `finally`-Block automatisch vom Compiler generiert wird und um einiges komplexer ist. **Wichtig** ist dabei, dass in der Klasse `Scanner` das Interface `AutoCloseable` implementiert ist, damit dort die `close()`-Funktion sicher existiert. Mit `;`-getrennt können mehrere Variabeln definiert werden.

```
Scanner s = new Scanner(System.in);
try{
    ...//work with s
}finally{
    if(s != null) {s.close();}
}
```


BY-NC-SA

Lambdas

Höherwertige Funktionen

- Funktionen, welche wiederum Funktionen als Parameter erwarten oder zurückgeben
- Funktionen werden wie Werte behandelt. (Referenz auf Funktion)
- Parametertyp ist ein Interface mit **genau einer** abstrakten Methode (**functional Interface**)
- übergebene Methode muss **typ-kompatibel** mit der Methode im Interface sein, sprich gleiche Signatur und Rückgabetyp

Methodenreferenz

- Referenz auf eine Methode (noch kein Aufruf!)
- Methode wird wie ein Objekt behandelt
- Methodenreferenzen haben keinen eigenen Typ (nicht wie in C++)

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

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

Syntax einiger nützlichen Methodenreferenzen:

```
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
```

Lambdas

Ein Lambda ist eine Referenz auf eine **anonyme** Methode. Der Syntax sieht wie folgt aus:

```
(Parameterliste) -> { Body }
```

Beispiel:

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

Dabei sind die geschweiften Klammern im Body optional, werden nur bei einem syntaktischer Ausdruck (Expression) gebraucht.

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

Bei nur einem Übergabeparameter sind sogar die runden Klammern überflüssig.

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

letzterer Ausdruck ist Analog zu **`this::isAdult`** mit:

```
boolean isAdult(Person p){
    return p.getAge() >= 18;
}
```

Lambdas sind auch ohne Parameterliste möglich, was dann wie folgt aussieht:

```
() -> System.out.println("Do nothing!");
```

Das benützen eines Lambdas und das Functional Inteface sieht dann wie folgt aus:

```
public void testAll(Predicate<T> criterion) {
    List<Integer> integerList =
                    Arrays.asList(10, 20, 18, 21);
    for(int elem: integerList){
        if(criterion.test(elem)){
            System.out.println(elem + '\n');
        }
    }
}
```

```
@FunctionalInterface
public interface Predicate<T> {
    boolean test(T element);
}
```

`@FunctionalInterface` ist nicht nötig, aber kann zur Information hingeschrieben werden. Für den Compiler ändert sicher aber nichts.

BY-NC-SA

Stream API

- Definiere was gesucht ist, nicht wie
- Framework zur Sequenz von Collection-Operationen (häufig höherwertige Funktionen mit Lambdas als Argumente)
- Regeln:
  - keine Interferenz (Darf Collection nicht selbst abändern), z.b.: `filter(p -> people.add(p))`
  - keine Seiteneffekte (keine Abh. zu äusseren änderbaren Variablen), z.b.: `map(p -> globalCount++; return p; )`
- braucht immer eine Terminaloperation am Schluss.

Beginn

Begonnen wird immer mit dem Aufruf von `stream()` bei Collection. Beispiel:

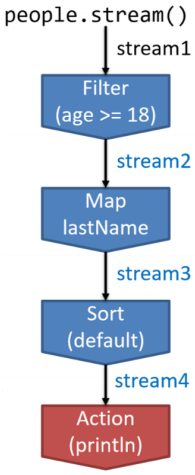
```
people.stream()
```

Method Chaining / fluent Interface

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

Ist äquivalent zu ↓

```
Stream<Person> stream1 = people.stream();
Stream<Person> stream2 = stream1.filter(p -> p.getAge() >= 18);
Stream<Person> stream3 = stream2.map(p -> p.getLastName());
Stream<Person> stream4 = stream3.sorted();
stream4.forEeach(s -> System.out.println(s));
```



Zwischenoperationen

```
filter(Predicate)   // Rauspicken gemaess Predicate-Funktionsobjekt/Lambda
map(Function)       // Projizieren gemaess Funktionsobjekt/Lambda
mapToInt(Function)  // Projizieren auf int, long double (primitiver Datentyp)
sorted()            // sortieren, mit oder ohne Comparator
distinct()          // keine gleichen Elemente gemaessss Equals()
limit(long n)       // erste n Elemente liefern, danach ignorieren
skip(long n)        // erste n Elemente ignorieren, danach weiterliefern
```

Terminaloperationen

```
forEach(Consumer)   // Pro Element Operation anwenden, meist mit Seiteneffekt
count()             // Anzahl Element
min(), max()        // Mit Comparator Argument
average(), sum()    // Nur bei Int/Long/Double Streams
findAny(), findFirst()  // Gibt irgendein bzw. erstes Element zurueck
```

Endliche Quellen

```
IntStream.range(1, 100)
Stream.of(2, 3, 5, 7, 13)
Stream.empty()  // Testzwecke
Collection.stream()
Stream.concat(stream1, stream2)
```

Unendliche Quellen

```
generate(random::nextInt)
iterate(0, i -> i+1)
```

Dies funktioniert, weil ein Element im Strom erst bereitgestellt wird, wenn der Nachfolger es wirklich anfordert. Als zweites Element kann zum Beispiel `.limit(1000)` folgen.

BY-NC-SA

Rückumwandlungen

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

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

Übersicht Collectors

```
Collectors.toList()     // in Liste abbilden
Collectors.toCollection(TreeSet::new)   // in beliebige Collection abbilden
                                        //          (Konstruktorreferenz)
Collectors.groupingBy(key, aggregator)  // Gruppierung mit optionalem Aggregator
                                        // 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

Byte-Streams

```
var in = new FileInputStream("myFile.data");    // bestehende Datei zum lesen oeffnen
int value = in.read();
while (value >= 0) {        // -1 => end of file
    byte b = (byte)value;   // gelesenes Byte (wenn positiv)
    // work with b
    value = in.read();
}
in.close();
```

```
var out = new FileOutputStream("test.data");// Datei neu anlegen bzw. ueberschreiben
while (...) {
    byte b = ...;
    out.write(b);
}
out.close();    // Wichtig! Herunterschreiben ("Flush") des Rests beim Schliessen

new FileOutputStream("test.data", true)     // Anhaengen an File, falls existiert
```

Besondere Stream-Methoden

```
int read(byte[] b, int offset, int length)
// lese length Bytes in Array b ab Index
   offset
```

```
void write(byte[] b, int offset, int length)
void flush()    // implizit bei close()
```

Standard Input/Output

- System.in → System.in
- System.out & System.err → PrintStream
- PrintsStream ist Subklasse von OutputStream

BY-NC-SA

Character-Streams

```
try (var reader = new FileReader("quotes.txt")) {   // Systemabhaengige Charcter Set
    int value = reader.read();
    while (value >= 0) {    // -1 => end of line
        char c = (char)value;   // 16-bit char
        // use character
        value = reader.read();
    }
}
```

```
try (var writer = new FileWriter("test.txt", true)) {   // true => Append
    writer.write("Hello!");     // String schreiben
    writer.wrtie('\n');         // Einzelner char schreiben
}
```

Einfachster Text-Datei Zugriff

```
// Ganze Text-Datei einlesen
List<String> lines = Files.readAllLines(Path.of("in.txt"), StandardCharsets.UTF_8);

// Alle Zeilen als StreamAPI:
Stream<String> lines = Files.lines(Path.of("in.txt"), StandardCharsets.UTF_8);

// Ganze Text-Datei schreiben
Files.write(Path.of("out.txt"), lines, StandardCharsets.UTF_8);
```

Enums

Definition

Eigener Datentyp mit endlichem Wertebereich.

```
public enum Weekday {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY,
    FRIDAY, SATURDAY, SUNDAY
}
// Java-Namenskonvention:   Enum-Werte nur in
//                          Grossbuchstaben
```

Verwendung

```
Weekday currentDay;

currentDay = Weekday.WEDNESDAY;
if(currentDay == Weekday.SUNDAY){...}


currentDay = null;
```

Enums mit Switch-Anweisung

```
void getActivity(Weekday day) {
    switch (day) {
        // Labels ohne Enum-Typ
        case MONDAY:
        case TUESDAY:
        case WEDNESDAY:
            return "consulting";
        case THURSDAY:
        case FRIDAY:
            return "backoffice";
        default:
            return "weekend"
    }
}
```

Logik in Enums

```
public enum Weekday {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY,
    FRIDAY, SATURDAY, SUNDAY;

    public boolean isWeekend() {
        return this == SATURDAY || this == SUNDAY;
    }
}

Weekday currentDay = ...;
if(currentDay.isWeekend()) {...}
```


BY-NC-SA