Cheatsheet

Links

- JUnit 5: https://junit.org/junit5/docs/current/user-guide/
- Java: https://docs.oracle.com/en/java/javase/15/docs/api/index.html

Strategie

1. Multiple Choice
2. Vererbung
3. Unit-Tests
4. Kleine Programmieraufgabe
5. Klassendiagramm
6. Grosse Programmieraufgabe

Refactoring-Aufgabe?

Datentypen

primitive Typen

- direkt in Variablen gespeichert
- verfügen nicht über Methoden

Typname	Beschreibung	Beispielkonstanten		
Ganze Zahlen:				
`byte`	ganze Zahl in 8 Bit	`24`	`-2`	
`short`	ganze Zahl in 16 Bit	`137`	`-119`	
`int`	ganze Zahl in 32 Bit	`5409`	`-2003`	
`long`	ganze Zahl in 64 Bit	`423266353`	`55L`	
Gleitpunktzahlen:				
`float`	einfache Fließkommazahl (32 Bit)	`43.889F`		
`double`	doppelte Fließkommazahl (64 Bit)	`42.63`	`2.4e5`	
Andere Typen:				
`char`	einzelnes Zeichen (16 Bit)	`'m'`	`'?'`	`'\u00F6'`
`boolean`	ein boolescher Wert (wahr oder falsch)	`true`	`false`	

Typ	Minimum	Maximum
`byte`	`-128`	`127`
`short`	`-32768`	`32767`
`int`	`-2147483648`	`2147483647`
`long`	`-9223372036854775808`	`9223372036854775807`
	Positives Minimum	**Positives Maximum**
`float`	`1.4e-45`	`3.4028235e38`
`double`	`4.9e-324`	`1.7976931348623157e308`

Objekttypen

- Referenzen auf Objekte
- nur die Referenz wird kopiert

Methoden

Sondierende Methoden (Inspektoren) ⇒ liefern Informationen über den aktuellen Zustand eines Objekts

Verändernde Methoden (Mutatoren) ⇒ ändern den internen Zustand eines Objekts

Konstruktor vs. Methode

Methoden besitzen einen Rückgabetyp (auch wenn er nur void ist), Konstruktoren aber nicht

Überladen

Eine Klasse kann mehr als einen Konstruktor oder mehr als eine Methode mit dem gleichen Namen enthalten, solange jede von ihnen einen unterscheidbaren Satz von Parametertypen definiert.

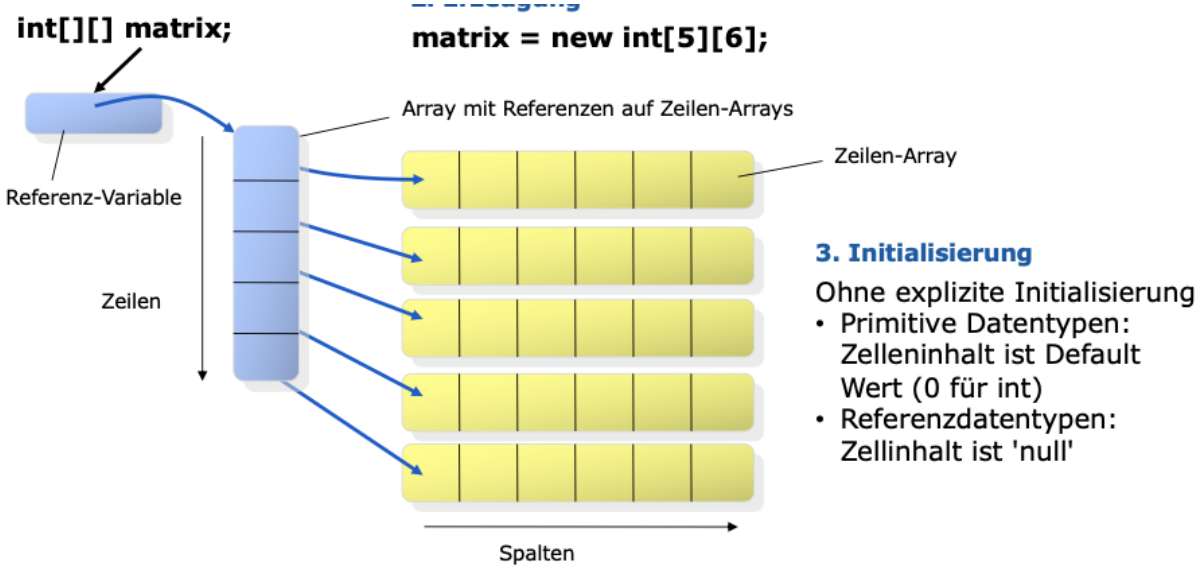
Zugriffsmodifikatoren

Modifier	Class	Package	Subclass	Global
Public	Yes	Yes	Yes	Yes
Protected	Yes	Yes	Yes	No
Default	Yes	Yes	No	No
Private	Yes	No	No	No

Arrays

mehrdimensionale Arrays

⇒ Dimension (Anzahl `[]`) entspricht Verschachtelungstiefe




```
int[][] matrix = new int[9][10];
int zaehler = 1;

for (int zeile = 0; zeile < matrix.length; zeile++) {
  for(int spalte = 0; spalte < matrix[zeile].length; spalte++) {
    matrix[zeile][spalte] = zaehler++;
  }
}
```

Iterieren

```
// for-each
for(Student eintrag : studenten) {
  System.out.println(eintrag.gibName());
}

// while mit Laufindex
int index = 0;
while(index < studenten.size()) {
```

```
  System.out.println(studenten.get(index).gibName());
  index++;
}

// while mit Iterator
Iterator<Student> iter = studenten.iterator();
while(iter.hasNext()) {
  System.out.println(iter.next().gibName());
}

//Stream
studenten.forEach(student -> System.out.println(student.gibName()));
```

💡 `remove` auf ein `ArrayList` ist nicht während der Iteration möglich! (⇒ Iterator verwenden) ⇒ ConcurrentModificationException

Arrays vs ArrayList

- Arrays: Speichern einer fixen und bei Erzeugung der Sammlung bekannten Anzahl von «Dingen»
- ArrayList: Anzahl benötigte Speicherplätze unbekannt

Pseudozufallszahlen

- `Random` erzeugt Pseudozufallszahlen

Ausgehend von einem Anfangswert (Seed), wird gemäss einer Berechnungsvorschrift die nächste Zahl eindeutig bestimmt
⇒ wenn man den Anfangswert kennt (Systemzeit z.B. kann man die Zufallssequenz berechnen)

```
Random random = new Random();
int max = 10;
int min = 5;

// [min, max]
random.nextInt((max - min) + 1) + min;

// [min, max)
random.nextInt((max - min)) + min;

// random item from List
List<String> names = List.of("Max", "Peter", "Olaf");
names.get(random.nextInt(names.size()));
```

Regex

Metasymbol	Beispiel	Bedeutung	Menge der gültigen Literale
*	ax*b	0 oder mehrere x	ab, axb, axxb, axxxb,...
+	ax+b	1 oder mehrere x	axb, axxb, axxxb, ...
?	ax?b	x optional	ab, axb
|	a|b	a oder b	a, b
()	x(a|b)x	Gruppierung	xax, xbx
.	a.b	Ein beliebiges Zeichen	aab, acb, aZb, a[b, ...
[ ]	[abc]x	1 Zeichen aus einer Klasse	ax, bx, cx
[-]	[a-h]	Zeichenbereich	a, b, c, ..., h

Bibliotheksklassen

Schnittstelle

Eine klare Beschreibung der Schnittstelle einer Klasse macht das Studieren der Implementation der Klasse, resp. des Quellcodes überflüssig. Alle zur korrekten Benutzung der Klasse notwendigen Informationen sind dokumentiert. Dies ist folglich ein Mittel, um von der Implementierung der Klasse zu abstrahieren.

Methodenschnittstelle = **Signatur** der Methode + Kommentar

Signatur = [Zugriffsmodifikator, Ergebnistyp, Methodenname, Parameterliste]

Geheimnisprinzip

⇒ besagt, dass die internen Details der Implementierung einer Klasse vor anderen Klassen verborgen sein sollten. Dies unterstützt eine bessere Modularisierung von Anwendungen

1. Keine Kenntnis über Interna benötigen
2. Keine Kenntnis über Interna erlauben (aus der Sicht einer Klasse)

⇒ lose Kopplung

▼ Beispiel

```
public class Auto {
	public String kennzeichen;
	...
	public boolean setAutokennzeichen(String autoKennzeichen) {
		if (istGueltigesKennzeichen(autoKennzeichen)) {
			kennzeichen = autoKennzeichen;
			return true;
		}
		return false;
	}
	...
}
```

Welches Geheimnisprinzip wird verletzt? Was ist das Problem, welches daraus entsteht? Wie kann es korrigiert werden?

„Keine Kenntnis über Interna erlauben". Es ist möglich auch ungültige Autokennzeichen zu speichern. Die Instanzvariable `kennezeichen` muss `private` deklariert werden.

Klassenentwurf

Kopplung

> 💡 Grad der Abhängigkeiten zwischen Klassen ⇒ “tell don’t ask”

⇒ in einer eng gekoppelten Klassenstruktur kann eine Änderung an einer Klasse viele Änderungen an anderen Klassen nach sich ziehen

Öffentliche Datenfelder

Öffentliche Datenfelder erhöhen das Potenzial für starke Kopplung weil dann bei einer Änderung der Art, wie die Ausgangsinformationen gespeichert werden, alle anderen Klassen, welche auf diese Datenfelder zugreifen, angepasst werden müssen. Bei einem Zugriff über eine Methode, spielt die Art, wie diese Information in der Klasse Raum gespeichert wird, keine Rolle.

Entwurf nach Zuständigkeiten

Jede Klasse ist für den Umgang mit ihren Daten verantwortlich. Methoden, die Daten verarbeiten sollten deshalb zu der Klasse gehören, die für diese Daten verantwortlich sind

implizite Kopplung

> 💡 wenn sich eine Klasse auf interne Informationen einer anderen Klasse abstützt

⇒ Fehler kann unentdeckt bleiben

Kohäsion

⇒ wie gut eine Programmeinheit eine logische Aufgabe oder Einheit abbildet

In einem System mit hoher Kohäsion ist jede Programmeinheit (eine Methode, eine Klasse oder ein Modul) verantwortlich für genau eine wohldefinierte Aufgabe oder Einheit.

⇒ Zweck: Wiederverwendung

Kapselung

> 💡 Informationen über die Implementierung verbergen

⇒ nur Informationen über das Was einer Klasse (was sie leistet) nach außen sichtbar sein sollten, nicht aber das Wie (ihre Realisierung)

⇒ Zweck: Implementierung ändern, ohne das andere Klassen davon betroffen sind

- public Datenfelder brechen die Kapselung
- Saubere Kapselung reduziert die Kopplung

Gesetz von Demeter

💡 Objekte sollen nur mit Objekten in ihrer unmittelbarer Umgebung kommunizieren

⇒ Verringerung der Kopplung = bessere Wartbarkeit

⇒ “Sprich nur zu Deinen nächsten Freunden”

Testing

Positives / Negatives Testen

⇒ Positiv: Testen der Fälle, die funktionieren sollten.
⇒ Negativ: Testen der Fälle, die fehlschlagen sollten

Regressionstests

⇒ Test, der bereits erfolgreich gelaufen ist und wiederholt wird, um eine neue Version der Software zu überprüfen.

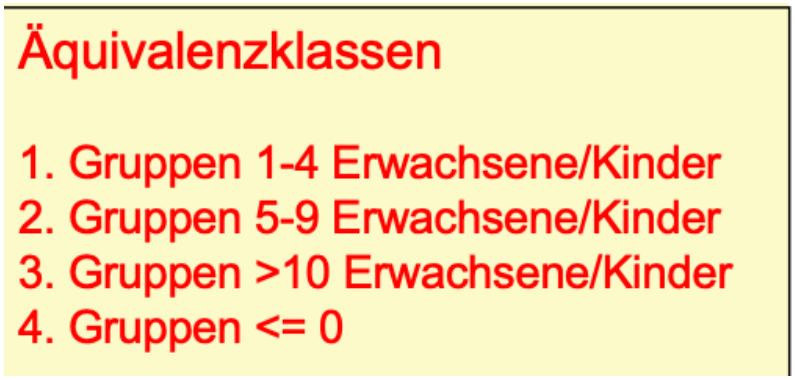
Äquivalenzklassen

⇒ Gruppierung von Werten mit dem gleichen Verhalten

▼ Beispiel

In einer vorherigen Vorlesung haben wir den Eintrittspreis für ein Erlebnisbad berechnet, der Preis pro Person sei wie folgt:

- 12.-- bei Gruppen von 1 – 4
- 10.50 bei Gruppen von 5 – 9
- 9.50 bei grösseren Gruppen
- Kinder erhalten 50% Rabatt



▼ Beispiel mit einem 2d Array

```java
public static int berechneMaxWert(int[][] umsatz) {
  int max = 0;
  for (int i = 0; i < umsatz.length; i++) {
    for (int j = 0; j < umsatz[i].length; j++) {
      if (umsatz[i][j] > max) {
        max = umsatz[i][j];
      }
    }
  }
  return max;
}
```

```
Äquivalenzklassen

1. Gültige Werte (alle >= 0)

2. Gültige Werte (beliebige, positiv und negativ)

3. Gültige Werte (alle < 0)

4. Leeres Array

5. Null-Array
```

⇒ aus den Äquivalenzklassen folgen dann die Testfälle

⇒ beim testen Grenzwerte berücksichtigen (`Integer.MIN_VALUE` , ...)

Exceptions

- Root class `Throwable`

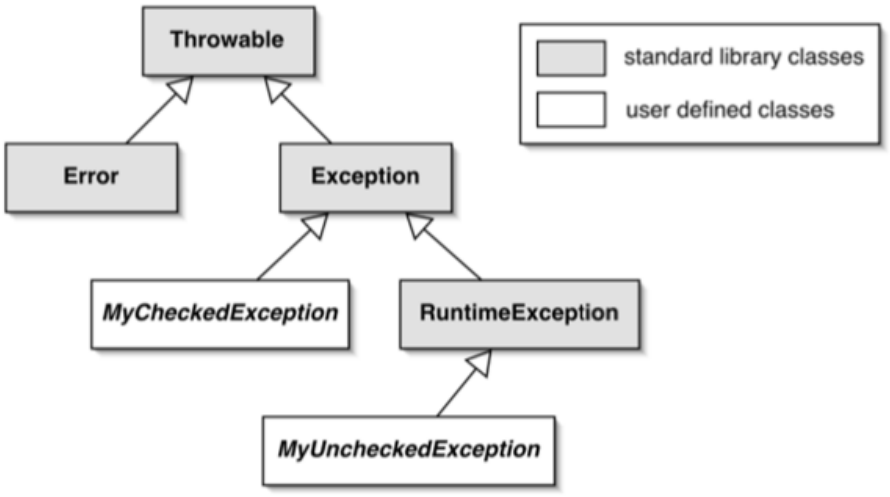
2 main subclasses:

- Error (severe system errors)
- Exception
  - checked (by compiler):
    - subclass of `Exception` ⇒ must be handled
    - anticipated failures
  - unchecked: `RuntimeException` handling is optional, do not have to be declared
    - unlikely recovery

Own exceptions

```
class NichtParkierbarException extends RuntimeException {
  public NichtParkierbarException() {}

  public NichtParkierbarException(String message) {
    super(message);
  }
}
```



Vererbung

Subklassen

- Konstruktoren der Subklasse können nicht auf die privaten Instanzvariablen der Superklasse zugreifen

`instanceof` überprüft ob Objekt eine Instanz von einer Klasse oder deren Subklassen ist

Casting

> 💡 Umwandlung eines Typs in einen anderen Typ

- Typen müssen in direkter Linie verwandt sein

⇒ Statische Typ muss entweder dem dynamischen Typ entsprechen oder im Vererbungsbaum weiter oben angesiedelt sein. ⇒ sonst passt das Objekt nicht rein!

Upcast

⇒ von einer Subklasse zu einer Superklasse
⇒ Passiert automatisch (implizit) im Vererbungsbaum aufwärts

```
Hund hund = new Hund();
Tier tier = hund; //-> unnötig: Tier tier = (Tier) hund;
```

Downcast

⇒ von einer Superklasse zu einer Subklasse
⇒ muss explizit gecastet werden

```
Tier tier = new Hund();
((Hund) tier).bellen() // -> die Methode bellen ist auf dem statischen Typ Tier nicht vorhanden
```

Methodensuche

- Start beim dynamischen Typ
- Dem Vererbungsbaum aufwärts folgend die erste Implementation verwenden.
- Ist die Methode im statischen Typ nicht vorhanden, reklamiert der Compiler und verlangt einen Typecast.
  - Passt das Objekt zur Laufzeit nicht zum Typecast kommt ein Laufzeitfehler.

equals

```
public boolean equals(Object obj) {
  if(this == obj) {
    return true; // Referenzgleichheit
  }
  if(!(obj instanceof Student)) {
    return false; // nicht derselbe Typ
  }
  // Zugriff auf die Datenfelder des anderen Studenten
  Student anderer = (Student) obj;
  return name.equals(anderer.name) && matrikelnummer.equals(anderer.matrikelnummer) && scheine == anderer.scheine;
}
```

💡 Immer, wenn die **equals-**Methode überschrieben wird, muss die **hashCode-**Methode ebenfalls überschrieben werden!

hashCode

⇒ um effizientes Suchen und Einfügen in `HashMap` oder `HashSet` z.B. zu ermöglichen

```
public int hashCode() {
  int ergebnis = 17;
  if (marke == null) {
    ergebnis = 37 * ergebnis;
  } else {
    ergebnis = 37 * ergebnis + marke.hashCode();
  }
  ergebnis = 37 * ergebnis + leistung;
  return ergebnis;
}
```

oder:

```
@Override
public int hashCode() {
  return Objects.hash(marke, leistung);
}
```

Abstraktion

💡 Abstraktion ist die Fähigkeit, Details von Bestandteilen zu ignorieren, um den Fokus der Betrachtung auf eine höhere Ebene lenken zu können.

Vorteile

- Vermeidung von Code-Duplizierung
- bessere Wart- und Erweiterbarkeit.

abstrakte Klassen

konkrete Klassen = Klassen die nicht abstrakt sind

- nur abstrakte Klassen dürfen abstrakte Methoden definieren
- eine konkrete Klasse muss alle abstrakten Methoden implementieren, sonst ist sie selber abstrakt

Interface

⇒ Spezifikation eines Typs

- keine Instanzfelder, Konstruktoren

⇒ Trennung der Spezifikation / Implementierung

implementierende Klassen muss alle Methoden implementieren! 🙂

Abstrakte Klassen vs. Interfaces

Interfaces lassen multiple Vererbung zu, die implementierende Klasse hat somit noch die Möglichkeit von einer Klasse zu erben.

Objektvergleiche

comparable vs Comparator (Interfaces) ⇒ falls Objekte keine natürliche Ordnung haben

comparable

- auf dem Objekt, Klasse muss verändert werden

```
Collections.sort(List)
```

```
class Student implements Comparable<Student> {
  int rollno;
  String name;
  int age;

  public int compareTo(Student st){
    if(age==st.age)
      return 0;
    else if(age>st.age)
      return 1;
    else
      return -1;
  }
}
```

Comparator

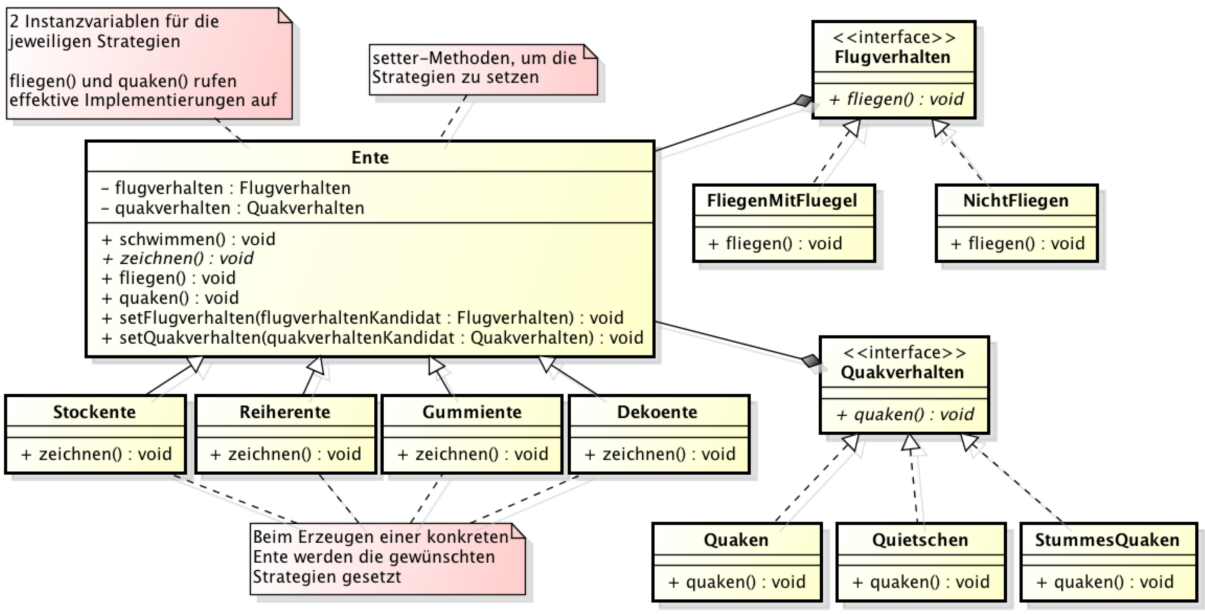
- eigene Klasse für den Vergleich

```
Collections.sort(List, Comparator);
```

```
class AgeComparator implements Comparator<Student>{
  public int compare(Student s1,Student s2){
    if(s1.age==s2.age)
    return 0;
    else if(s1.age>s2.age)
    return 1;
    else
    return -1;
  }
}
```

Strategy pattern

⇒ Klassen die Strategien definieren




```
public class Gummiente extends Ente {
  public Gummiente() {
    setFlugverhalten(new NichtFliegen());
    setQuakverhalten(new Quietschen());
  }

  @Override
  void zeichnen() {
    System.out.println("Ich sehe aus wie eine Gummiente");
  }
}
```

```
public abstract class Ente {
  private Flugverhalten flugverhalten;
  private Quakverhalten quakverhalten;
```

```
  public void setFlugverhalten (Flugverhalten flugverhaltenKandidat) {
    flugverhalten = flugverhaltenKandidat;
  }

  public void setQuakverhalten(Quakverhalten quakverhaltenKandidat) {
    quakverhalten = quakverhaltenKandidat;
  }

  abstract void zeichnen();

  public void fliegen() {
    flugverhalten.fliegen();
  }

  public void quaken() {
    quakverhalten.quaken();
  }

  public void schwimmen() {
    System.out.println("Alle Enten schwimmen, sogar Modellenten!");
  }
}
```