# Git

1. **git add abc**  fügt die Datei abc (mit all ihren Änderungen seit dem letzten Commit) zur Staging Area hinzu, d. h. alle Änderungen in abc werden für den nächsten Commit vorgemerktNotiz: Reihenfolge letztes Jahr: 1. fügt die Datei abc (mit all ihren Änderungen seit dem letzten Commit) zur Staging Area hinzu, d. h. alle Änderungen in abc werden für den nächsten Commit vorgemerkt  2. erstellt einen neuen Commit auf dem aktuellen Branch mit allen Änderungen, die gerade in der Staging Area sind; zum Schreiben der Commit-Nachricht wird der für git konfigurierte Standard-Editor geöffnet  3. Abkürzung zum Fetchen und Mergen des passenden Remote-Branches  4. kopiert alle Commits des aktuellen Branches auf den konfigurierten Standard-Remote (typischerweise origin)  5. im aktuellen Verzeichnis eine Kopie des Repositories, das im Ordner /tmp/abc/.git liegt, anlegen und den Standard-Branch auschecken  6. alle Commits vom Standard-Remote in das lokale git-Repository kopieren, aber ohne die Änderungen in die lokalen Branches zu kopieren  7. die Änderungen vom Branch abc in den aktuellen Branch integrieren (durch Erstellen eines Merge-Commits)  8. auf den Branch abc wechseln  9. einen neuen Branch abc anlegen  10. auf dem aktuellen Branch einen neuen Commit erzeugen, der dieselben Änderungen beinhaltet wie der Commit abc  11. das letzte git commit auf dem aktuellen Branch rückgängig machen, aber ohne die Änderungen des Commits zu verwerfen (d. h. anschließend gibt es nicht committete Änderungern)  12. die Commits des aktuellen Branches X neu auf den Stand von abc anwenden und den Branch X auf den letzten dieser Commits zeigen lassen  13. einen neuen Commits erzeugen, der die Änderungen von Commit abc rückgängig macht  14. zeit an, wer welche Zeilen der Datei abc in welchem Commit zuletzt geändert hat  15. zeigt eine Übersicht über alle Commits auf dem aktuellen Branch an  16. zeigt Informationen zum Commit abc an, inkl. gemachte Änderungen  17. zeigt alle Änderungen seit dem letzten Commit an, die noch nicht in der Staging Area sind
2. **git commit** erstellt einen neuen Commit auf dem aktuellen Branch mit allen Änderungen, die gerade in der Staging Area sind; zum Schreiben der Commit-Nachricht wird der für git konfigurierte Standard-Editor geöffnet
3. **git push** kopiert alle Commits des aktuellen Branches auf den konfigurierten Standard-Remote (typischerweise origin)
4. **git pull** Abkürzung zum Fetchen und Mergen des passenden Remote-Branches
5. **git fetch** alle Commits vom Standard-Remote in das lokale git-Repository kopieren, aber ohne die Änderungen in die lokalen Branches zu kopieren
6. **git clone temp/abc/.git** im aktuellen Verzeichnis eine Kopie des Repositories, das im Ordner /tmp/abc/.git liegt, anlegen und den Standard-Branch auschecken
7. **git cherry-pick abc** auf dem aktuellen Branch einen neuen Commit erzeugen, der dieselben Änderungen beinhaltet wie der Commit abc
8. **git branch abc** einen neuen Branch abc anlegen
9. **git checkout abc** auf den Branch abc wechseln
10. **git merge abc** die Änderungen vom Branch abc in den aktuellen Branch integrieren (durch Erstellen eines Merge-Commits)
11. **git rebase abc** die Commits des aktuellen Branches X neu auf den Stand von abc anwenden und den Branch X auf den letzten dieser Commits zeigen lassen
12. **git reset HEAD~** das letzte git commit auf dem aktuellen Branch rückgängig machen, aber ohne die Änderungen des Commits zu verwerfen (d. h. anschließend gibt es nicht committete Änderungern)
13. **git revert abc** einen neuen Commits erzeugen, der die Änderungen von Commit abc rückgängig macht
14. **git log** zeigt eine Übersicht über alle Commits auf dem aktuellen Branch an
15. **git show abc** zeigt Informationen zum Commit abc an, inkl. gemachte Änderungen
16. **git blame abc** zeigt an, wer welche Zeilen der Datei abc in welchem Commit zuletzt geändert hat
17. **git diff** zeigt alle Änderungen seit dem letzten Commit an, die noch nicht in der Staging Area sind

# Vorlesung 1

## Interfaces und Klassen - Collection Framework

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **List Interface** | | **Set Interface  -> Reihenfolge egal**  **-> keine Duplikate** | | **Map Interface** | | **Queue Interface** | | **Comparable** |
| **ArrayList** | - Wenn schneller Zugriff auf Elemente über den Index benötigt wird  - bietet effiziente Implementierung für zufälligen Zugriff    - weniger effizient beim Einfügen oder Löschen von Elementen in Mitte | **HashSet** | - Wenn keine Duplikate & Reihenfolge egal  - Beste Leistungen für Suchen, Einfügen und Löschen von Elementen | **HashMap** | - **schnellste** Operation für Einfügen, Löschen und Suchen **Schlüssel-Wert-Paare**  **-** Reihenfolge spielt keine Rolle | **Priority Queue** | Wenn Elemente gemäß Prio abgearbeitet werden sollen  ordnet Elemente basierend auf ihrer natürlichen Ordnung oder einem Comparator. | wenn Objekte einer Klasse in einer bestimmten Reihenfolge sortieren werden sollen, die als "natürliche Ordnung" bezeichnet wird.  Die natürliche Ordnung ist die Standardreihenfolge, in der Objekte verglichen werden sollen, basierend auf den inhärenten Eigenschaften der Objekte. |
| **LinkedList** | - häufiges Einfügen und Löschen von Elementen an beliebigen Stellen    - schnelleres Einfügen und Löschen im Vergleich zur ArrayList  - langsameren Zugriff auf zufällige Elemente | **Linked HashSet** | - wenn Reihenfolge wichtig aber keine Duplikate erlaubt  - Reihenfolge: wie eingefügt | **Linked**  **HashMap** | Wie HashMap + Reihenfolge wie eingefügt  Wenn **Iterations-reihenfolge** wichtig | **Array Deque** | Implementiert Double-Ended-Queues (Deque)  -> Elemente von beiden Enden der Queue hinzuzufügen oder zu entfernen |
|  |  | **TreeSet** | - sortierte Reihenfolge der Elemente  - speichert Elemente in aufsteigender Reihenfolge | **TreeMap** | -Schlüssel in sortierter Reihenfolge  -Wenn durchgehend sortierte Map benötigt |  |  |

Utility-Klassen können den Umgang mit den Interfaces erleichtern. Das sind Befehle, zum Umgang mit den Interfaces.

|  |  |
| --- | --- |
| retainAll | Schnittmenge aus zwei Collection Frameworks (mainNames.retainAll(namesToRetain); |

## Idiomatischen Iterationscode schreiben

**Definition**: Code der den gängigen Konventionen und Stilrichtlinien für die jeweilige Programmiersprache, das Framework oder die Bibliothek folgt.

|  |  |  |
| --- | --- | --- |
| **Thema** | **Beschreibung** | **Wann nutzen** |
| **Rekursion** | Hoher Speicherverbrauch, was zu StackOverflowError führen kann, wenn Stack voll ist | Ganz selten nutzen |
| **Foreach Schleife** | auch "enhanced for loop" - ermöglicht jedes **Element einer Sammlung oder eines Arrays sequenziell zu durchlaufen**, ohne einen Index oder Iterator explizit handhaben zu müssen | Iterieren von Sammlungen immer nutzen |

## Funktionale Interfaces, Lambda-Ausdrücke

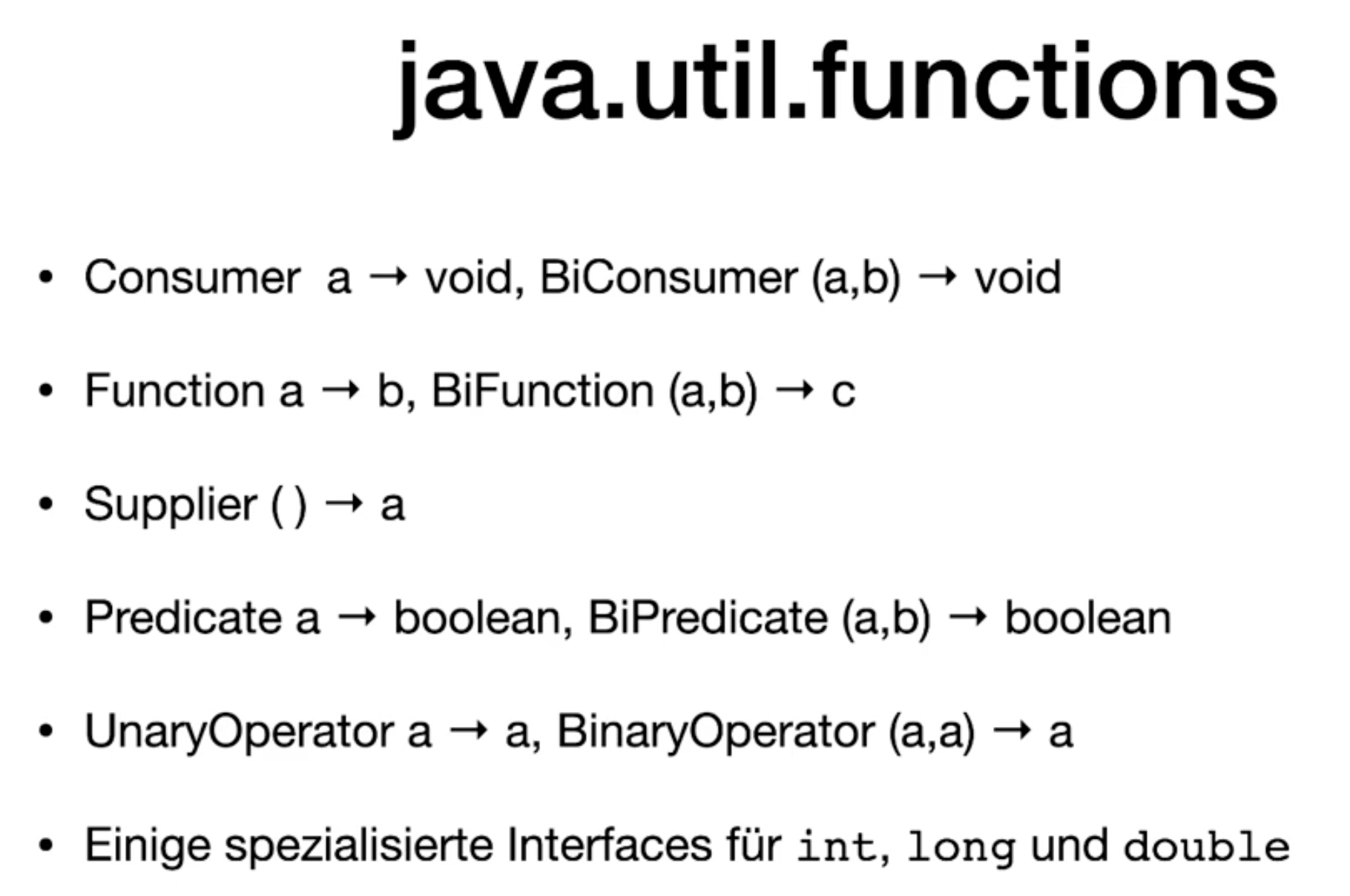
Im **Interface** werden nur **Methodenköpfe** implementiert. Methodenkörper wird in Klasse definiert, welche das Interface nutzt.

**Lambda + „foreach“:** Iteration wird umgekehrt. Statt Datenstruktur in Code, bringe ich Code in Datenstruktur   
-> Alternative

**Zusammengefasst:**

* Ein funktionales Interface ist ein Interface, das genau eine (abstrakte) Methode beinhaltet.
* Ein Lambda-Ausdruck kann überall dort eingesetzt werden, wo ein funktionales Interface erwartet wird.
* Möchte ich **idiomatisch** alle Einträge einer Liste ausgeben, ist die **forEach**-Methode eine gute Wahl -> Demnach brauche ich auch **Lambda**

## Java-Interfaces aus java.util.function kennen und verwenden

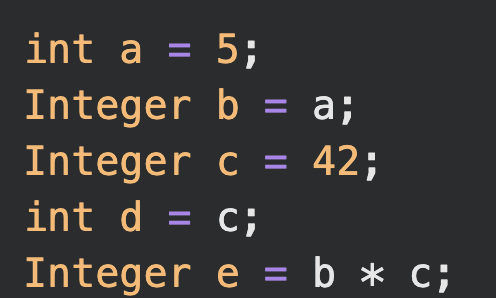


## Schleifen

|  |  |
| --- | --- |
| **Schleife** | **Zweck** |
| for | -Mehr als 1 Element gleichzeitig oder Index bekannt sein soll  -Anzahl Iteration vorher bekannt -Für Arrays und Sammlungen iterieren  -Fehleranfällig (erst foreach testen) |
| foreach | -Iterieren einer Sammlung oder Array, ohne Index zu verfolgen -Auch bekannt als enhanced loop |
| while | -wenn keine festen Grenzen  -Anzahl Iterationen vorher nicht bekannt -Schleife so lange ausführe, wie bekannt -> bis best. Ereignis erreicht |
| Do- while | Wie while, Bedingung erst am Ende geprüft, dadurch Schleifenrumpf mind. einmal ausgeführt |

## Primitives, Objekte und Auto-(Un)boxing

1. **Primitives**: Primitive Datentypen sind die grundlegendsten Datentypen, die eine Sprache direkt unterstützt. In Java beispielsweise sind das Datentypen wie **int**, **float**, **double**, **char**, **boolean**, usw. Diese Typen speichern einfache Werte und sind nicht Objekte. Primitives benötigen im Allgemeinen weniger Speicher und ihre Verarbeitung ist schneller als bei Objekten.
2. **Objekte**: Im Gegensatz zu Primitives repräsentieren Objekte **komplexe Strukturen** und können **Methoden enthalten**, mit denen auf ihre Daten zugegriffen und diese manipuliert werden können. In Java ist alles ein Objekt, außer den primitiven Datentypen. Klassen wie String, Integer, Double sind Beispiele für Objekttypen. Objekte werden im Heap-Speicher abgelegt und durch Referenzen verwaltet.
3. **Auto-(Un)Boxing**: Auto-Boxing und Unboxing sind **Prozesse** in Java, die automatisch zwischen **primitiven Datentypen** und ihren entsprechenden **Objektklassen konvertieren**.
   1. Bsp. Boxing: int automatisch zu Integer Objekt, wenn erforderlich.
   2. Bsp. Unboxing: Integer zu int



## Git

**1. Git clone**

Kopiert ein bestehendes Git-Repository von einem Server auf deinen lokalen Computer.

**2. Git add**

Fügt geänderte Dateien zur Staging-Area hinzu, um sie auf einen Commit vorzubereiten.

**3. Git commit**

Speichert die in der Staging-Area vorbereiteten Änderungen in der lokalen Repository-Historie.

-> In den meisten Texteditoren speicherst du die Nachricht und schließt den Editor, indem du die entsprechende Tastenkombination verwendest. Zum Beispiel in Vim drückst du Esc, dann :wq (für speichern und schließen) und drückst Enter. In Nano drückst du Ctrl + O um zu speichern und Ctrl + X um den Editor zu schließen.

**4. Git push**

Überträgt Commits aus deinem lokalen Repository auf ein entferntes Repository.

**5. Git status**

Zeigt den Status deines Arbeitsverzeichnisses und der Staging-Area, einschließlich geänderter, hinzugefügter oder gelöschter Dateien.

## Zusammenfassung:

**Lernen:**

* Welche Collections / Interfaces
* Welche Schleifen
* Idomatischen Code schreiben / begründen
* Java.util.function auswendig lernen
* Git clone, commit, push, add, status erklären

**Üben:**

* Funktionale Interfaces
* Lambda Ausdrücke

# Vorlesung 2

## Classpath erklären und konfigurieren

* + Werkzeug, um Java Programme in der Kommandozeile auszuführen oder übersetzen / kompilieren

*Der Classpath ist ein Parameter in der Java-Umgebung, der dem Java-Laufzeitsystem und dem Compiler mitteilt, wo nach Klassen und Paketen zu suchen ist. Er kann aus einer Liste von Verzeichnissen, JAR-Archiven und ZIP-Archiven bestehen.*

Konfigurieren des Classpaths:

Command Line: Sie können den Classpath beim Starten der JVM mit dem -cp oder -classpath Argument setzen.

Zum Beispiel:

java -cp "C:\myproject\classes;C:\myproject\lib\mylib.jar" com.example.MyClass

Umgebungsvariable: Sie können auch eine Umgebungsvariable CLASSPATH setzen, die von Java automatisch gelesen wird.

**Anwendung:**

cp oder classpath möglich

javac -classpath /home/uni/propra1/pfad /home/uni/propra1/pfad/Foo.java

* + In pfad ist Hauptdatei
  + Danach mit Leerzeichen ist Pfad zur eigentlichen Datei
  + Foo.java ist nun Hauptklasse

java -cp /home/uni/propra1/ pfad.Foo

* + Verzeichnis, wo Hauptklasse liegt / package
  + Und mit pfad.Foo wird Hauptklasse ausgeführt
  + Gleiches, wenn ich Package definiert habe, also package.pfad;

**Wann java wann javac beim classpath?**

**Verwendung von** *javac*:

* **Kompilieren**
* Bei Änderungen / erstmaligen kompilieren
* Beispiel: javac -classpath .:/path/to/libs/\* MyClass.java

**Verwendung von** *java*:

* **Auszuführen**
* Bereits kompiliert
* Beispiel: java -classpath .:/path/to/libs/\* MyClass

## Klassen in Packages strukturieren und Klassen anderer Packages verwenden

javac -cp /path/to/Java/src /path/to/Java/src2/Main.java /path/to/Java/src/MyClass.java

Pakete in Java:

* Organisieren Klassen und verhindern Namenskonflikte
* Wenn Klasse in einem Unterordner sind

Strukturieren von Klassen in Paketen:

* Verwenden Sie package.meinPaket; am Anfang der Quellcodedatei.

Klassen anderer Pakete verwenden:

* Importieren mit import paketname.Klassenname;
* Alternativ: import paketname.\*; für alle Klassen im Paket

Beispiel:

* import meinPaket.MyClass;

Zugriff auf Klassen ohne Paket:

* Klassen im Standardpaket können direkt verwendet werden

## Sichtbarkeitsregeln erklären und passende Sichtbarkeiten von Klassen, Methoden und Variablen wählen

* **Sichtbarkeitsregeln in Java**:
  + Bestimmen, welche Teile des Codes von anderen Teilen des Codes zugänglich sind.
  + Wird durch Zugriffsmodifizierer wie public, protected, private und das Fehlen eines Modifizierers festgelegt.
* **Sichtbarkeiten von Klassen**:
  + public: Klasse ist für alle anderen Klassen sichtbar.
  + Kein Modifizierer: Klasse ist nur im gleichen Paket sichtbar (Paketprivat).
* **Sichtbarkeiten von Methoden und Variablen**:
  + public: Methoden und Variablen sind für alle anderen Klassen sichtbar.
  + protected: Zugriff ist nur für Unterklassen und Klassen im gleichen Paket möglich.
  + Kein Modifizierer: Sichtbarkeit auf Klassen im gleichen Paket beschränkt (Paketprivat).
  + private: Nur innerhalb derselben Klasse sichtbar.
* **Wahl der passenden Sichtbarkeit**:
  + Verwenden Sie public für Teile, die von überall aus zugänglich sein müssen.
  + Verwenden Sie protected, wenn der Zugriff auf Unterklassen beschränkt sein soll.
  + Verwenden Sie das Fehlen eines Modifizierers für Klassen, Methoden und Variablen, die nur im selben Paket verwendet werden sollen.
  + Verwenden Sie private für Teile, die nur intern in derselben Klasse verwendet werden.

## Klassen in jar-Dateien verpacken und ausführbare jar-Dateien (mit und ohne Gradle) erstellen

1. **Erstellen einer ausführbaren JAR-Datei mit dem**jar**-Tool**:
   * Verwenden Sie das jar-Tool mit der Option e, um den Eintrittspunkt (die Klasse mit der main-Methode) anzugeben.
   * Beispiel: jar cfe hello.jar Hello Hello.class
   * -> Klasse Hello liegt in der Main-Methode (Hello.java) -> muss erst kompiliert werden
2. **Ausführen der ausführbaren JAR-Datei**:
   * Verwenden Sie den Befehl java -jar hello.jar, um das Programm auszuführen.
   * Alternativ können Sie die JAR-Datei durch einen Doppelklick in vielen Betriebssystemen ausführen.

## Ein Gradle-Java-Projekt mit Standardkonfiguration von Hand anlegen

**1. Projektverzeichnis erstellen:**

mkdir MeinGradleProjekt

cd MeinGradleProjekt

**2. Gradle Verzeichnisstruktur anlegen**

mkdir -p src/main/java

mkdir -p src/main/resources

mkdir -p src/test/java

mkdir -p src/test/resources

**3. build.gradle Datei erstellen**

touch build.gradle

ODER

code build.gradle

**4. Inhalte in build.gradle**

plugins {

 id 'java'

}

 repositories {

 mavenCentral()

}

 dependencies {

 implementation 'LINK DER BIBLIOTHEK‘

}

## Die Gradle-Tasks assemble, build, distZip, jar und run erklären und verwenden

* assemble: Kompiliert und verpackt das Projekt ohne Testausführung.  
  Befehl: ./gradlew assemble
* build: Führt assemble aus und testet das Projekt, erstellt ein verteilbares Format (z.B. JAR).  
  Befehl: ./gradlew build
* distZip: Erstellt ein ZIP-Archiv, das das Projekt und alle seine Abhängigkeiten enthält.  
  Befehl: ./gradlew distZip
* jar: Erzeugt eine JAR-Datei aus den Klassen und Ressourcen des Projekts.  
  Befehl: ./gradlew jar
* run: Startet eine Anwendung, die eine main()-Methode enthält, direkt über Gradle.  
  Befehl: ./gradlew run

## Java-Bibliotheken für einen bestimmten Zweck finden und als Abhängigkeit in ein Gradle-Projekt einbinden

## Ein vorhandenes Gradle-Projekt in eine IDE importieren und ausführen

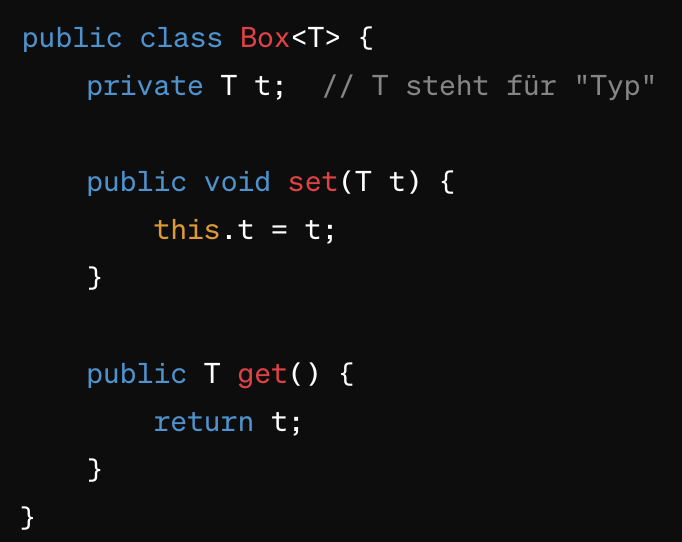
## Code mit Generics erklären und eigene generische Klassen und Methoden schreiben

### Grundlagen:

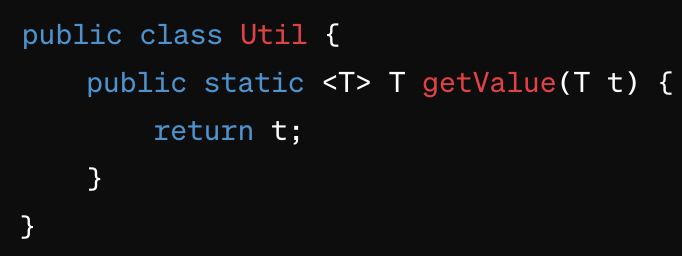
**Typsicherheit**: Generics erhöhen die Typsicherheit, indem sie es Ihnen erlauben, den Typ der Elemente, die eine Sammlung aufnehmen kann, zu spezifizieren. Dies verhindert Laufzeitfehler, da Probleme bereits bei der Kompilierung erkannt werden.

**Wiederverwendung von Code**: Durch die Verwendung von Generics können Sie mit einem einzigen Algorithmus arbeiten, der auf verschiedene Typen angewendet wird.

**Generische Klassen**: Eine Klasse kann als generisch deklariert werden, indem ein Typ-Parameter in spitzen Klammern (<>) nach dem Klassennamen hinzugefügt wird.

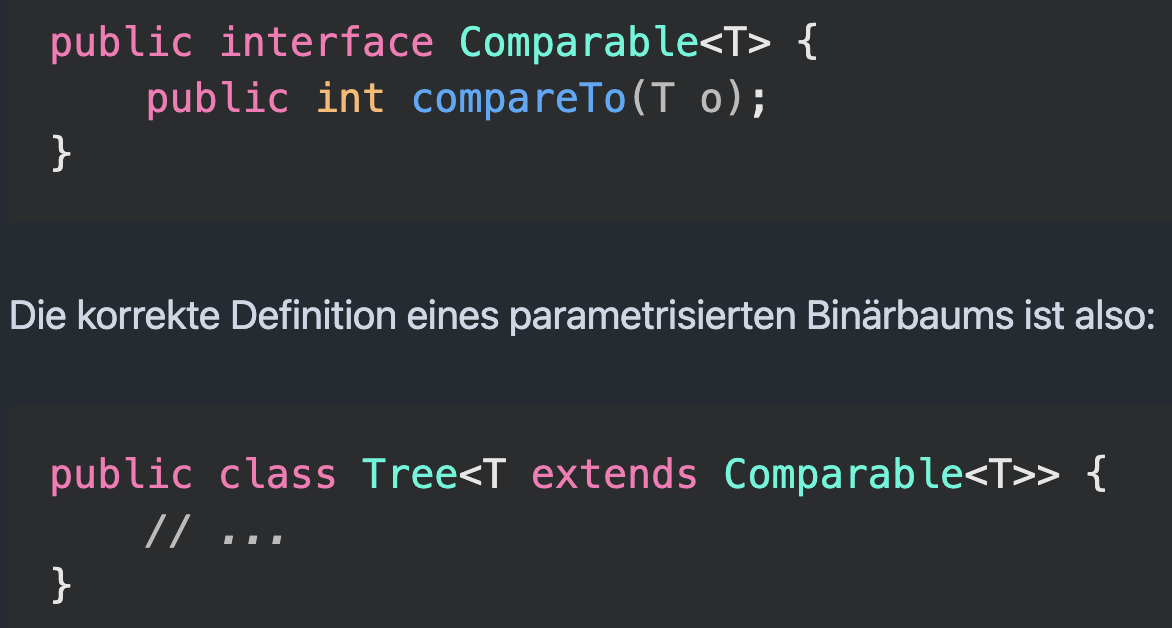


**Generische Methoden**: Eine Methode innerhalb einer Klasse kann auch ihre eigenen Typ-Parameter haben.



### Typenbeschränkung

Durch extends kann Typ eingeschränkt werden. Bspw.:



Beachte, Comparable ist auch eine Klasse, weshalb diese zuerst implementiert werden muss. Extends lässt in diesem Fall zu, dass nur Untertypen von Comparable erlaubt sind.

## Zusammenfassung

Lernen

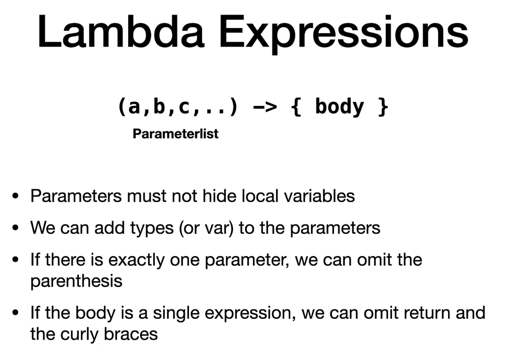
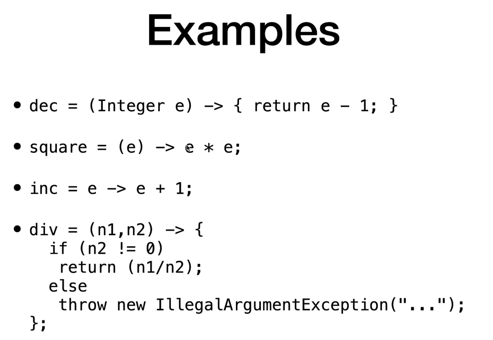
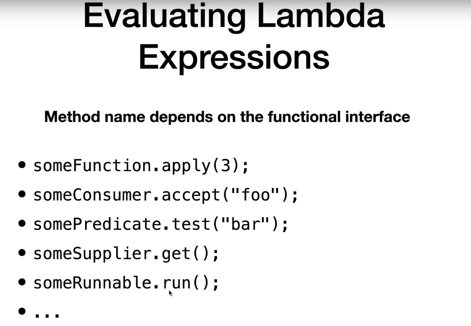
* Gradle Befehle
* Verzeichniss erstellen
* Aufbau build.gradle
* Code mit Generics erklären

Üben:

* Gradle nutzen
* Classpath -kompilieren & ausführen
* Java-Bibliotheken finden
* Generische Klassen & Methoden schreiben

# Vorlesung 3

## Lambda-Ausdrücke durch Methodenreferenzen ersetzen.



## Funktionen höherer Ordnung erklären, verwenden und implementieren.

Funktionen höherer Ordnung nehmen oder geben Funktionen als Parameter auf / zurück. Es kann auch beides gleichzeitig geschehen, also Funktion aufnehmen und zurückgeben.

Reduce Funktion: Baut so lange alle Ausdrücke, die eingegeben werden mit ein, bis keine mehr da sind

## Collections mithilfe der Stream-API verarbeiten.

* **toList**: Sammelt die Elemente eines Streams in einer Liste.
  + Syntax: toList()
  + Beispiel: List<Integer> list = stream.collect(Collectors.toList())
* **toSet**: Sammelt die Elemente eines Streams in einem Set (entfernt Duplikate).
  + Syntax: toSet()
  + Beispiel: Set<Integer> set = stream.collect(Collectors.toSet())
* **toMap**: Sammelt die Elemente eines Streams in einer Map, wobei ein Schlüssel und ein Wert für jedes Element festgelegt werden.
  + Syntax: toMap(Function<? super T, ? extends K> keyMapper, Function<? super T, ? extends U> valueMapper)
  + Beispiel: Map<String, Integer> map = stream.collect(Collectors.toMap(Object::toString, Object::hashCode))
* **joining**: Kombiniert die Elemente eines Streams zu einer einzigen Zeichenfolge.
  + Syntax: joining()
  + Beispiel: String result = stream.collect(Collectors.joining(", "))
* **summarizingInt**: Berechnet statistische Informationen für Integer-Elemente eines Streams (z.B. Summe, Durchschnitt, Minimum, Maximum).
  + Syntax: summarizingInt(ToIntFunction<? super T> mapper)
  + Beispiel: IntSummaryStatistics stats = stream.collect(Collectors.summarizingInt(x -> x))
* **partitioningBy**: Teilt die Elemente eines Streams basierend auf einer Bedingung in zwei Gruppen auf (true/false).
  + Syntax: partitioningBy(Predicate<? super T> predicate)
  + Beispiel: Map<Boolean, List<Integer>> partitioned = stream.collect(Collectors.partitioningBy(x -> x % 2 == 0))
* **groupingBy**: Gruppiert die Elemente eines Streams basierend auf einer bestimmten Klassifizierungsfunktion.
  + Syntax: groupingBy(Function<? super T, ? extends K> classifier)
  + Beispiel: Map<Integer, List<Integer>> grouped = stream.collect(Collectors.groupingBy(x -> x / 10))

## Das Collector-Interface erklären können.

Streams sind leere Objekte. Um die Ergebnisse / die Methoden, die der stream ausführt weiter nutzen zu können, brauchen wir Collector, um sie zu sammeln und weiter nutzen zu können.

stream.collect(collector)

## Die in Collectors mitgelieferten Implementierungen des Collector-Interfaces kennen und verwenden.

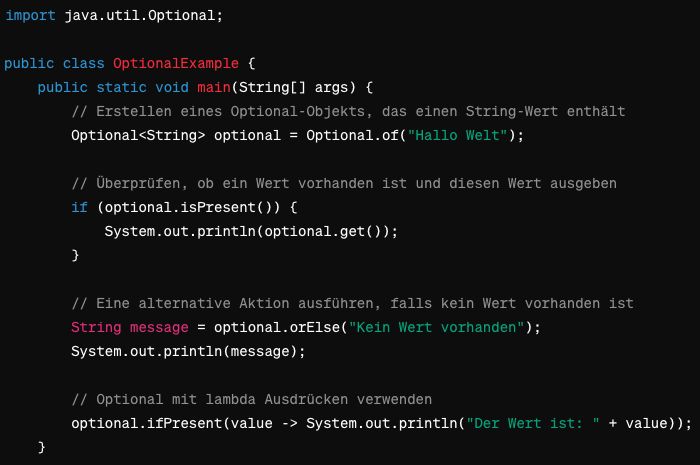
1. **toList**: Sammelt die Elemente eines Streams in einer Liste.
   * Syntax: toList()
   * Beispiel: List<Integer> list = stream.collect(Collectors.toList())
2. **toSet**: Sammelt die Elemente eines Streams in einem Set (entfernt Duplikate).
   * Syntax: toSet()
   * Beispiel: Set<Integer> set = stream.collect(Collectors.toSet())
3. **toMap**: Sammelt die Elemente eines Streams in einer Map, wobei ein Schlüssel und ein Wert für jedes Element festgelegt werden.
   * Syntax: toMap(Function<? super T, ? extends K> keyMapper, Function<? super T, ? extends U> valueMapper)
   * Beispiel: Map<String, Integer> map = stream.collect(Collectors.toMap(Object::toString, Object::hashCode))
4. **joining**: Kombiniert die Elemente eines Streams zu einer einzigen Zeichenfolge.
   * Syntax: joining()
   * Beispiel: String result = stream.collect(Collectors.joining(", "))
5. **summarizingInt**: Berechnet statistische Informationen für Integer-Elemente eines Streams (z.B. Summe, Durchschnitt, Minimum, Maximum).
   * Syntax: summarizingInt(ToIntFunction<? super T> mapper)
   * Beispiel: IntSummaryStatistics stats = stream.collect(Collectors.summarizingInt(x -> x))
6. **partitioningBy**: Teilt die Elemente eines Streams basierend auf einer Bedingung in zwei Gruppen auf (true/false).
   * Syntax: partitioningBy(Predicate<? super T> predicate)
   * Beispiel: Map<Boolean, List<Integer>> partitioned = stream.collect(Collectors.partitioningBy(x -> x % 2 == 0))
7. **groupingBy**: Gruppiert die Elemente eines Streams basierend auf einer bestimmten Klassifizierungsfunktion.
   * Syntax: groupingBy(Function<? super T, ? extends K> classifier)
   * Beispiel: Map<Integer, List<Integer>> grouped = stream.collect(Collectors.groupingBy(x -> x / 10))

## Eigene Implementierungen von Collector schreiben.

## Code erklären und schreiben, der Optional verwendet.

Optional ist eine Container-Klasse in Java, die dazu dient, den Umgang mit null zu verbessern. Sie repräsentiert ein Objekt, das entweder einen Wert enthält oder leer ist (also keinen Wert hat). Optional wird häufig verwendet, um die Lesbarkeit des Codes zu verbessern und NullPointerExceptions zu vermeiden.

Beispiel für Optional



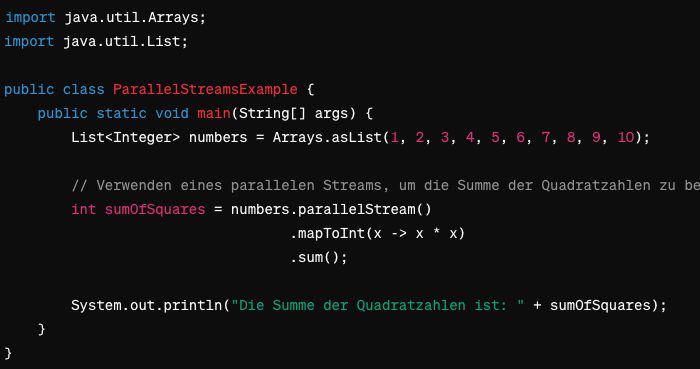
In diesem Beispiel sehen wir verschiedene Methoden, um mit Optional zu arbeiten:

* Optional.of(value): Erstellt ein Optional-Objekt, das einen Wert enthält. Wirft eine NullPointerException, wenn der Wert null ist.
* isPresent(): Überprüft, ob ein Wert vorhanden ist.
* get(): Gibt den Wert zurück, wenn er vorhanden ist; wirft sonst eine NoSuchElementException.
* orElse(defaultValue): Gibt den Wert zurück, wenn er vorhanden ist; sonst gibt es den Standardwert defaultValue.
* ifPresent(consumer): Führt den gegebenen Lambda-Ausdruck aus, wenn ein Wert vorhanden ist.

## Code erklären, der parallele Streams verwendet.

Parallele Streams sind eine Erweiterung der Stream-API in Java, die parallele Verarbeitung von Daten ermöglicht, um die Leistung auf Mehrkernprozessoren zu verbessern. Sie nutzen die ForkJoinPool-Klasse, die im Hintergrund für die Aufteilung der Aufgaben und deren parallele Verarbeitung verantwortlich ist.

Beispiel:



* parallelStream(): Erzeugt einen parallelen Stream aus der Liste.
* mapToInt(x -> x \* x): Wendet eine Funktion auf jedes Element an, um sein Quadrat zu berechnen.
* sum(): Summiert die Ergebnisse.

Parallele Streams eignen sich besonders für rechenintensive Operationen über große Datenmengen, können jedoch bei kleinen Datenmengen oder Operationen, die nicht teuer in Bezug auf CPU-Zeit sind, ineffizient sein.

## Lernziele

Üben:

* Streams
* Middleware
* Methodenreferenzen
* Sammlungsbefehle

Lernen:

Übung

1. **Java Stream Basics:** Schreibe eine Methode, die eine Liste von Ganzzahlen entgegennimmt und die Summe aller ungeraden Zahlen berechnet.
2. **Middleware in C#:** Implementiere eine Middleware-Klasse in C#, die eine Funktion mit einem vorherigen und einem nachfolgenden Transformator umhüllt. Die Klasse sollte in der Lage sein, die Eingabe zu transformieren, die gewrappte Funktion aufzurufen und schließlich die Ausgabe zu transformieren.
3. **Qualitätskontrolle mit Streams:** Schreibe eine Methode in Java, die die Qualität von Würfeln für Casinos überprüft. Die Methode sollte eine Liste von Testwürfen entgegennehmen und ein Histogramm der Augenzahlen zurückgeben, das angibt, wie oft jede Augenzahl geworfen wurde.
4. **Java Interface und Lambda-Expression:** Definiere ein Interface in Java, das eine Methode zum Verdoppeln einer Zahl deklariert. Implementiere dann diese Methode mit Hilfe einer Lambda-Expression und teste sie.
5. **Java Stream und Gruppierung:** Schreibe eine Methode in Java, die eine Liste von Studentenobjekten entgegennimmt und diese nach ihrem Studiengang gruppiert. Gib dann die Anzahl der Studenten in jedem Studiengang aus.
6. **C# LINQ-Abfrage:** Verwende LINQ in C#, um eine Liste von Wörtern alphabetisch zu sortieren und dann die Länge jedes Wortes auszugeben.
7. **Java Comparator und Streams:** Implementiere eine Methode in Java, die eine Liste von Benutzerobjekten entgegennimmt und sie nach ihrem Alter sortiert. Verwende dabei einen Comparator und Streams.
8. **C# Funktionale Programmierung:** Schreibe eine Funktion in C#, die eine Liste von Zeichenfolgen entgegennimmt und eine neue Liste zurückgibt, in der jede Zeichenfolge in Großbuchstaben konvertiert ist.
9. **Java Optional und Stream:** Schreibe eine Methode in Java, die eine Liste von Optional-Integer-Werten entgegennimmt und die Summe aller vorhandenen Integer berechnet.
10. **Java Parallel Streams:** Implementiere eine Methode in Java, die die Quadratwurzel aller Elemente einer großen Liste von Ganzzahlen parallel berechnet und das Ergebnis zurückgibt.

# Vorlesung 4

**Integration Tests:** Testen mehrere Dinge gleichzeitig -> teurer und langsamer

**Unit Tests:** Nur eine Sache wird getestet -> schneller und günstiger

## Was können Tests

**Regression verhindern:** Verhindert, dass behobene Bugs nicht wieder eingeführt werden, da Test in Sammlung bestehen bleibt und auf genau diesen Bug testet.

**Sicherheitsnetz bei Änderungen:** Sicherheit, dass wir keine groben Fehler machen, wenn wir bestehenden Code ändern.

**Codebeispiele Geben / Dokumentation:** Test gibt Verwendungsbeispiele für eine Methode

**-> Tests garantieren nicht, dass der gesamte Code fehlerfrei ist**

## Wichtige Eigenschaften Tests

* Tests müssen **einfacher** sein **als** der **Produktivcode**
* Nach Möglichkeit **linearer Code**
* **Keine komplizierte Logik**
* Immer nur **ein Konzept** pro Tests

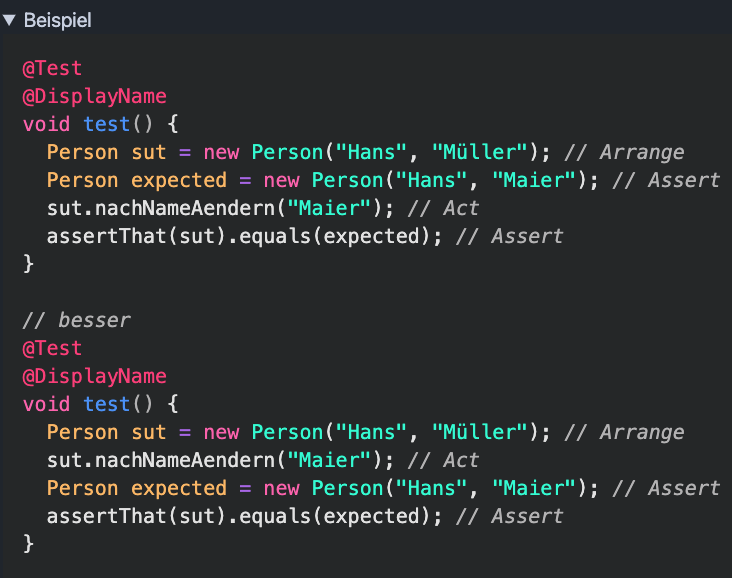
## JUnit Tests

* Normale Java Klassen
* Getrennt vom Prodiktivcode
  + Produktivcode: src/main/java
  + Testcode: src/test/java

Methoden mit **@Test** werden als Test ausgeführt

## Tests mithilfe von AssertJ nach dem AAA-Schema schreiben.

Es ist eine gute Idee, sich bei der Implementierung von Tests an das AAA-Schema zu halten. Dabei wird ein Test in drei Phasen zerlegt. Die erste Phase (Arrange) dient dazu, den Rahmen für den Test aufzusetzen. Zum Beispiel würden wir in dieser Phase das zu testende Objekt erzeugen. In der zweiten Phase (Act) rufen wir den zu testenden Code auf. In der letzten Phase (Assert) überprüfen wir, ob der aufgerufene Code das korrekte Resultat geliefert hat.



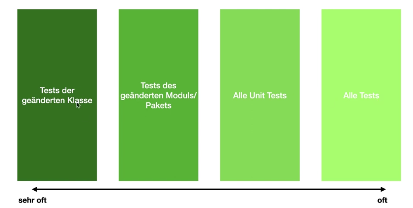
## Tests mithilfe von Gradle und der IDE ausführen.

„gradle check“ testet den Code und gibt hier Report aus:

„build/reports/tests/test/index.html“

## Erklären, wie gut automatisch testbarer Code aussieht (FIRST), und selbst solchen Code schreiben.

**1.** **Fast:** Code muss sehr **schnell** sein. Das schafft man durch z.B. **Partitionierung:**

****

**2. Isolated:** Tests müssen auswählbar sein. Heißt, ich kann mit aussuchen, welche ich laufen lasse, wie viele gleichzeitig / parallel, wie oft, etc. -> Test dürfen keine Infos teilen / haben eigenen Kontext, damit sie sich nicht gegenseitig beeinflussen

**3. Repeatable:** Zuverlässigkeit der Tests, also immer das gleiche Ergebnis, egal wo er ausgeführt wird (PC, Server, egal wo)

**4. Self evaluating:** Tests haben 2 Ergebnisse: Pass oder Fail. Das muss der Test selbst begutachten und entscheiden können

**5. Timely:** Tests und Produktivcode soll parallel gecoded werden.

## Erklären, was Seiteneffekte, seiteneffektfreie Funktionen und pure functions auszeichnet.

**Pure function:** Ausgabe ist nur von Eingabeparameter der Funktion abhängig und die Funktion hat keine Seiteneffekte.

**Seiteneffekt:** Ein Seiteneffekt ist eine Änderung im Zustand des Systems oder eine sichtbare Interaktion mit der Außenwelt, die während der Ausführung einer Funktion auftritt. -> Mit der Umgebung außerhalb der Funktion agiert. Bsp.:

* Änderung des Zustands eines Objektes
* Ausgabe auf der Konsole
* Abschicken einer Mail

**Seiteneffektfreie Funktionen:** Funktionen ohne Seiteneffekte, die keine äußeren Zustände verändern oder von externen Zuständen abhängen.

## Vor- und Nachteile von (nicht) pure functions im Hinblick auf die Wart- und Testbarkeit erläutern.

## Testgetriebene Entwicklung und deren Vor- und Nachteile erklären.

## Code testgetrieben entwickeln.

assertThat(FUNKTION/METHODE).ASSERTION()

**1. Allgemeine Assertions**

* .isEqualTo(expected): Überprüft, ob das Objekt gleich dem erwarteten Wert ist.
* .isNotEqualTo(expected): Stellt sicher, dass das Objekt nicht gleich dem erwarteten Wert ist.
* .isNull(): Überprüft, ob das Objekt null ist.
* .isNotNull(): Überprüft, ob das Objekt nicht null ist.
* .isTrue(): Stellt sicher, dass ein boolescher Wert wahr ist.
* .isFalse(): Stellt sicher, dass ein boolescher Wert falsch ist.

**2. String-spezifische Assertions**

* .contains(sequence): Überprüft, ob der String die angegebene Sequenz enthält.
* .doesNotContain(sequence): Überprüft, dass der String die angegebene Sequenz nicht enthält.
* .startsWith(prefix): Überprüft, ob der String mit dem angegebenen Präfix beginnt.
* .endsWith(suffix): Überprüft, ob der String mit dem angegebenen Suffix endet.
* .matches(regex): Überprüft, ob der String dem angegebenen regulären Ausdruck entspricht.
* .isEqualToIgnoringWhitespace(expected): Vergleicht zwei Strings, wobei Leerzeichen ignoriert werden.

**3. Collection-spezifische Assertions**

* .isEmpty(): Überprüft, ob die Sammlung leer ist.
* .isNotEmpty(): Stellt sicher, dass die Sammlung nicht leer ist.
* .hasSize(expectedSize): Überprüft, ob die Sammlung die erwartete Größe hat.
* .contains(elements): Überprüft, ob die Sammlung die angegebenen Elemente enthält.
* .containsExactly(elements): Überprüft, ob die Sammlung genau und in der angegebenen Reihenfolge die Elemente enthält.
* .doesNotContain(elements): Überprüft, dass die Sammlung die angegebenen Elemente nicht enthält.

**4. Numerische Assertions**

* .isGreaterThan(value): Überprüft, ob die Zahl größer als der angegebene Wert ist.
* .isLessThan(value): Überprüft, ob die Zahl kleiner als der angegebene Wert ist.
* .isBetween(start, end): Überprüft, ob die Zahl innerhalb des angegebenen Bereichs liegt.

## Code erklären und schreiben, der Records verwendet.

Records sind eine beliebte und einfache Art, Typen aufzuschreiben, die als reine Datenspeicher verwendet werden. Da mithilfe von Records Klassen mit mehreren Feldern kompakt notiert werden können, kommen Sie häufig bei Code-Vorgaben in Klausuren vor.

## Was ist das Vorgehen von Testgetriebene Entwicklung, Einführung TDD?

1. Wir schreiben einen **fehlschlagenden** Test (red)

2. Wir schreiben den **notwendigen Code**, der den Test durchlaufen lässt. Es sind **alle Tricks erlaubt**.

3. Wir verbessern den Code in einem **Refactoring**-Schritt. -> erst wenn alle Tests Grün!

Erst Test, dann Code schreiben:

1. **RED**: Es wird ein fehlschlagender Test geschrieben. Da fehlschlagende Tests in den meisten Werkzeugen rot dargestellt werden, wird diese erste Phase auch als RED bezeichnet.

* Man sollte dabei auch **fehlschlagende Tests** schreiben. Falls sie nicht fehlschlagen wissen wir, dass dort ein Problem ist.

2. **Minimaler:** Es wird (minimaler) Produktivcode geschrieben, sodass der Test durchläuft. (GREEN)

* **Alles erlaubt:** ChatGPT, Stackflow, selbst schreiben -> schnell und einfach

3. **Refactor:** Die interne Qualität des Codes wird verbessert.

* Erst wenn alle Tests Grün sind
* Interne Struktur anpassen, nicht äußeres Ergebnis

Man sollte dabei auch fehlschlagende Tests schreiben. Falls sie nicht fehlschlagen wissen wir, dass dort ein Problem ist.

## Welche Arten von Unit Tests gibt es?

# Vorlesung 5 - Smells

## Namenskonventionen in Java

**Attribute:** CamelCase Anfangswort kleingeschrieben

**Klassen:** CamelCase Anfangswort großgeschrieben -> Nomen

**Methoden:** CamelCase Anfangswort kleingeschrieben -> Verben

**Konstanten:** komplett großgeschrieben

**Interfaces:**

* Einfach das was es ist: Map, ArrayList, etc. im CamelCase Style mit Anfangs groß
* Adjektive: wenn Sortierbarkeit oder Iterierbarkeit ausdrücken soll (Dann: *Sortable* oder *Iterable* )

**1. Klassen oder Interfaces:**

* Erkennung: Beginnen mit einem Großbuchstaben und verwenden CamelCase. Klassen verwenden Nomen, Interfaces manchmal Adjektive.
* Beispiel: CustomerSegmenter, Sortable

**2. Methoden und Attribute:**

* Erkennung: Beginnen mit einem Kleinbuchstaben und verwenden CamelCase. Methoden verwenden Verben, Attribute können Nomen sein.
* Beispiel: printAuszug, getCustomerSegment (Methoden), auszugPrinter (Attribut)

**3. Konstanten:**

* Erkennung: Werden komplett großgeschrieben und Wörter durch Unterstriche getrennt (Screaming Snake Case).
* Beispiel: PI, ANZAHL\_MONATE

## Code Smells in bestehendem Code identifizieren, deren Folgen analysieren und (ggf. mithilfe der IDE) refactorn. (insbes. Mehrere Verantwortlichkeiten, Mysterious Name, Namenskonventionen, Verdächtige Kommentare, Lange Methoden, Lange Parameterlisten)

### Mysteriuous Name

Wahl des richtigen Namens einer Methode oder Variable kann sehr schwer sein und irreführend, falls falsch gewählt. Merke:

* Namen dürfen niemals lügen
* Namen müssen spezifisch sein, denk dran, wer den Code später lesen muss
* Verständlich!

Regeln / idomatisch:

* Konstaten (final / static-Attribute) = großgeschrieben
* Klassen = Nomen
* Methoden = Verben

### Kommentare

Begründe warum wir den Code geschrieben haben, wie wir ihn geschrieben haben. Nicht Was genau er macht im Detail. Kurz und nicht verwirrend.

### Methoden

1. Long Method:

* Eine Methode = eine Funktion -> Namensfindung einfacher
* Möglichst kurz
* Sinnvolle Unterteilungen

2. Anzahl Parameter klein halten

* Lange Parameterlisten sollten unterteilt werden und eigene Objekte implementiert werden
  + Bsp. Statt 6 Parameter für Weg im Dreidimensionale Raum, kann man auch 2 Punkte im Raum angeben. Dafür muss aber so ein Objekt existieren.

## Die Prinzipien SLAP und DRY erklären und erläutern, welche Konsequenzen Verletzungen haben.

## SLAP (Single-Layer-of-Abstraction-Prinzip)

* + Anweisungen innerhalb einer Methode haben denselben Detailgrad

**Definition:** Das Single Level of Abstraction Principle besagt, dass alle Statements innerhalb einer Methode denselben Abstraktionsgrad haben sollten.

**Ziele:**

* Erhöht die Verständlichkeit und Lesbarkeit des Codes
* Erleichtert die Wartung und Erweiterung

**Anwendung:**

* Abstraktionsgrad nicht mischen:
  + **Hoher Abstraktionsgrad:** Methoden sollten entweder nur andere Methoden aufrufen, die spezifische Aufgaben erledigen.
  + **Niedriger Abstraktionsgrad:** Oder sie sollten detaillierte Implementierungen enthalten, die konkrete Arbeitsschritte ausführen.
  + Nicht mischen: Methoden sollten nicht sowohl abstrakte Aufrufe als auch detaillierte Implementierungen enthalten.
* Jeder Methodenaufruf sollte eine klare, einzelne Aufgabe haben.

## DRY / Duplicated Code (Don’t repeat yourself)

**Definition:** Das DRY-Prinzip besagt, dass Code- oder Informationsduplikation vermieden werden sollte. Jede Wissenseinheit sollte eine einzige, eindeutige und autoritative Darstellung im System haben.

**Ziele:**

* Vermeidung von Redundanz
* Erhöhung der Wartbarkeit
* Erleichterung von Änderungen und Erweiterungen

**Anwendung:**

* Gemeinsame Logik in Methoden oder Funktionen auslagern.
* Wiederverwendbare Komponenten erstellen.
* Duplizierten Code durch Schleifen, Funktionen oder Klassen ersetzen.

Prinzip der Softwareentwicklung, das besagt, dass man Informationen oder Codes in einem System nicht mehrfach wiederholen soll. Die Idee dahinter ist, Redundanz zu vermeiden, um die Software wartungsfreundlicher zu machen.

Es ist wichtig das Wissen im Code ständig zu warten und anzupassen durch Spezifikation, Implementierung und Tests. Wissen ändert sich ständig durch neue Erkenntnisse, externe Vorschriften oder Rückmeldungen. Daher ist es wichtig, auf dem neuesten Stand zu bleiben und flexibel mit Informationen umzugehen, um Softwareprobleme zu vermeiden und den Code robust und relevant zu halten.

## Code erklären und schreiben, der Enumerations verwendet.

## Code erklären, der den ternären Operator verwendet.

## Code erklären, der das var-Keyword verwendet.

## Die Testabdeckung des eigenen Codes bestimmen und sinnvoll deuten.

# Vorlesung 6

## Systematische Fehlersuche

### Arten von Fehlern

* Syntax und Compilerfehler, Fehler beim Starten oder Bauen
* Logische Fehler
* Probleme mit Ressourcen
* Integrationsfehler

### Genereller Ansatz / Regel für Debugging:

RTFM: Read The FehlerMeldung!

### Schritte der Fehlerbehebung / Debugging bei logischen Fehlern:

1. Fehler reproduzieren
2. Diagnose der Fehlerursache
3. Reparieren
4. Reflektieren

### 1. Fehler reproduzieren

**Beobachtung**

* Was sind die Umstände unter denen der Fehler auftritt? (Kontext)
* Welches Ergebnis wird erwartet? (Soll Zustand / Erwartung)
* Welches Ergebnis bekommen wir? (Ist Zustand / Ergebnis)
* Kritisch eigene Annahmen hinterfragen, Doku anschauen!
* RTFM!

**Reproduktion**

* Keine Reproduktion = Kein Nachweis, ob der Code repariert wurde
* Reproduktion ist der **erste** Schritt!

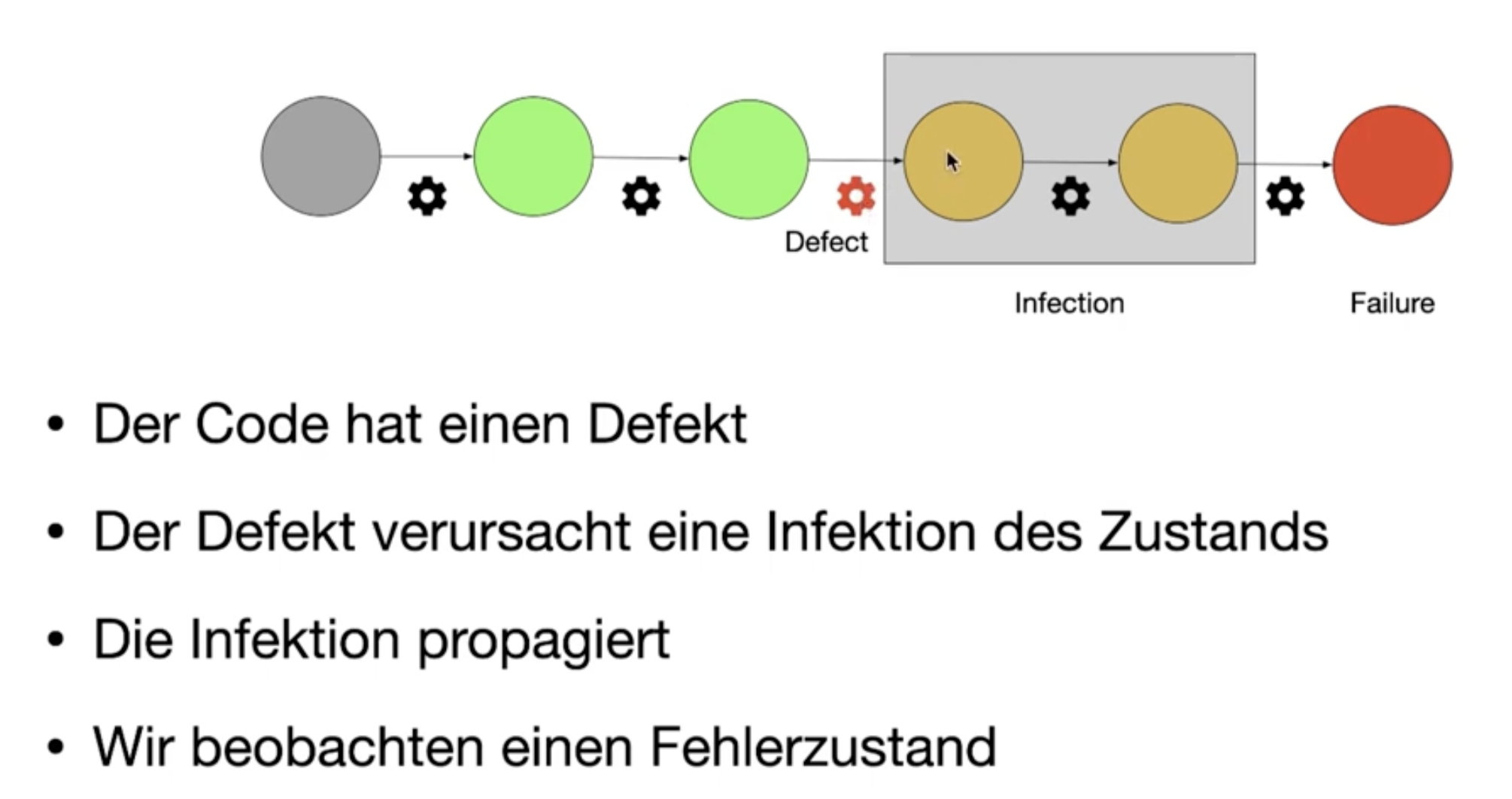
**Minimieren und Automatisieren**

* Der Bug soll möglichst automatisch provoziert werden können
  + Test schreiben
* Ausführung und Daten sollen möglichst minimal sein
  + Binäre Suche auf Daten
  + Annähern an den minimalen Input, der das Problem verursacht, damit man daraus besser herausfinden kann, woran es liegt

**Probleme**

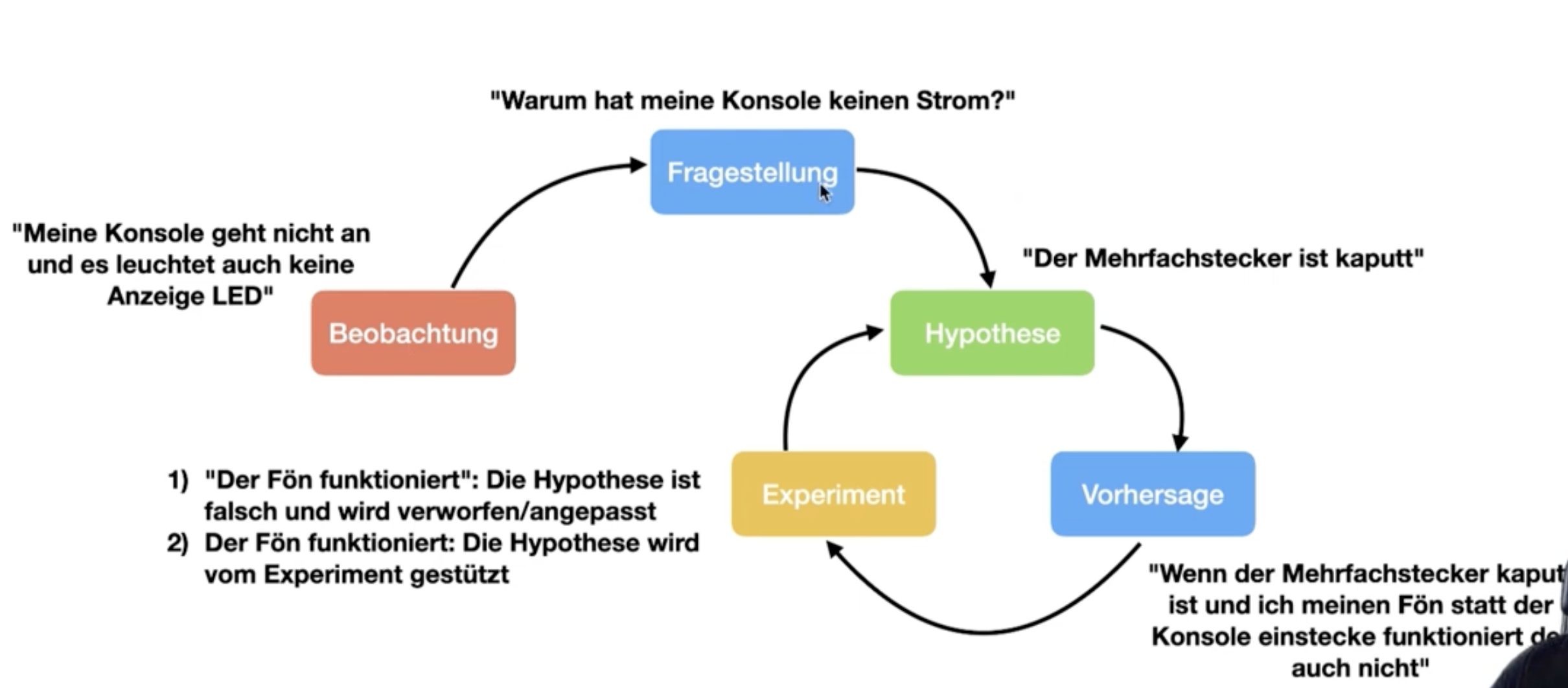
* Bugs, die nur unter schwer reproduzierbaren Bedingungen auftreten
* Nichtdeterministisch auftretende Fehler
  + nur unter bestimmten Voraussetzungen tritt Fehler auf
* Heisenbugs
  + Sobald Debugger angemacht wird, kommt Fehler nicht mehr

### 2. Diagnose der Fehlerursache



Defekt erzeugt Infektion, welche irgendwann einen Fehler verursacht

Wichtig: Nicht jeder Defekt = Fehler



**Scientific Debugging**

* Was wissen wir über das Problem? Was könnte eine mögliche Ursace sein? Aufstellen einer Hypothese (muss im Code fundiert sein)
* Wie könnte ein Experiment aussehen, das die Hypothese widerlegen kann?
* Wenn das Experiment die Hypothese widerlegt wird eine neue Hypothese gesucht
* Nach genug Experimenten sind wir sicher, dass wir die Ursache gefunden haben

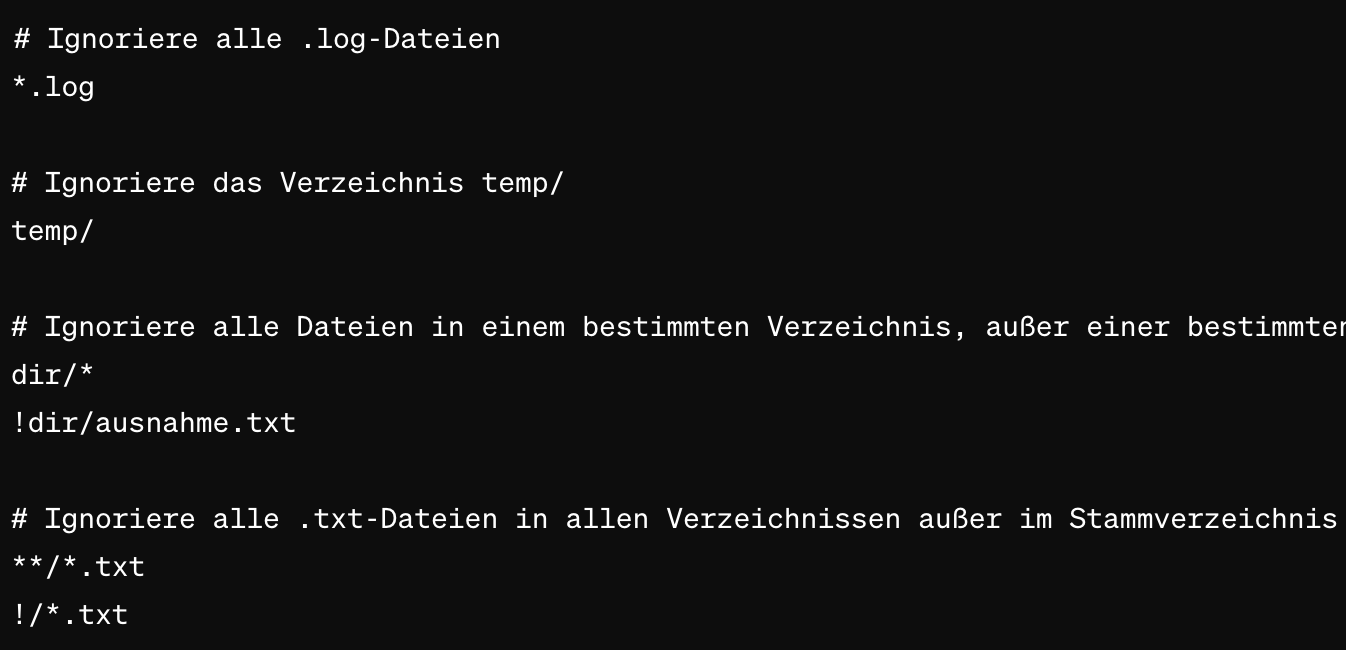
### 3. Reparieren

* Mit einem frischen System anfangen (ggf. geänderte Dateien kopieren
* Demonstrieren, dass der Fehler da ist (durch einen Test)
* Die **Fehlerursache** beheben (hoffentlich ist der Code wartbar)
* Demonstrieren, dass der Fehler behoben wurde

### 4. Reflektion

* Wie ist es zu dem Problem gekommen?
* Wie verhindern wir derartige Fehler in der Zukunft?

## Commit Befehle

* git log: Zeigt Dokumentation aller commits an
  + Syntax: git log [<options>] [<revision-range>] [[--] <path>…​]
  + **git log** --oneline Übersicht über alle Commits in einem Repository auf dem aktuellen Entwicklungszweig
  + **git log** --oneline --abbrev-commit --all --graph --decorate Übersicht über alle Commits in einem Repository, inkl. aller Verzweigungen
* git show: zeigt einzelne Commits an
  + **git show** 5581d0d4 Anzeige der Informationen zu einem einzelnen Commit, inkl. gemachte Änderungen; entspricht der [Anzeige des Commits auf Github](https://github.com/corona-warn-app/cwa-server/commit/5581d0d43dacdca797af25ba402c024b0400c144)
* git blame: Zeigt Personen der Änderungen an
  + **git blame** pom.xml Anzeige, wer welche Zeilen der Datei pom.xml zuletzt geändert hat; entspricht der [Blame-Ansicht auf Github](https://github.com/corona-warn-app/cwa-server/blame/main/pom.xml)
* git diff: Zeigt Änderungen zwischen, nach oder vor bestimmten Commits an
  + **git diff** Anzeige aller Änderungen seit dem letzten Commit
  + **git diff** ecc54380 4fb4322c Anzeige aller Änderungen zwischen den beiden angegebenen Commits
* git revert: Macht Commit rückgängig
* git ignore: legt fest, welche Ordner und Strukturen von Git ignoriert werden
  + 

## Sinnvolle Commit Formulierungen

**1. Erste Zeile:** Betreff im Imperativ; manchmal Präfix: FIX für Fehlerbehebung oder FEATURE für Funktion

**2. Leerzeile**

**3. Dritte Zeile: Ausführliche Beschreibung**

* Warum wurde die Änderung vorgenommen?
* Was wurde geändert? (sofern das nicht aus den Codeänderungen offensichtlich wird)
* Wirkt sich der Commit auf andere Programmteile aus?
* ggf. Verlinkung zu relevanten Issues, Änderungsvorschlägen oder Diskussionen

**Wichtig:**

* Ein Commit nur für zusammenhängende Änderungen. Viele kleinere Änderungen sollten in jeweils eigene Commits
* Wenn Formulierung schwierig ist, ist das ein Hinweis darauf, dass man zu viel in einen Commit packen will (Stichwort: und)

## Commits und Tests

1. Nach der Green-Phase sind wir an einem Punkt, wo wir unsere Arbeit commiten können, denn wir haben gerade ein neues Verhalten hinzugefügt. Dieses Verhalten kann im Grunde nicht zu kompliziert sein, denn wir haben ja nur einen einzigen Test hinzugefügt.

2. Nach einem Refactoring Schritt ist ebenfalls ein Punkt, wo sich ein Commit anbietet. Auch hier haben wir eine relativ überschaubare (und im Falle von Refactorings sogar in der Regel reversible) Änderung gemacht.

## Erklären, wie sich statische Methoden und Attribute von Instanzmethoden und -variablen unterscheiden.

## Die Dokumentation von Methoden und Klassen finden und dort beschriebene Funktionalität in eigenem Code verwenden.

## Das Mutationstest-Werkzeug PIT verwenden und die Ergebnisse deuten.

# Vorlesung 7

## Qualität von Software

### ISO 25010:

* **Functional Suitability:**Software funktioniert korrekt und vollständig
* **Performance / Efficiency:**Leistung des Systems, z.B. Laufzeit, Framerate
* **Compatibility:**Kompatibilität mit anderen Systemen
* **Usability:**Barrierefreiheit / Einfachheit der Nutzbarkeit
* **Reliability:**Fehlertoleranz, Verfügbarkeit
* **Security:**Datenschutz, Absicherung von Daten, Zugriffskontrolle
* **Protability:**Betreibbarkeit auf verschiedenen Systemen
* **Maintanability:**Wartbarkeit

**Warum nicht alles?**

* Nicht widerspruchsfrei
  + Laufzeit und Kompatibilität stehen im Konflikt, da Laufzeit verloren geht, wenn Ressourcen für andere Systeme gleichzeitig verwendet werden müssen
* Wir treffen spezifische Auswahl für ein Projekt

## Wartbarkeit

* Wartbarkeitskosten dominieren in vielen Fällen die Gesamtkosten
* Wartbarkeit wird reduziert durch
  + Ungeschickte Strukturierung
  + Kopplung zwischen den Strukturelementen

## Bausteine / Komponente / Modul

### Schnittstellen

*Schnittstelle* meint hier alle von anderen Komponenten sichtbaren Bestandteile einer Komponente, nicht Java-Interfaces (obwohl die natürlich auch Bestandteil einer Komponentenschnittstelle sein können).

1. Abstraktion: Eine Schnittstelle definiert nur die Methoden, die eine Klasse implementieren muss, nicht aber, wie diese Methoden funktionieren.

2. Entkopplung: Sie trennt die Verwendung von der Implementierung. Du kannst den Code schreiben, der die Schnittstelle benutzt, ohne die Details der Implementierung zu kennen.

Schnittstellen von Bausteinen: Was müssen wir über Methode wissen, damit wir sie verwenden können?

**Methoden**

Wenn wir eine Methode verwenden wollen, müssen wir ihren Namen, die Typen und Reihenfolge der Parameter und in der Regel den Rückgabetyp kennen. Wir müssen aber zum Beispiel *nicht* die Namen der Parameter kennen.

**Klassen**

Wenn wir eine Klasse verwenden wollen, müssen wir wissen, wie die Klasse heißt. Wir müssen aber auch wissen, wie wir eine Instanz der Klasse erzeugen können, also z. B. welche Konstruktoren existieren und welche Parameter die Konstruktoren haben. Sobald wir eine Instanz einer Klasse haben, können wir auf die **sichtbaren** Methoden und Attribute zugreifen.

**Package**

Package können wir nicht erzeugen oder aufrufen, dennoch ist es ein Baustein, da es Code strukturiert.

Die Schnittstelle eines Paketes sind die darin enthaltenen **öffentlichen** Klassen. Alle paketprivaten Klassen sind von außen nicht „sichtbar” und folglich nicht Teil der Schnittstelle.

**JARs**

Eine Bibliothek besteht aus einem oder mehreren Paketen und die Schnittstelle der Bibliothek ist die Vereinigung der Schnittstellen dieser Pakete.

**Modul**

Wir kennen aus der Vergangenheit schon die Möglichkeit über die Standard-Eingabe/-Ausgabe mit einem Programm zu kommunizieren. Es gibt aber auch viele weitere Möglichkeiten wie zum Beispiel Shared Memory, Dateisystem, Datenbank, Netzwerkkommunikation u.v.m.

### Anforderungen Zerlegung in Bausteine

Zerlegung so vornehmen, damit Änderungen immer nur einen Baustein betreffen, also nur eine Methode oder Klasse geändert werden muss. Wir versuchen lokal zu bleiben.

Außerdem sollte der Blast Radius abschätzbar sein. Änderungen sollten weitgehend isoliert sein, wodurch Bugs weniger wahrscheinlich sind.

* Änderungen in einer Komponente lokalisieren und den Effekt auf anderen Komponenten reduzieren

### Richtigen Bausteine bilden

**Single Responsibility Prinzip (SRP)**

* Dinge die sich aus demselben Grund ändern, sollten in denselben Komponenten untergebracht werden.
* Dinge die sich aus unterschiedlichen Gründen ändern, sollten in unterschiedlichen Komponenten untergebracht werden.

Erhöht die Wahrscheinlichkeit, dass Änderungen nur in einer Komponente durchgeführt werden müssen

**Information Hiding Prinzip**

Komponenten so schneiden, damit Entscheidungen gekapselt und nach außen versteckt werden. Dadurch müssen Änderungen nicht außerhalb der Komponente angepasst werden.

Konkret: Schnittstelle eines Bausteins bewusst so designen, dass die Details der Umsetzung nach außen hin verstecken.

### Bausteine bilden

**Die Regel der Abstraktion:**

Bei der Erstellung von Software die Realität auf ein hilfreiches Modell reduzieren, indem man unwichtige Details weglässt. Es geht darum, Konzepte von ihrer konkreten Umsetzung zu trennen, um die Komplexität zu verringern und sich auf wesentliche Aspekte zu konzentrieren. Beispielsweise müssen wir bei der Verwendung der Methode void sort(List) nur deren Funktion und Schnittstelle kennen, nicht aber die genauen Implementierungsdetails des Sortieralgorithmus.

Bei der Abstraktion kombinieren wir Code oder Daten zu neuen Einheiten, die einen Namen erhalten und als Einheit verwendet werden können. In objektorientierten Sprachen wie Java werden Daten und Prozeduren zu Objekten zusammengefasst und in Packages organisiert. Andere Sprachen nutzen ähnliche Konzepte, wie Namespaces in Clojure oder Module in C.

Auf höherer Ebene können wir größere Systeme in einzelne, kooperierende Programme zerlegen, wie Unix-Kommandos oder moderne Microservice-Architekturen. Java bietet mit dem Modulsystem zusätzlich eine Möglichkeit, Packages zu strukturieren und ihre Sichtbarkeit zu steuern.

## Kopplung

Bindungen zwischen Komponenten:

A und B sind gekoppelt, wenn in B Anpassungen vorgenommen werden müssen, falls A verändert wird. Umgekehrt muss das nicht zutreffen. Das muss erst geprüft werden.

**Kopplung im Code erkennen:**

Wenn Methode A in B verwendet wird und B eventuell nicht mehr kompiliert werden kann, sobald wir A ändern, dann ist B an A gekoppelt. A ist aber nicht an B gekoppelt, da Veränderungen in B keine Auswirkung auf A haben.

**Kopplung in Daten**

Es gibt versteckte Abhängigkeiten im Code. Wenn du zum Beispiel die Reihenfolge der Zahlen in main änderst, muss auch die Suchfunktion (Methode Suche) angepasst werden, weil sie z.B. sortierte Zahlen erwartet.

**Unsichtbare Kopplung**

Es gibt unsichtbare Abhängigkeiten im System. Zum Beispiel, wenn zwei Firmen ihre Server im selben Rack haben und eine Firma ihre Backupstrategie ändert, kann das die Leistung der anderen Firma beeinträchtigen, obwohl sie scheinbar nichts miteinander zu tun haben.

**Kopplungsstärke:**

Je mehr Annahme eine Komponente über die Schnittstellt einer anderen Komponente macht, desto stärker ist die Kopplung. Wir versuchen ein geringe / lose Kopplung zu erreichen.

Z.B. Wenn iterationsfolge unbestimmt und wenn sie fest geregelt ist, so ist die Kopplungsstärke unterschiedlich

**Kopplung minimieren:**

* Verwende abstrakte Schnittstellen: Arbeite mit Interfaces oder abstrakten Klassen anstatt mit konkreten Implementierungen. Dadurch wird die Abhängigkeit auf eine feste Implementierung reduziert.
* Einschränkung der Annahmen: Mache möglichst wenige Annahmen über die Datenstruktur oder die Implementierungsdetails anderer Komponenten. Zum Beispiel, arbeite mit generischen Sammlungen (Collection) statt spezifischen (List).
* Entkopple Komponenten: Verwende Entwurfsmuster wie Abhängigkeitsinjektion (Dependency Injection) oder Beobachter (Observer), um die direkte Abhängigkeit zwischen Komponenten zu reduzieren.
* Kapselung: Verstecke die Implementierungsdetails hinter klar definierten Schnittstellen, so dass Änderungen in der Implementierung nicht die abhängigen Komponenten betreffen.
* Modularisierung: Teile dein System in klar getrennte Module, die über wohl definierte Schnittstellen interagieren. Dies reduziert die Wahrscheinlichkeit, dass Änderungen in einem Modul andere Module beeinflussen.
* Lose Kopplung bevorzugen: Verwende Mechanismen wie Ereignis-basierte Kommunikation oder Nachrichtenwarteschlangen, anstatt direkte Aufrufe zwischen Komponenten. So können Änderungen an einer Komponente ohne direkte Auswirkungen auf andere vorgenommen werden.

### Entstehung von Kopplung

**1. Kopplung durch Verletzung des Information Hiding Prinzips:**

Wenn mehrere Teile eines Programms dieselben Daten verwenden und ändern können, entsteht eine sehr starke Kopplung. Diese sollte unbedingt vermieden werden, da eine Änderung am geteilten Zustand sehr viele Änderungen mit sich zieht. Fehler am geteilten Zustand können viele Ursachen besitzen.

### Law of Demeter:

**Das Gesetz von Demeter sagt, dass Komponenten nur mit direkten „Nachbarn“ kommunizieren sollen. Wir sollen uns also keine Objekte über einen direkten Kollaborator holen und dort Code aufrufen.**

**Das Law of Demeter (LoD) besagt, dass eine Methode einer Klasse nur Methoden ihrer unmittelbaren Felder, ihrer Parameter oder von neu erstellten Objekten aufrufen sollte.**

**2. Kopplung durch Konstruktion & Aufrufe**

Wenn die Konstruktion innerhalb einer Klasse entsteht, dann wird die Klasse immer mit dieser einen Konstruktion arbeiten müssen. Findet die Konstruktion jedoch außerhalb statt und wir übergeben es als Parameter, so kann jede beliebige Konstruktion (mit den gleichen Methoden / Eigenschaften) übergeben und damit genutzt werden. Dadurch führen Änderungen, wie z.B. Segmentierung durch Region statt Alter in der Summarizing Klasse zu keinen Problemen, sollte die Segmentierung eine Klasse sein, und die Konstruktion von Region außerhalb stattfinden. Denn nun übergeben wir einfach statt Alter, die Region. Bei Summarizing muss demnach nichts angepasst werden und die Wartbarkeit ist wesentlich einfach und weniger fehleranfällig.

Kann vermieden werden, indem Konstruktion außerhalb von Klassen stattfindet und in die Klasse übergeben wird.

Gemeinsame Interfaces einführen z.B. AgeSegmentation & PLZSegmentation bekommen Segmentation Interface

**Dependency Inversion** -> Konkrete Implementierung durch Interface ersetzen, wodurch Compilezeit sich umdreht

**Dependency Injektion** -> Konstruktion außerhalb der Klasse, wodurch dann Übergabe als Parameter stattfindet, statt Konstruktion in der Klasse

Nicht alle Konstruktionen in Klasse sind falsch!

**3. Kopplung durch Vererbung**

## Kohäsion

Kohäsion ist Bindung innerhalb von Komponenten.

Wir wollen eine hohe Kohäsion *innerhalb* einer Komponente und eine lose Kopplung *zwischen* den Komponenten. Dieses Prinzip wird als LCHC (**low coupling, high cohesion**) bezeichnet.

## Code im Hinblick auf IHP, SRP, SLAP, LCHC und Law of Demeter analysieren und die Folgen abwägen.

## Code entsprechend der Prinzipien, Regeln und Praktiken in Methoden, Klassen und Packages zerlegen.

# Vorlesung 8

## Durch Kapselung Invarianten sicherstellen.

### Kapselung (Encapsulation):

* Definition: Konzept der Objektorientierten Programmierung (OOP), das Daten und Methoden, die auf diese Daten zugreifen, in einer Klasse zusammenfasst.
* Ziel: Schutz der Datenintegrität durch Einschränkung des direkten Zugriffs auf die Daten.
* Mittel: Verwendung von Zugriffsspezifikatoren (private, protected, public).

### Invarianten:

* Definition: Bedingungen, die zu jedem Zeitpunkt während der Lebensdauer eines Objekts wahr sein müssen.
* Ziel: Sicherstellung der Konsistenz und Korrektheit des Objektzustands.
* Bsp.: Alter, Bankkonto darf nicht negativ sein, Geburtsdatum muss in Vergangenheit liegen

### Umsetzung:

1. **Private Felder:** Datenfelder einer Klasse als privat deklarieren, um direkten Zugriff von außen zu verhindern.

*private int age;*

1. **Getter und Setter:** Zugriff auf private Felder über öffentliche Methoden (Getter und Setter) kontrollieren.

*public int getAge() { return age; }*

*public void setAge(int age) {*

*if (age > 0) { // Invariante sicherstellen this.age = age;   
} else {*

*throw new IllegalArgumentException("Age must be positive");*

*}*

*}*

1. **Konstruktoren:** Initialisierung der Objekte durch Konstruktoren, die die Invarianten überprüfen.

*public Person(int age) {*

*setAge(age); // Nutzung von Setter zur Invariantenprüfung*

*}*

### Wichtige Punkte:

**Information Hiding:** Details der Implementierung verbergen, um Änderungen zu erleichtern und die Integrität der Daten zu wahren.

**Wartbarkeit:** Durch Kapselung können Änderungen intern durchgeführt werden, ohne externe Schnittstellen zu beeinflussen.

## Code Smells in bestehendem Code identifizieren, deren Folgen analysieren und (ggf. mithilfe der IDE) refactorn. (insbes. Large Class, Primitive Obsession, Data Clumps, Divergent Change, Shotgun Surgery, Feature Envy, Message Chains)

### Code Smells innerhalb von Klassen:

### Large Class

Hier ist der Name eigentlich schon Programm. Wenn wir Klassen mit vielen Methoden haben, dann kann es sehr gut sein, dass diese Klassen zu viele Verantwortlichkeiten haben und möglicherweise das Single Responsibility Prinzip verletzen.

### Primitive Obsession

Primitive Datentypen (wie int, string, float) werden übermäßig und unangemessen anstelle von besser geeigneten Klassen oder Objekten verwendet werden. Erstellung von Klassen hilft bei der Kapselung.

**Probleme:**

* Schlechte Lesbarkeit: Der Code wird schwer verständlich, da primitive Datentypen keinen Kontext bieten.
* Wiederholter Code: Validierungslogik und andere Operationen werden oft dupliziert.
* Schwierige Wartbarkeit: Änderungen erfordern Anpassungen an vielen Stellen im Code.

### Data Clumps

Bestimmte Gruppen von Daten (Felder, Parameter) treten oft zusammen auf, sind aber nicht in einer eigenen Klasse oder Struktur gekapselt.

**Probleme:**

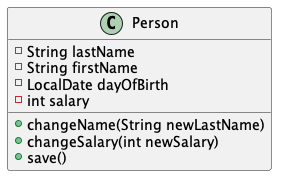
* Wiederholter Code: Gleiche Gruppen von Daten werden in mehreren Methoden oder Klassen verwendet.
* Schlechte Lesbarkeit: Der Zusammenhang zwischen den Daten ist nicht klar erkennbar.
* Schwierige Wartbarkeit: Änderungen an der Gruppe von Daten müssen an vielen Stellen im Code durchgeführt werden.

### Code Smells zwischen Programmen und Beziehungen zwischen Klassen

### Divergent Change

Divergent Change tritt auf, wenn eine Klasse aufgrund unterschiedlicher Gründe mehrfach geändert werden muss.

Bsp.



Muss aber nicht automatisch schlecht sein. Hier liegen Code Smells vor, die aber gewollt sein können.

### Shotgun Surgery

Shotgun Surgery tritt auf, wenn eine kleine Änderung viele verschiedene Klassen betrifft.

### Feature Envy

Niedrige Kohäsion, hohe Kopplung: wenn eine Methode einer Klasse mehr Interesse an den Daten einer anderen Klasse zeigt als an ihren eigenen.

### Message Chains

Tritt oft auf, wenn wir gegen das Law of Demeter verstoßen. Das Gesetz von Demeter sagt, dass Komponenten nur mit direkten „Nachbarn“ kommunizieren sollen. Wir sollen uns also keine Objekte über einen direkten Kollaborator holen und dort Code aufrufen.

Das Law of Demeter (LoD) besagt, dass eine Methode einer Klasse nur Methoden ihrer unmittelbaren Felder, ihrer Parameter oder von neu erstellten Objekten aufrufen sollte.

Message Chains entstehen, wenn ein Objekt eine lange Kette von Methodenaufrufen auf anderen Objekten ausführt.

## Code Smells beheben oder nicht?

Code Smells sind immer nur ein Hinweis auf ein mögliches Problem und in machen Fällen gehören sie notwendigerweise zu bestimmten Mustern und lassen sich nicht vermeiden, ohne andere Prinzipien zu verletzten.

Bevor wir uns an die Aufgabe machen, einen Code Smell zu beseitigen, sollten wir uns die Frage stellen, ob das notwendig ist.

1. Welche Art von Änderung würde in der Zukunft durch den Code Smell erschwert werden und wie wahrscheinlich ist es, dass wir diese Änderung durchführen werden?

2. Entstehen durch die Beseitigung des Code Smells im vorliegenden Code neue Probleme? Haben diese neuen Probleme schlimmere Konsequenzen als der ursprüngliche Code Smell?

Die Verbesserung der Wartbarkeit einer Software ist immer ein Trade-Off und kein Selbstzweck.

## Erklären, welche unterschiedlichen Arten von Annotationen es gibt.

Annotationen sind Metadaten, die zu Code-Elementen (Klassen, Methoden, Feldern) hinzugefügt werden und zusätzliche Informationen oder Anweisungen bereitstellen.

Allgemeine Annotationen

* **@Override**
  + Markiert eine Methode, die eine Methode der Superklasse überschreibt.
* **@Deprecated**
  + Markiert eine Methode, Klasse oder ein Feld als veraltet und rät von deren Verwendung ab.
* **@SuppressWarnings**
  + Unterdrückt bestimmte Compiler-Warnungen.

Meta-Annotationen (Annotationen für Annotationen)

* **@Retention**
  + Gibt an, wie lange die Annotation behalten werden soll (z.B. Quelle, Bytecode, Laufzeit).
* **@Target**
  + Gibt an, auf welche Code-Elemente die Annotation angewendet werden kann (z.B. Methoden, Felder).
* **@Documented**
  + Gibt an, dass die Annotation in der Javadoc-Dokumentation erscheinen soll.
* **@Inherited**
  + Gibt an, dass eine Annotation von einer Superklasse vererbt werden kann.

Annotations in JPA (Java Persistence API)

* **@Entity**
  + Markiert eine Klasse als JPA-Entität.
* **@Table**
  + Gibt den Tabellennamen in der Datenbank an, mit dem die Entität verknüpft ist.
* **@Id**
  + Markiert ein Feld als Primärschlüssel.
* **@GeneratedValue**
  + Gibt die Strategie zur automatischen Generierung des Primärschlüssels an.
* **@Column**
  + Gibt die Eigenschaften der Datenbankspalte an, die einem Feld zugeordnet ist.

Annotations in Spring

* **@Component**
  + Markiert eine Klasse als Bean zur automatischen Erkennung durch Spring.
* **@Autowired**
  + Ermöglicht die automatische Injektion von Abhängigkeiten.
* **@Service**
  + Spezifische Art von @Component, die eine Serviceklasse kennzeichnet.
* **@Repository**
  + Spezifische Art von @Component, die eine Datenzugriffsschicht kennzeichnet.
* **@Controller**
  + Spezifische Art von @Component, die einen Spring MVC-Controller kennzeichnet.

Annotations für Tests (JUnit)

* **@Test**
  + Markiert eine Methode als Testmethode.
* **@Before**
  + Markiert eine Methode, die vor jedem Test ausgeführt wird.
* **@After**
  + Markiert eine Methode, die nach jedem Test ausgeführt wird.
* **@BeforeClass**
  + Markiert eine Methode, die einmalig vor allen Tests in der Klasse ausgeführt wird.
* **@AfterClass**
  + Markiert eine Methode, die einmalig nach allen Tests in der Klasse ausgeführt wird.

## Erklären, welche Möglichkeiten das Reflection-Framework bietet.

Reflection ist die Fähigkeit eines Programms, zur Laufzeit Informationen über sich selbst zu erhalten oder zu ändern.

**Möglichkeiten:**

* Klassen-Informationen abrufen
* Methoden aufrufen
* Instanziierung von Objekten
* Zugriff auf Felder
* Ändern von Attributen und Methoden
* Analyse und Manipulation von Annotationen

# Vorlesung 9

## Polymorphismus

**Arten:**

1. Ad-hoc Polymorphismus  
   **Funktionsüberladung:** z.B. gleiche Klasse mit int und mit double. Compiler wählt dann korrekte  
   **Operatorüberladung:** Operatoren, wie ‚+‘ können genutzt werden. In Java jedoch nicht modifiziert werden
2. Parametrischer Polymorphismus  
   siehe Generics
3. Vererbungspolymorphismus   
     
   Der Hauptzweck von Vererbungspolymorphismus ist, dass wir neue Fälle zu einem Programm hinzufügen können, ohne den restlichen Code anfassen zu müssen.  
     
   **Open-Close-Prinzip:** offen für Erweiterungen (z. B. neue Klassen), ohne dass bestehender Code verändert werden muss (geschlossen für Veränderung)

**-**  **Offen für Erweiterung:** Du solltest in der Lage sein, das Verhalten eines Softwaremoduls zu erweitern, ohne seinen bestehenden Code zu ändern. Dies erreicht man oft durch Vererbung, Interfaces oder Abstraktionen.

**- Geschlossen für Veränderung:** Sobald ein Modul implementiert und getestet ist, sollte es nicht mehr geändert werden müssen. Änderungen sollten durch Hinzufügen neuen Codes erfolgen, nicht durch Modifikation des bestehenden Codes.   
  
-> Neue Methoden für neue Berechnungen oder so einfügen, statt es in bestehenden Code einfügen zu müssen. Dadurch macht man sich weniger kaputt. Nach dem Prinzip, wenn der Test grün war, soll er es auch bleiben.

Wenn dasselbe switch-case-Konstrukt an mehreren Stellen in einem Programm auftaucht, ist es sinnvoll, kurz über Polymorphie nachzudenken.  
  
Reduziert Kopplung (Dependency Inversion), da zwar Schnittstellen bestehen, das sind aber nur Namen von Methoden und Funktionen, keine ganzen Code Texte.  
  
**(java.lang.Objekt** ganz oben in Hierarchie -> jedes Objekt erbt davon  
  
Das Liskovsche Substitutionsprinzip Wenn U eine Unterklasse von O ist, kann ich immer, wenn ein O-Objekt erwartet wird, ein U-Objekt einsetzen. Alles, was mit O-Objekten geht, muss auch mit U-Objekten gehen und dieselben Invarianten einhalten.  
  
**Vorteil:** **Open/Close Prinzip:** Wenn wir neue Unterklassen hinzufügen wollen, geht das schnell. Aber wir sind von außen vor Modifizierungen geschützt. Unterklassen, die Eigenarten bei Funktionen / Methoden haben, können so einfach implementiert werden.  
  
**Nachteil:** Wenn ich eine neue Methode / Funktion in der Basisklasse einfüge, dann muss ich sie auch überall implementieren, wodurch Änderung mehr verstreut wird.)

## Vererbung und Kopplung

### Interface Segregation Prinzip

Eine Hilfe bei der Definition von Interfaces ist das sogenannte Interface Segregation Prinzip (ISP). Das ISP besagt, dass wir unsere Schnittstellen so klein wie möglich, aber so groß wie nötig entwerfen sollen. Wir möchten im benutzenden Code die Informationen über Dependencys möglichst klein halten.

Wenn wir ein Interface definieren, dann sollte dieses möglichst nur die Methoden enthalten, die vom Client benötigt werden. Das Interface sollte also nicht alle möglichen denkbaren Funktionen enthalten.

Damit halten wir die Kopplung durch Vererbung gering, wodurch Änderungen an Interfaces weniger Änderungen an Klassen nach sich ziehen.

### Kopplung bei Vererbung mit Klassen

Aus der starken Kopplung durch Vererbung folgt, dass wir Ober- und Unterklassen immer gemeinsam weiterentwickeln müssen. Wenn wir Vererbung verwenden, dann sollten wir nach Möglichkeit die Superklasse und ihre Subklassen innerhalb einer größeren Komponente, z. B. einem Package, isolieren und immer gemeinsam entwickeln. Vererbung über Komponenten-Grenzen hinweg sollten wir dagegen möglichst vermeiden.

### Refused Bequest Smell

Der Refused Bequest Smell (abgelehnter Nachlass) tritt auf, wenn eine Unterklasse Methoden oder Eigenschaften von ihrer Basisklasse erbt, diese aber nicht benötigt oder verwendet. Dies deutet oft darauf hin, dass die Vererbungshierarchie nicht korrekt oder suboptimal gestaltet ist.

## Polymorphismus nutzen, um Kopplung zu reduzieren und offen für Erweiterungen zu sein.

## Begründet zwischen Klassen-Vererbung, Interface-Vererbung und Komposition im Hinblick auf Wartbarkeit auswählen.

## SOLID-Prinzipien beim Entwickeln anwenden und Verletzung im Quellcode erkennen, deren Folgen zu erläutern und die Probleme zu beheben.

## Code erklären, der die java.time-Klassen verwendet.

# Vorlesung 10

Test-Double:

Test-Doubles sind wesentliche Werkzeuge in der Testautomatisierung und in der Test-Driven Development (TDD) Praxis, da sie helfen, Tests unabhängig von externen Systemen und Abhängigkeiten zu gestalten, was zu stabileren und zuverlässigeren Tests führt. Vor allem dafür da künstliche Umgebungen für Tests zu schaffen

Stubs: Stubs ersetzen ein kollaborierendes Objekt in einem Test und simulieren das echte Objekt für ein bestimmtes Testszenario. Stubs haben keine Logik, sondern geben nur vordefinierte Werte zurück, die für das spezifische Testszenario hartcodiert sind.

Dummy: Ein Dummy ist ein Objekt, das im getesteten Code gar nicht genutzt wird und nur dazu dient, einen Parameter zu befüllen. Beispielsweise, wenn man einen Kunden erstellen muss, damit man die Methode testen kann, dann erstellt man einen Kunden „Dummy McDummyFace“ oder so, damit wir einen haben, den man am Ende gar nicht braucht. Ein Dummy sollte nicht ausgegeben werden, bei dem Test. Er sollte nur als nicht-verwendetes Zwischenobjekt genutzt werden.

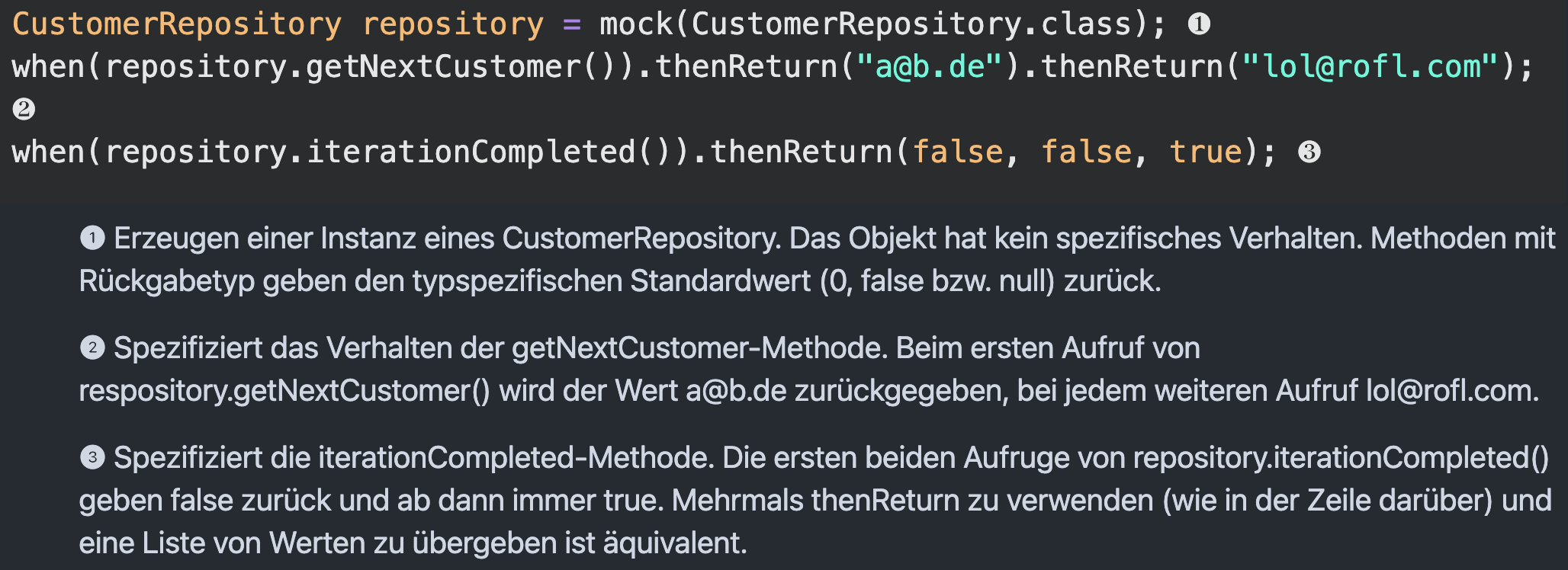
Mock: Ein Mock ist ein speziell konfiguriertes Testobjekt, das das Verhalten eines echten Objekts simuliert, um Interaktionen und Methodenaufrufe während des Tests zu überprüfen. Es nimmt am Assert-Schritt teil. Beispielsweise die Funktion neue Mails in einer Mailliste zu speichern. Dann kann ein Test geschrieben werden, welcher genau das implementiert.

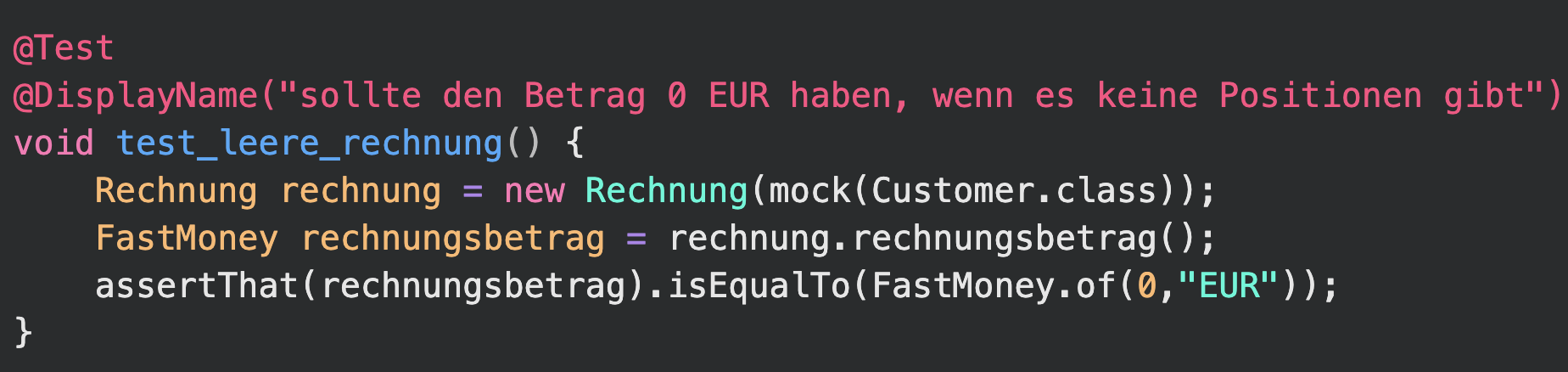
Mockito:

Klasse, die importiert werden kann. Damit können dann Test-Doubles erstellt werden. Welche genau das sind liest sich aus dem Kontext, da nicht gezeigt wird, ob Stub, Mock, oder Dummy. Die Methode erzeugt eine Instanz, die denselben Typ hat wie das übergebene Class-Objekt und deren Methoden durch speziellen Testcode ersetzt wurde.

* + Das MockFramework lohnt sich nur dann, wenn die Erstellung per Hand umständlicher ist und länger dauern wurde. Sonst ist per Hand lesbarer und schneller.

Datenbank Stub: Ein Stub ist ein Testobjekt, das vordefinierte Antworten auf bestimmte Aufrufe liefert, ohne die eigentliche Logik auszuführen. Es wird verwendet, um kontrollierte Rückgabewerte für bestimmte Methodenaufrufe zu liefern.



Customer-Dummy: Mit einem mock, können auch Dummys erstellt werden, wie hier im Bsp. 

Statt bei der Rechnung einen echten Kunden zu übergeben, erstellen wir einen mock und übergeben die dazugehörige Klasse. mock(customer.class) ist ein stub, hilft uns aber dabei ein Dummy zu erstellen. Es verlangt hier auch keine Parameter o.ä.

verify(KLASSE).METHODE(AUSGABE,DIE RAUSKOMMEN SOLLTE) -> mit verify kann man auch assert durchführen.

## Unterschiedliche Arten von Test-Doubles verwenden und mit und ohne Mockito implementieren.

## Erläutern, ob ein Test-Double in einem Test als Stub, Dummy oder Mock verwendet wird.

## Folgen der Verwendung von Mock-Objekten in Tests im Hinblick auf Kopplung erklären.

## Code erklären, der @Tag verwendet.

# Vorlesung 11

## Konsequenzen von Kopplung zwischen Tests und Produktivcode erläutern und (insbes. mithilfe von Factory-Methoden, dem Builder Pattern, Templates, passenden Assertions und eigenen Assertions) minimieren.

## Überspezifikation und DRY-Verletzungen in Tests analysieren und beheben.

## Erklären, was eine Fluent API auszeichnet.

## Parametrisierten Test-Code erklären.

## Die Ausführungsreihenfolge von Test-Methoden mit @BeforeAll, @AfterEach usw. erklären.

# Vorlesung 12

**Fachlicher Code:** Bezieht sich auf allgemeine Geschäftsprozesse. Kann schwer wiederverwendet werden, da es sich auf fachlich spezifische Themen bezieht.

**Technischer Code:** Kann super wiederverwendet werden, da es sich immer um dasselbe handelt. Zum Beispiel Zugriff auf eine Datenbank.

**Framework:** Gibt Rahmen vor. Darin kann fachlicher Code eingefügt werden. Technischer Code, wird vom Framework übernommen. Beispiele: Java Enterprise Edition, Spring / Spring Boot, Micronaut, Quarkus.

## Mithilfe des Spring Frameworks Dependency Injection umsetzen. (insbes. @Component, @Configuration, @Bean, @Primary, @Qualifier)

@SpringBootApplication

Hauptklasse, die alles steuert. Markiert Spring Application. Mit: ‚SpringApplication.run(KLASSENNAME.class, args);’ werden @bean Methoden gesucht und ausgeführt

@Bean

In eine bean kann jede andere bean / component injeziert werden. Z.B. Greeter in CommandLineRunner. Sobald ein Objekt als Parameter angegeben wird, sucht Spring nach einer @Component Klassen die mit der Objektklasse übereinstimmt und gibt eine Instanz hinein.

@Primary

Wenn mehrere Beans des gleichen Typs vorhanden sind, kann @Primary verwendet werden, um die bevorzugte Bean zu markieren. Kommt vor, wenn mehrere Klassen Interface implementieren und als Parameter Interface gegeben wird. Unter @Component

@Qualifier

Die @Qualifier-Annotation wird verwendet, um explizit anzugeben, welche Bean injiziert werden soll, wenn mehrere Beans desselben Typs existieren. Dies überschreibt die @Primary-Annotation, falls vorhanden. In Parameterliste vor Parameter @Qualifier "A", dann wird die vor anderen Klassen genommen. Oder Klassenname, aber mit Anfangsbuchstabe klein (das vermeiden, wegen Umbennenungen).

@Autowired

Wenn mehrere Konstruktoren einer in einer @Component-Klasse stehen und es keinen Default-Konstruktor gibt, weiß Spring erstmal nicht, welcher Konstruktor für die Objekterzeugung bei einer Injection genutzt werden soll. Mit @Autowired kann ein Konstruktor für diesen Zweck explizit festgelegt werden. Bei Field-Injection nutzen. Das sind Klassen ohne Konstruktor. Das sollte man vermeiden. Geht auch bei Setter-Injection.

**Injizierbare Klassen:**

@ Component

Die @Component-Annotation markiert eine Klasse als Bean (d.h. ein verwaltetes Objekt), die von Spring verwaltet wird. Mit @Component ("A") wird Klasse umbenannt.

@ Configuration

Die @Configuration-Annotation markiert eine Klasse, die eine oder mehrere @Bean-Methoden definiert. Eine @Bean-Methode gibt eine Bean zurück, die im Spring Context registriert wird. Heißt, die Klasse, die die @Bean-Methode ausgibt, fungiert wie ein @Component.

* + Wenn Klasse nicht mit @Component markierte ist, oder werden kann, kann dann trotzdem von der @Bean Methode injiziert werden. Die @Bean Methode muss dann in einer @Configuration Klasse sein

**Zirkuläre Dependencies:**

LHCH-Verletzung, da eher auf andere Klassen zugegriffen wird. Zwischen Spring Beans sollte das niemals erzeugt werden.

**Optionale Dependencies:**

SRP-Verletzung, da zu viele unterschiedliche Aufgaben.

## Erklären, was bei mutable Objekten bei der Injection zu beachten ist, und dieses Verhalten modifizieren. (@Scope("prototype"))

Mutable Objekte: Objekte, die nach Einstellung verändert werden können, z.B. Listen

Singletone-Scope: Spring verwaltet Beans normalerweise im Singleton-Umfang, also nur eine Instanz eines Beans pro Scoping-Container. Bei Mutable Objekten kann das problematisch sein, da diese teilweise mehr Zustände brauchen. Man bekommt bei Singleton also immer nur eine Instanz einer Klasse, welche durchgehend genutzt wird.

Prorotype Scope: Wenn ein Bean im Prototype-Scope deklariert wird, erstellt Spring eine neue Instanz des Beans jedes Mal, wenn es benötigt wird. Dies ist besonders nützlich für mutable Objekte, da jede Anforderung eine eigene, unabhängige Instanz erhält.Dabei bekommt man jedesmal eine neue Instanz einer Klasse.

Syntax:

@Component   
@Scope("prototype") / @Scope("Singleton")

## Externe Konfigurationsdateien anlegen und Werte daraus injizieren lassen. (@Value)

In Spring-Anwendungen können Konfigurationswerte in einer separaten Datei, typischerweise einer application.properties oder application.yml, gespeichert werden. Diese Dateien enthalten Schlüssel-Wert-Paare, die von Spring verwaltet und bei Bedarf in Beans injiziert werden können. In Properties können außerdem einige Einstellungen eingeführt werden, wie den Banner auszustellen, oä. YAML sind hierarchisch aufgebaut, was angenehmer zu lesen ist.

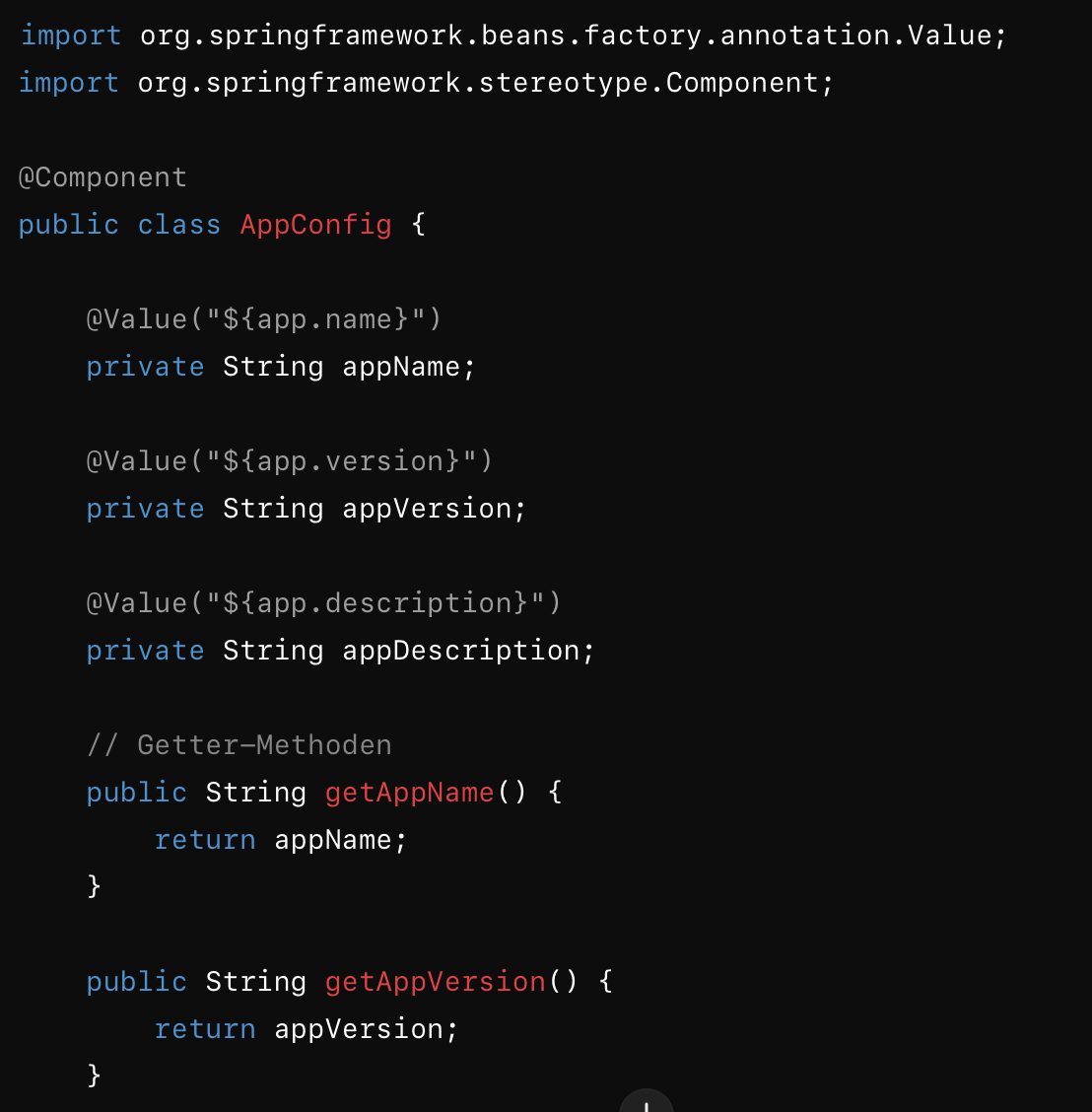
**Schritt 1:** Anlegen der Konfigurationsdatei

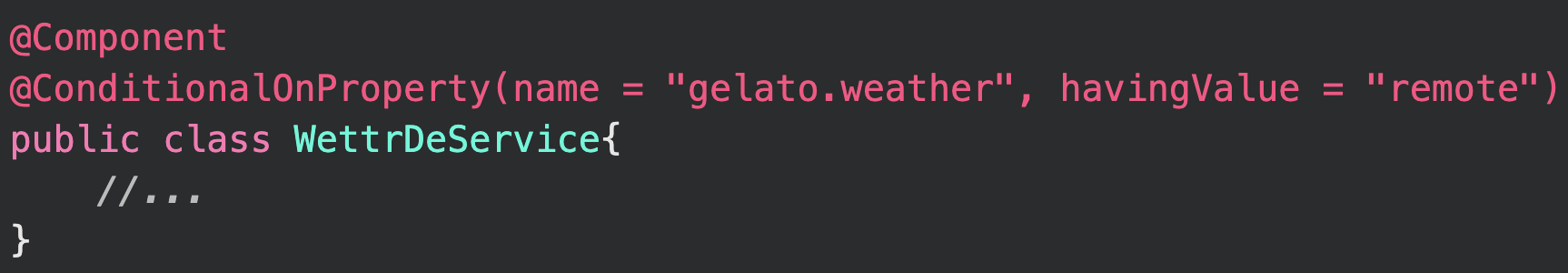
Erstellen Sie eine application.properties Datei im src/main/resources Verzeichnis Ihrer Spring-Anwendung:

# src/main/resources/application.properties   
app.name=MeinAppName   
app.version=1.0.0   
app.description=Das ist eine Beispielanwendung.

Schritt 2: Verwendung der @Value-Annotation

Mit der @Value-Annotation können Sie die Werte aus der Konfigurationsdatei in Ihre Beans injizieren.





Es können auch mehrere gleichzeitig verwendet werden.

Schritt 3: Verwendung konfigurierten Beans

AppConfig kann nun in anderen Komponenten verwendet werden, um auf die Werte zuzugreifen.



## Injection innerhalb von JUnit-Tests verwenden. (@ExtendWith)

## Erklären, warum die Verwendung eines Logging-Frameworks sinnvoll ist.

## Code erklären, der int-, float-, long- und Textblock-Literale verwendet.

# Vorlesung 13

Thread ist Aufgabe, die vom CPU ausgeführt werden muss. Dabei werden große Aufgaben immer in mehrere kleine Threads aufgeteilt.

Concurrent: Der CPU kann nicht gleichzeitig Threads ausführen. Simulierter Parallelism.

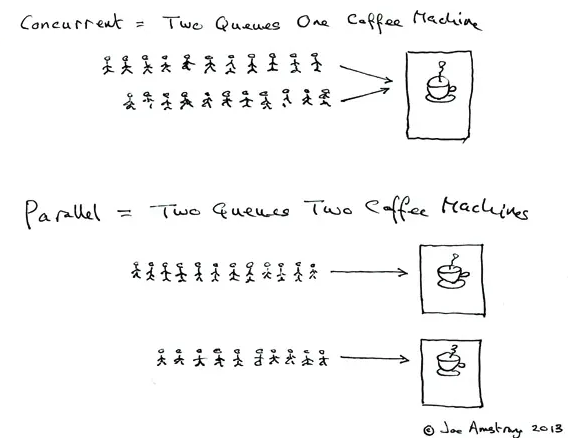
Stattdessen arbeitet er ein wenig an einem Thread und dann ein wenig an einem anderen Thread und so weiter. Dabei wird eine Aufgabe in kleinere Aufgaben aufgeteilt, die dann nach abwechselnd ausgeführt werden. So laufen mehrere Aufgaben scheinbar gleichzeitig.

Parallel: Mehrere Threads werden auf unterschiedlichen CPUs verarbeitet.

Mit mehreren CPU kann man auch mehrere Aufgaben parallel ausführen. Eine Aufgabe kann auch auf mehrere CPUs verteilt werden. Zudem können mehrere Aufgaben auf mehreren CPUs laufen, also 4 auf zwei. Dann findet Parallel und Concurrent gleichzeitig statt.

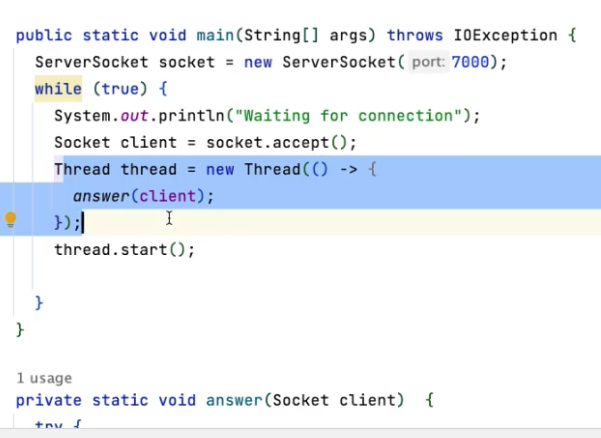
Demnach ist das folgende nöglich:

* Concurrent not Parallel
* Parallel not Concurrent
* Concurrent and Parallel
* Neither -> so kleine Aufgabe, dass sie nicht aufgeteilt wird



SingleThread Methoden: Schlecht, da erst eine Anfrage beendet werden muss, bevor andere bearbeitet wird. Bei Websiten wäre das schlecht, da User dann angewiesen wären auf Anzahl anderer User vor Ihnen und deren Last.

## Thread mit runnable



Für jede Anfrage wird mit dieser Syntax dann ein eigener Thread erstellt.

Der runnable ist dabei der Lambda Ausdruck (supplier), welcher die Methode ausführt die ausgeführt werden soll.

### Synchronized:

* synchronized bekommt Objekt: synchronized(object){...}
* Wird in run Methode erstellt
* Der Thread darf solange nicht auf andere Synchronized Blöcke zugreifen, bis der laufende fertig ausgeführt wurde
* Threads ohne synchronized können gleichzeitig ausgeführt werden
* Dadurch erreichen wir **Konsistenz** innerhalb von Threads
* Instanzmethode, die als synchronized deklariert ist, ist äquivalent zu einem synchronized(this)-Block

## Erklären, welche Probleme es in nebenläufigen Kontexten geben kann.

## Durch Kapselung und Synchronisierung Invarianten auch in nebenläufigen Kontexten sicherstellen.

## Aggregate, Entitäten und Wert-Objekte erklären, voneinander abgrenzen und implementieren.

## KI-Tools zur Softwareentwicklung einsetzen.