Karlsruher Institut für Technologie

9. Tutorium
Java-API, Prinzipien der ObjektOrientierung

Tutorium 14

Péter Bohner | 18.01.2023

Inhaltsverzeichnis

Wiederholung

Welche der Aussagen treffen auf Interfaces zu?

- Definieren eine Schnittstelle
- Definieren eine Sammlung von Methodensignaturen
- Eine Klasse kann mehrere Interfaces implementieren
- Ein Interface kann von beliebig vielen anderen Interfaces erben

Was stellt eine Exception dar?

Eine Ausnahme

Mit welchen Schlüsselwörtern wirft man in Java eine Exception?

```
throw new ExceptionName();
```

Wiederholung

Wie fange und behandle ich eine Exception?

```
try {
    // code
} catch (ExceptionType e) {
    // exception-handling
}
```

Welche der Aussagen treffen idealerweise auf Exceptions zu?

- Steuern **niemals** den Kontrollfluss
- Stellen die Ausnahme dar
- Fangen vorhersehbare und behandelbare Fehler ab
- Trennen Programmlogik und Fehlerbehandlung sauber voneinander

Wiederholung

Welche der Aussagen treffen auf Rekursive-Methoden zu?

- Speicher-/Zeitaufwändiger als iteratives Äquivalent
- Termination kann nicht vorausgesagt werden
- Vorteilhaft, falls Anzahl der Iterationen noch nicht bekannt
- Falls bekannt, besser iterativ

Was passiert in Java, wenn eine Methode aufgerufen wird?

- Lokale Variablen (inkl. Paramter),
- Operanden,
- Rücksprungadresse

  ... werden auf den Aufrufstapel geschrieben

Wie deklariert man in Java die Verwendung eines Generics `T` in einer Klasse?

```
public class ClassName<T> { ... }
```

Wie gibt man den konkreten Datentyp bei der Instanzierung einer Klasse die Generics nutzt an?

```
ClassName<String> objectName = new ClassName<>();
```

Übungsblatt 4

Allgemein

- A 9 und B 15 Abgaben
- Die meisten sollten ihren Übungsschein haben
- **Logik Hardcoden bringt nichts!** *Wieso muss ich das beim 4. ÜB noch sagen?*
  Wenn ihr z.B. den BFS nicht hin bekommt, wird das mit den Abschlussaufgaben schwer.

Code Stil

- Leere Zeilen zwischen Abschnitten (oder besser in Hilfsmethoden auslagern)
- Bedingung und Block nicht auf eine Zeile.
- *Haut einfach den Auto-Formatter vor der Abgabe drauf*
- *Achtet auf die Rechtschreibung, vermeidet Stilblüten*

Übungsblatt 4

Javadoc

- Beschreibt die Funktionalität von Klassen und (öffentlichen) Methoden / Attribute
- Nur mit Hilfe der Javadoc soll klar sein, was genau das dokumentierte Objekt soll
- Deshalb: (in-) Valide Parameter (Kombinationen) beschreiben
- Beschreiben, was in Randfällen, Fehlerzuständen passiert
- Welche Exceptions werden wann geworfen?

Übungsblatt 4 - Aufgabe A

Trennung von UI und Model

- Was gehört in die UI?
- Ein und Ausgabe (System.out, Scanner)
- Das Parsen von Eingaben
- Die Serialisation von Daten, die Ausgegeben werden sollen
- Was gehört ins Model?
- Die Modellierung des wahren Zustandes und Logik
- **Keine Ein und Ausgaben!**
- Das Model sollte keine invaliden Daten darstellen (nur richtig übertragenes Terrain sollte auch in einem Terrain-Objekt gespeichert werden)

Übungsblatt 4

Häufige Fehler in Aufgabe A

- Code-Duplication statt Hilfsmethoden parametrisieren
- UI und IO vermischen
- Schlechte OO-Modellierung
- Kontrollfluss mit Exceptions

Häufige Fehler in Aufgabe B

- Unvollständige JavaDoc (z.B. @throws fehlt)
- Falscher Exception-Typ

Übungsblatt 4 - Aufgabe B Exceptions

NullPointerException

Thrown when an application attempts to use null in a case where an object is required.

IllegalStateException

Signals that a method has been invoked at an illegal or inappropriate time. In other words, the Java environment or Java application is not in an appropriate state for the requested operation.

IllegalArgumentException

Thrown to indicate that a method has been passed an illegal or inappropriate argument.

Übungsblatt 4 - Aufgabe B Exceptions

Welche Exception passt am besten?

- `if(sweets == null) throw new Something();`
  `if(sweets == null) throw new NullPointerException("sweets Argument may not be null.");`
- `if(day < 0 || day > NUMBER_OF_DAYS) throw new Something();`
  `if(day < 0 || day > NUMBER_OF_DAYS) throw new IllegalArgumentException("Day is out of range.");`
- `if(!canOpenDoor(day)) throw new Something();`
  `if(!canOpenDoor(day)) throw new IllegalStateException("The door may not be opened yet");`
- Anmerkung: Error messages bitte als Konstanten

JavaDoc - Was ist das?

Javadoc...

- ... beschreibt Klassen, Methoden, Konstuktoren und Felder
- ... wird direkt in den Code geschreiben
- ... erleichtert das Verständnis

Regeln und Ziele

- **einheitlich** auf Deutsch oder auf Englisch
- vollständige und aussagekräftige Beschreibung

`Javadoc zur Java API`

JavaDoc für Klassen

- beschreibt, welchen Zweck die Klasse erfüllt und welche Funktionalität sie aufweist
- mögliche Verwendungszwecke können erwähnt werden
- der Autor wird angegeben
- die Versionsnummer wird angegeben

Syntax

```
/**
 * Kurze aber aussagekräftige Beschreibung der Klasse.
 * @author Autor
 * @version versionsNummer
 */
class ClassName {
```

JavaDoc für Klassen - Beispiel

```java
/**
 * Modelliert einen zweidimensionalen Vektor.
 * Der Vektor besteht aus einer x- und y-Komponente.
 * @author Peter Bohner
 * @author uqknc
 * @version 1.0
 */
class Vector2D {
    double x;
    double y;
}
```

JavaDoc für Konstruktoren

- beschreibt Zweck der Konstruktors
- beschreibt, in welchem Kontext der Kontruktor verwendet werden kann
  - Spezialfälle, die das Verhalten betreffen
- alle Parameter werden beschrieben (Zweck und eventuell Wertebereich)

Syntax

```
/**
 * Kurze aber aussagekräftige Beschreibung von Zweck und Kontext des Konstruktors.
 * Sonderfälle werden beschrieben.
 * @param parameter Beschreibung des Parameters
 */
KlassenName(Datentyp parameter, ...) {
```

JavaDoc für Konstuktoren - Beispiel

```java
/**
 * Konstruiert einen zwei dimensionalen Vektor mit zwei übergebenen Komponenten.
 *
 * @param x die x-Komponente des Vektors
 * @param y die y-Komponente des Vektors
 */
Vector2D(double x, double y) {
    this.x = x;
    this.y = y;
}
```

Exceptions - Einführung

Exception

- eine *Ausnahme*
- Zur Laufzeit des Programms
- Zur Unterbrechung des normalen Kontrollflusses

Verwendung einer Exception

- Ein *Problem* tritt auf
- Normales Fortfahren nicht möglich
- Lokale Reaktion darauf nicht sinnvoll/möglich
- Behandlung des Problems an anderer Stelle nötig

Exceptions

Exceptions in Java

Ausnahme in Java

- echtes Objekt (Methoden, Attribute, . . .)
- Von Klasse `Exception` abgeleitet
- Mindestens zwei Konstruktoren: `Default` & mit `String`-Parameter (mit zusätzlichen Informationen)
- Methoden: `getMessage()` & `printStackTrace()`
- Erzeugung mit **`new`**
- Auslösen mit **`throw`**

Exceptions - Beispiel

```java
public void setMonth(int month) {
    if ((month < 1) || (month > 12)) {
        throw new IllegalArgumentException(
                "Wrong month: " + month);
    }
    this.month = month;
}
```

Exceptions - Arten von Fehlern

Error

(Katastrophale) Probleme, die eigentlich nicht auftreten dürfen.
Speicher voll, Illegaler Byte-Code, JVM-Fehler, ...

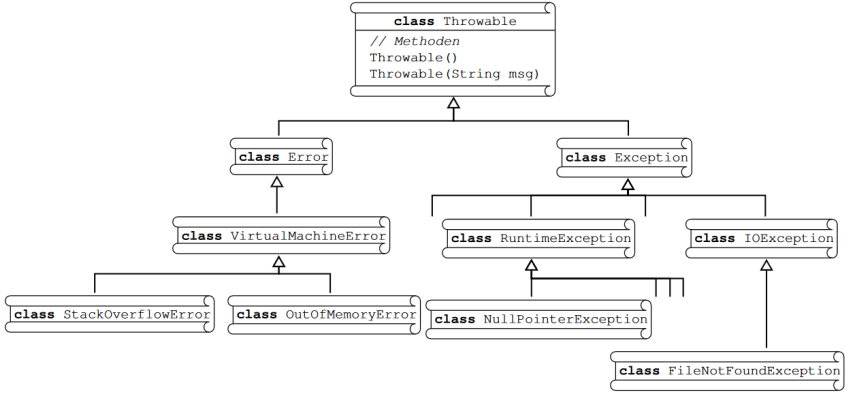
`RuntimeException`

Durch *fremde* Fehler erzeugte Probleme.
falsche Benutzung einer Klasse, Programmierfehler

Geprüfte Exception (*checked Exception*)

Vorhersehbare und behandelbare Fehler.
Datei nicht vorhanden, Festplatte voll, Fehler beim Parsen, ...

Exceptions - Hierarchie



Exceptions

Ausnahmebehandlung in Java

```
try {
    // hier koennte eine Exception auftreten
} catch (ExceptionType1 e) {
    // Fehlerbehandlung fuer ExceptionType1
} catch (ExceptionType2 e) {
    // Fehlerbehandlung fuer ExceptionType2
}
```

Fall-through, die Zweite

- Java ruft den ***ersten*** passenden catch Block auf!
- Alle weiteren werden *ignoriert*

Exceptions

Beispiel

```
try {
    FileReader fr = new FileReader(".test");
    int nextChar = fr.read();
    while (nextChar != -1) {
        nextChar = fr.read();
    }
} catch (FileNotFoundException e) {
    System.out.println("Nicht gefunden.");
} catch (IOException e) {
    System.out.println("Ooops.");
}
```

Exceptions

Exception Handler (= `catch`-Block)

- Behandlung einer Ausnahme
- an einer Stelle
- irgendwo im Aufrufstack
- getrennt von normalen Programmcode

Catch or specify

Jede ausgelöste geprüfte (checked) Exception muss

- behandelt (Exception Handler) oder
- deklariert (`throws`)

werden.

Exceptions

Deklaration von Ausnahmen

- Deklaration im Methodenkopf:
  ```
  private String readFile(String filename) throws IOException, FileNotFoundException {}
  ```
- Aufrufer muss sich um Exception kümmern
- `throws` ist Teil der Signatur (Vorsicht beim Überschreiben)
  Exceptions können in überschriebenen Methoden weggelassen werden, aber nicht hinzukommen.
- Nicht deklarationspflichtig sind `RuntimeException` & `Error` (sowie deren Unterklassen)
- Jede Exception mittels @throws im Javadoc beschrieben werden

Anmerkung: Da `IOException` Oberklasse von `FileNotFoundException` ist, müsste letzteres nicht extra deklariert werden. Dokumentationszwecke!

Exceptions

Ort der Behandlung

Finden der passenden Ausnahmebehandlung:

- Suche im Aufrufstack nach umgebenden `try-catch`-Blöcken, gehe zu erstem passenden `catch`-Block
- Nach der Behandlung: Fortsetzung am Ende des `try-catch`-Block

Exceptions - Konventionen

Behandlung?

- **Error und Unterklassen:** Nein, nicht sinnvoll behandelbar
- **Exception:** Nein, viel zu allgemein
- **RuntimeException:** Prinzipiell Nein
- **dessen Unterklassen:** Programmierfehler beheben! (Ausnahme: `NumberFormatException`)
- **Andere:** Ja, wenn sinnvoll behandelbar
- `try`-Block so klein wie möglich halten

Exceptions - Konventionen

Werfen?

- **Error:** Nein.
- **Exception:** Niemals, nur als eigene Unterklasse
- **RuntimeException:** Ja, eigene (semantisch passende) Unterklasse

Beispiel

```
if ((month < 1) || (month > 12)) {
    throw new IllegalArgumentException(
            "Wrong month: %s", month);
}
switch (month) {
    case 1: break; // ...
    default: throw new Error();
}
```

Exceptions - Konventionen

Verwendung

Exceptions sollen:

- zur Vereinfachung dienen
- die absolute Ausnahme darstellen
- mittels `@throws` im Javadoc beschrieben werden
- NICHT den normalen Kontrollfluss steuern

Böse!

```
try {
    while (character != array[i]) { i++; }
} catch (Exception e) {
    System.out.println("Element nicht gefunden.");
}
```

Exceptions - Konventionen

Verboten!

- `try`-Block um das ganze Programm
- Leerer `catch`-Block
- Explizites Fangen des Typs `Exception`
- Explizites Fangen des Typs `Throwable`

Exceptions

Eigene Exceptions

- Ableiten einer eigenen Unterklasse von Exception oder RuntimeException
- Implementierung der zwei Standard-Konstruktoren
- Definition einer eigenen, sinnvollen Exception-Hierarchie (bei Bedarf)
- Verwendung von vorhandenen Exceptions nur für dafür vorgesehene Zwecke (Javadoc anschauen)

Beispiele in der Java-API

- `IllegalArgumentException`
- `IllegalStateException`
- `UnsupportedOperationException`
- `NullPointerException`

Exceptions - Zusammenfassung

Ausnahmen

- werden ausgelöst (throw) und behandelt (try-catch) oder
- deklariert (throws)
- sollen die Ausnahme bleiben
- trennen sauber Programmlogik und Fehlerbehandlung

Fehlererkennung

- so früh wie möglich
- defensiv
- mittels if Exceptions

Java-API - Application Programming Interface

- Sammlung von Klassen und Paketen
- Enthalten häufig verwendete Funktionalität

Bekannte Klassen

- `Object`
- `String`, `Math`, `Enum`
- `Comparable`, `Integer`, `Double`, ...

Pakete

- `java.lang` Basisfunktionen (oben genannte Klassen)
- `java.util` Java Collections, Zeit-/Datumsfunktionen
- `java.io` Ein-/Ausgabe

Java-API - Collection

Das Interface `java.util.Collection<E>`

- `java.util.List`
- `java.util.Set`
- `java.util.SortedSet`
- `java.util.Queue`

Wichtige Methoden

- `boolean contains(Object element)`
- `boolean add(E element)`
- `boolean remove(Object element)`

Java-API - Map

Das Interface `java.util.Map<K, V>`

- Paare werden eindeutig nach ihrem `Key` gespeichert
- Ein `Key` darf nur einmal enthalten sein

Wichtige Methoden

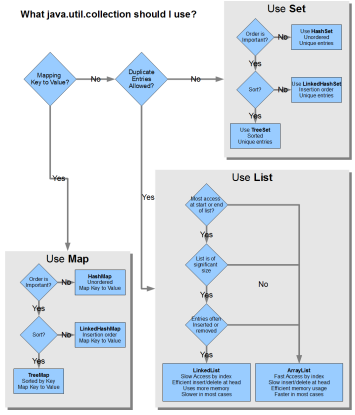
- `V put(K key, V value)`
- `V get(Object key)`
- `V remove(Object key)`
- `boolean containsKey(Object key)`
- `boolean containsValue(Object value)`

Java-API - Map

Beispiel

```java
public static void main(String[] args) {
    Map<Integer, Integer> intMap = new HashMap<Integer, Integer>();
    intMap.put(5, 20);
    intMap.put(10, 30);
    if (intMap.containsKey(5)) {
        System.out.println(intMap.get(5)); // 20
    }
}
```

Java-API - Welche Collection nehme ich wann?



Quelle: `https://i.stack.imgur.com/aSDsG.png`

Java-API - Reguläre Ausdrücke

`java.util.regex` und seine Klassen

- Pattern (`java.util.regex.Pattern`)
- Matcher (`java.util.regex.Matcher`)

```
String positiveIntegerPattern = "\\+?[1-9]\\d*";
Pattern pattern = Pattern.compile(positiveIntegerPattern);
Matcher matcher = pattern.matcher("+529");
boolean isMatch = matcher.matches(); // true
```

Java-API - Reguläre Ausdrücke

Reguläre Ausdrücke werden wie in GBI gebildet

Character

- Klassen
  - `\d` Ziffer
  - `\w` Zeichen
  - `[a-g]` Ein Zeichen von a-g, `[1-9]` Ziffer von 1-9
- Quantoren
  - `*` Ausdruck davor 0 mal oder mehr, + Ausdruck davor mindestens 1 mal
  - `?` Ausdruck davor 0 oder 1 mal
  - `a{5}` 5 mal a

Reguläre Ausdrücke testen: `https://regexr.com/`

Aufgabe

Pattern

Schreibe jeweils für die folgenden Muster einen regulären Ausdruck, sodass diese mit Hilfe der Klasse `Pattern` von Java erkannt werden würden.

5-stellige positive Ganzzahl (ohne +)	\d{5}
Ganzzahl (mit + und -)	(+|-)?\d*
kleingeschriebenes Wort, anschließend Doppelpunkt	\w+:
positive Kommazahl (mit + und -)	(+|-)?\d+(,\d+)?

Java-API - Wichtige Links

- **Allgemein**

  `https://docs.oracle.com/en/java/javase/11/docs/api/index.html`

- `java.util.Collection<E>`

  `https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Collection.html`

- `java.util.Map<K, V>`

  `https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Map.html`

- `java.util.regex`

  `https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/regex/package-summary.html`

Objektorientierte Design-Prinzipien

OO-Design-Prinzipien

1. Datenkapselung
2. Komposition vor Vererbung
3. Gegen Schnittstellen programmieren
4. `SOLID`-Prinzipien
5. Guter Schnittstellenentwurf

Datenkapselung

- private Attribute
- Entwurfsentscheidung hinter Schnittstelle verdecken
- Keine Objektattribute in gettern direkt zurückgeben
  - Kopie von Collection erstellen und diese zurückgeben

Komposition vor Vererbung

- Komposition
  - Verwendung von Klassen als Attributtypen
  - Bestehende Objekte werden zusammengesetzt, um Funktionalität zu erweitern
- Vererbung
  - Ableitung von Eigenschaften und Funktionalität
  - Implementierung wird erweitert
- Komposition sollte vor Vererbung genutzt werden, insofern das möglich ist
- Beispiel Klasse `AccessControl`
  - Attribut vom Typ `Map` zur Nutzerverwaltung
  - Man könnte auch `AccessControl` von `HashMap` ableiten und somit die Funktionalität zur Verfügung stellen
  - Ist aber unübersichtlich und verkompliziert den Code
  - Verletzt außerdem die is-a-Beziehung: `AccessControl` ist keine `HashMap`

Gegen Schnittstellen programmieren

Nicht gegen die Implementierung programmieren!

- Leichte Austauschbarkeit
- Bessere Wiederverwendbarkeit

```
public int getStretchedCollectionSize(Collection<?> collection) {
    return collection.size() * this.factor;
}
```

SOLID-Prinzipien

- Jede Klasse sollte eine Verantwortung haben
- Open-Close Prinzip (nächste Folie)
- Eine Instanz einer Unterklasse sollte sich so verhalten, dass sie jederzeit auch als Instanz der Oberklasse verwendet werden könnte
- Klassen sollten durch ein Interface nicht unnötige Methoden implementieren müssen $\Rightarrow$ Interface aufteilen
- Abstraktes darf nicht von Details abhängen

Abstraktes darf nicht von Details abhängen

Schlecht

```java
public class UserDatabase {
    LinkedList<User> users;

    public UserDatabase() {
        this.users = new LinkedList<>();
    }
}
```

Gut

```java
public class UserDatabase {
    List<User> users;

    public UserDatabase() {
        this.users = new LinkedList<>();
    }
}
```

Open-Close Prinzip

- Offen für Erweiterung ⇔ Geschlossen für Veränderung
- Programmmodule sollen nicht verändert werden müssen
- Erweiterbaren Code schreiben
- Code soll nicht verändert werden

Schnittstellenentwurf

- Implementierungsdetails verbergen
- Konsistentes und angemessenen Abstraktionsniveau
- Generelle Schnittstellen verwenden (`void addUsers(Collection<User> users)`)

Weitere Prinzipien

- Niemals gleichzeitig in einer Methode Objekt-Zustand zurückgeben und ändern
- **Idempotente** Methoden schreiben (Hintereinanderausführung liefert konsistente Ergebnisse)
- Richtige Methodennamen wählen (Beschreiben das Verhalten der Methode und sind konsistent)

Bis zum nächsten Tutorium am 25.01.2022.