Karlsruher Institut für Technologie

8. Tutorium

Exceptions, Interfaces, Iteratoren, Rekursion, Java API

Tutorium 14

Péter Bohner | 11.01.2022

Inhaltsverzeichnis

Vererbung und dynamische Bindung

```java
public class Mutter {
    private String name;
    public Mutter(String name) {
        this.name = name;
    }
    public void sageEtwas(int kuchenMenge) {
        System.out.println(kuchenMenge + " Stücke Kuchen reichen!");
    }
}
```

```java
public class Oma extends Mutter {
    public Oma(String name) {
        super(name);
    }
    public void sageEtwas(int muffinMenge) {
        System.out.println("Nimm " + muffinMenge + " Muffins!");
    }
}
```

Welche Zeile ist nicht gültig?

```java
Mutter mutter = new Mutter("Julia");
Mutter mutter2 = new Oma("Jutta");
Oma oma1  = new Mutter("Julia");
Oma oma2 = new Oma("Jutta");
```

Was wird ausgegeben?

```java
mutter.sageEtwas(2);          // 2 Stücke Kuchen reichen!
mutter2.sageEtwas(2);         // Nimm 2 Muffins!
oma2.sageEtwas(2);            // Nimm 2 Muffins!
```

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, ...

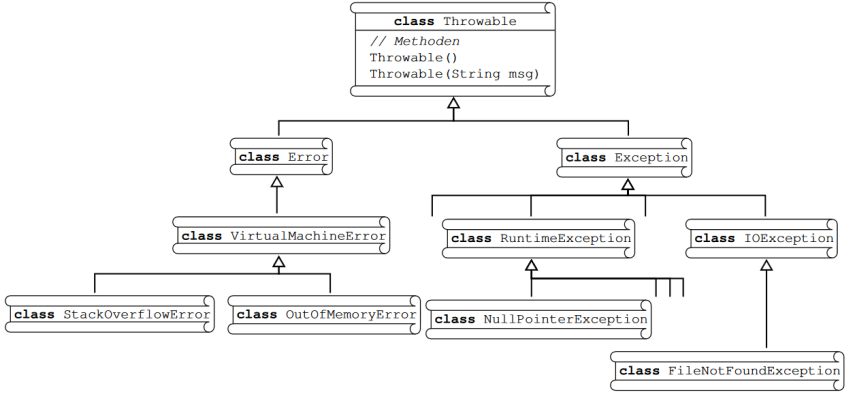
`RuntimeException`

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

Was sind Interfaces?

- **Interface** oder auch *Schnittstelle*
- Fast wie abstrakte Klassen
- Beschreiben ein bestimmtes Verhalten
- Definieren eine Sammlung von Methodensignaturen
  - Diese können von anderen Klassen implementiert werden
  - Aber nicht direkt im Interface
- Interfaces werden zur **Typisierung** verwendet
  - Klasse A, die das Interface I implementiert, ist ein Subtyp von I
  - Alle Klassen, die I implementieren, können nun an Stellen verwendet werden, an denen die Funktionalität der Schnittstelle erwartet wird

Was sind Interfaces?

- Eine Klasse, die ein Interface implementiert, implementiert alle Methoden, die im Interface definiert wurden (oder macht sie abstrakt)
- Eine Klasse kann **mehrere** Interfaces **implementieren**
- Ein Interface kann von **mehreren anderen** Interfaces erben (mittels `interface`)
- Interfaces können nicht instanziiert werden

Was sind Interfaces?

Karlsruher Institut für Technologie

Beispiel

```java
public interface Printable {
    /**
     * Prints the object's state
     */
    public void print();
}
```

Syntax

- `interface Name` definiert ein Interface
- Methoden werden wie sonst auch definiert, allerdings:
- Geschweifte Klammern und Rumpf weglassen
- Semikolon nicht vergessen

Was sind Interfaces?

Karlsruher Institut für Technologie

Beispiel

```
public class Person implements Printable {
    private String forename;
    private String lastname;

    public Person(String forename, String lastname) {
        this.forename = forename;
        this.lastname = lastname;
    }

    @Override
    public void print() {
        System.out.println(forename + " " + lastname);
    }
}
```

Syntax

- `class Classname implements InterfaceA, InterfaceB` implementiert ein Interface
- Methoden der Schnittstelle werden mit `@Override` annotiert

Was sind Interfaces?

Karlsruher Institut für Technologie

Beispiel

```
public class Main {
    public static void main(String[] args) {
        Printable p = new Person("Max", "Müller");
        p.print();
    }
}
```

Syntax

- `Interfacename objectname = new Classname();` erstellt ein Objekt vom Typ des Interfaces
- Alle Methoden der Schnittstelle können darauf aufgerufen werden

Bereits existierende Interfaces

Es existieren bereits einige vorgefertigte Interfaces der Java-API, die mit Klassen der API kompatibel sind. Folgende Interfaces **muss**/**sollte** man kennen:

- **Comparable**
  (`https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Comparable.html`)
  - Ermöglicht einfaches Vergleichen von Objekten einer Klasse (zum Beispiel zum Sortieren)
- **Iterable**
  (`https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Iterable.html`)
  - Klassen die das Interface implementieren, ermöglichen eine Schnittstelle zu einem Iterator

⇒ Mit Interfaces werden Verhaltensweisen festgelegt, weshalb Namen der Interfaces oft auf *-able* enden

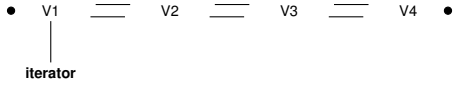
Iteratoren

- **Iterieren**: Direkt hintereinander auf alle Elemente einer Datenstruktur zugreifen
- Über Arrays iteriert man mit einer for-Schleife

```
for (int i = 0; i < a.length; i++) {
    a[i] = 10;
}
```

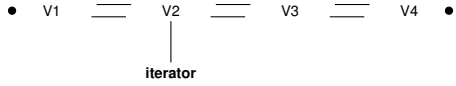
- Über verkettete Liste ist das nicht so leicht möglich
- Deshalb gibt es den Iterator
- **Iterator**: Zeiger, der nacheinander über alle Elemente einer Datenstruktur iteriert

Initialisieren



- Iterator initialisieren
- Er zeigt auf das erste Element der Liste
- Knotenelement kann abgefragt werden

Iterieren



- Ein Element weitergehen
- Er zeigt auf das zweite Element der Liste
- Knotenelement kann abgefragt werden

Iteratoren

Implementierung

- Könnt ihr Euch in der VL anschauen
- Hauptsache, ihr habt das Prinzip verstanden

Bibliotheken nutzen

- `import java.util.Iterator;`
- Iterator<Datentyp> iterator = list.iterator();
- Methoden:

  `boolean hasNext()`

  `Datentyp next()` (Gibt aktuelles Element zurück und springt zum nächsten)

  `void remove()`

Anwendung

```java
import java.util.LinkedList;
import java.util.Iterator;
public class IteratorApp {
    public static void main(String[] args) {
        LinkedList<Integer> intList = new LinkedList<Integer>();
        for (int i = 0; i < 10; i++) {
            intList.add(i);
        }
        Iterator<Integer> iterator = intList.iterator();
        while (iterator.hasNext()) {
            System.out.println(iterator.next());
        }
    }
}
```

Was sind Generics?

- *generisch*: auf nichts Spezifisches Bezug nehmen
- $\Rightarrow$ Allgemeine Typen verwenden
- Damit können Listen allgemein implementiert werden

Beispiel

- `java.util.LinkedList<E>`
  - Das **E** steht hierbei für einen generischen Typen
  - Instanzen der Klasse **E** können zu der Liste hinzugefügt werden

Anwendung

```java
public class Pocket<T> {
    private T content;

    public T getContent() {
        return this.content;
    }

    public void setContent(T content) {
        this.content = content;
    }
}
```

```java
public class PocketTest {
    public static void main(String[] args) {
        Pocket<String> stringPocket = new Pocket<>();
        stringPocket.setContent("Test");
        System.out.println(stringPocket.getContent());
    }
}
```

Syntax

- Erstellen
  - `class Name<Typ-Parameter> {}`
  - `interface Name<Typ-Parameter> {}`
- Verwenden
  - `Name<Typ> objectName`
- Beispiel
  - `class LinkedList<E> {}`
  - `LinkedList<Point> points = new LinkedList<Point>();`

Namenskonventionen

Es gibt verschiedene Platzhalter, z.B. `class Name<T>`
Wofür steht das T?

- **T**: Typ
- **E**: Element
- **K**: Schlüssel (key)
- **V**: Wert (value)

Wildcard

Wenn der generische Typ beispielsweise nur Unterklassen einer bestimmten Klasse sein soll:

- Obere Schranke
  - `<(? extends C)>`
  - Instanzen von Typ C oder spezifischer
- Untere Schranke
  - `<(? super D)>`
  - Instanzen von Typ D oder allgemeiner

Generische Methoden

Beispiel

```java
import java.math.BigInteger;

public class GenericMethods {
    public static <T extends Number> int addThree(T number) {
        return number.intValue() + 3;
    }

    public static void main(String[] args) {
        System.out.println(addThree(2)); // 5
        System.out.println(addThree(2.3)); // 5
        System.out.println(addThree(BigInteger.TEN)); // 13
    }
}
```

Bekannte generische Interfaces

- **Comparable<T>**
  (`https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Comparable.html`)
  - Ermöglicht einfaches Vergleichen von Objekten einer Klasse (zum Beispiel zum Sortieren)
- **Iterable<T>**
  (`https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Iterable.html`)
  - Klassen die das Interface implementieren, ermöglichen eine Schnittstelle zu einem Iterator

Aufgabe

Generische Liste

Im ILIAS findet ihr die Klasse `StringList`. Erweitert diese Klasse zu einer **generischen Klasse**, mit der man beliebige Datentypen in der Liste speichern kann.

Aufgabe

Anwendung zu Interfaces und Exceptions

Ladet Euch die Klasse `NumericalSequence` (Zahlenfolge) aus dem GIT herunter. Die Klasse soll um zwei Funktionalitäten erweitert werden:
Einerseits soll es eine neue Methode geben, die ein neues Element am Ende der Zahlenfolge einfügt. Ist die übergebene Zahl negativ, soll eine **IllegalArgumentException** geworfen werden. Andererseits soll die Klasse das Interface **Comparable** (siehe Java-API) implementieren. Verglichen wird die Summe aller Elemente der Zahlenfolge (siehe Methode `getSum()`). Je kleiner die Summe, desto weiter vorne soll das Objekt einsortiert werden.

Tipp

Bedenkt, dass **Comparable** generisch ist. Es muss mit `Comparable<Objectname>` implementiert werden!

Rekursion - Einführung

Divide and Conquer-Prinzip

Um ein Problem zu lösen, teile es in mehrere Teilprobleme nach dem „Divide and Conquer-Prinzip“ („teile und herrsche“).

Prinzip

Führe denselben Berechnungsablauf immer wieder mit kleineren Eingabedaten aus, bis die Eingabe lösbar wird. Danach werden die Teilergebnisse wieder zusammengeführt.

Umsetzung $\Rightarrow$ Eine Methode, die sich direkt oder indirekt immer wieder selbst aufruft.

Rekursion - Beispiel Fakultät

Erinnerung: $n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$
Beispiele: $3! = 3 \cdot 2 \cdot 1 = 6$ und $0! = 1 = 1!$

Rekursive Schreibweise:

$$n! = \begin{cases} n \cdot (n-1)! & \text{, falls } n > 0 \\ 1 & \text{, falls } n = 0 \end{cases}$$

Java:

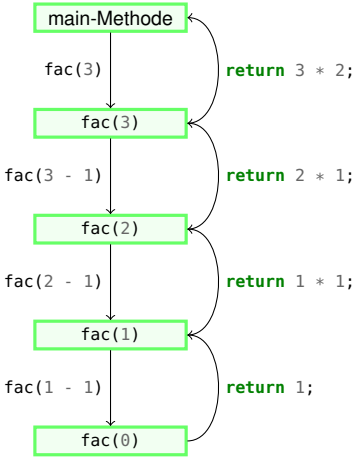
```java
public static int fac(int n) {
    if (n > 0) {
        return n * fac(n-1);
    }
    return 1;
}
```

Rekursion

Lokale Variable (und Parameter) einer Methode werden im Aufrufstapel (call stack oder Laufzeitkeller) abgelegt.
Für jede Methode wird ein neuer Speicherbereich (Schachtel bzw. frame) auf dem Aufrufstapel angelegt mit u.a.

- Lokalen Variablen(inkl. Parameter)
- Operanden
- Rücksprungadresse

Rekursion




```java
public static void main(String args[]) {
    int result = fac(3);
}
public static int fac(3) {
    if (3 > 0) {
        return 3 * fac(3 - 1);
    }
    return 1;
}
public static int fac(2) {
    if (2 > 0) {
        return 2 * fac(2 - 1);
    }
    return 1;
}
```

Rekursion vs. Iteration

Rekursion

- Vorteilhaft, wenn die Anzahl der Iterationen noch unklar
- Viele Methodenaufrufe
  → Zeitaufwendiger
- Belegen des Stacks für die Werte der aktuellen und lokalen Variablen
  → Speicheraufwendiger
- Nicht auf dem ersten Blick abschätzbar, ob sie terminiert.

Iteration

- Vorteilhaft, wenn die Anzahl der Iterationen bekannt ist.
- Komplexität leichter abschätzbar
- Leichter abschätzbar, ob die Iteration terminiert.

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

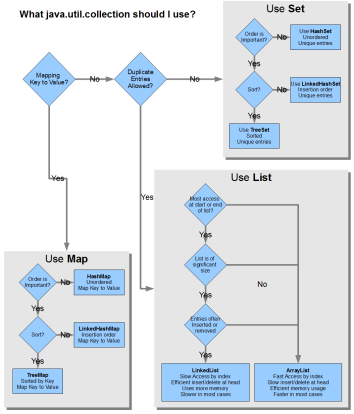
- `V put(K key, V value)`
- `V get(Object key)`
- `V remove(Object key)`
- `boolean containsKey(Object key)`
- `boolean containsValue(Object value)`

Java-API - Map

Beispiel

```
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

Bis zum nächsten Tutorium am 18.01.2023.