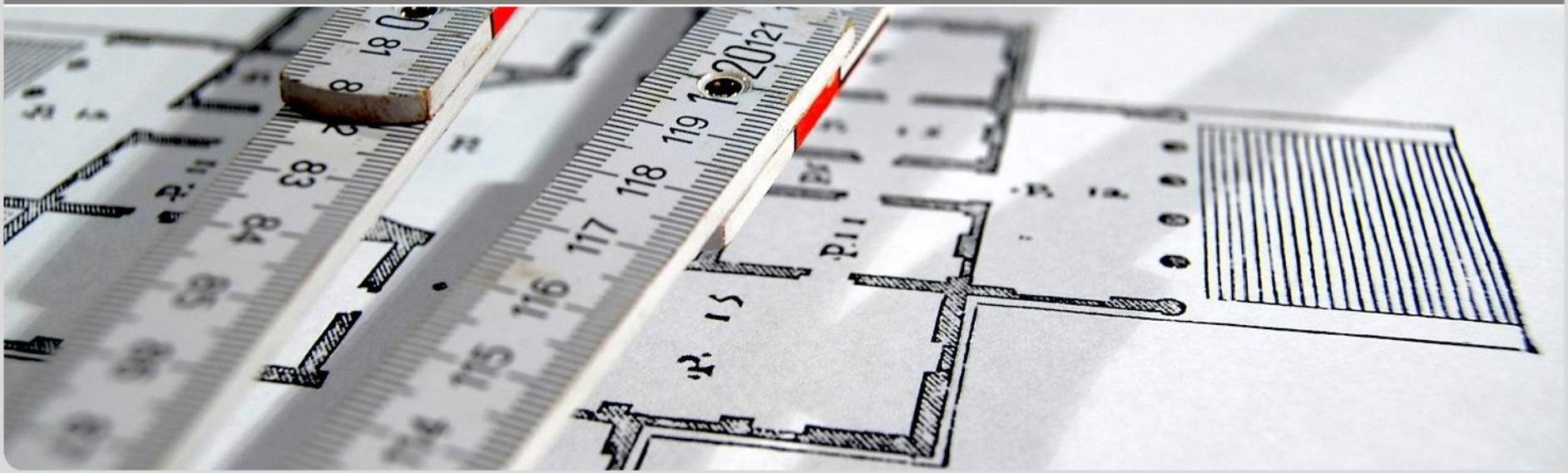
Karlsruher Institut für Technologie

Programmieren-Tutorium Nr. 10

10. Tutorium | Jonas Ludwig

Generic Interfaces, Exceptions, Rekursion

Architecture-driven Requirements Engineering – Institut für Programmstrukturen und Datenorganisation – Fakultät für Informatik



Was machen wir Heute?

- Generics Interfaces
- Lambda Ausdrücke
- Exceptions
- Rekursion
- Softwarequalität / Programmierstil

Das Interface Comparable<T>

- Das Interface `Comparable<T>` definiert die Methode
  - `public int compareTo(T o)`
- und damit eine totale Ordnung auf Objekten vom Typ T
- Sollen Objekte z.B. sortiert werden, muss man sie miteinander vergleichen können. Dies wird am besten über eine einheitliche Schnittstelle gemacht:
  - < 0, falls this kleiner ist als other
  - == 0, falls this gleich groß ist wie other
  - > 0, falls this größer ist als other

- Beispiel: Sortierung von Integer

```
List<Integer> l = new ArrayList<Integer>();
l.add(18); l.add(46); l.add(18); l.add(12);
Collections.sort(l);
System.out.println(l); // [12,18,18,46]
```

Das Interface Comparable<T>

```java
/**
 * This interface imposes a total ordering on the
 * objects of each class that implements it. This
 * ordering is referred to as the class's natural
 * ordering, and the class's compareTo method is
 * referred to as its natural comparison method.
 */
interface Comparable<T> {

    /**
     * Compares this object with the specified
     * object for order. Returns a negative integer,
     * zero, or a positive integer as this object is
     * less than, equal to, or greater than the
     * specified object.
     */
    int compareTo(T o);
}
```

Das Interface Comparable<T>

- Beispiel-Implementierung des Interface bei gleichzeitiger Instanziierung:

```
public class Student implements Comparable<Student> {
   private final String name;
   private final int matrNr;

   // ...

   @Override
   public int compareTo(Student other) {
      int ret = 0;
      if (!this.equals(other)) {
         ret = this.name.compareTo(other.name);
         if (ret == 0) {
            ret = this.matrNr - other.matrNr;
         }
      }
      return ret;
   }
}
```

Das Interface Comparator<T>

- Das Interface `Comparator<T>` definiert die Methode
  - `public int compare(T o1, T o2)`
- und damit eine totale Ordnung auf Objekten vom Typ T
- Rückgabewert analog zu compareTo des Interfaces Comparable
  - < 0, falls o1 kleiner ist als o2
  - == 0, falls o1 gleich groß ist wie o2
  - > 0, falls o1 größer ist als o2

- Beispiel: Inverse Sortierung von Integer

```
class ReverseIntegerOrdering implements Comparator<Integer> {
   public int compare(Integer i1, Integer i2) { return i2 - i1; }
}

List<Integer> l = new ArrayList<Integer>();
l.add(18); l.add(46); l.add(18); l.add(12);
Collections.sort(l, new ReverseIntegerOrdering()); // [46,18,18,12]
```

Das Interface Comparator<T>

- Üblicherweise Verwendung mittels einer anonymen Klasse:

```
List<Integer> l = new ArrayList<Integer>();
l.add(18); l.add(46); l.add(18); l.add(12);

Collections.sort(l, new Comparator<Integer>() {
    public int compare(Integer i1, Integer i2) {
        return i2 - i1;
    }
});

System.out.println(l); // [46,18,18,12]
```

- Erzeugt eine anonyme Klasse, die `Comparator<Integer>` implementiert und instanziiert diese Klasse einmalig

Anonyme innere Klassen

- Anonyme Klassen gehen noch einen Schritt weiter als lokale Klassen
- Sie haben keinen Namen und erzeugen immer automatisch ein Objekt
  - Klassendeklaration und Objekterzeugung sind zu einem Sprachkonstrukt verbunden
- `new KlasseOderSchnittstelle() { /* ... */ }`
- In dem Block geschweifter Klammern lassen sich nun Methoden und Attribute deklarieren oder Methoden überschreiben

Lambda Ausdrücke

- Lambda-Ausdrücke sind eine kompakte Schreibweise zur Formulierung einer anonymen Klasse mit einer Methode
- Häufig benötigt um (funktionale) Schnittstellen zu implementieren, welche die Signatur einer Methode exakt definieren
- Erst seit Java 8:
  - http://cr.openjdk.java.net/~briangoetz/lambda/lambda-state-final.html
  - https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html

Lambda Ausdrücke – Syntax

- Parameterliste in runden Klammern gefolgt von einem Pfeiloperator und einem Ausdruck, welcher die Parameter verarbeitet

- (Parameterliste) -> Ausdruck

- Bei mehreren Kommandos sind die Ausdrücke von geschweiften Klammern umschlossen
- Wenn der Ausdruck ein Ergebnis liefert, muss es wie bei einer Methode mit `return` zurückgegeben werden

- (Parameterliste) -> {

```
        Ausdruck1;
        Ausdruck2;
        return Wert;
}
```

Lambda Ausdrücke – Beispiele 1

```
import java.util.ArrayList; import java.util.Collections;
import java.util.Comparator; import java.util.List;

public class Lambda {
   public static void main(String[] args) {
      List<Integer> list = new ArrayList<>();
      list.add(18); list.add(46); list.add(18); list.add(12);

      Collections.sort(list, new Comparator<Integer>() {
         @Override
         public int compare(Integer a, Integer b) {
            return b - a;
         }
      });

      Collections.sort(list, (i1, i2) -> i2 - i1);

      Comparator<Integer> reverseIntegerOrdering = (x, y) -> (y - x);
      Collections.sort(list, reverseIntegerOrdering);
   }
}
```

Lambda Ausdrücke

- Alle drei Aufrufe der Sortiermethode im Beispiel machen exakt das gleiche. Nur unterschiedlich mit:
  - Anonyme Klasse
  - Direkt übergebener Lambda Ausdruck
  - Lambda Ausdruck welcher zuvor einer Variablen zugewiesen wurde
- In den meisten Fällen kann der Java-Compiler aus dem Kontext den Datentyp der Parameter eines Lambda-Ausdrucks erkennen
- Darum ist es meist nicht erforderlich, eine Typbezeichnung voranzustellen

Lambda Ausdrücke – Gültigkeitsebenen

- Der Lambda-Ausdruck kann direkt auf Variablen zugreifen, die in derselben Codeebene zugänglich sind, in der er definiert wird
  - Anonyme Klassen können dies in der Regel nicht!

- Dabei gibt es allerdings eine wesentliche Einschränkung: Lokale Variablen müssen final deklariert sein oder sich zumindest so verhalten
  - Variablen dürfen nach ihrer ersten Zuweisung nicht mehr verändert werden, so das der Java-Compiler eine Deklaration mit final ohne Fehlermeldung akzeptieren würde

```
(final) int reverse = -1;
Comparator<Integer> integerOrdering  =
                                (i1, i2) -> reverse * (i1 - i2);
Collections.sort(list, integerOrdering);
```

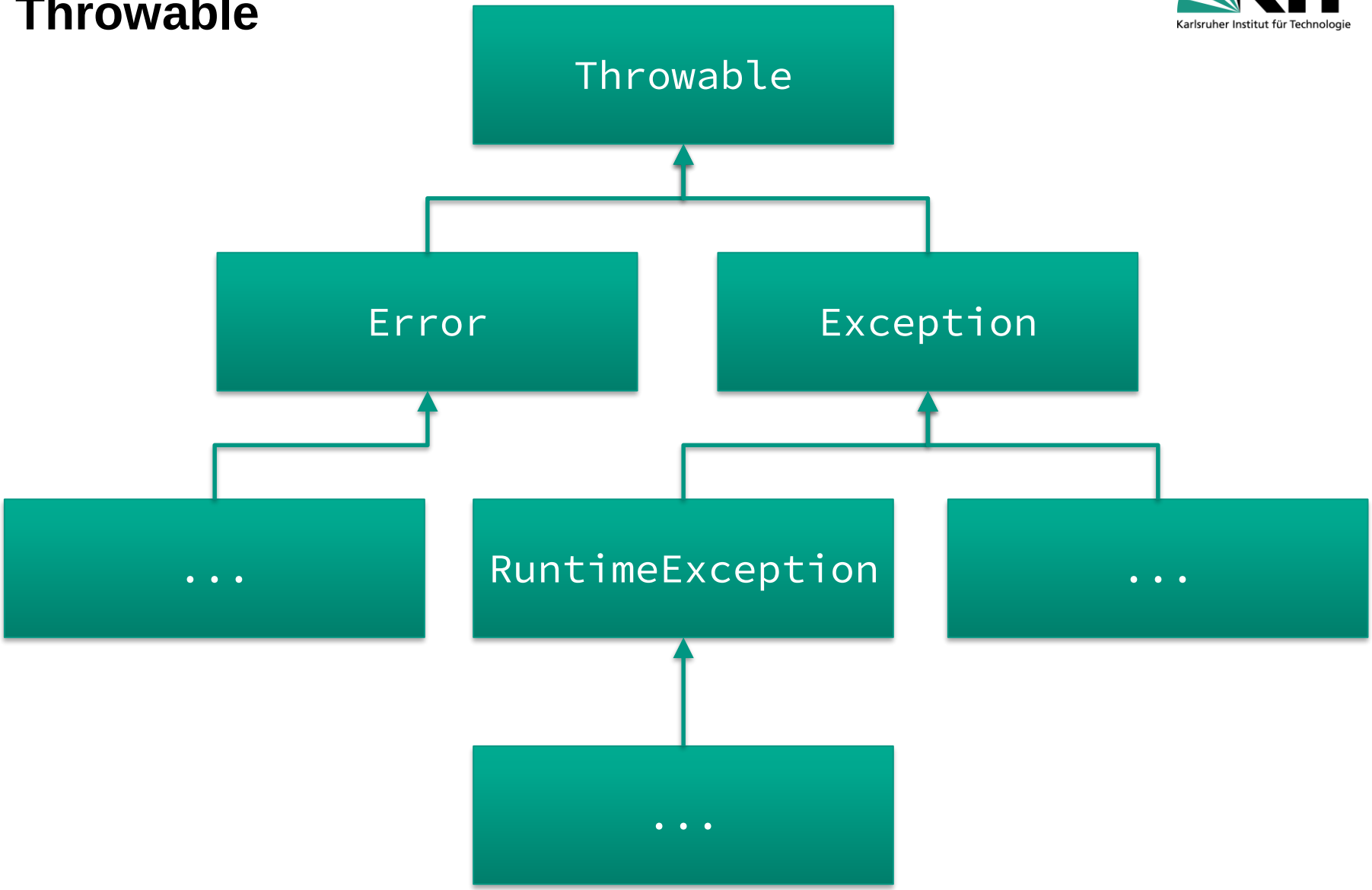
Exception

- Exceptions sind:
  - Ausnahme zur Laufzeit des Programms
  - Unterbrechungen des normalen Ablauf eines Programms
- Eine nicht aufgefangene Ausnahme führt zum Programmabbruch!
- Verwendung einer Exception:
  1. Ein Problem tritt auf
  2. Normales Fortfahren nicht möglich
  3. Lokale Reaktion darauf nicht sinnvoll möglich
  4. Behandlung des Problems an anderer Stelle nötig

Exception

- Exceptions sind in Java echte Objekte vom Typ `Throwable`
  - Erzeugung mit `new`
  - Auslösen mit `throw`
- Eine Exception hat in der Regel mindestens zwei Konstruktoren:
  - Default-Konstruktor ohne Parameter
  - Konstruktor mit einem String-Parameter
- Die Klasse `Throwable` definiert mehrere Methoden, zum Beispiel:
  - `getMessage()` – Nachricht der Ausnahme
  - `printStackTrace()` – Ausgabe des Aufrufstacks

Throwable



Exception – Deklaration

- Manche Ausnahmen müssen überprüft und explizit behandelt werden
  → Ausnahmen, die kein `Error` oder keine `RuntimeException` sind
- Andere Ausnahmen hingegen müssen nicht zwingend überprüft werden
  → Ausnahmen, die ein `Error` oder eine `RuntimeException` sind

- Die Idee ist, dass Ausnahmen, deren Auftreten auf einen Programmierfehler hindeuten, nicht überprüft werden müssen
  - `RuntimeException`

- Ausnahmen hingegen, deren Auftreten von Zeit zu Zeit unvermeidlich ist, müssen dagegen entweder aufgefangen werden oder explizit als möglich deklariert werden
  - `IOException`
  - `InterruptedException`

Exception – Deklaration

- Exceptions, welche nicht von `RuntimeException` erben oder bereits gefangen werden, müssen angekündigt werden

- Ankündigen funktioniert über JavaDoc-Annotation `@throws` und in der Methodensignatur mit `throws`

```
/**
 * The great boom method.
 *
 * @throws UniverseExplodeException when the universe
 *         is going to explode
 */
public void boom() throws UniverseExplodeException {
        if (Math.random() < 0.5) {
                throw new UniverseExplodeException();
        }
}
```

RuntimeException

Kindklasse von `RuntimeException`	Was den Fehler auslöst
`ArithmeticException`	Ganzzahlige Division durch 0
`ArrayIndexOutOfBoundsException`	Array-Indexgrenzen wurden missachtet
`StringIndexOutOfBoundsException`	String-Indexgrenzen wurden missachtet
`ClassCastException`	Typanpassung ist zur Laufzeit nicht möglich
`EmptyStackException`	Stapelspeicher ist leer
`IllegalArgumentException`	Parameter nicht korrekt angegeben
`NumberFormatException`	Konvertierung in eine Zahl ist nicht möglich
`NullPointerException`	Instanzname enthält keine Instanz

RuntimeException – Aufgabe

```
double[] array = new double[-1];
```
NegativeArraySizeException

```
array = new double[0];
array[1] = 1;
```
ArrayIndexOutOfBoundsException

```
String s = null;
s.toLowerCase();
```
NullPointerException

```
s = "abc";
char c = s.charAt(42);
```
StringIndexOutOfBoundsException

```
int x = 1 / 0;
```
ArithmeticException

```
x = Integer.parseInt("zwei");
```
NumberFormatException

```
Stack<String> stack = new Stack<>();
s = stack.pop();
```
EmptyStackException

```
Vector<String> vector = new Vector<>();

if (stack instanceof Vector) {
    Stack<String> cast = (Stack) vector;
}
```
ClassCastException

Exception – Fangen

```
try {
   // Programmcode, der eine Ausnahme ausführen kann
} catch (/* Exception... */) {
   // Programmcode zum Behandeln der Ausnahme
}
// Es geht ganz normal weiter...
```

Exception – Fangen

- Ein besonders ausgezeichnetes Programmstück überwacht mögliche Fehler und ruft gegebenenfalls speziellen Programmcode zur Behandlung auf
- Den überwachten Programmbereich leitet das Schlüsselwort **`try`** ein
- Dem `try`-Block folgt in der Regel ein oder mehrere **`catch`**-Block, in dem Programmcode steht, welcher den Fehler behandelt
- Wird im `try`-Block eine Ausnahme geworfen, so werden der Reihe nach alle `catch`-Blöcke durchgegangen
- Der erste Block, dessen Parameter zur geworfenen Ausnahme passt, kommt zur Anwendung und der entsprechende Code wird ausgeführt

Exception – Aufgabe

```
class ExceptionX extends Exception { }
class ExceptionY extends ExceptionX { }

public class Test {
   public static void main(String[] args)    {
      try {
         throw new ExceptionY();
      } catch (ExceptionX e) {
         System.out.println("ExceptionX");
      } catch (Exception e) {
         System.out.println("Exception");
      }
   }
}
```

Exception – Aufgabe

```
public class Test {
    public static void main(String[] args) {
        try {
            int i = 1 / 0;
        } catch (Exception e) {
            System.out.println("Exception");
        } catch (ArithmeticException ae) {
            System.out.println("ArithmeticException");
        }
    }
}
```

error: exception ArithmeticException has already been caught

Exception - Aufgabe

- Implementieren Sie eine möglichst einfache statische Methode, welche einen übergebenen `String`-Parameter in den dementsprechenden `int`-Wert umwandet. Verwenden Sie hierzu die Methode `Integer.parseInt(String s)`. Wenn der übergebenen Parameter nicht in sinnvoll eine Zahl umgewandelt werden kann, soll immer der Wert `-1` zurückgeben werden.

```
public static int parseInt(String input) {
        try {
                return Integer.parseInt(input);
        } catch (NumberFormatException e) {
                return -1;
        }
}
```

Exception – Faustregel 1

Der try-Block sollte so klein wie möglich sein!

- Beim lesen des Quelltextes wird klarer, wo das Problem auftreten kann

Exception – Faustregel 1 – Anti-Pattern

```
try {

    ...
    ...
    ... Hier stehen jede Menge Zeilen an Quelltext ...
    ...
    ...
    ...

} catch (SomeException e) {
    ...
} catch (OtherException e) {
    ...
} catch (ThirdException e) {
    ...
}
```

Exception – Faustregel 2

Exceptions des expliziten Typs Exception und Throwable dürfen NIE gefangen werden!

- Bei unterschiedlichen Fehlern will man meist unterschiedlich weiter machen, zum Beispiel:
  - `IOException`: nochmals versuchen
  - `NullPointerException`: Fehlerbericht an den Entwickler schicken
  - `IllegalArgumentException`: Fehlerausgabe an den Nutzer
- Durch die verschiedenen catch-Blöcke zeigst man, dass an die verschiedenen Fehlerarten gedacht wurde

Exception – Faustregel 2 – Anti-Pattern

```
try {
    ...
} catch (Exception e) {
    ...
} catch (Throwable t) {
    ...
}
```

Exception – Faustregel 3

Ein catch-Block darf niemals leer sein!

- Die Fehler werden klammheimlich unterdrückt
- Das Mindeste ist eine minimale Fehlerausgabe

Exception – Faustregel 3 – Anti-Pattern

```
int x = 0;
try {
    x = Integer.parseInt(string);
} catch (NumberFormatException e) { }
```

Exception – Faustregel 4

Exceptions sollten nicht den normalen Kontrollfluss steuern!

- Exceptions sollen nur die absolute Ausnahme darstellen

Exception – Faustregel 4 – Anti-Pattern

```
try {
   x = Integer.parseInt(args[0]);

   ...

   if (x == 1) {
      throw new NumberFormatException();
   }

   ...

} catch (NumberFormatException e) {
   ...
}
```

Exception – Faustregel 5

RuntimeException und Unterklassen dürfen im Allgemeinen nicht behandelt werden!

- Tritt eine `RuntimeException` auf, liegt in der Regel ein Denkfehler des Programmierers zugrunde
- Programmierfehler beheben, nicht behandeln!
- Ausnahme: `NumberFormatException`, `IllegalArgumentException`

Exception – Faustregel 5 – Anti-Pattern

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

Exception – Deklaration eigener Exceptions

- Exceptions werden genauso wie Klassen deklariert und sind ganz normale Objekte
  - Erben direkt oder indirekt von Exception

- Implementierung mindestens der zwei Standard-Konstruktoren:

```
public class UniverseExplodeException extends Exception {

        public UniverseExplodeException() {
                super();
        }

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

Exception – Aufgabe

- Schreiben Sie eine eigenen IllegalMatrixException, welche direkt von IllegalArgumentException erbt. Die Exception soll mindestens zwei Konstruktoren zu Verfügung stellen.
  - public class IllegalMatrixException
- Schreiben Sie eine statische Methode, welche die Dimension einer quadratischen Matrix bestimmt. Wenn die übergeben Matrix nicht Quadratisch ist, soll eine IllegalMatrixException geschmissen werden. Kündigen Sie dies über JavaDoc und in der Methodensignatur an.
  - public static int getSizeOfSquareMatrix(int[][] matrix)

Exception – Aufgabe – Lösung

```
public class IllegalMatrixException extends IllegalArgumentException {
    public IllegalMatrixException() {
        super();
    }

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

Exception – Aufgabe – Lösung

```
/**
 * @param matrix ...
 * @return n ...
 * @throws IllegalMatrixException ...
 */
public static int getSizeOfSquareMatrix(int[][] matrix)
                                throws IllegalMatrixException {
   if ((matrix == null) || (matrix.length == 0)) {
      throw new IllegalMatrixException("Invalid number of rows.");
   }
   final int n = matrix.length;
   for (int i = 0; i < n; i++) {
      if ((matrix[i] == null) || (matrix[i].length != n)) {
         throw new IllegalMatrixException("Invalid value " +
                                  "for the column: " + i);
      }
   }
   return n;
}
```

Fibonacci

- Die ersten neun Zahlen der Fibonacci-Folge sehen so aus:
  - `0, 1, 1, 2, 3, 5, 8, 13, 21, ...`
- Hier ist jede Zahl (ab der zweiten Position) gleich der Summe der beiden Vorgängerzahlen
- Mathematische Definition:
  - `Fib(0) = 0, Fib(1) = 1 und`
  - `Fib(n) = Fib(n-1) + Fib(n-2) , für n>1`

Rekursion

- Prinzip der Rekursion:
  - Man führe das gleiche Berechnungsmuster immer wieder mit einfacheren beziehungsweise kleineren Eingabedaten aus
  - Bis man zu einer trivialen Eingabe gelangt

- Realisierung von Rekursion:
  - Methoden, die sich direkt oder indirekt selbst aufrufen
  - Bei jedem rekursiven Aufruf wird eine neue Instanz der jeweiligen Methode gestartet
  - Jede Instanz hat ihre eigenen lokalen Variablen und Parameter, welche „von außen“ nicht sichtbar sind

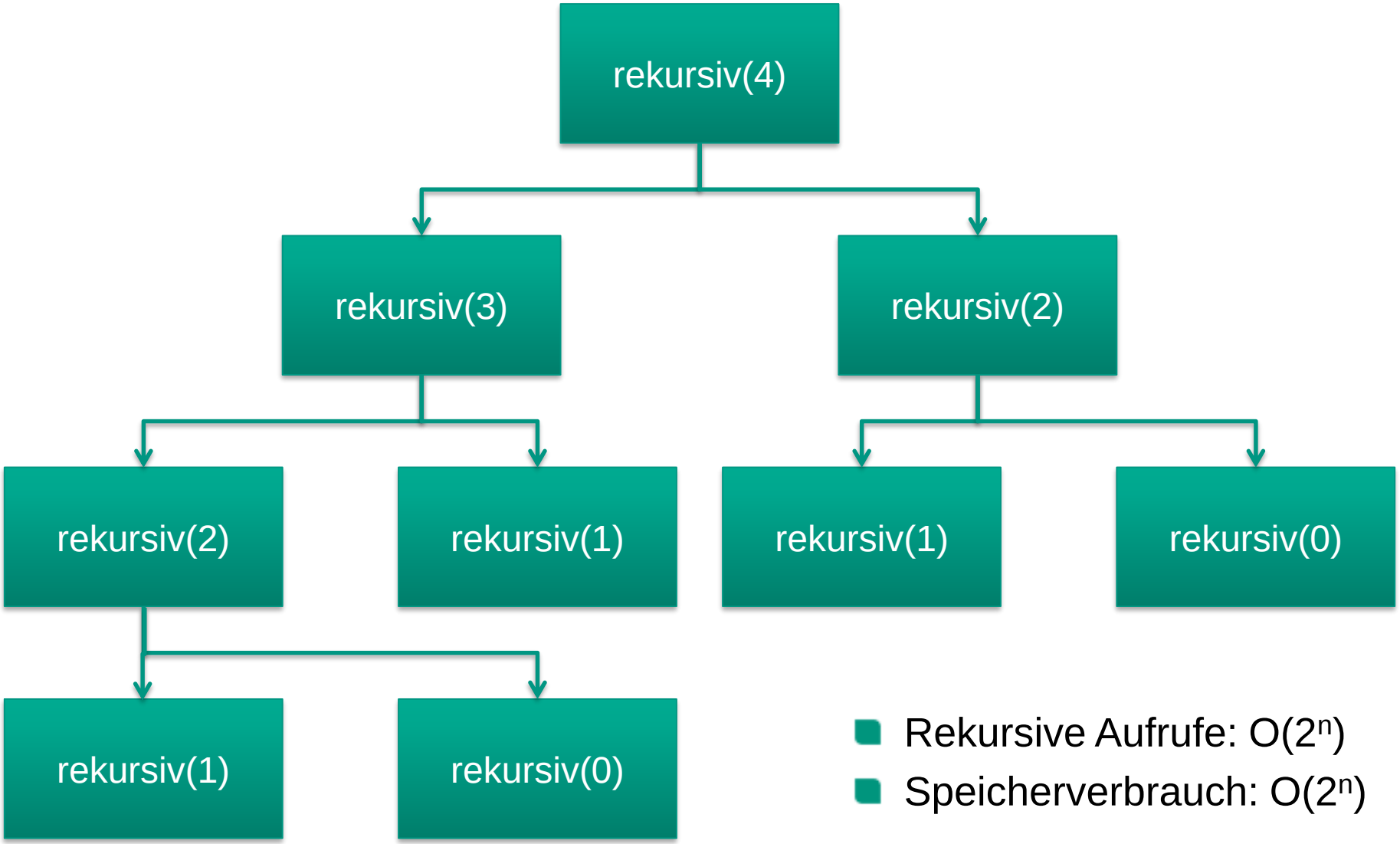
Fibonacci – Aufgabe

- Implementieren Sie eine rekursive Methode, die die n-te Fibonacci-Zahl berechnet

```
public static long rekursiv(final long number) {
        if (number <= 0) {
                return 0;               //Fib(0) = 0
        } else if (number == 1) {
                return 1;               //Fib(1) = 1
        }

        //Fib(n) = Fib(n-1) + Fib(n-2) , für n>1
        return rekursiv(number - 1) + rekursiv(number - 2);
}
```

Probleme bei Rekursion



- Rekursive Aufrufe: $O(2^n)$
- Speicherverbrauch: $O(2^n)$

Probleme bei Rekursion

- Rekursive Methoden haben in der Regel keine gute Performance!
- Durch die wiederholten Methoden Aufrufe wird immer wieder derselbe Methodeneintrittscode bearbeitet und bei jeder Inkarnation der Kontext gesichert
- Dies führt zu einem höherem Arbeitsspeicherverbrauch
- Alle rekursiven Algorithmen lassen sich jedoch auch durch iterative Programmierung implementieren (und umgekehrt)

Endrekursion

- Eine linear rekursive Funktion heißt **endrekursiv**, wenn in jedem Zweig der rekursive Aufruf nicht in andere Aufrufe eingebettet ist
- Endrekursion ermöglicht speicher-effiziente Auswertung

```
public static long tailRekursiv(long number) {
    return tailRekursiv(number, 0, 1);
}

private static long tailRekursiv(long number, long akumulator1,
                                 long akumulator2) {
    if (number <= 0) {
        return akumulator1;
    } else if (number == 1) {
        return akumulator2;
    }
    return tailRekursiv((number - 1), akumulator2,
            (akumulator1 + akumulator2));
}
```

- Übergebe Zwischenergebnisse in Hilfsparametern

Rekursion – Aufgabe

- Implementieren Sie eine Methode, welche die Summe der ersten n Elemente eines Arrays rekursiv nach dem Schema der Induktion berechnet und zurückgibt:
  - `public static int sum(int[] array, int to)`
- Implementieren Sie eine Methode, welche die Summe der Elemente in einem Intervall rekursiv nach dem teile und herrsche Paradigma berechnet und zurückgibt:
  - `public static int sum(int[] array, int from, int to)`
- Implementieren Sie eine Methode, welche das Maximum der Elemente in einem Intervall rekursiv nach dem teile und herrsche Paradigma berechnet und zurückgibt:
  - `public static int max(int[] array, int from, int to)`

Rekursion – Aufgabe – Lösung

```
public static int sum(int[] array, int to) {
    if (to < 0) {
        return 0;
    }
    return array[to] + sum(array, to - 1);
}
```

Rekursion – Aufgabe – Lösung

```
public static int sum(int[] array, int from, int to) {
    if (from == to) {
        return array[from];
    }

    return sum(array, from, (from + to) / 2)
            + sum(array, ((from + to) / 2) + 1, to);
}
```

Rekursion – Aufgabe – Lösung

```
public static int max(int[] array, int from, int to) {
    if (from == to) {
        return array[from];
    }

    int m1 = max(array, from, (from + to) / 2);
    int m2 = max(array, ((from + to) / 2) + 1, to);

    if (m1 < m2) {
        return m2;
    }
    return m1;
}
```

Softwarequalität

- *Software sind Programme und gegebenen Falles die zugehörige Dokumentation und weitere Daten, die zum Betrieb eines Computers notwendig sind.* (ISO 24765:2010)

- *Qualität ist der Grad, in dem ein Satz inhärenter Merkmale eines Objekts Anforderungen erfüllt.* (ISO 9000:2015-11)

- *Die Qualität eines Softwaresystems ist der Grad, in dem das System den genannten und den implizierten Bedürfnissen seiner verschiedenen Teilhaber genügt.* (ISO 25010:2011)

- **Was wären beispielhaft solche Teilhaber Bedürfnisse?**

Softwarequalität nach ISO/IEC 25010:2011

- Funktionelle Eignung
  - Funktionelle Vollständigkeit
  - Funktionale Korrektheit
  - Funktionale Angemessenheit
- Leistungsfähigkeit
  - Zeitverhalten
  - Ressourcennutzung
  - Kapazität
- Portabilität
  - Anpassungsfähigkeit
  - Einbaubarkeit
  - Austauschbarkeit
- Benutzerfreundlichkeit
  - Angemessenheit Erkennbarkeit
  - Erlernbarkeit
  - Bedienbarkeit
  - Benutzerfehlerschutz
  - Ästhetik der Benutzeroberfläche
  - Zugänglichkeit
- Zuverlässigkeit
  - Reife
  - Verfügbarkeit
  - Fehlertoleranz
  - Wiederherstellbarkeit
- Sicherheit
  - Vertraulichkeit
  - Integrität
  - Nachweisbarkeit
  - Zurechenbarkeit
  - Authentizität
- Wartbarkeit
  - Modularität
  - Wiederverwendbarkeit
  - Analysierbarkeit
  - Modifizierbarkeit
  - Prüfbarkeit
- Kompatibilität
  - Koexistenz
  - Interoperabilität

Funktional gleichwertig aber gleiche Qualität?

```java
public boolean isLeapYear() {
    if (((this.year % 100) == 0) && ((this.year % 400) != 0)) {
        return false;
    }
    return ((this.year % 400) == 0) || (((this.year % 4) == 0)
           && !((this.year % 100) == 0));
}

public boolean isLeapYear() {
    return ((year % 400) == 0)
           || (((year % 4) == 0) && ((year % 100) != 0));
}
```

Funktional gleichwertig aber gleiche Qualität?

```
 public boolean isEqual(Date other) {
    if (other.year == year) {
        if (other.month == month) {
            if (other.day == day) {
                return true;
            } else {
                return false;
            }
        } else {
            return false;
        }
    } else {
        return false;
    }
}
public boolean isEqual(Date other) {
    return (this.year == other.year) && (this.month == other.month)
           && (this.day == other.day);
}
```

Warum verständlich programmieren?

- **Warum sollte man Wert darauf legen, dass Programme funktionieren, und außerdem noch lesbar sind?**

Warum verständlich programmieren?

Warum wird Wert darauf gelegt, dass Programme funktionieren, und außerdem noch lesbar sind?

- Andere lesen Programme
- Software-Wartung erzeugt 80% der Kosten
- Quelltext ist die einzige verlässliche Dokumentation
- Software wird mit Quelltexten ausgeliefert

Ein gut strukturierter, objektorientierter Programmcode mit einem konsistentem Programmierstil:

- Erleichtert die Lesbarkeit und Übersicht
- Beschleunigt die Einarbeitung bei Personalwechsel
- Beschleunigt die eigene Wiedereinarbeitung
- Erhöht die Zeitersparnis bei Fehlerfindung, Erweiterung und Pflege

Java Code Conventions (JCC)

- Java Code Conventions als internationaler Standard
  - http://www.oracle.com/technetwork/java/codeconventions-150003.pdf
- Über den Sinn einzelner Regeln der JCC kann man sicher diskutieren
- Die JCC sind ein internationaler Standard und verbindlich für alle vorgegeben
- Auf den Tutoriums- und Vorlesungs-Folien wird aus platzgründen öfters Quelltext präsentiert, der nicht den JCC genügt!

Bezeichner

Sind die Bezeichner aussagekräftig und konsistent gewählt?

- Selbsterklärende, eindeutige Namen für Methoden und Variablen
- Verwendung einzelner Buchstaben nur als Laufindizes (i, j, k) oder Parameter einfacher mathematischer Funktionen (x, y, z)
- Keine weiteren Informationen im Namen kodiert
- Groß/Kleinschreibung (und Unterstriche) zur Trennung von Wortteilen
- Einheitliche Groß/Kleinschreibung für Konstanten, Variablen, Methoden und Klassen

Namenskonventionen

- UpperCamelCase bei Klassennamen
  - `class Example {}`
  - `class GamingPiece {}`
  - `class InterestListElement {}`
- lowerCamelCase bei Variablen und Attribute
  - `String example;`
  - `boolean hasSomething = true;`
  - `double maxValue = 13.37;`
  - `int peoplePerHousehold;`
- BIG_AND_FAT bei Konstanten
  - `final static int INTERVALL = 100;`
  - `final static double GRAVITY_ON_THE_MOON = 1.622;`

- **Alle Bezeichner in Englisch!**

Binnenmajuskeln

- Immer ganze Wörter Verwenden! Akronyme und Abkürzungen sind zu vermeiden, es sei denn, sie sind viel weiter verbreitet als ihre ursprüngliche lange Form

- Einige Worte können in der englischen Sprache mehrdeutig zusammengeschrieben werden:
  - nonempty oder non-empty ⇒ checkNonempty oder checkNonEmpty

Natürliche Sprache	Richtig	Falsch
XML HTTP request	XmlHttpRequest	XMLHTTPRequest
new customer ID	newCustomerId	newCustomerID
inner stopwatch	innerStopwatch	innerStopWatch
supports IPv6 on iOS?	supportsIpv6OnIos	supportsIPv6OnIOS
YouTube importer	YouTubeImporter	YoutubeImporter

Googles (fast) deterministisches Schema

- Konvertieren aller Wörter in reine ASCII-Buchstaben
  - Müller's algorithm ⇒ Muellers algorithm
- An verbleibenden Leerraumzeichen und sonstigen Satzzeichen in einzelne Wörter zerlegen. Wenn ein Wort bereits im CamelCase ist, wird auch dieses in seine Bestandteile zerlegt
  - AdWords ⇒ Ad Words
- Alles in Kleinbuchstaben konvertieren, einschließlich Akronyme
  - User ID ⇒ user id
- Den ersten Buchstaben eines Wortes in einen Großbuchstaben konvertieren, bei...
  - jedem Wort, für den UpperCamelCase.
  - jedem Wort außer dem ersten, für den lowerCamelCase.
- Alle Wörter zu einem Wort zusammensetzen

Einrückung

Wird die Programm-Struktur durch konsistente Einrückung und Leerzeilen verdeutlicht?

- Eine neue Zeile für jeden Befehl
- Absetzen von Strukturen:
  - Tiefere Einrückung kontrollierter Befehle (konsistent!)
  - Keine übertriebene Einrückung (4 Leerzeichen)
  - Vor und hinter der Struktur (incl. Kommentar) genau eine Leerzeile
- Strukturierung von Operatoren und Operanden durch Leerzeichen (konsistent!)
- Leerzeilen zwischen Methoden (konsistent!)

Blockstruktur

- Ein Block wird geschweifte Klammern `{ }` und durch gemeinsame Einrückung des Quelltextes gekennzeichnet

- Die Einrückungstiefe der Anweisungen im Block durch vier Leerzeichen erhöht
  - Wenn der Block zu Ende ist, wird zu der vorherigen Einrückungstiefe zurückgekehrt

- Regeln für die Verwendung von geschleiften Klammern:
  - Keinen Zeilenumbruch bevor einer geöffneten geschleiften Klammer
  - Ein Zeilenumbruch, nach einer geöffneten geschleiften Klammer
  - Ein Zeilenumbruch bevor einer schließenden geschleiften Klammer
  - Ein Zeilenumbruch nach einer schließenden geschleiften Klammer, wenn diese Klammer eine Anweisung terminiert oder den Rumpf von einer Methode, Konstruktor oder eine Klasse schließt
  - Die Verwendung erfolgt auch, wenn dessen Rumpf leer ist oder aus nur einer einzigen Anweisung besteht

Leerzeichenkonventionen

- Kein Leerzeichen nach:
  - . Punktoperator
  - (linke runde Klammer
  - ! logische Negation
  - ~ bitweises Komplement
  - -- präfix Dekrement
  - - unäres Minus
  - + unäres Plus
  - ++ präfix Inkrement
- Ein Leerzeichen vor und nach:
  - Alle anderen Operatoren
  - Alle anderen Anweisungen
- Kein Leerzeichen vor:
  - ; Semikolon
  -) rechte runde Klammer
  - -- postfix Dekrement
  - ++ postfix Inkrement
- Ein Leerzeichen nach:
  - , Komma
  - ; Semikolon
  - (Typ) Typumwandlung

Überlange Zeilen

Passt ein Ausdruck nicht auf eine Zeile,
muss er umgebrochen werden:

- Maximale Breite: 120 Zeichen
- Hinter einem Komma, aber vor Operatoren
- Trenne die äußerste Struktur
- Richte neue Zeile gemäß des umgebenden Ausdrucks aus Benutze Addition für überlange Zeichenketten
- Vermeide verwirrende Einrückungen

Struktur

Wird das Verfolgen des Programm-Ablaufs durch kleine, wohlstrukturierte Prozeduren unterstützt?

- Was immer in eine eigene Methode ausgelagert werden kann, sollte ausgelagert werden:
  - Algorithmen
  - Methoden zur Realisierung einer Funktion aus der Aufgabenstellung
  - Elementare Operationen auf Datenstrukturen

Anpassbarkeit

Kann das Programm ohne großen Aufwand an veränderte Randbedingungen angepasst werden?

- Verwendung von Konstanten
  - Wiederkehrende Werte an einer Stelle definiert
  - Änderungen nur an einer Stelle nötig
- Jede Komponente nur verwendbar, wenn und wo nötig
  - Verwendung von Zugriffsmethoden anstelle direkter Zugriffe auf Variablen fremder Klassen
  - Definition von Zugriffsmethoden bei Bedarf und nicht „auf Vorrat“
- Vermeidung von redundantem Code

Programmierstil Richtlinien der Übungsblätter

- Achten Sie darauf nicht zu lange Zeilen, Methoden und Dateien zu erstellen
- Programmcode muss in englischer Sprache verfasst sein
- Kommentieren Sie Ihren Code angemessen: So viel wie nötig, so wenig wie möglich
- Wählen Sie geeignete Sichtbarkeiten für Ihre Klassen, Methoden und Attribute
- Verwenden Sie keine Klassen der Java-Bibliotheken ausgenommen Klassen der Pakete `java.lang` und `java.io`
- Achten Sie auf fehlerfrei kompilierenden Programmcode
- Halten Sie alle Whitespace-Regeln ein
- Halten Sie die Regeln zu Variablen-, Methoden und Paketbenennung ein und wählen Sie aussagekräftige Namen

Bewertungskriterien

- Methodik
  - Strukturierter Programmcode
  - Eindeutige und aussagekräftige Namen
  - Einhaltung der Namenskonventionen
  - Ausreichende und richtige Dokumentation
  - Einhalten der Checkstyle-Richtlinien
- Funktionalität
  - Konkrete Aufgabenstellung erfühlen
  - Objektorientierte Programmierung
  - Einsatz von sinnvollen Datentypen
  - Kein überflüssiger Quelltext
  - Programm Stabilität

Fragen?

Was machen wir nächste Woche?

- Exceptions
- Testen & Assertions

Vielen Danke für eure Aufmerksamkeit!