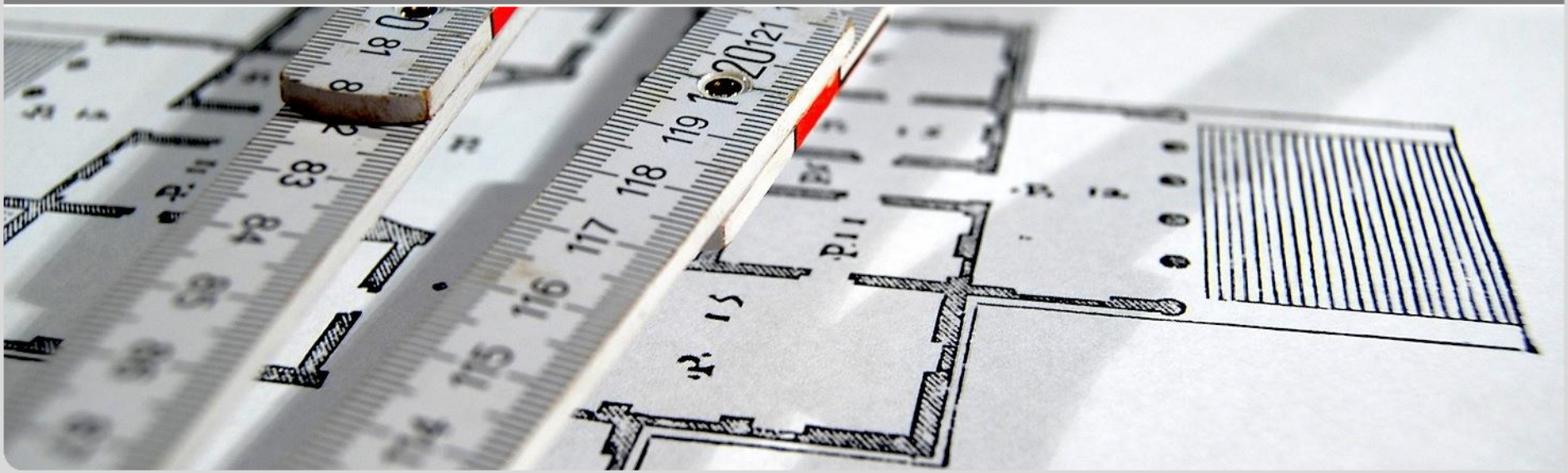
Karlsruher Institut für Technologie

Programmieren-Tutorium Nr. 10

9. Tutorium | Jonas Ludwig
Wrapper-Klassen, Generics, Rekursion

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



Was machen wir heute?

- Wrapper-Klassen
- Generics
- Lambda-Ausdrücke
- Rekursion

Organisatorisches

- Morgen: Präsenzübung

Übungsblatt 3

- Aufgabe A *Ø5.85P von 10P*
  - Trennung zwischen Benutzerschnittstelle, Programmlogik und Datenmodell einhalten!
  - Randfälle betrachten
    - Verhalten bei leerem Stack
    - Division durch 0
  - Benutzereingaben sehr genau überprüfen
    - Leerzeichen mehr oder weniger gibt Abzug!
  - JavaDoc => Englisch
  - Kommentare => Deutsch oder Englisch

Wrapper-Klasse

- Eine Wrapper-Klasse umhüllt primitive Datentypen und ermöglich so grundsätzlich objektorientierten „Methoden“ für primitive Datentypen
- Ein Wrapper-Objekt kann nach dem Erzeugen nicht verändert werden
- Wrapper-Klassen bieten zusätzlich Methoden zur Konvertierung eines Datentyps in einen String und vom String in den Datentyp
- Generische-Typparameter können keine primitiven Datentypen sein

Integer Wrapper-Klasse

```
public final class Integer
        extends Number implements Comparable<Integer> {

   public static int MAX_VALUE;
   public static int MIN_VALUE;
   private final int value;
   public Integer (int i) { ... }
   public Integer (String s) { ... }
   public static int compare(int x, int y) { ... }
   public static String toHexString(int i) { ... }
   public static String toString(int i) { ... }
   public static Integer valueOf(int i) { ... }
   public int compareTo(Integer i) { ... }
   public boolean equals(Object o) { ... }
   public int intValue() { ... }
   public String toString() { ... }

   ...
}
```

- http://docs.oracle.com/javase/8/docs/api/java/lang/Integer.html

Weitere Wrapper-Klassen

Wrapper-Klasse	Primitiver Datentyp
`Boolean`	`boolean`
`Character`	`char`
`Byte`	`byte`
`Short`	`short`
`Integer`	`int`
`Long`	`long`
`Double`	`double`
`Float`	`float`
`Void`	`void`

- Für **`void`** (welches kein Datentyp ist) existiert die Klasse `Void`, sie deklariert nur eine Konstante vom Typ `Class<Void>`

Autoboxing – Boxing und Unboxing

- Primitive Datentypen und Wrapper-Objekte werden bei Bedarf ineinander umgewandelt

```
int     i = 42;
Integer j = i;    // 1. steht für  j = Integer.valueOf(i)
int     k = j;    // 2. steht für  k = j.intValue()
```

- 1. nennt sich *Boxing* und erstellt automatisch ein Wrapper-Objekt, sofern erforderlich
- 2. ist das *Unboxing* und steht für das Beziehen des Elements aus dem Wrapper-Objekt

Konstruktor versus valueOf()

- Eine statische Methode muss Objekte nicht immer neu erzeugen, sondern kann auch auf vorkonstruierte Objekte zurückgreifen
- Stammen die Ganzzahlen aus dem Wertebereich `-128` bis `+127`, so greift `valueOf()` auf vorbereitete Objekte aus einem Speicher zurück
- Die JavaVM kann bei der Initialisierung auch angewiesen werden den Wertebereich `-128` bis zu `+2147483647` zu generieren

```
Integer i1 = new Integer(42);
Integer i2 = new Integer(42);
Integer i3 = Integer.valueOf(42);
Integer i4 = Integer.valueOf(42);
System.out.println(i1 == i2); // false
System.out.println(i2 == i3); // false
System.out.println(i3 == i4); // true
```

Wrapper – Aufgabe

```
final Integer int1 = Integer.valueOf("42");
final Integer int2 = new Integer(42);
final Integer int3 = 42;
final int int4 = 42;
```

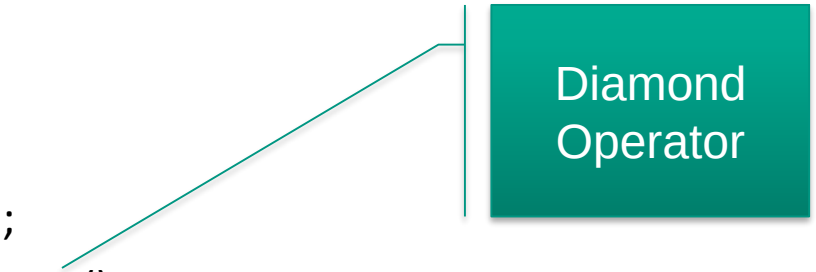
`System.out.println(int1 == int2);`	false
`System.out.println(int1 == int3);`	true
`System.out.println(int1 == int4);`	true
`System.out.println(int2 == int3);`	false
`System.out.println(int2 == int4);`	true
`System.out.println(int3 == int4);`	true
`System.out.println(int1.equals(int2));`	true
`System.out.println(int1.equals(int3));`	true
`System.out.println(int1.equals(int4));`	true
`System.out.println(int2.equals(int3));`	true
`System.out.println(int2.equals(int4));`	true
`System.out.println(int3.equals(int4));`	true
`System.out.println(int4.equals(int4));`	Fehler beim kompilieren
`System.out.println(int4.equals(int1));`	Fehler beim kompilieren

Generics – Konzept

- **Aufgabe aus dem Tutorium:** Implementieren Sie eine Klasse `LinkedPointList`, welche Punkte (Klasse `Point`) verwaltet.
- **Feststellung:** Der Code für die Liste ist an sich unabhängig vom Typ der Basisdaten, verwendet aber überall die Klasse `Point`.
- **Idee:** Implementierung einer Klasse `LinkedList`, welche einen beliebigen Datentyp mittels einer Liste verwaltet. Beim Entwurf der Liste ist es egal, welchen Typ die enthaltenen Objekte haben.
- **Lösung:** Mache den Typ der Basisdaten zu einem Parameter der Datenstruktur, da auch Typen Parameter sein können.

Generics – Syntax

- Syntax
  - class Name<Typ-Parameter> { ... }
  - interface Name<Typ-Parameter> { ... }
- Deklaration
  - class LinkedList<E> { ... }
  - interface Map<K,V> { ... }
- Verwendung
  - List<Point> list = new LinkedList<Point>();
  - Map<Integer, Point> map = new HashMap<>();

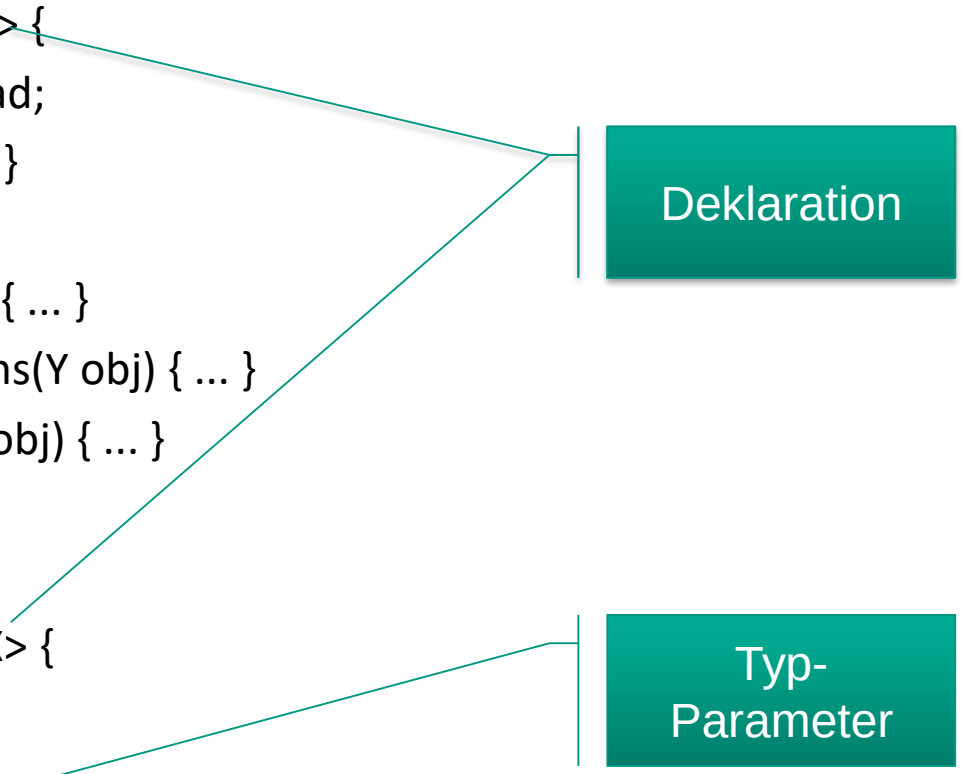


Generics – Beispiel

```
public class LinkedList<Y> {
    private ListCell<Y> head;
    public LinkedList() { ... }

    public void add(Y obj) { ... }
    public boolean contains(Y obj) { ... }
    public void remove(Y obj) { ... }
    public int size() { ... }

    private class ListCell<X> {
        X content;
        ListCell<X> next;
        ListCell(X content) { ... }
    }
}
```



Generics

- Ersetzen eines konkreten Typs durch einen Typ-Parameter
  - Erlaubt Wiederverwendung und Generalisierung von Programmcode
  - Können nur Referenzen sein: Klassen, Schnittstellen und Aufzählungen

- **Namenskonvention:** Formale Typparameter sind in der Regel einzelne Großbuchstaben, da sie nur Platzhalter sind und keine wirklichen Typen
  - **T** (steht für Typ)
  - **E** (Element)
  - **K** (Key/Schlüssel)
  - **V** (Value/Wert)

Generics – Aufgabe

- Implementieren Sie eine generische Klasse `Pair`, um zwei Objekte zusammenzufassen. Hierbei müssen beide Objekte nicht zwingend den gleichen Typ haben. Fügen Sie dieser Klassen einen geeigneten Konstruktor und Methoden zur Datenkapseln hinzu.

```java
public class Pair<K, V> {
    private K key;
    private V value;

    public Pair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public K getKey() {
        return key;
    }
    public void setKey(K key) {
        this.key = key;
    }
    public V getValue() {
        return value;
    }
    public void setValue(V value) {
        this.value = value;
    }
}
```

Generics - Typeinschränkung

- Oft ist ein beliebiger Typ zu allgemein: Es ist erforderlich, dass es eine bestimmte Methode gibt → Vererbung / Interfaces
- Gewünschte Typeinschränkungen müssen explizit kodiert werden
- Einschränken der Typparameter möglich:
  - Unterklasse: (`T extends SomeSuperClass`): T ist Unterklasse
  - Interface (`T extends SomeInterface`): T implementiert Interface

Generics – Kovarianz und Wildcards

- Obwohl `Integer` eine Unterklasse von `Number` ist, ist `List<Integer>` keine Unterklasse von `List<Number>`

```
Number number = Integer.valueOf(1);
List<Number> list;
list = new LinkedList<Integer>();
list.add(Double.valueOf(2.0));
```

- Gewünschte Klassenbeziehungen müssen explizit kodiert werden
- Wildcard **?** ist anonymer Typparameter
  - Obere Schranke (`? extends C`): Lesen mit Typ `C` möglich
  - Untere Schranke (`? super D`): Zuweisen mit Typ `D` möglich

PECS – Producer Extends Consumer Super

- Ein Container, aus dem Daten gelesen werden (= Producer), wird mit einer `extends`-Wildcard versehen: `List<? extends Integer>`
- Ein Container, in dem Daten abgespeichert werden (= Consumer), wird mit einer `super`-Wildcard versehen: `List<? super Integer>`
- Ein Container, aus dem sowohl Daten gelesen als auch abgespeichert werden, wird mit keinem Wildcard versehen: `List<Integer>`

```
public static <T> void copy(List<? super T> dest,
                            List<? extends T> src) {
    for (int i = 0; i < src.size(); i++) {
        dest.set(i, src.get(i));
    }
}
```

Generics – Interfaces

- Das Interface `Iterable<T>`
- Das Interface `Iterator<T>`
- Das Interface `Comparable<T>`
- Das Interface `Comparator<T>`

Das Interface Iterator<T>

- Das Interface Iterator<E> definiert unter anderem die Methoden
  - `boolean hasNext()`
  - `E next`
- Ein Iterator bezeichnet einen Zeiger (cursor), mit dem über die Elemente einer Kollektion gewandert (iteriert) werden kann

Das Interface Iterable<T>

- Das Interface Iterable<T> definiert die Methode
  - Iterator<T> iterator()
- Die implementierende Klasse muss somit die Methode iterator() zur Verfügung stellen, die einen Iterator<T> als Ergebnis hat
- Beispiel: Eine Klasse IterableString, bei der über die einzelnen Character des Strings iteriert werden kann

```
class IterableString implements Iterable<Character> {
    private String str;
    public IterableString(String str) { this.str = str; }

     public Iterator<Character> iterator() {
        return new IterableStringIterator(str);
     }
}
```

Iterator<T> und Iterable<T>

- Verwendung 1:

```
IterableString s = new IterableString("Iteratoren...");

Iterator<Character> it = s.iterator();

while (it.hasNext()) { System.out.print(it.next()); }
```

- Verwendung 2:

```
IterableString s = new IterableString("...sind toll!");

for (Character c : s) { System.out.print(c); }
```

Iterator<T> und Iterable<T> – Aufgabe

- Implementieren Sie eine Klasse `IterableStringIterator`, welche selbst die `Iterator<Character>` Schnittstelle implementiert.

```
public class IterableStringIterator implements Iterator<Character> {
   private String str;
   private int count = 0;

   public IterableStringIterator(String str) { this.str = str; }

   public boolean hasNext() { return count < str.length(); }

   public Character next() {
      if (count == str.length()) {
         throw new NoSuchElementException();
      }
      return str.charAt(count++);
   }
}
```

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 Comparable<T> – Aufgabe

- Betrachten Sie die Java-API des Interfaces java.lang.Comparable. Implementieren Sie dieses Interface für die Klasse Point und machen Sie somit die Punkte vergleichbar. Verwenden Sie hierbei die Euklidische Norm um zwei Punkte miteinander zu vergleichen. Stellen Sie die Verträglichkeit der compareTo-Methode mit der equals-Methode sicher. Falls die Normen zweier Punkte gleich sind, soll zuerst nach dem x-Wert und gegebenenfalls nach dem y-Wert verglichen werden.
- $p = \begin{pmatrix} x \\ y \end{pmatrix} \quad \|p\| = \sqrt{x^2 + y^2}$

```
public class Point {
        private final double x;
        private final double y;
        //...
}
```

Das Interface Comparable<T> – Lösung

```
public class Point implements Comparable<Point> {

        private final Double x;
        private final Double y;

        public Point() {
                    final Double zero = Double.valueOf(0.0);
                    x = zero;
                    y = zero;
        }

        public Point(final double x, final double y) {
                    this.x = Double.valueOf(x);
                    this.y = Double.valueOf(y);
        }
```

Das Interface Comparable<T> – Lösung

```
public double getX() {
	return x.doubleValue();
}

public double getY() {
	return y.doubleValue();
}

public double euclideanNorm() {
	return Math.sqrt((getX() * getX()) + (getY() *
getY()));
}
```

Das Interface Comparable<T> – Lösung

```
@Override
public boolean equals(final Object obj) {
	if (this == obj) {
		return true;
	} else if (obj == null) {
		return false;
	}

	return (this.getClass() == obj.getClass())
		&& x.equals(((Point) obj).x)
		&& y.equals(((Point) obj).y);
}
```

Das Interface Comparable<T> – Lösung

```
@Override
public int compareTo(final Point other) {
        return Point.compare(this, other);
    }
```

Das Interface Comparable<T> – Lösung

```
public static int compare(final Point p1, final Point p2) {
        if ((p1 == null) && (p2 == null)) {
                return 0;
        } else if (p1 == null) {
                return -1;
        } else if (p2 == null) {
                return 1;
        }

        int answer = 0;
```

Das Interface Comparable<T> – Lösung

```
        if (!p1.equals(p2)) {
                answer = Double.compare(p1.euclideanNorm(),
                                        p2.euclideanNorm());
                if (answer == 0) {
                        answer = p1.x.compareTo(p2.x);
                        if (answer == 0) {
                                answer = p1.y.compareTo(p2.y);
                        }
                }
        }
        return answer;
}
```

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);
```

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

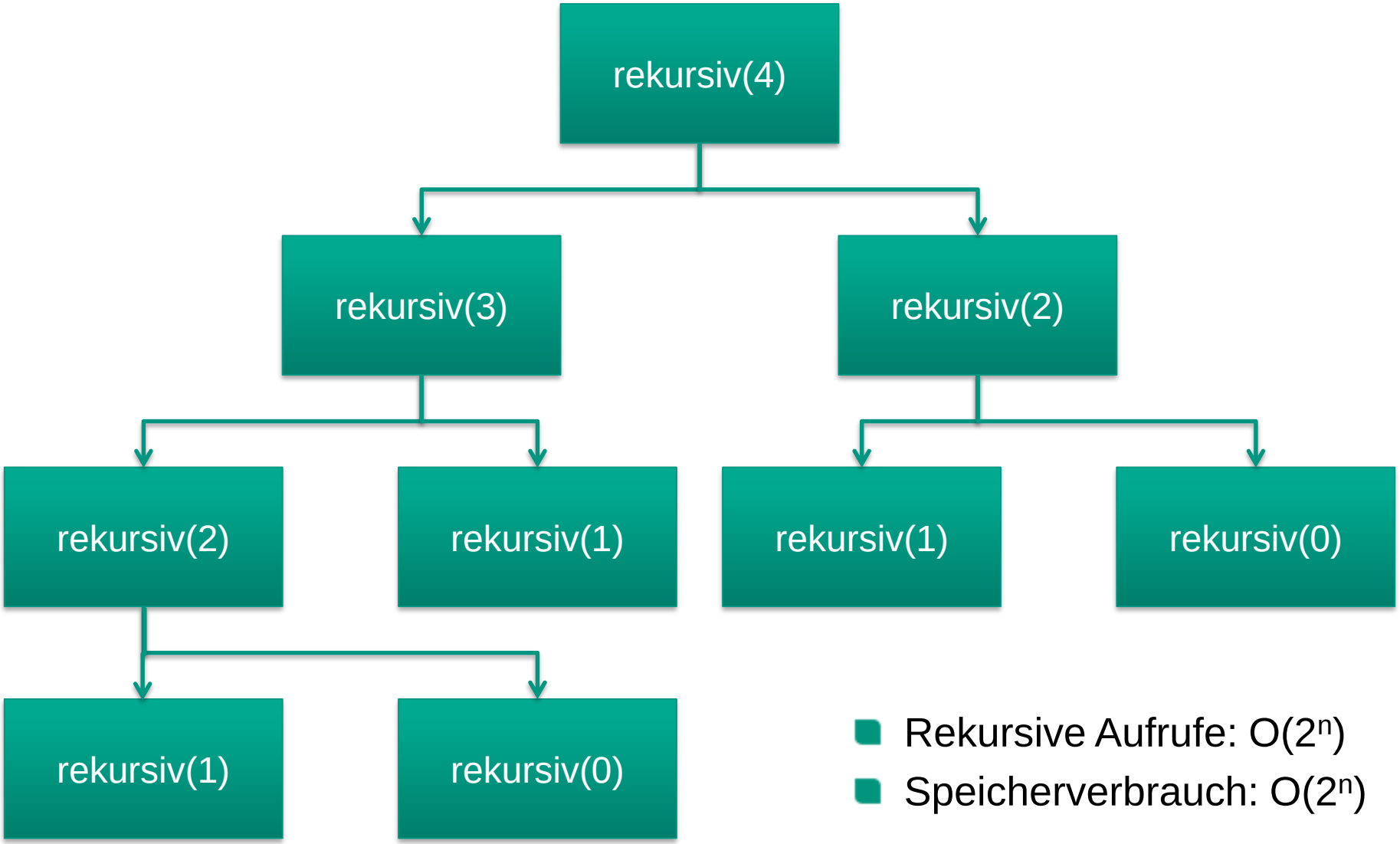
Fibonacci – Aufgabe

- 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;
}
```

Karlsruher Institut für Technologie

Fragen?

Was machen wir nächste Woche?

- Exceptions
- Testen & Assertions

Vielen Danke für eure Aufmerksamkeit!