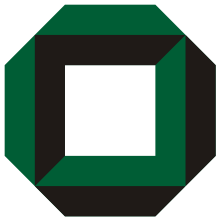
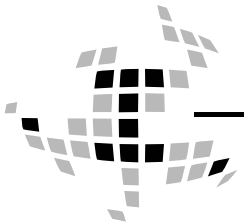


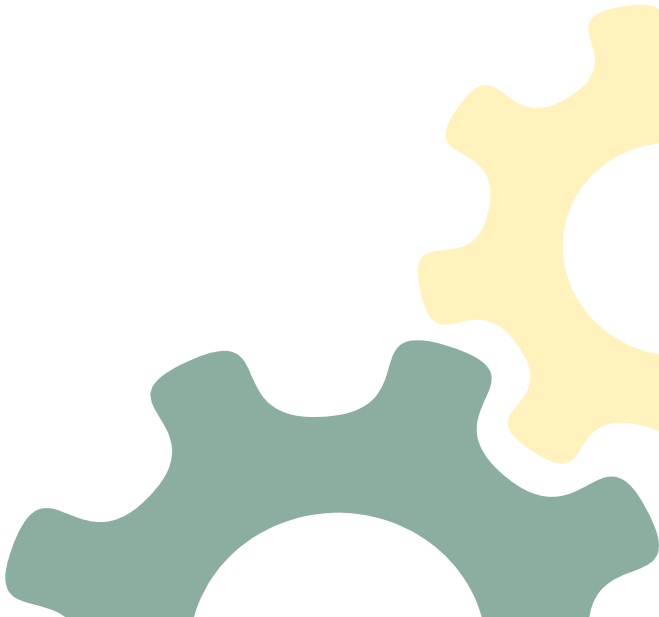
Universität Karlsruhe (TH)
Forschungsuniversität · gegründet 1825

# Softwaretechnik 1
# Übung 4

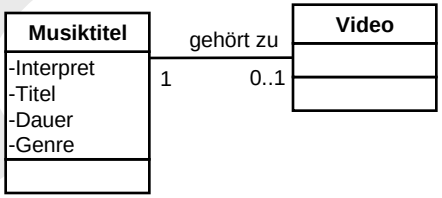
18.6.2009



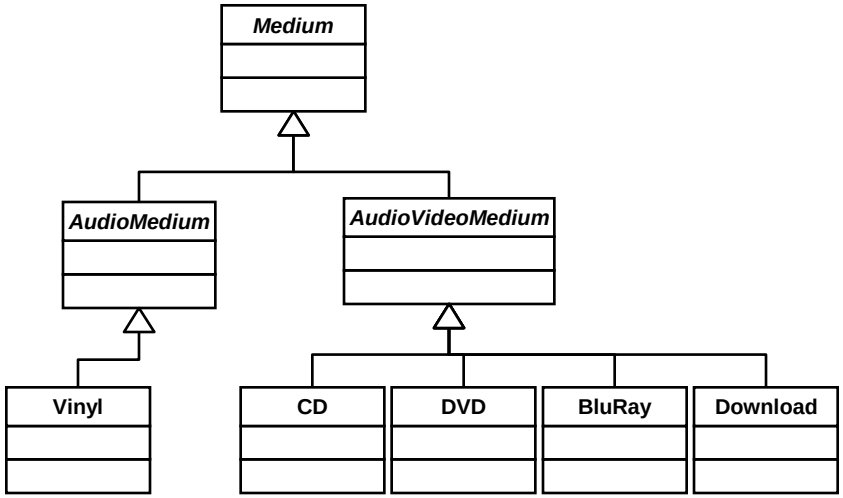
Fakultät für **Informatik**
Lehrstuhl für Programmiersysteme



# Aufgabe 1)

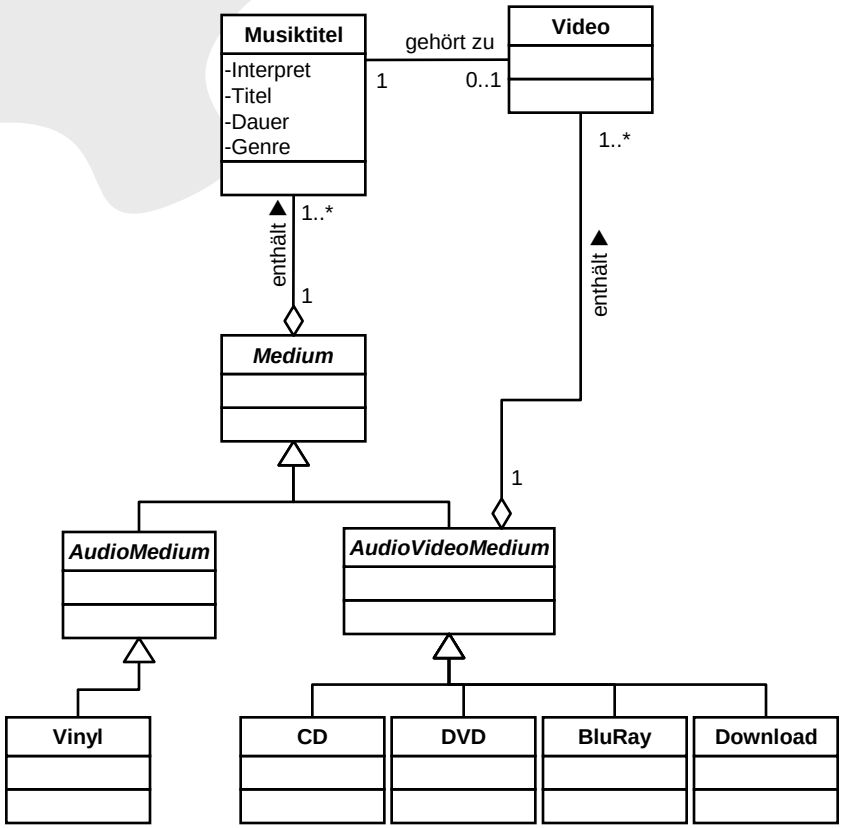



Ein Musiktitel enthält den *Titel* des Musikstückes sowie Informationen über den *Interpreten*, das *Genre* sowie die *Dauer* des Titels. Der Händler unterscheidet zwischen Musiktiteln ohne Video und Musiktiteln, zu welchen optional das Musikvideo geordert werden kann. Zu einem Musiktitel kann es maximal ein Musikvideo geben.




Zur Auswahl stehen Medien für Musiktitel ohne Video sowie Musiktitel mit Video. Musiktitel ohne Video können auf Vinyl, CD, DVD, BluRay bestellt oder als Download heruntergeladen werden. Musiktitel mit Video können nicht auf Vinyl bestellt werden.

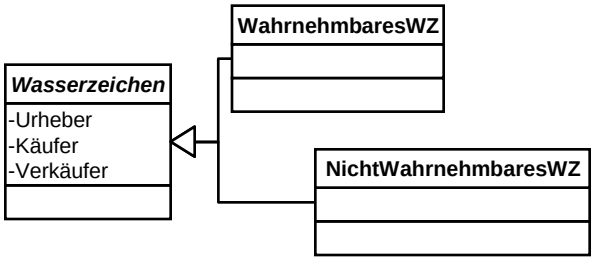
# Aufgabe 1)



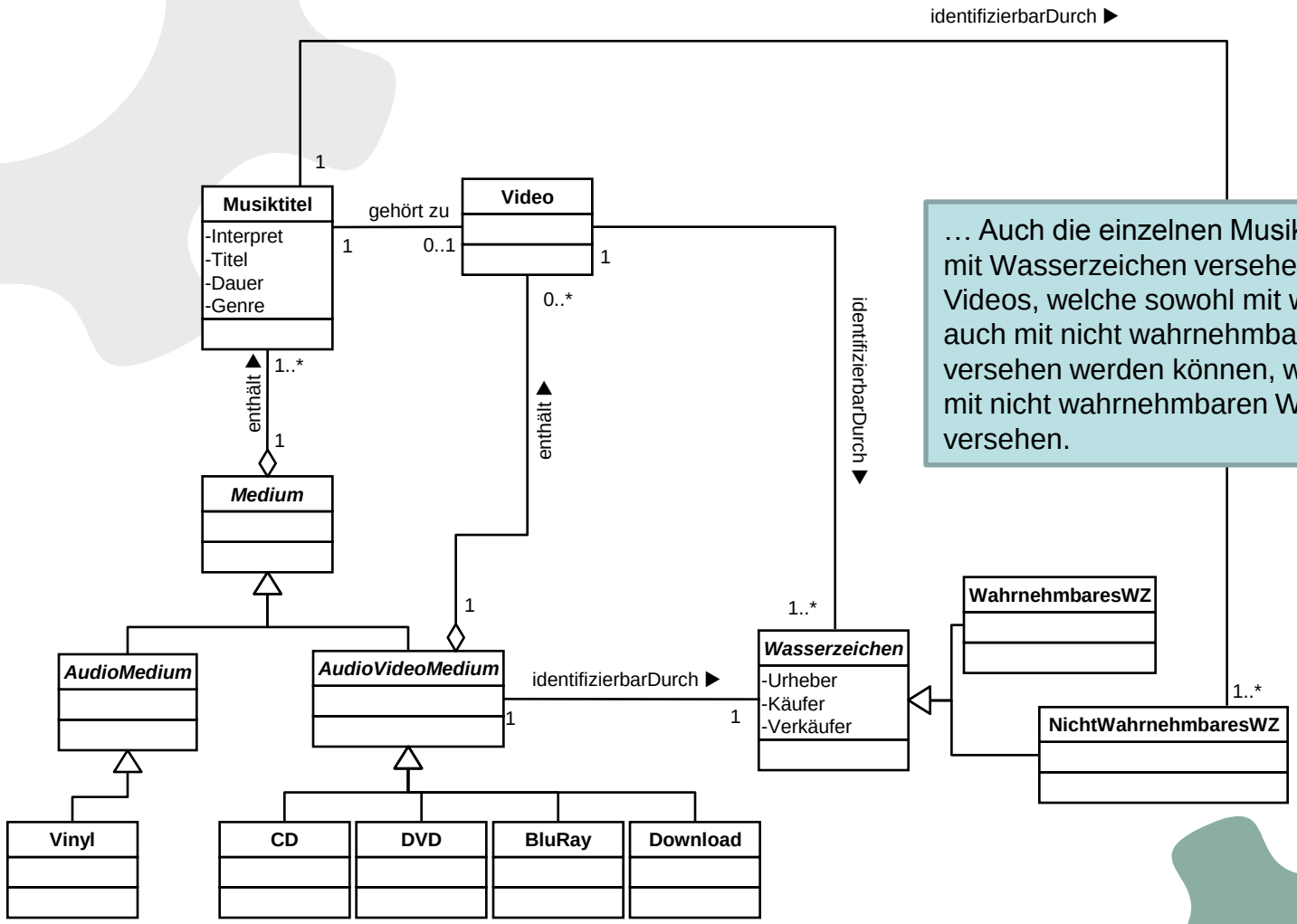

Zur Auswahl stehen Medien für Musiktitel ohne Video sowie Musiktitel mit Video. Musiktitel ohne Video können auf Vinyl, CD, DVD, BluRay bestellt oder als Download heruntergeladen werden. Musiktitel mit Video können nicht auf Vinyl bestellt werden.

# Aufgabe 1)

Die Medien für Musiktitel mit Video werden mit einem Wasserzeichen versehen. Dieses kann sowohl ein wahrnehmbares Wasserzeichen als auch ein nicht wahrnehmbares Wasserzeichen sein…
Die Wasserzeichen dienen dazu, den *Urheber*, den *Verkäufer* sowie den *Käufer* identifizieren zu können.



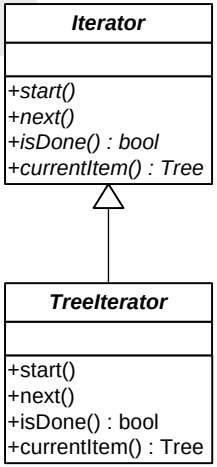
# Aufgabe 1)




… Auch die einzelnen Musiktitel und Videos sind mit Wasserzeichen versehen. Im Gegensatz zu Videos, welche sowohl mit wahrnehmbaren, als auch mit nicht wahrnehmbaren Wasserzeichen versehen werden können, werden Musiktitel nur mit nicht wahrnehmbaren Wasserzeichen versehen.

# Aufgabe 2a)

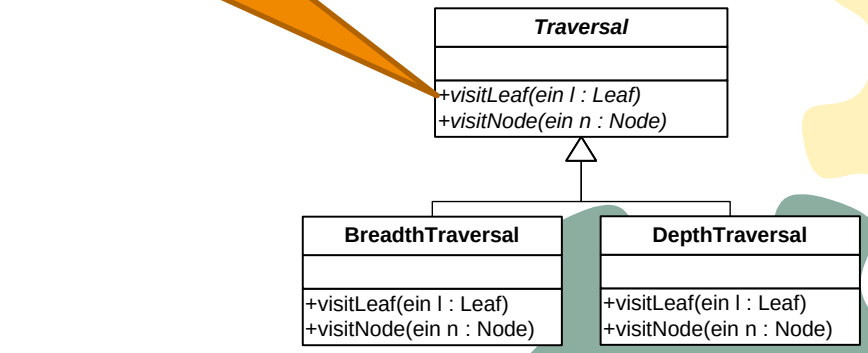
Für das Verarbeiten des Baumes möchten Sie verschiedene Varianten implementieren, wie der Baum mit einem `Iterator` traversiert werden soll.




Für das aktuelle Einsatzgebiet der Anwendung genügen Ihnen zwei verschiedene Traversierungsarten (**Traversal**):

- *Ebenenweise (***BreadthTraversal***)*
- *Hauptreihenfolge* (**DepthTraversal***)*

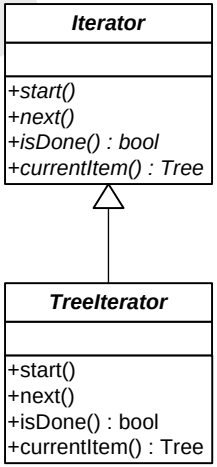
Operationen des Besucher-Musters.

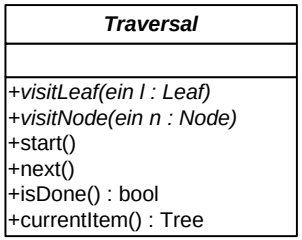


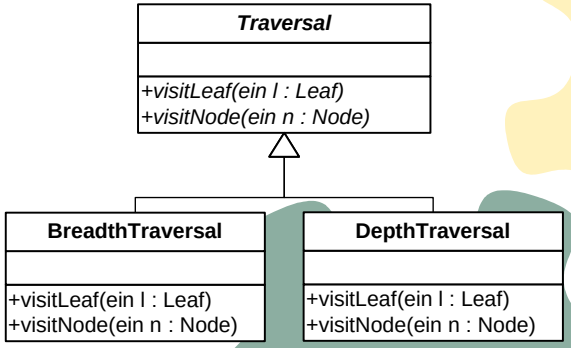
# Aufgabe 2a)

Für das Verarbeiten des Baumes möchten Sie verschiedene Varianten implementieren, wie der Baum mit einem `Iterator` traversiert werden soll.

Für das aktuelle Einsatzgebiet der Anwendung genügen Ihnen zwei verschiedene Traversierungsarten (**Traversal**):
- *Ebenenweise (***BreadthTraversal***)*
- *Hauptreihenfolge* (**DepthTraversal***)*



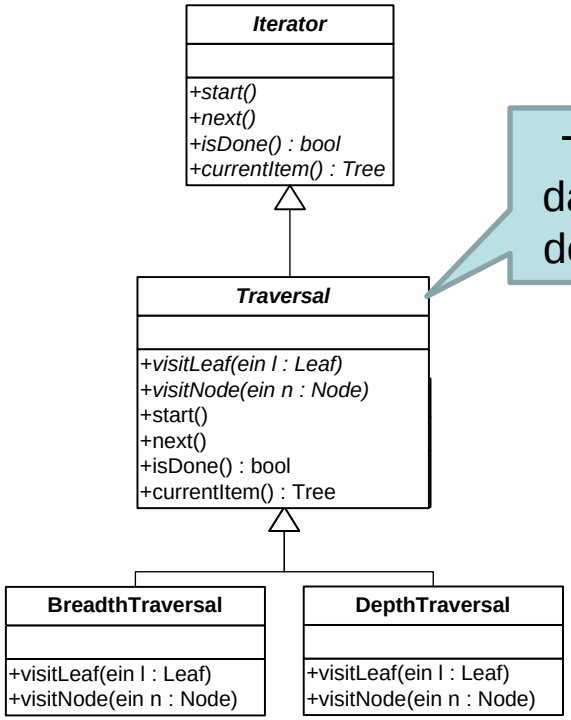



# Aufgabe 2a)

Für das Verarbeiten des Baumes möchten Sie verschiedene Varianten implementieren, wie der Baum mit einem `Iterator` traversiert werden soll.

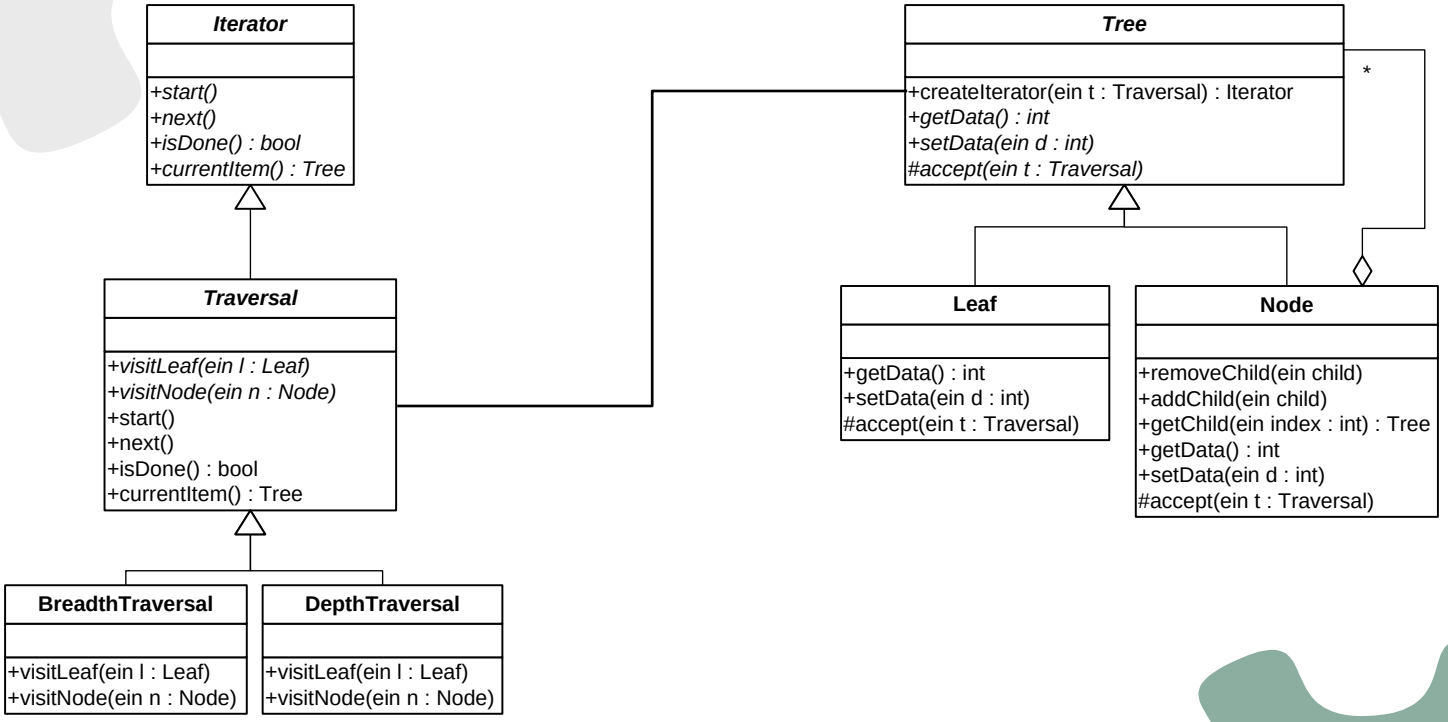
Für das aktuelle Einsatzgebiet der Anwendung genügen Ihnen zwei verschiedene Traversierungsarten (**Traversal**):
- *Ebenenweise (***BreadthTraversal***)*
- *Hauptreihenfolge* (**DepthTraversal***)*




`Traversal` verbindet das Iterator-Muster mit dem Besucher-Muster.

# Aufgabe 2a)

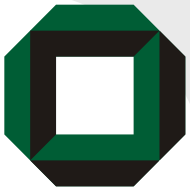
Erweitern der Baum-Datenstruktur um die notwendigen Operationen **`accept`** und **`createIterator`**.



# Aufgabe 2b)

- Begründen Sie kurz die Nützlichkeit des Entwurfsmusters Besucher in dem gegebenen Szenario.

- Eine mögliche Antwort:
  Der Besucher vereinfacht es, weitere Traversierungsarten zu implementieren, ohne vorhandene Klassen zu ändern.



# Aufgabe 3a)

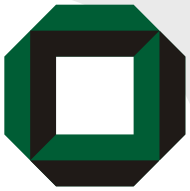
Gegeben sei folgender Java-Quelltext aus der Vorlesung, welcher von 2 Ausführungsfäden gleichzeitig ausgeführt wird:

```
if (globalVar > 0) {
  globalVar--;
}
```

Wie auch in der Vorlesung ist die Variable **`globalVar`** eine globale Variable vom Typ **`int`**, welche den Wert 1 enthält und auf welche von beiden Fäden zugegriffen werden kann. Zusätzlich sei diese Variable nun mit **`volatile`** definiert:

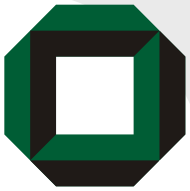
```
volatile int globalVar = 1;
```

Kann mit dieser Änderung auch eine Wettlaufsituation auftreten?



# Aufgabe 3a)

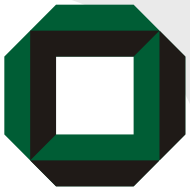
- Was bedeutet `volatile`?
  - Ist eine Variable als `volatile` deklariert, stellt die Java VM sicher, dass die Variable nicht im lokalen Cache (des Prozessors) zwischengespeichert wird, sondern die Aktualisierung der Variablen direkt im Hauptspeicher stattfindet.



# Aufgabe 3a)

- Auch bei der Deklaration der Variablen mit **`volatile`** kommt es in diesem Fall zu einem Wettlauf, d.h. die Variable **`globalvar`** wird negativ:

```
                // volatile int globalVar = 1

if (globalVar > 0) {
                                if (globalVar > 0) {
                                  globalVar--;
                                }
  globalVar--;
}
```



# Aufgabe 3b)

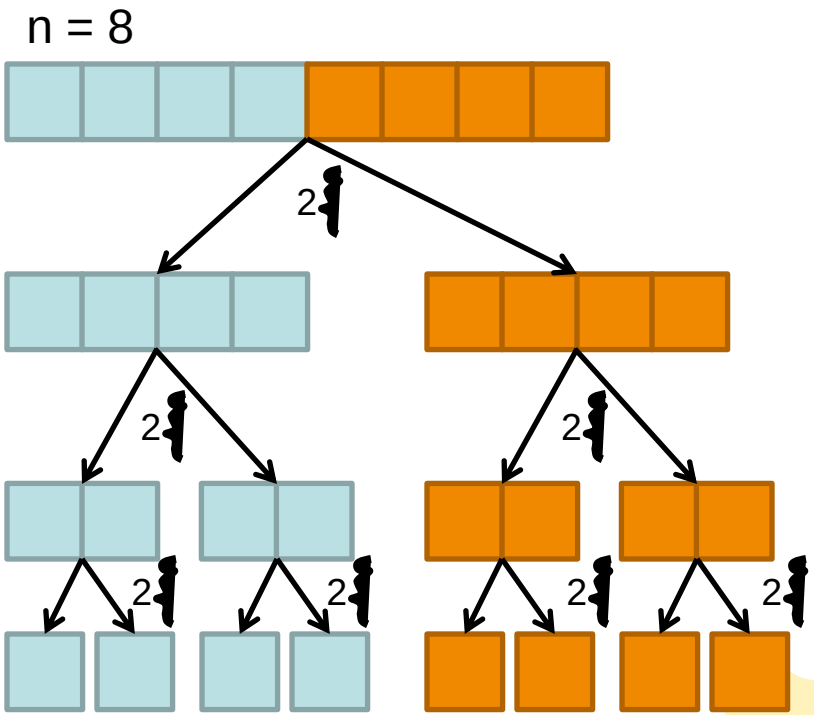
Gegeben seien eine sequentielle und eine parallele Implementierung eines Algorithmus, welcher die Summe aller Elemente in einem Feld bildet und auf dem Teile-und-Herrsche Prinzip basiert.

Angenommen, in der parallelen Implementierung des Algorithmus wird für jeden Aufruf von `sum(…)` einer neuer Faden erzeugt und gestartet.

1) Wie viele Fäden werden gestartet, wenn die parallele Implementierung der Methode `sum(…)` mit einem Feld der Größe `n` aufgerufen wird?
2) Wieso ist die sequentielle Implementierung i.d.R. schneller (bezogen auf die Laufzeit des Algorithmus) als die parallele Implementierung mit Java Threads?

# Aufgabe 3b)

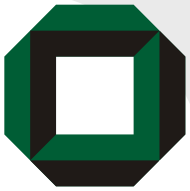
```
public int sum(int[] array) {
  // Abbruchbedingung
  if (array.length <= 1) {
    return array;
  }

  // startet neuen Faden
  int r1 =
    sum(getLeftSubArray(array));

  // startet neuen Faden
  int r2 =
    sum(getRightSubArray(array));

  int result = r1 + r2;

  return result;
}
```




- 7*2 = 14 neue Fäden werden erstellt
- Mit Hauptfaden also 15 Fäden

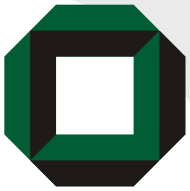
➔ Fäden insgesamt: $2^{(\log(n) + 1)} - 1$

$= 2n - 1$



# Aufgabe 3b)

- Wieso ist die sequentielle Implementierung i.d.R. schneller als die parallele Implementierung?
  - Das Erzeugen von neuen Fäden ist sehr zeitaufwändig.
  - Insbesondere in den letzten Zerteilen-Schritten übersteigt das Erstellen eines neuen Fadens die Zeit, die für das sequentielle Sortieren von x (abhängig vom verwendeten System) Elementen notwendig wäre.



# Aufgabe 4)

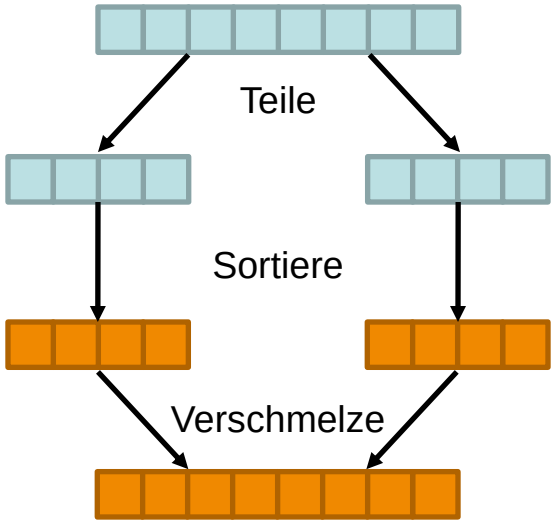
a) Implementieren Sie eine sequentielle Variante des Merge-Sort-Algorithmus zum Sortieren von Integer-Feldern (`int[]`).

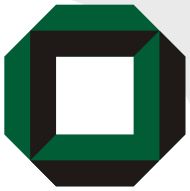
b) Parallelisieren Sie die sequentielle Variante des Merge-Sort-Algorithmus aus Teilaufgabe a). Verwenden Sie hierzu Java Threads.

- Verwenden Sie die zur Verfügung gestellte Klasse **`IOHelper`**, um Eingaben entgegenzunehmen (**`getNextIntegerArray()`**) und Ihre Ergebnisse auszugeben (**`printResult(int[] array)`**).

# Aufgabe 4)
# Funktionsweise von Merge-Sort

- Merge-Sort funktioniert nach dem Teile-Herrsche-Prinzip:





# Aufgabe 4a)

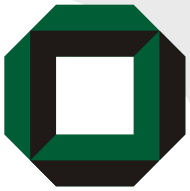
```
public class SequentialMergeSort {
  public int[] sort(int[] array) {
    if (array.length <= 1) { return array; }

    int[] part1 = new int[array.length / 2];
    int[] part2 = new int[array.length - part1.length];
    System.arraycopy(array, 0, part1, 0, array.length / 2);
    System.arraycopy(array, array.length / 2, part2, 0,
      array.length - part1.length);

    part1 = sort(s.getPart1());  part2 = sort(s.getPart2());

    int[] result = new int[part1.length + part2.length];
    int i1 = 0, i2 = 0;
    for (int k = 0; k < result.length;) {
      if ((i2 >= part2.length) ||
          (i1 < part1.length && part1[i1] <= part2[i2])) {
        result[k++] = part1[i1++];
      } else {
        result[k++] = part2[i2++];
      }
    }
    return result;
  }
}
```

Abbruchbedingung für den „Teilen“-Vorgang.

Integer-Feld in zwei gleiche Teile aufteilen.

Beide Teil-Felder sortieren.

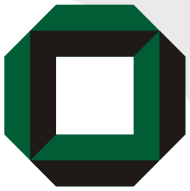
Beide Teil-Felder zu einem Feld zusammenfassen. Dabei auf die Sortierung achten!



# Aufgabe 4a) Schablonenmethode

- **Ziel:** Gemeinsames Verhalten aus Unterklassen herausfaktorisieren.
- In diesem Fall:
  - Das Aufteilen des Feldes (Split)
  - Das Zusammenführen der Teilergebnisse (Merge)

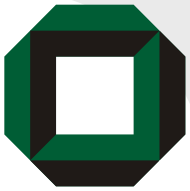
- Daher: Erstelle neue Klasse **`MergeSort`**, welche Funktionalität zum Aufteilen und Zusammenführen der Ergebnisse enthält.



# Aufgabe 4a)

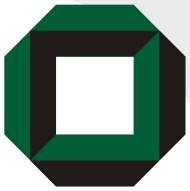
```
public abstract class MergeSort {
  protected final SplitArray split(int[] array) {
    return new SplitArray(array);
  }

  protected final int[] merge(int[] part1, int[]
   part2) {
    int[] array = new int[part1.length +
   part2.length];
    int i1 = 0, i2 = 0;
    for (int k = 0; k < array.length;) {
      if ((i2 >= part2.length) ||
          (i1 < part1.length && part1[i1] <=
   part2[i2])) {
        array[k++] = part1[i1++];
      } else {
        array[k++] = part2[i2++];
      }
    }
    return array;
  }

  public abstract int[] sort(int[] array);
}
```

```
public class SplitArray {
  public SplitArray(int[] array) {
    part1 = new int[array.length / 2];
    part2 = new int[array.length - part1.length];
    System.arraycopy(array, 0, part1,
      0, array.length / 2);
    System.arraycopy(array, array.length / 2,
      part2, 0, array.length - part1.length);
  }

  public int[] getPart1() {
    return part1;
  }

  public int[] getPart2() {
    return part2;
  }

  private int[] part1;
  private int[] part2;
}
```



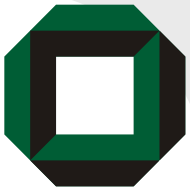
# Aufgabe 4a)

```
public class SequentialMergeSort extends MergeSort {

  @Override
  public int[] sort(int[] array) {
    if (array.length <= 1) {
      return array;
    }

    SplitArray s = split(array);

    int[] part1 = sort(s.getPart1());
    int[] part2 = sort(s.getPart2());

    int[] result = merge(part1, part2);

    return result;
  }
}
```



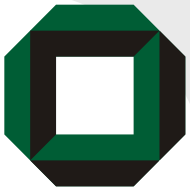
# Aufgabe 4b)

```
public class ParallelMergeSort extends MergeSort {
  private static final int MIN_SIZE_FOR_PARALLEL_SORT = 64;
  private static final SequentialMergeSort SEQUENTIAL_INST = new SequentialMergeSort();

  @Override
  public int[] sort(int[] array) {
    MergeSortWorker msw = new MergeSortWorker(array);
    Thread master = new Thread(msw);
    master.start();

    try {
      master.join();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }

    return msw.getSortedArray();
  }
```



# Aufgabe 4b)

```
private class MergeSortWorker implements Runnable {
  private int[] array = null;

  public MergeSortWorker(int[] array) { this.array = array; }
  public int[] getSortedArray() { return array; }

  public void run() {
    if (MIN_SIZE_FOR_PARALLEL_SORT < array.length) {
      SplitArray s = split(array);
      int[] part1 = s.getPart1();  int[] part2 = s.getPart2();

      MergeSortWorker msw1 = new MergeSortWorker(part1);
      MergeSortWorker msw2 = new MergeSortWorker(part2);
      Thread worker1 = new Thread(msw1); Thread worker2 = new Thread(msw2);
      worker1.start(); worker2.start();

      try { worker1.join(); worker2.join(); }
      catch (InterruptedException e) { e.printStackTrace(); }

      array = merge(msw1.getSortedArray(), msw2.getSortedArray());
    } else {
      array = SEQUENTIAL_INST.sort(array);
    }
  }
} }
```



# Aufgabe 4)

```
public final class TestRunner {
  final static int ARRAY_SIZE = 32;

  public static void main(String[] args) {
    IOHelper ioh = new IOHelper();
    int[] arrayPar = ioh.getNextIntegerArray();

    MergeSort par = new ParallelMergeSort();
    int[] resultPar = par.sort(arrayPar);

    ioh.printResult(resultPar);
  }
}
```