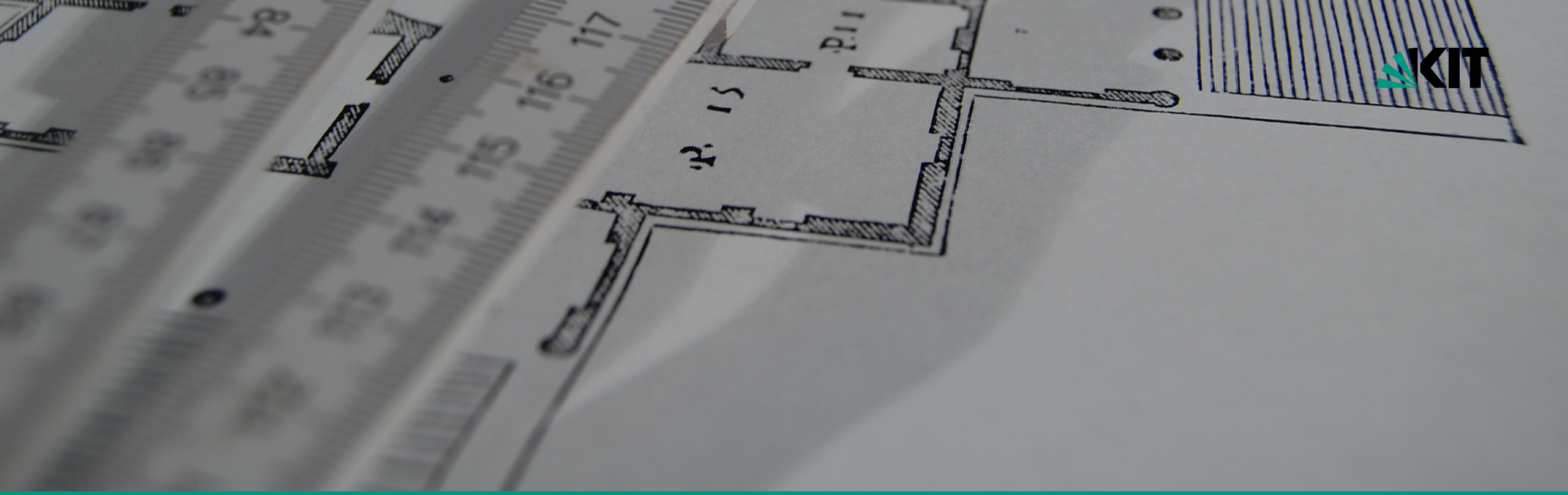

Generics und Rekursion

Programmieren Tutorium Nr.12

Aleksandr Zakharov | 20. Januar 2026

Generics

Die Idee dahinter

Nur einmalige Realisierung eines Programmierkonzepts (z.B. Liste) und zwar unabhängig vom Typ, der den Basisdaten zugrunde liegt.

Merke

Generics abstrahieren vom zugrunde liegenden Basistyp. (Interface abstrahiert von der Implementierung.)

Generics - Zum Nachdenken …

Problem bis Java 5 . . .

Datenstrukturen waren prinzipiell offen für jeden Typ.

- beim Speichern wurden Objekte vom allg. Typ ’Object’ entgegengenommen
- Rückgabewerte waren ebenfalls vom Typ ’Object’

Eine Liste soll aber z.B. nur T-Objekte und keine X-, Y- oder Z-Objekte enthalten. Das kann mit dem allg. Typ ’Object’ jedoch nicht verhindert werden.

Generics - Zum Nachdenken II ...

Beispiel

```
1 List myIntList = new ArrayList();
2 myIntList.add(new Integer(0));
3 Integer x = (Integer) myIntList.iterator().next();
```

Der ’cast’ in Zeile 3 ist notwendig, da Iterator nur Daten vom Typ ’Object’ liefert.

Generics - Zum Nachdenken II ...

Beispiel

```
1 List myIntList = new ArrayList();
2 myIntList.add(new Integer(0));
3 Integer x = (Integer) myIntList.iterator().next();
```

Der 'cast' in Zeile 3 ist notwendig, da Iterator nur Daten vom Typ 'Object' liefert.

- Warum kann hier ein Runtime Error auftreten, falls der Entwickler mit Integer falsch liegt?

Generics - Zum Nachdenken II ...

Beispiel

```
1 List myIntList = new ArrayList();
2 myIntList.add(new Integer(0));
3 Integer x = (Integer) myIntList.iterator().next();
```

Der 'cast' in Zeile 3 ist notwendig, da Iterator nur Daten vom Typ 'Object' liefert.

- Warum kann hier ein Runtime Error auftreten, falls der Entwickler mit Integer falsch liegt?
- Wie könnte man das Problem vermeiden?

Generics - Zum Nachdenken II ...

Beispiel

```
1 List myIntList = new ArrayList();
2 myIntList.add(new Integer(0));
3 Integer x = (Integer) myIntList.iterator().next();
```

Der 'cast' in Zeile 3 ist notwendig, da Iterator nur Daten vom Typ 'Object' liefert.

- Warum kann hier ein Runtime Error auftreten, falls der Entwickler mit Integer falsch liegt?
- Wie könnte man das Problem vermeiden?
- Gefährlich, falls andere sie modifizieren können:

```
1 List myIntList = new ArrayList();
2 myIntList.add("Evil Incarnate, Inc."); // Schade :(
```

Generics - Zum Nachdenken III ...

Beispiel zur Vermeidung

```
1 List<Integer> myIntList = new LinkedList<Integer>();
2 myIntList.add(new Integer(0));
3 Integer x = myIntList.iterator().next();
```

Generics - Zum Nachdenken III ...

Beispiel zur Vermeidung

```
1 List<Integer> myIntList = new LinkedList<Integer>();
2 myIntList.add(new Integer(0));
3 Integer x = myIntList.iterator().next();
```

Seit Java 5 ...

Mit Java 5 nutzt die Collection-API sehr intensiv Generics. Daraus folgt bessere Typsicherheit (nur spezielle Objekte in der Datenstruktur), da angegeben werden kann, welche Typen in der Liste erlaubt sind

Generics - Zum Nachdenken III ...

Beispiel zur Vermeidung

```
1 List<Integer> myIntList = new LinkedList<Integer>();
2 myIntList.add(new Integer(0));
3 Integer x = myIntList.iterator().next();
```

Seit Java 5 ...

Mit Java 5 nutzt die Collection-API sehr intensiv Generics. Daraus folgt bessere Typsicherheit (nur spezielle Objekte in der Datenstruktur), da angegeben werden kann, welche Typen in der Liste erlaubt sind

Beispiel

Was ist hier erlaubt

```
1 List<Disco> discos = new ArrayList<Disco>();
```

Generics - Typeinschränkung

Typeinschränkung möglich!

Es ist möglich, den Typ des Parameters einzuschränken!

```
1 class Name<T extends Point> {
2         ...
3 }
```

Generics und Vererbung

Subtypen

- Ist dies möglich:

```
1 List<Dog> dogs = new ArrayList<>();
2 // Liste von Hunden ist-ein Liste von Tieren
3 List<Animal> animals = dogs;
```

Generics und Vererbung

Subtypen

- Ist dies möglich:

```
1 List<Dog> dogs = new ArrayList<>();
2 // Liste von Hunden ist-ein Liste von Tieren
3 List<Animal> animals = dogs;
```

- **Nein!** Warum nicht?

Generics und Vererbung

Subtypen

- Ist dies möglich:

```
1 List<Dog> dogs = new ArrayList<>();
2 // Liste von Hunden ist-ein Liste von Tieren
3 List<Animal> animals = dogs;
```

- **Nein!** Warum nicht?

```
1 List<Dog> dogs = new ArrayList<>();
2 List<Animal> animals = dogs;
3 animals.add(new Cat()); // Autsch :(
```

- Was ist mit Arrays?

Generics und Vererbung

Subtypen

- Ist dies möglich:

```
1 List<Dog> dogs = new ArrayList<>();
2 // Liste von Hunden ist-ein Liste von Tieren
3 List<Animal> animals = dogs;
```

- **Nein!** Warum nicht?

```
1 List<Dog> dogs = new ArrayList<>();
2 List<Animal> animals = dogs;
3 animals.add(new Cat()); // Autsch :(
```

- Was ist mit Arrays?
  Da geht das! **Aufpassen!**

Generics - Zusammenfassung

Syntax

```
1 class name<Typ–Parameter> { ... }
```

```
1 interface Name<Typ–Parameter> { ... }
```

Beispiel

- Klassendeklaration:

```
1 class LinkedList<Data> {  ...  }
```

Generics - Zusammenfassung

Syntax

```
1 class name<Typ–Parameter> { ... }
```

```
1 interface Name<Typ–Parameter> { ... }
```

Beispiel

- Klassendeklaration:

```
1 class LinkedList<Data> {  ...  }
```

- Verwendung:

```
1 LinkedList<Point> ps = new LinkedList<Point>();
```

Generics - Zusammenfassung

Wichtig!

Die Typ-Parameter dürfen NUR mit Referenz-Datentypen instantiiert werden!

Primitive Datentypen sind also NICHT erlaubt!

Rekursion

Divide and Conquer

’divide and conquer’ - ’Teile und Herrsche’ ist ein wichtiges Grundprinzip der Algorithmik:

Rekursion

Divide and Conquer

’divide and conquer’ - ’Teile und Herrsche’ ist ein wichtiges Grundprinzip der Algorithmik:

- Teile ein Problem in mehrere Teilprobleme, um es zu lösen

Rekursion

Divide and Conquer

’divide and conquer’ - ’Teile und Herrsche’ ist ein wichtiges Grundprinzip der Algorithmik:

- Teile ein Problem in mehrere Teilprobleme, um es zu lösen
- Löse jedes einzelne (kleinere) Teilproblem

Rekursion

Divide and Conquer

'divide and conquer' - 'Teile und Herrsche' ist ein wichtiges Grundprinzip der Algorithmik:

- Teile ein Problem in mehrere Teilprobleme, um es zu lösen
- Löse jedes einzelne (kleinere) Teilproblem
- Füge die einzelnen (kleineren) Teillösungen zur Gesamtlösung zusammen

Rekursionsprinzip

Man führe das gleiche Berechnungsmuster immer wieder mit einfacheren, bzw. kleineren Eingabedaten aus, bis man zu einer trivialen Eingabe gelangt.
Realisierung: Methoden rufen sich direkt oder indirekt selbst auf
Rekursion ist die Standardimplementierung von Divide and Conquer

Rekursive Methoden

Definition

- Eine Methode f heißt **(direkt) rekursiv**, wenn im Rumpf von f Aufrufe von f vorkommen.

[1] De facto heißt das ein neuer Stackframe usw...

Rekursive Methoden

Definition

- Eine Methode f heißt **(direkt) rekursiv**, wenn im Rumpf von f Aufrufe von f vorkommen.
- Die Methode f heißt **indirekt rekursiv**, wenn im Rumpf von f eine Methode g aufgerufen wird, die ihrerseits direkt oder indirekt auf Aufrufe von f führt.

[1] De facto heißt das ein neuer Stackframe usw...

Rekursive Methoden

Definition

- Eine Methode f heißt **(direkt) rekursiv**, wenn im Rumpf von f Aufrufe von f vorkommen.
- Die Methode f heißt **indirekt rekursiv**, wenn im Rumpf von f eine Methode g aufgerufen wird, die ihrerseits direkt oder indirekt auf Aufrufe von f führt.

Merke

- Bei jedem rekursiven Aufruf wird eine **neue Instanz**[1] der jeweiligen Methode gestartet.
  $\Rightarrow$ Jede Instanz hat ihre eigenen lokalen Variablen und Parameter, welche 'von außen' nicht sichtbar sind.
- Wenn die aufgerufene Untermethode terminiert, wird in der aufrufenden Methode weiter gemacht!

[1] De facto heißt das ein neuer Stackframe usw...

Rekursion - Obligatorisches Meme

[Quelle: https://www.smbc-comics.com/comic/recursion - Recursion]

Beispiel: Fakultät

Fakultät

Die Fakultätsfunktion n! berechnet das Produkt der Zahlen 1, 2, . . . , n. Rekursiv lässt sich n! daher so berechnen:

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

In Java:

Beispiel: Fakultät

Fakultät

Die Fakultätsfunktion n! berechnet das Produkt der Zahlen 1, 2, . . . , n. Rekursiv lässt sich n! daher so berechnen:

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

In Java:

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

Beispiel: Fakultät - was passiert

Wie sieht das nun aus, wenn wir fac(4) aufrufen?

```
fac(4)
```

Beispiel: Fakultät - was passiert

Wie sieht das nun aus, wenn wir fac(4) aufrufen?

```
fac(4)
= ( if (4>0) { 4*fac(4-1) } else { 1 } ) // Einsetzen
```

Beispiel: Fakultät - was passiert

Wie sieht das nun aus, wenn wir fac(4) aufrufen?

```
fac(4)
= ( if (4>0) { 4*fac(4-1) } else { 1 } ) // Einsetzen
= 4*fac(3)                               // Auswerten
```

Beispiel: Fakultät - was passiert

Wie sieht das nun aus, wenn wir fac(4) aufrufen?

```
fac(4)
= ( if (4>0) { 4*fac(4-1) } else { 1 } )   // Einsetzen
= 4*fac(3)                                 // Auswerten
= 4*( if (3>0) { 3*fac(3-1) } else { 1 } ) // Einsetzen
```

Beispiel: Fakultät - was passiert

Wie sieht das nun aus, wenn wir fac(4) aufrufen?

```
fac(4)
= ( if (4>0) { 4*fac(4-1) } else { 1 } )   // Einsetzen
= 4*fac(3)                                 // Auswerten
= 4*( if (3>0) { 3*fac(3-1) } else { 1 } ) // Einsetzen
= 4*3*fac(2)                               // Auswerten
```

Beispiel: Fakultät - was passiert

Wie sieht das nun aus, wenn wir fac(4) aufrufen?

```
fac(4)
= ( if (4>0) { 4*fac(4-1) } else { 1 } )     // Einsetzen
= 4*fac(3)                                   // Auswerten
= 4*( if (3>0) { 3*fac(3-1) } else { 1 } )   // Einsetzen
= 4*3*fac(2)                                 // Auswerten
= 4*3*( if (2>0) { 2*fac(2-1) } else { 1 } ) // Einsetzen
```

Beispiel: Fakultät - was passiert

Wie sieht das nun aus, wenn wir fac(4) aufrufen?

```
fac(4)
= ( if (4>0) { 4*fac(4-1) } else { 1 } )     // Einsetzen
= 4*fac(3)                                   // Auswerten
= 4*( if (3>0) { 3*fac(3-1) } else { 1 } )   // Einsetzen
= 4*3*fac(2)                                 // Auswerten
= 4*3*( if (2>0) { 2*fac(2-1) } else { 1 } ) // Einsetzen
= 4*3*2*fac(1)                               // Auswerten
```

Beispiel: Fakultät - was passiert

Wie sieht das nun aus, wenn wir fac(4) aufrufen?

```
fac(4)
= ( if (4>0) { 4*fac(4-1) } else { 1 } )       // Einsetzen
= 4*fac(3)                                     // Auswerten
= 4*( if (3>0) { 3*fac(3-1) } else { 1 } )     // Einsetzen
= 4*3*fac(2)                                   // Auswerten
= 4*3*( if (2>0) { 2*fac(2-1) } else { 1 } )   // Einsetzen
= 4*3*2*fac(1)                                 // Auswerten
= 4*3*2*( if (1>0) { 1*fac(1-1) } else { 1 } ) // Einsetzen
```

Beispiel: Fakultät - was passiert

Wie sieht das nun aus, wenn wir fac(4) aufrufen?

```
fac(4)
= ( if (4>0) { 4*fac(4-1) } else { 1 } )       // Einsetzen
= 4*fac(3)                                     // Auswerten
= 4*( if (3>0) { 3*fac(3-1) } else { 1 } )     // Einsetzen
= 4*3*fac(2)                                   // Auswerten
= 4*3*( if (2>0) { 2*fac(2-1) } else { 1 } )   // Einsetzen
= 4*3*2*fac(1)                                 // Auswerten
= 4*3*2*( if (1>0) { 1*fac(1-1) } else { 1 } ) // Einsetzen
= 4*3*2*1*fac(0)                               // Auswerten
```

Beispiel: Fakultät - was passiert

Wie sieht das nun aus, wenn wir fac(4) aufrufen?

```
fac(4)
= ( if (4>0) { 4*fac(4-1) } else { 1 } )          // Einsetzen
= 4*fac(3)                                        // Auswerten
= 4*( if (3>0) { 3*fac(3-1) } else { 1 } )        // Einsetzen
= 4*3*fac(2)                                      // Auswerten
= 4*3*( if (2>0) { 2*fac(2-1) } else { 1 } )      // Einsetzen
= 4*3*2*fac(1)                                    // Auswerten
= 4*3*2*( if (1>0) { 1*fac(1-1) } else { 1 } )    // Einsetzen
= 4*3*2*1*fac(0)                                  // Auswerten
= 4*3*2*1*( if (0>0) { 0*fac(0-1) } else { 1 } ) // Einsetzen
```

Beispiel: Fakultät - was passiert

Wie sieht das nun aus, wenn wir fac(4) aufrufen?

```
fac(4)
= ( if (4>0) { 4*fac(4-1) } else { 1 } )          // Einsetzen
= 4*fac(3)                                        // Auswerten
= 4*( if (3>0) { 3*fac(3-1) } else { 1 } )        // Einsetzen
= 4*3*fac(2)                                      // Auswerten
= 4*3*( if (2>0) { 2*fac(2-1) } else { 1 } )      // Einsetzen
= 4*3*2*fac(1)                                    // Auswerten
= 4*3*2*( if (1>0) { 1*fac(1-1) } else { 1 } )    // Einsetzen
= 4*3*2*1*fac(0)                                  // Auswerten
= 4*3*2*1*( if (0>0) { 0*fac(0-1) } else { 1 } ) // Einsetzen
= 4*3*2*1*1                                       // Auswerten
```

Beispiel: Fakultät - was passiert

Wie sieht das nun aus, wenn wir fac(4) aufrufen?

```
fac(4)
= ( if (4>0) { 4*fac(4-1) } else { 1 } )          // Einsetzen
= 4*fac(3)                                        // Auswerten
= 4*( if (3>0) { 3*fac(3-1) } else { 1 } )        // Einsetzen
= 4*3*fac(2)                                      // Auswerten
= 4*3*( if (2>0) { 2*fac(2-1) } else { 1 } )      // Einsetzen
= 4*3*2*fac(1)                                    // Auswerten
= 4*3*2*( if (1>0) { 1*fac(1-1) } else { 1 } )    // Einsetzen
= 4*3*2*1*fac(0)                                  // Auswerten
= 4*3*2*1*( if (0>0) { 0*fac(0-1) } else { 1 } ) // Einsetzen
= 4*3*2*1*1                                       // Auswerten
= 24
```

Beispiel: Quicksort - rekursives Sortieren

Lösungsmethode

Wir können Quicksort wunderbar rekursiv lösen, indem wir *Divide and Conquer* umsetzen . . .

Beispiel: Quicksort - rekursives Sortieren

Lösungsmethode

Wir können Quicksort wunderbar rekursiv lösen, indem wir *Divide and Conquer* umsetzen ...

- Wähle beliebiges Pivot-Element p und sortiere das Array in zwei Teilen

Beispiel: Quicksort - rekursives Sortieren

Lösungsmethode

Wir können Quicksort wunderbar rekursiv lösen, indem wir *Divide and Conquer* umsetzen ...

- Wähle beliebiges Pivot-Element p und sortiere das Array in zwei Teilen
  - Links Elemente < p
  - Mitte Elemente == p
  - Rechts Elemente > p
  - Wenn man das im Array selbst tut, steht p anschließend an der richtigen Stelle.

Beispiel: Quicksort - rekursives Sortieren

Lösungsmethode

Wir können Quicksort wunderbar rekursiv lösen, indem wir *Divide and Conquer* umsetzen ...

- Wähle beliebiges Pivot-Element p und sortiere das Array in zwei Teilen
  - Links Elemente < p
  - Mitte Elemente == p
  - Rechts Elemente > p
  - Wenn man das im Array selbst tut, steht p anschließend an der richtigen Stelle.
- Verfahre rekursiv mit dem linken und rechten Teil

Beispiel: Quicksort - rekursives Sortieren

Lösungsmethode

Wir können Quicksort wunderbar rekursiv lösen, indem wir *Divide and Conquer* umsetzen ...

- Wähle beliebiges Pivot-Element p und sortiere das Array in zwei Teilen
  - Links Elemente < p
  - Mitte Elemente == p
  - Rechts Elemente > p
  - Wenn man das im Array selbst tut, steht p anschließend an der richtigen Stelle.
- Verfahre rekursiv mit dem linken und rechten Teil
- Gibt es nur noch elementare Blöcke, so ist das Array sortiert

Beispiel: Quicksort - rekursives Sortieren

Lösungsmethode

Wir können Quicksort wunderbar rekursiv lösen, indem wir *Divide and Conquer* umsetzen ...

- Wähle beliebiges Pivot-Element p und sortiere das Array in zwei Teilen
  - Links Elemente < p
  - Mitte Elemente == p
  - Rechts Elemente > p
  - Wenn man das im Array selbst tut, steht p anschließend an der richtigen Stelle.
- Verfahre rekursiv mit dem linken und rechten Teil
- Gibt es nur noch elementare Blöcke, so ist das Array sortiert
- Der ganze Sortieraufwand steckt also in der Zerlegung in kleinere Teilprobleme (und nicht im Vereinen der Lösungen)

Quicksort - Implementierung

```java
public class Quicksort {
    public static void sort(long[] a) {
        qsort(a, 0, a.length - 1);
    }
    /* Sortiert Array a[left .. right] */
    private static void qsort(long[] a, int left, int right) {
        int i = left;
        int j = right;
        long p = pivot(a, left, right); // Waehle einbel. Element
        while (i <= j) {
          while (a[i] < p) { i++; } // Suche El. links >= p
          while (a[j] > p) { j--; } // Suche El. rechts <= p
          if (i <= j) {
              long tmp = a[i];
              a[i] = a[j]; a[j] = tmp; // Vertausche
              i++; j--;
          }
        }
        if (left < j) { qsort(a, left, j); } // Sortiere linken Teil
        if (i < right) { qsort(a, i, right); } // Sortiere rechten Teil
    }
}
```

Präsenzübung

Übungsschein – Zusammengefasst

Präsenz/Blatt	> 50%	<= 50%
> 75%	Schein	Kein Schein
<= 75%	Kein Schein	Kein Schein

Bei triftigem Grund (z.B. Krankheit)

Bei Rücktritt von der Präsenzübung aus triftigem Grund und > 50% in Übungsblättern, müsst ihr nur die Präsenzübung wiederholen!

Ergebnisse & Einsicht

- Ergebnis wird im Artemis veröffentlicht
- Einsicht im Tutorium

Präsenzübung

Zeit/Ort

- Wann: 21.01.2026 17:30 - 20:00 Genaue Zeit unter https://news.praktomat.cs.kit.edu/
- Wo: Ort individuell unter https://news.praktomat.cs.kit.edu/

Präsenzübung

Formalitäten

- Keine Hilfsmittel
- Dokumentechter Stift (nicht rot), Studentenausweis, Personalausweis (!!!)
- Nach Abgabe warten bis alle Abgaben gezählt sind
- Wegen Kontrolle mindestens 15 Minuten früher da sein
- Tutoriumsnummer wissen (Wir sind Tutorium 17)
- **Jeder Betrugsversuch zählt sofort als „nicht bestanden“!**

Präsenzübung

F R A G E N ? ? ?

Präsenzübung

Beispiele

Lösen wir ein paar Beispiele . . .