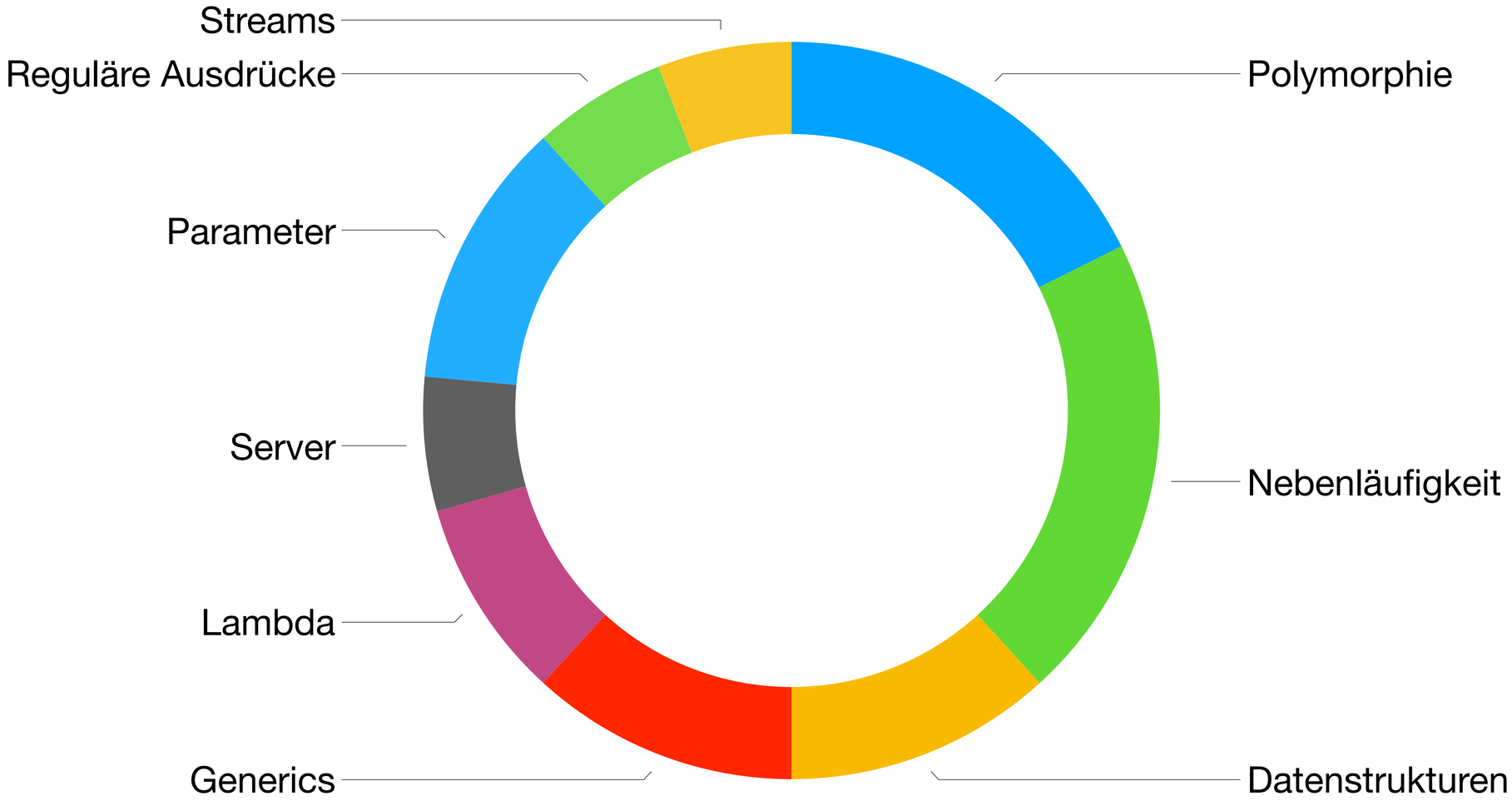
PGdP

- Parameter
- Garbage Collection
- RegEx
- Generics
- Datenstrukturen
- Streams
- Lambda
- Polymorphie
- Server
- Nebenläufigkeit

Folien: go.tum.de/904005

PGdP

- Parameter
- Garbage Collection
- RegEx
- Generics
- Datenstrukturen
- Streams
- Lambda
- Polymorphie
- Server
- Nebenläufigkeit



Eure Wünsche

Parameter
Garbage Collection
RegEx
Generics
Datenstrukturen
Streams
Lambda
Polymorphie
Server
Nebenläufigkeit

Parameter

Java ist 'Call-by-Value' > Kopie des Wertes wird übergeben

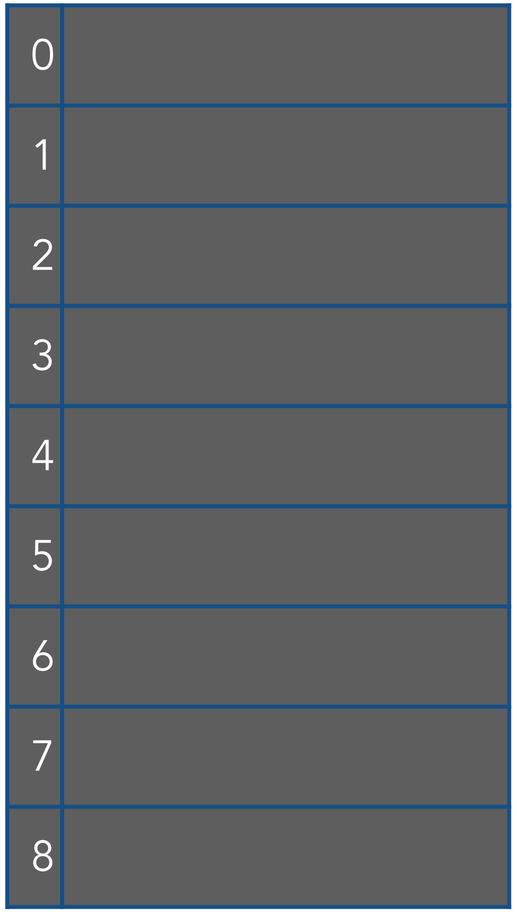
- Innerhalb der Methode wird eine **Kopie eines primitiven Datentypen** verändert > keine Veränderung des Originals
- Innerhalb einer Methode wird mithilfe einer **Kopie der Referenz auf einen komplexen Datentyp** die Stelle im Speicher verändert, an der sich das Original befindet
  > Veränderung des Originals
  > Referenz kann auch überschrieben werden

Parameter
Garbage Collection
RegEx
Generics
Datenstrukturen
Streams
Lambda
Polymorphie
Server
Nebenläufigkeit

Parameter

Primitive Datentypen

```
1   public static void
2    changeWert(int b) {
3     b = 0;
4     return;
5   }
6
7   int zahl = 2;
8   changeWert(zahl);
9   write("Zahl: " + zahl);
10
11
12
```



Parameter
Garbage Collection
RegEx
Generics
Datenstrukturen
Streams
Lambda
Polymorphie
Server
Nebenläufigkeit

Parameter

Primitive Datentypen

```
1   public static void
2    changeWert(int b) {
3     b = 0;
4     return;
5   }
6
7   int zahl = 2;
8   changeWert(zahl);
9   write("Zahl: " + zahl);
10
11
12
```

Parameter
Garbage Collection
RegEx
Generics
Datenstrukturen
Streams
Lambda
Polymorphie
Server
Nebenläufigkeit

Parameter

Primitive Datentypen

```
1   public static void
2    changeWert(int b) {
3     b = 0;
4     return;
5   }
6
7   int zahl = 2;
8   changeWert(zahl);
9   write("Zahl: " + zahl);
10
11
12
```

Wert aus dem Speicher wird bei Methoden-aufruf kopiert.

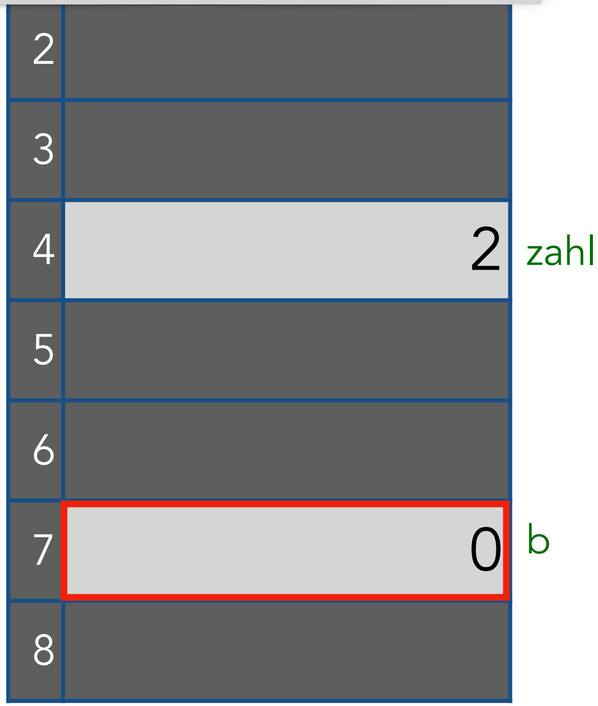
Parameter
Garbage Collection
RegEx
Generics
Datenstrukturen
Streams
Lambda
Polymorphie
Server
Nebenläufigkeit

Parameter

Primitive Datentypen

```
1   public static void
2    changeWert(int b) {
3     b = 0;
4     return;
5   }
6
7   int zahl = 2;
8   changeWert(zahl);
9   write("Zahl: " + zahl);
10
11
12
```

Änderungen nur auf der Kopie.



Parameter
Garbage Collection
RegEx
Generics
Datenstrukturen
Streams
Lambda
Polymorphie
Server
Nebenläufigkeit

Parameter

Primitive Datentypen

```
1   public static void
2    changeWert(int b) {
3     b = 0;
4     return;
5   }
6
7   int zahl = 2;
8   changeWert(zahl);
9   write("Zahl: " + zahl);
10
11
12
```

Lokale Variablen der Methode werden beim verlassen gelöscht.

Parameter
Garbage Collection
RegEx
Generics
Datenstrukturen
Streams
Lambda
Polymorphie
Server
Nebenläufigkeit

Parameter

Primitive Datentypen

```
1   public static void
2    changeWert(int b) {
3     b = 0;
4     return;
5   }
6
7   int zahl = 2;
8   changeWert(zahl);
9   write("Zahl: " + zahl);
10
11
12
```

```
Zahl: 2
```

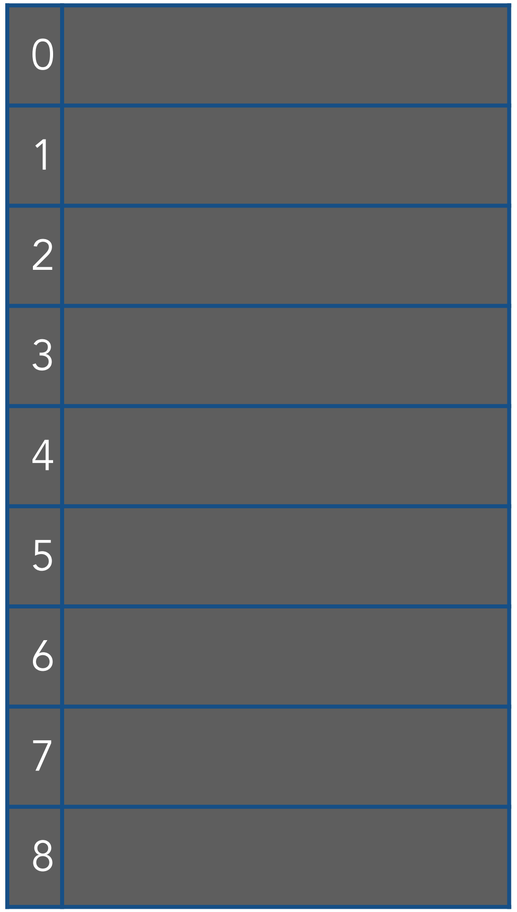
Übergebener primitiver Wert bleibt unverändert.

Parameter
Garbage Collection
RegEx
Generics
Datenstrukturen
Streams
Lambda
Polymorphie
Server
Nebenläufigkeit

Parameter

Komplexe Datentypen (Referenztypen)

```
1   public static void
2    changeWert(int[] arr) {
3     arr[0] = 6;
4     arr = new int[2];
5     return;
6   }
7
8   int[] feld = new int[1];
9   feld[0] = 7;
10  changeWert(feld);
11  write("Feld: " + feld[0]);
12
```

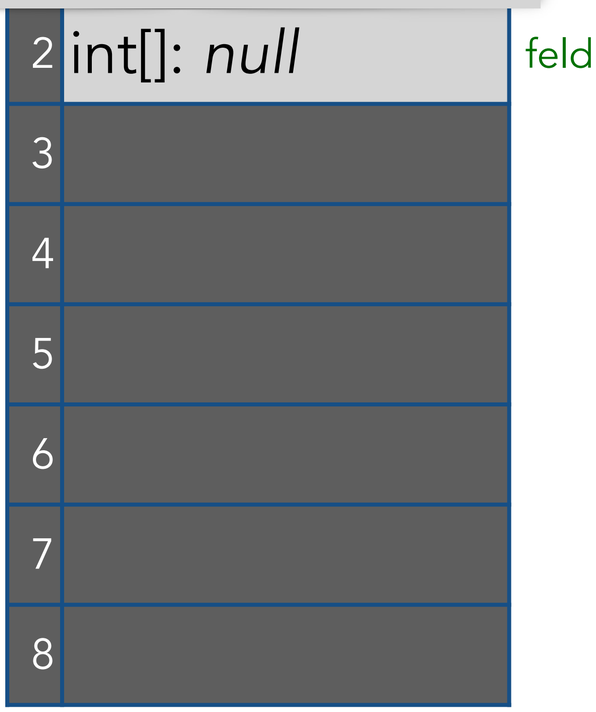


Parameter

Komplexe Datentypen (Referenztypen)

```
1   public static void
2    changeWert(int[] arr) {
3     arr[0] = 6;
4     arr = new int[2];
5     return;
6   }
7
8   int[] feld = new int[1];
9   feld[0] = 7;
10  changeWert(feld);
11  write("Feld: " + feld[0]);
12
```

Referenztyp für int-Arrays wird angelegt.



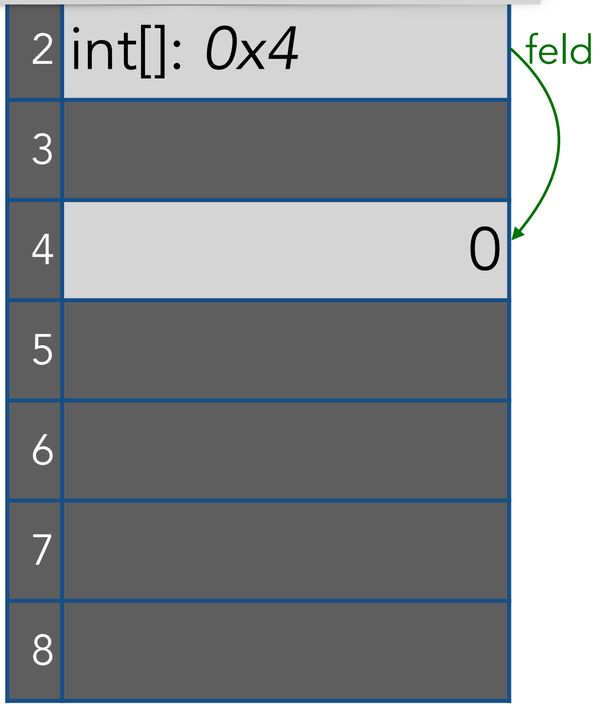
Parameter
Garbage Collection
RegEx
Generics
Datenstrukturen
Streams
Lambda
Polymorphie
Server
Nebenläufigkeit

Parameter

Komplexe Datentypen (Referenztypen)

```
1   public static void
2    changeWert(int[] arr) {
3     arr[0] = 6;
4     arr = new int[2];
5     return;
6   }
7
8   int[] feld = new int[1];
9   feld[0] = 7;
10  changeWert(feld);
11  write("Feld: " + feld[0]);
12
```

int-Array wird initialisiert.

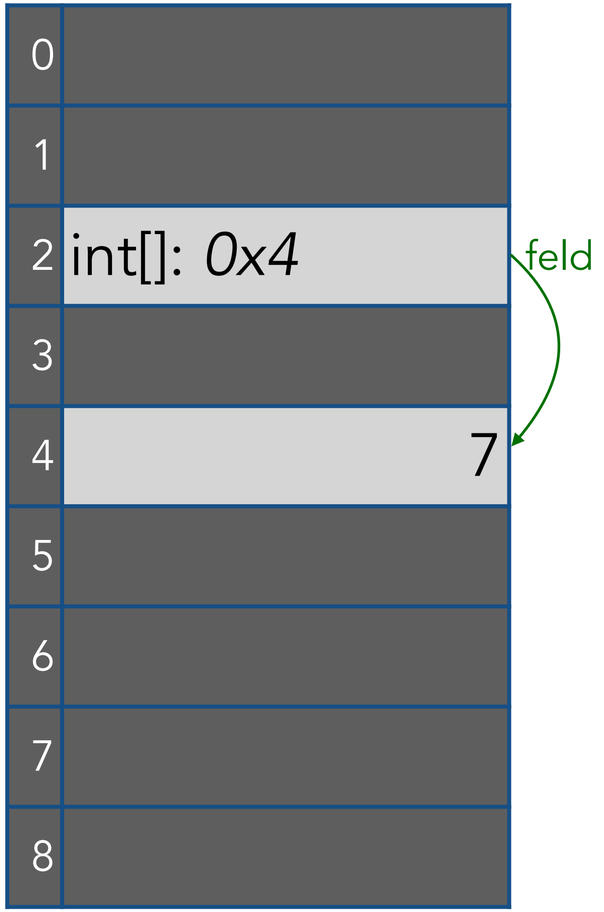


Parameter
Garbage Collection
RegEx
Generics
Datenstrukturen
Streams
Lambda
Polymorphie
Server
Nebenläufigkeit

Parameter

Komplexe Datentypen (Referenztypen)

```
1   public static void
2    changeWert(int[] arr) {
3     arr[0] = 6;
4     arr = new int[2];
5     return;
6   }
7
8   int[] feld = new int[1];
9   feld[0] = 7;
10  changeWert(feld);
11  write("Feld: " + feld[0]);
12
```



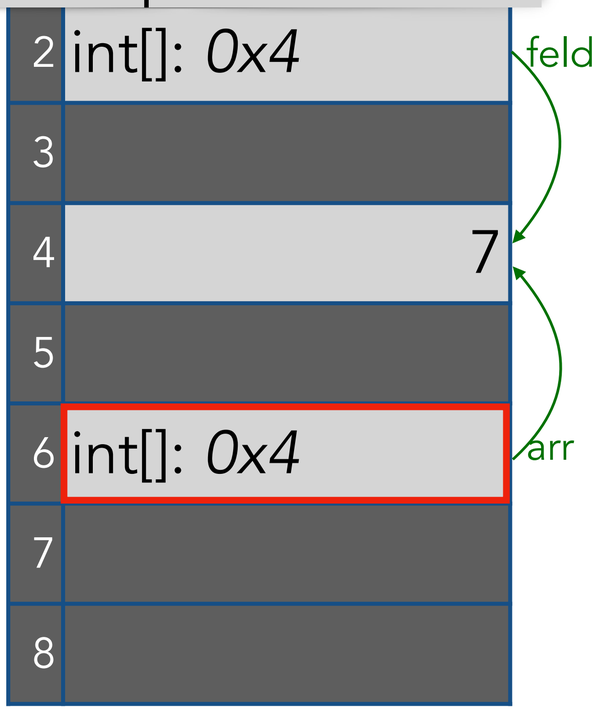
Parameter
Garbage Collection
RegEx
Generics
Datenstrukturen
Streams
Lambda
Polymorphie
Server
Nebenläufigkeit

Parameter

Komplexe Datentypen (Referenztypen)

```
1   public static void
2    changeWert(int[] arr) {
3      arr[0] = 6;
4      arr = new int[2];
5      return;
6   }
7
8   int[] feld = new int[1];
9   feld[0] = 7;
10  changeWert(feld);
11  write("Feld: " + feld[0]);
12
```

Referenzen aus Speicher werden bei Methodenaufruf kopiert.



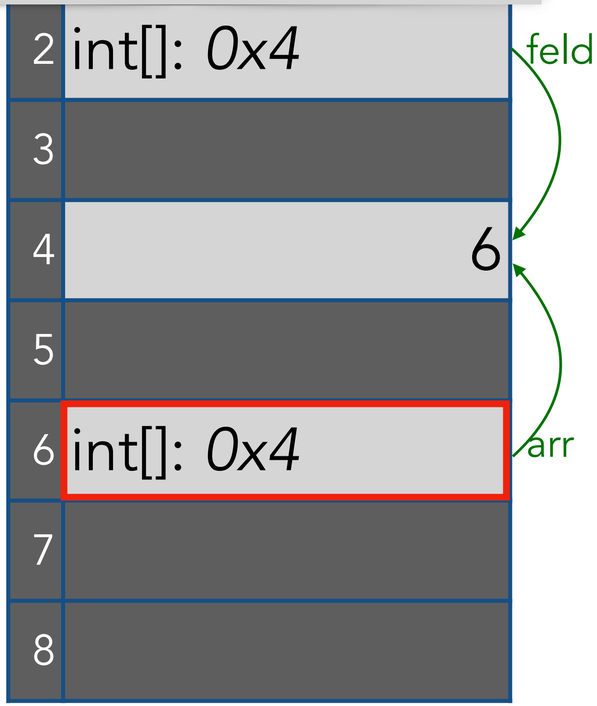
Parameter
Garbage Collection
RegEx
Generics
Datenstrukturen
Streams
Lambda
Polymorphie
Server
Nebenläufigkeit

Parameter

Komplexe Datentypen (Referenztypen)

```
1   public static void
2    changeWert(int[] arr) {
3     arr[0] = 6;
4     arr = new int[2];
5     return;
6   }
7
8   int[] feld = new int[1];
9   feld[0] = 7;
10  changeWert(feld);
11  write("Feld: " + feld[0]);
12
```

Durch die identische Referenz wird das alte Objekt/Array verändert.



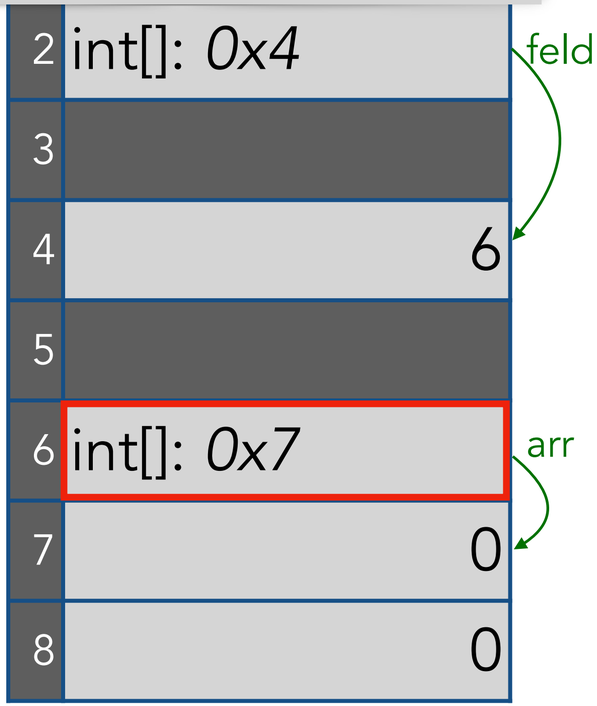
Parameter
Garbage Collection
RegEx
Generics
Datenstrukturen
Streams
Lambda
Polymorphie
Server
Nebenläufigkeit

Parameter

Komplexe Datentypen (Referenztypen)

```
1   public static void
2    changeWert(int[] arr) {
3     arr[0] = 6;
4     arr = new int[2];
5     return;
6   }
7
8   int[] feld = new int[1];
9   feld[0] = 7;
10  changeWert(feld);
11  write("Feld: " + feld[0]);
12
```

Kopierter Referenz wird durch 'new' überschrieben.



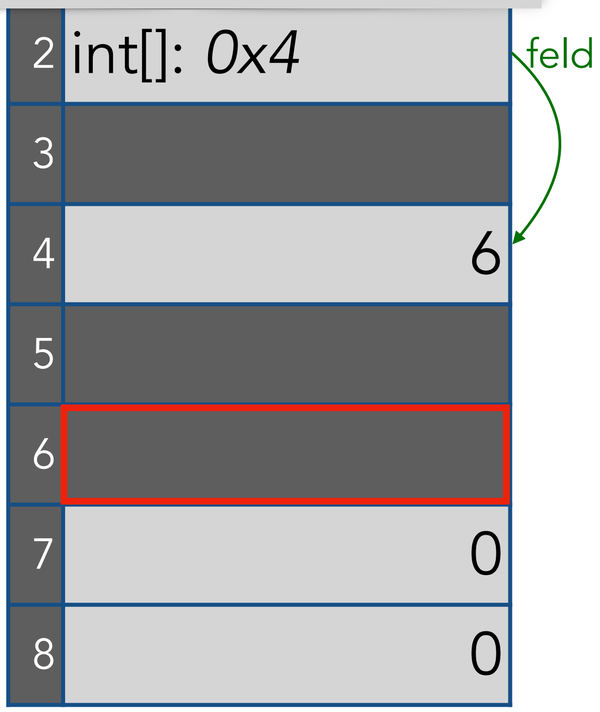
Parameter
Garbage Collection
RegEx
Generics
Datenstrukturen
Streams
Lambda
Polymorphie
Server
Nebenläufigkeit

Parameter

Komplexe Datentypen (Referenztypen)

```
1   public static void
2    changeWert(int[] arr) {
3     arr[0] = 6;
4     arr = new int[2];
5     return;
6   }
7
8   int[] feld = new int[1];
9   feld[0] = 7;
10  changeWert(feld);
11  write("Feld: " + feld[0]);
12
```

Lokale Variablen der Methode werden beim verlassen gelöscht.



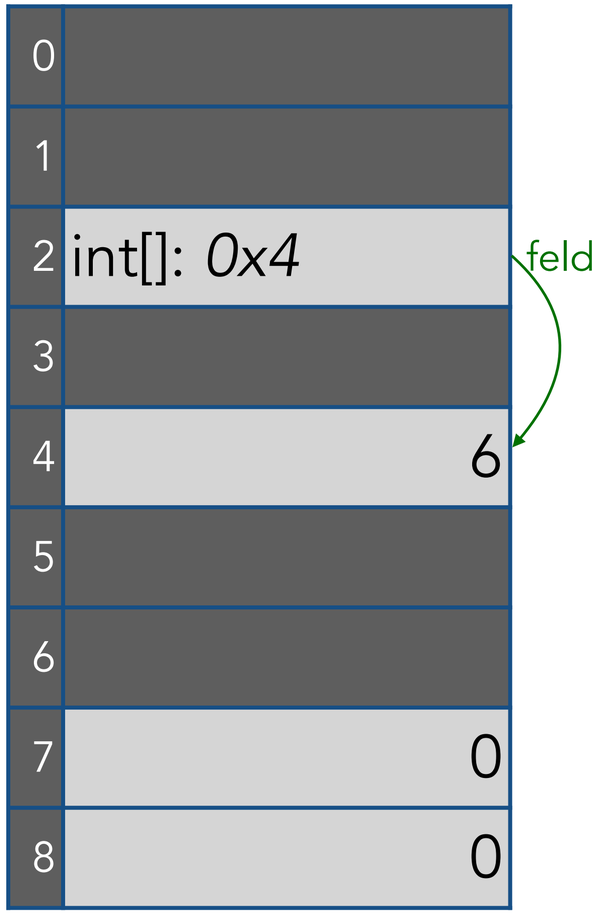
Parameter
Garbage Collection
RegEx
Generics
Datenstrukturen
Streams
Lambda
Polymorphie
Server
Nebenläufigkeit

Parameter

Komplexe Datentypen (Referenztypen)

```
1   public static void
2    changeWert(int[] arr) {
3     arr[0] = 6;
4     arr = new int[2];
5     return;
6   }
7
8   int[] feld = new int[1];
9   feld[0] = 7;
10  changeWert(feld);
11  write("Feld: " + feld[0]);
12
```

```
Feld: 6
```

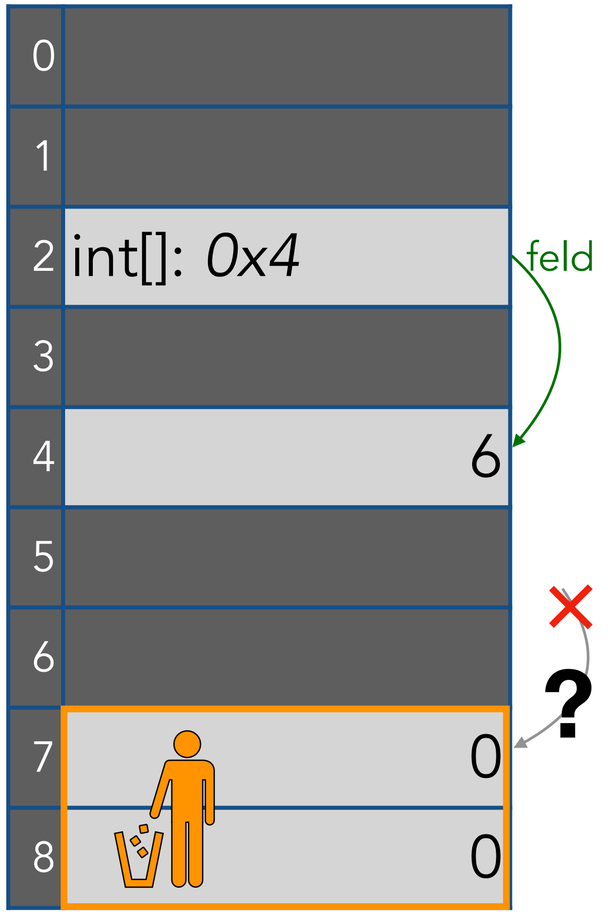


Garbage Collection

Garbage Collector entfernt Objekte ohne Referenzen

```
1   public static void
2    changeWert(int[] arr) {
3     arr[0] = 6;
4     arr = new int[2];
5     return;
6   }
7
8   int[] feld = new int[1];
9   feld[0] = 7;
10  changeWert(feld);
11  write("Feld: " + feld[0]);
12
```

```
Feld: 6
```

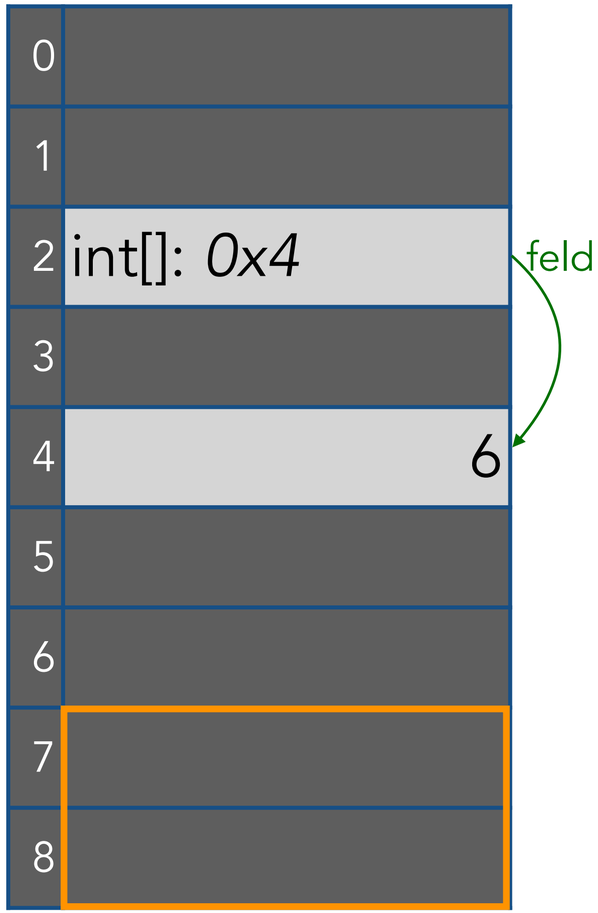


Garbage Collection

Garbage Collector entfernt Objekte ohne Referenzen

```
1   public static void
2    changeWert(int[] arr) {
3     arr[0] = 6;
4     arr = new int[2];
5     return;
6   }
7
8   int[] feld = new int[1];
9   feld[0] = 7;
10  changeWert(feld);
11  write("Feld: " + feld[0]);
12
```

```
Feld: 6
```



Reguläre Ausdrücke

Ausdruck	Bedeutung
`(text)`	Bündelung einer Zeichenkette
`+`	Konkatenation, zusammenhängen
`|`	Alternative, linkes oder rechtes Element
`*`	Iteration, keinmal bis unendlich mal; $[0,\infty]$
`?`	Option, einmal oder keinmal; $[0,1]$

```
(J|Y)an(n)?i(ck|k|c)
```

```
> Janic, Jannic, Janik, Jannik, Janick, Jannick
  Yanic, Yannic, Yanik, Yannik, Yanick, Yannick
```

Reguläre Ausdrücke

Es gibt in Java Methoden, die Regulä… …le als Parameter annehmen.

Nicht Klausurrelevant.

```
public String[] split(String regEx);
public String replaceAll(String regEx);
```

Mehr dazu: hier, nicht Klausurrelevant!

Generics

Generische Datentypen dienen als Platzhalter, damit ein Datentyp variabel festgelegt werden kann.

"Variablen für Datentypen"

```
class ListNode<T extends Number> {
  ListNode<T> next;
  T value;
}
```

Datentyp muss von Number erben

```
ListNode<Double> l = new ListNode<Double>();
```

Liste für Double

Liste für Double

~~`ListNode<Object> k = new ListNode<Object>();`~~

Generics

Generische Datentypen dienen als Platzhalter, damit ein Datentyp variabel festgelegt werden kann, Wildcards

`<?>` ≡ `<? extends Object>`

`<? extends H>`

Alle, die H als Oberklasse haben. H kann auch als Typ verwendet werden.

`<? super H>`

Alle, die in der Hierarchie über H stehen. Object muss in der Methode als Typ verwendet werden.

(?): Wildcards werden verwendet, um bei Methodenaufrufen, die Parameter mit generischen Typargumenten enthalten, eine höhere Flexibilität zu erreichen. Dazu ein kleines Beispiel:

```
public static void printCollection(LinkedList<Object> col) {
  for(Object s : col)
    System.out.println(s);
}
```

Die Methode kann nur mit einer Collection mit dem generischen Typen Object aufgerufen werden. Folgendes würde also einen Compilerfehler auslösen

```
LinkedList<String> stringListe = new LinkedList<String>();
printCollection(strinListe);
```

Um das zu verhindern, kann man für die Typparameter Wildcards definieren, bei denen die Methode dann ebenfalls funktioniert.

```
public static void printCollectionNew(LinkedList<? extends Object> col) {
  for(Object s : col)
    System.out.println(s);
}
LinkedList<String> stringListe = new LinkedList<String>();
printCollectionNew(strinListe);
```

In diesem Fall erlaubt man alle Typparameter (?), die als Oberklasse Object haben (also von Object erben; *Klasse* extends Object). Da alle Klassen von Object erben, kann man äquivalent dazu schreiben

```
public static void printCollectionNewSimplified(LinkedList<?> col) {
  for(Object s : col)
    System.out.println(s);
}
```

Neben 'super' kann man den Parameter mit ‚extends' auch nach unten beschränken (man erlaubt dann nur Oberklassen von dem, was nach super steht). Beispiel

```
public static void printCollectionNewLowerBounder(LinkedList<? super Integer> col) {
  for(Object s : col)
    System.out.println(s);
}
```

Hier wäre eine Liste von Numbers erlaubt, eine Liste von Doubles würde aber zu einem Fehler führen. (Siehe Klassenhierachie https://docs.oracle.com/javase/tutorial/java/data/numberclasses.html)

Parameter
Garbage Collection
RegEx
Generics
Datenstrukturen
Streams
Lambda
Polymorphie
Server
Nebenläufigkeit

Datenstrukturen

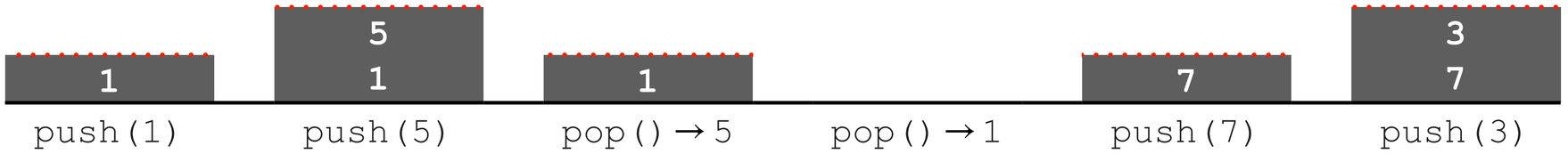
- Stack, Stapelspeicher
  > LIFO, Anwendung bei MiniJVM
- Queue, Schlange
  > FIFO
- List
  > Indexzugriffe
- (Binär-)Bäume
- Iteratoren

Datenstrukturen

Stack

- Last in, First out; LIFO
- Operationen:

```
> push(T value):void
> pop():T
> ...
```



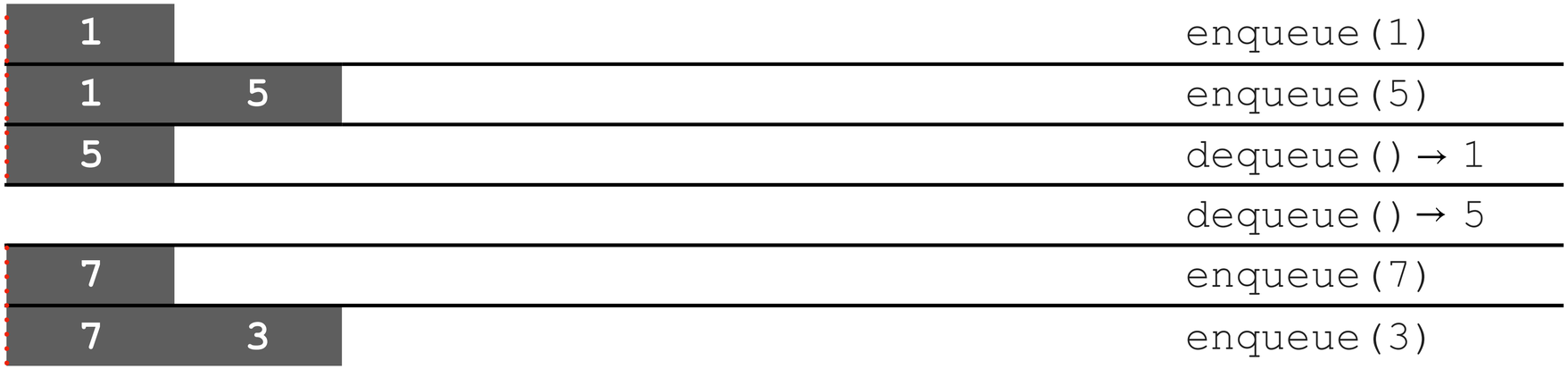
Parameter
Garbage Collection
RegEx
Generics
Datenstrukturen
Streams
Lambda
Polymorphie
Server
Nebenläufigkeit

Datenstrukturen

Queue

- First in, First out; FIFO
- Operationen:

```
> enqueue(T value):void
> dequeue():T
> ...
```

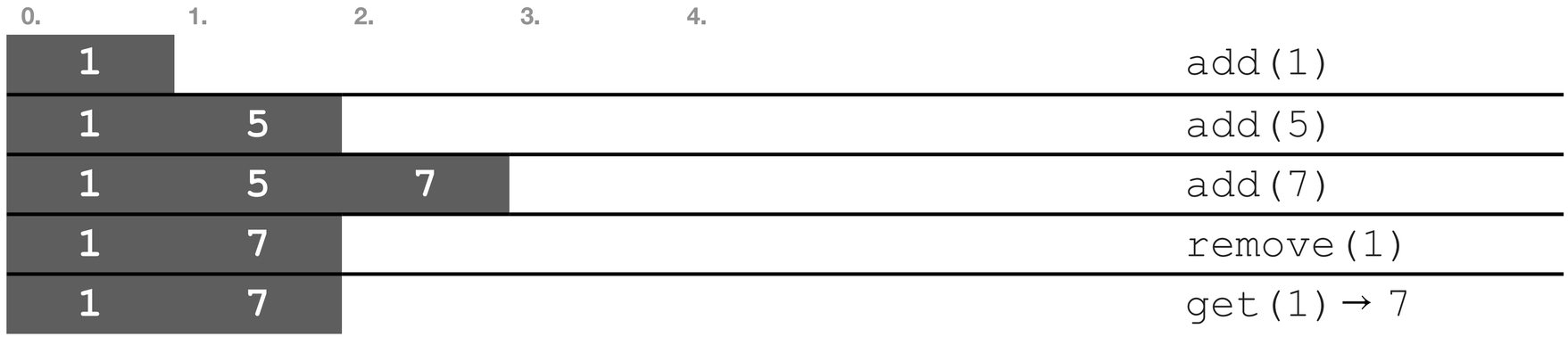


Datenstrukturen

List

- Indexzugriffe
- Operationen:

```
> add(T value):void
> get(int index):T
> remove(int index):void
> ...
```



Parameter
Garbage Collection
RegEx
Generics
Datenstrukturen
Streams
Lambda
Polymorphie
Server
Nebenläufigkeit

Datenstrukturen

List, Realisierung: ArrayList

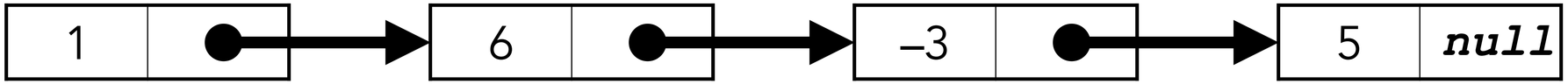
- Array mit fester Größe
- Bei zu vielen Elementen neues Array allozieren und altes Array kopieren

Parameter
Garbage Collection
RegEx
Generics
Datenstrukturen
Streams
Lambda
Polymorphie
Server
Nebenläufigkeit

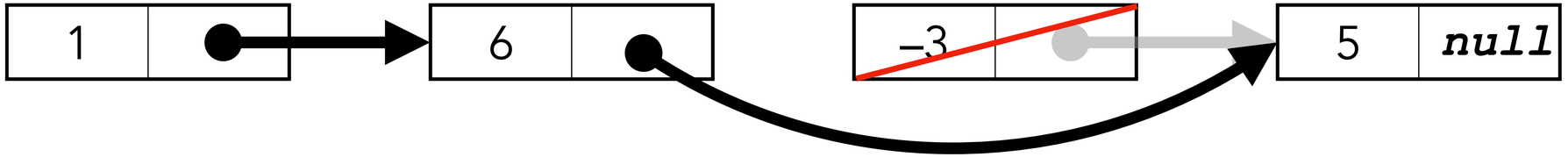
Datenstrukturen

List, Realisierung: LinkedList

- Durch Referenzen verknüpfte Listenknoten



- Löschen durch entfernen einer Referenz



Datenstrukturen

(Binär-)Bäume

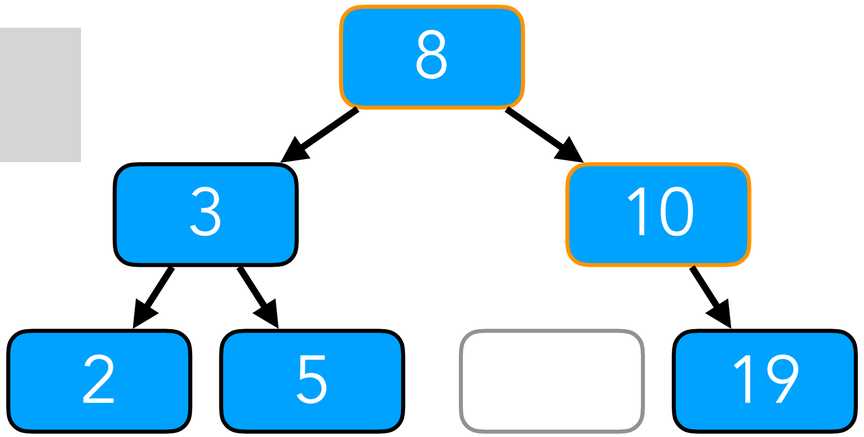
Wahrscheinlich nicht Klausurrelevant.

- Operationen:

```
> add(T value):void
> contains(T value):boolean
> remove(T value):void
```

- Invariante, z.B. links ≤ root ∧ rechts > root

```
add(9);
```



Datenstrukturen

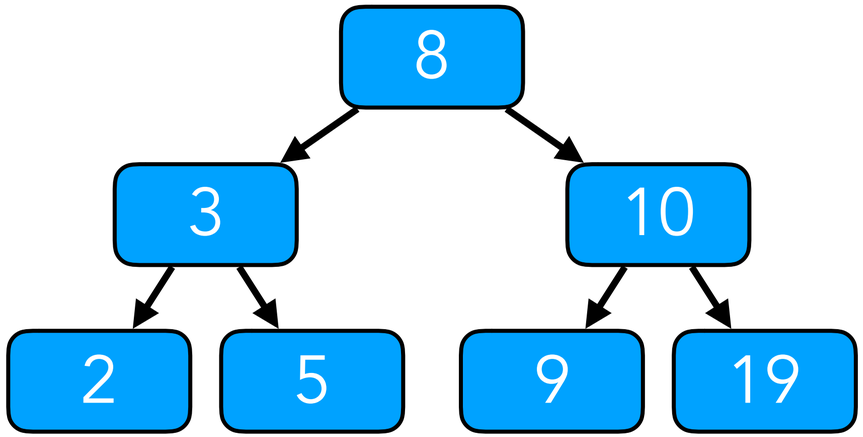
(Binär-)Bäume

Wahrscheinlich nicht Klausurrelevant.

- Operationen:

```
> add(T value):void
> contains(T value):boolean
> remove(T value):void
```

- Invariante, z.B. links ≤ root ∧ rechts > root



Datenstrukturen

(Binär-)Bäume

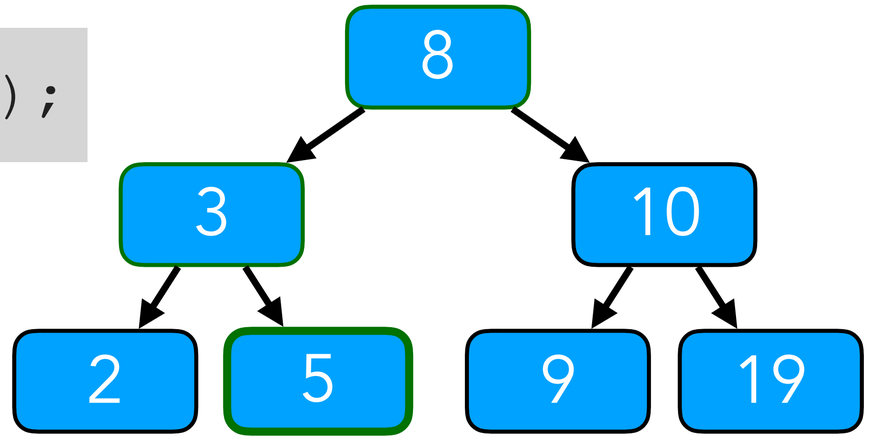
Wahrscheinlich nicht Klausurrelevant.

- Operationen:

```
> add(T value):void
> contains(T value):boolean
> remove(T value):void
```

- Invariante, z.B. links ≤ root ∧ rechts > root

```
contains(5);
```



Datenstrukturen

(Binär-)Bäume

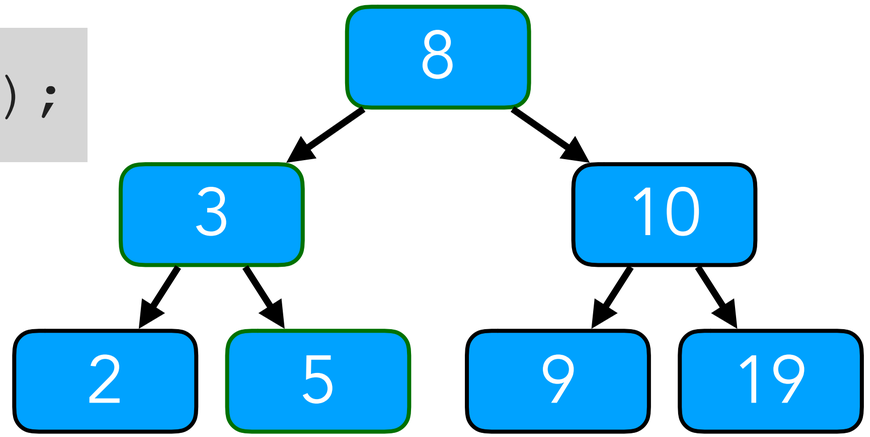
Wahrscheinlich nicht Klausurrelevant.

- Operationen:

```
> add(T value):void
> contains(T value):boolean
> remove(T value):void
```

- Invariante, z.B. links ≤ root ∧ rechts > root

```
contains(6);
```



Datenstrukturen

(Binär-)Bäume

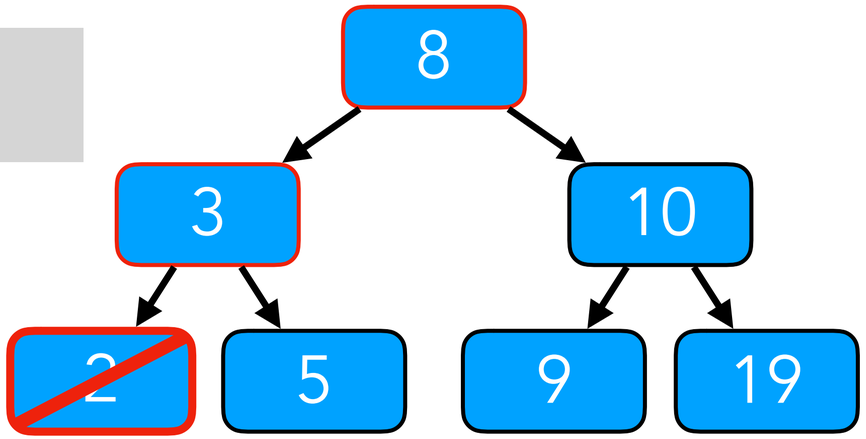
Wahrscheinlich nicht Klausurrelevant.

- Operationen:

```
> add(T value):void
> contains(T value):boolean
> remove(T value):void
```

- Invariante, z.B. links ≤ root ∧ rechts > root

```
remove(2);
```



Datenstrukturen

(Binär-)Bäume

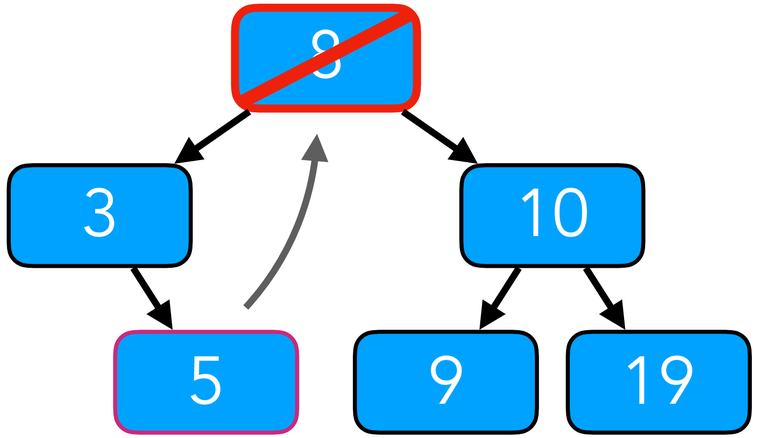
- Operationen:

```
> add(T value):void
> contains(T value):boolean
> remove(T value):void
```

- Invariante, z.B. links ≤ root ∧ rechts > root

Durch größtes Element des linken Teilbaumes ersetzen.

```
remove(8);
```



Datenstrukturen

(Binär-)Bäume

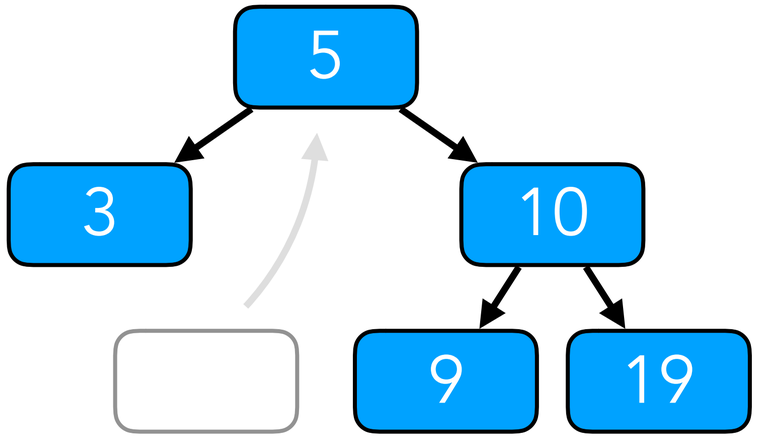
- Operationen:

```
> add(T value):void
> contains(T value):boolean
> remove(T value):void
```

- Invariante, z.B. links ≤ root ∧ rechts > root

Durch größtes Element des linken Teilbaumes ersetzen.

```
remove(8);
```



Datenstrukturen

Iterator, zum iterieren über Datenstrukturen

- Collections implementieren `Iterable`
  > sie verfügen über eine `iterator()` Funktion
  > es können forEach Schleifen verwendet werden
- Iterator besteht aus zwei Methoden (+ evtl. Variablen)
  > `hasNext()` gibt an, ob es ein weiteres Element gibt
  > `next()` gibt das nächste Element zurück

Datenstrukturen

Iterator, zum iterieren über Datenstrukturen

```
class MyArray<T> implements Iterable<T> {
  private T[] arr;
  // Implementierung
  public Iterator<T> iterator() {
    return new Iterator<T>() {
      private int pointer = 0;
      public boolean hasNext() {
        return this.pointer < arr.length;
      }
      public T next() {
        return arr[this.pointer++];
      }
    }
  };
}
```

Datenstrukturen

Iterator, zum iterieren über Datenstrukturen

```
class MyArray<T> implements Iterable<T> {
  private T[] arr;
  // Implementierung
  public Iterator<T> iterator() {
    return new Iterator<T>() {
      private int pointer = 0;
      public boolean hasNext() {
        return this.pointer < arr.length;
      }
      public T next() {
        return arr[this.pointe
      }
    }
  };
}
```

Pointer in MyArray Klasse würde paralleles iterieren falsch machen.

Parameter
Garbage Collection
RegEx
Generics
Datenstrukturen
Streams
Lambda
Polymorphie
Server
Nebenläufigkeit

Streams

Strom von Referenzen, auf dem Operationen ausgeführt werden können.

Quelle (erzeugt Stream):

```
- Stream.of( */ values */ );
- Arrays.stream( */ array */ );
- Collection.stream; //für alle Collections
- IntStream.range( /* from */ , /* to */ );
```

Basierend auf: Java 8 Streams Tutorial - Joe James

Streams

Strom von Referenzen, auf dem Operationen ausgeführt werden können.

Intermediäre Operationen (verändern Stream):

```
- map(Function<From, To>)
- filter(Predicate<Type>)
- distinct()
- sorted(Comparator<Type>)
- boxed()  //wandelt primitiven Stream zu Wrapper
- ...
```

Basierend auf: Java 8 Streams Tutorial - Joe James

Streams

Strom von Referenzen, auf dem Operationen ausgeführt werden können.

Terminale Operation (beenden Stream):

- `forEach(Consumer<Type>)`
- `reduce(Type, BinaryOperator<Type>)`

Identität (Startwert)

Basierend auf: Java 8 Streams Tutorial - Joe James

Streams

Klausuraufgabe: Wandeln Sie das gegebene Array mit Hilfe von `Arrays.stream()` in ein Stream- Objekt um. Verwenden Sie die `map()`, `forEach()`, `sorted()`, `filter()` und Lambda-Ausdrücke auf dem Stream, um hintereinander jede Zahl zu quadrieren, die Zahlen zu sortieren und auf der Kommandozeile auszugeben. Stellen Sie zudem sicher, dass nur Zahlen größer oder gleich 10 ausgegeben werden.

```
Arrays.stream(arr).map(i -> i*i).filter(
 i -> i >= 10).sorted().forEach(
 System.out::println);
```

Parameter
Garbage Collection
RegEx
Generics
Datenstrukturen
Streams
Lambda
Polymorphie
Server
Nebenläufigkeit

Lambda

Lambda Ausdrücke beschreiben Methoden als Abbildungen.
Vergleichbar ist das mit mathematischen Abbildungen.

$f : \mathbb{R} \to \mathbb{R}, x \mapsto x^2$, analog dazu `(x) -> (x * x)`

```
interface BinOp {
  public int calc(int z1, int z2);
}

BinOp sum = (summand1, summand2)
 -> (summand1 + summand2);

BinOp diff = (minuend, subtrahend)
 -> (minuend - subtrahend);

write(diff.calc(7, 2));    // printet 5
write(sum.calc(7, 2));     // printet 9
```

Parameter
Garbage Collection
RegEx
Generics
Datenstrukturen
Streams
Lambda
Polymorphie
Server
Nebenläufigkeit

Lambda

Lambda Ausdrücke beschreiben Methoden als Abbildungen.
Vergleichbar ist das mit mathematischen Abbildungen.

```
interface Function {
  public To apply(From value);
}
```

Die intermediäre Streamoperation `map(Function<From, To>)` erwartet als Parameter eine Funktion.

```
.map((a) -> (a + a));
```

Lambda

Lambda Ausdrücke beschreiben Methoden als Abbildungen.
Vergleichbar ist das mit mathematischen Abbildungen.

```
interface Function {
  public To apply(From value);
}
```

Die intermediäre Streamoperation `map(Function<From, To>)` erwartet als Parameter eine Funktion.

```
.map((a) -> {
  String s = Integer.toString(a);
  s = s.substring(0, 2);
  return s;
});
```

Mehrere Operationen in **{}**,
1: int zu String umwandeln
2: die ersten drei Zeichen herausfiltern
3: String zurückgeben

Parameter
Garbage Collection
RegEx
Generics
Datenstrukturen
Streams
Lambda
Polymorphie
Server
Nebenläufigkeit

Polymorphie

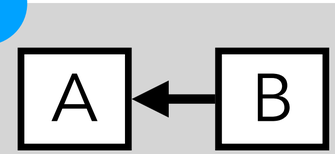
Herangehensweise:

1. Vererbungsstruktur als Diagramm aufzeichnen.
2. Statische und dynamische Typen der Objekte (auf denen Methoden aufgerufen werden) und Parameter bestimmen.
3. Kompatible Methoden anhand der statischen Typen bestimmen.
4. Beste Signatur wählen.
5. Ist die Methode nicht statisch? Dann anhand des dynamischen Typen nach einer 'näheren' Methode mit gleicher Signatur finden.

Polymorphie

```
1   public class A {
2    void f(A a) { write("A.f(A)"); }
3   }
4   public class B extends A {
5    void f(A a) { write("B.f(A)"); }
6    void f(B b) { write("B.f(B)"); }
7   }
8   A a1 = new A();
9   A a2 = new B();
10  B b1 = new B();
11  a1.f(a1);
12  a2.f(a2);
13  b1.f(a2);
```

1



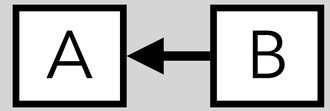
2

Var.	a1	a2	b1
st.	A	A	B
dy.	A	B	B

Polymorphie

```
1   public class A {
2    void f(A a) { write("A.f(A)"); }
3   }
4   public class B extends A {
5    void f(A a) { write("B.f(A)"); }
6    void f(B b) { write("B.f(B)"); }
7   }
8   A a1 = new A();
9   A a2 = new B();
10  B b1 = new B();
11  a1.f(a1);
12  a2.f(a2);
13  b1.f(a2);
```

1



2

Var.	a1	a2	b1
st.	A	A	B
dy.	A	B	B

3

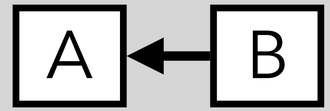
st. Signatur: A.f(A)

komp. Methode:
- Zeile 2

Polymorphie

```
1   public class A {
2    void f(A a) { write("A.f(A)"); }
3   }
4   public class B extends A {
5    void f(A a) { write("B.f(A)"); }
6    void f(B b) { write("B.f(B)"); }
7   }
8   A a1 = new A();
9   A a2 = new B();
10  B b1 = new B();
11  a1.f(a1);
12  a2.f(a2);
13  b1.f(a2);
```

1



2

Var.	a1	a2	b1
st.	A	A	B
dy.	A	B	B

3

st. Signatur: A.f(A)

4

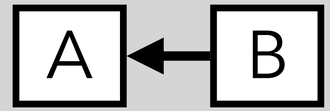
komp. Methode:
- Zeile 2

Polymorphie

```
1   public class A {
2    void f(A a) { write("A.f(A)"); }
3   }
4   public class B extends A {
5    void f(A a) { write("B.f(A)"); }
6    void f(B b) { write("B.f(B)"); }
7   }
8   A a1 = new A();
9   A a2 = new B();
10  B b1 = new B();
11  a1.f(a1);
12  a2.f(a2);
13  b1.f(a2);
```

```
A.f(A)
```

1



2

Var.	a1	a2	b1
st.	A	A	B
dy.	A	B	B

3

st. Signatur: A.f(A)

4

komp. Methode:
- Zeile 2

5

s:✗

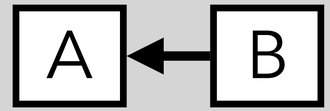
d:A

komp. Meth.:
- Zeile 2

Polymorphie

```
1   public class A {
2    void f(A a) { write("A.f(A)"); }
3   }
4   public class B extends A {
5    void f(A a) { write("B.f(A)"); }
6    void f(B b) { write("B.f(B)"); }
7   }
8   A a1 = new A();
9   A a2 = new B();
10  B b1 = new B();
11  a1.f(a1);
12  a2.f(a2);
13  b1.f(a2);
```

1



2

Var.	a1	a2	b1
st.	A	A	B
dy.	A	B	B

3

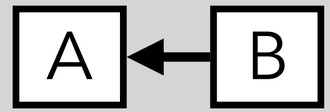
st. Signatur: A.f(A)

komp. Methode:
- Zeile 2

Polymorphie

```
1   public class A {
2    void f(A a) { write("A.f(A)"); }
3   }
4   public class B extends A {
5    void f(A a) { write("B.f(A)"); }
6    void f(B b) { write("B.f(B)"); }
7   }
8   A a1 = new A();
9   A a2 = new B();
10  B b1 = new B();
11  a1.f(a1);
12  a2.f(a2);
13  b1.f(a2);
```

1



2

Var.	a1	a2	b1
st.	A	A	B
dy.	A	B	B

3

st. Signatur: A.f(A)

4

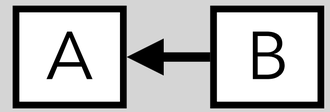
komp. Methode:
- Zeile 2

Polymorphie

```
1   public class A {
2    void f(A a) { write("A.f(A)"); }
3   }
4   public class B extends A {
5    void f(A a) { write("B.f(A)"); }
6    void f(B b) { write("B.f(B)"); }
7   }
8   A a1 = new A();
9   A a2 = new B();
10  B b1 = new B();
11  a1.f(a1);
12  a2.f(a2);
13  b1.f(a2);
```

```
B.f(A)
```

1



2

Var.	a1	a2	b1
st.	A	A	B
dy.	A	B	B

3

st. Signatur: A.f(A)

4

komp. Methode:
- Zeile 2

5

s:✗

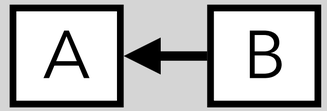
d:B

komp. Meth.:
- Zeile 2
- Zeile 5

Polymorphie

```
1   public class A {
2    void f(A a) { write("A.f(A)"); }
3   }
4   public class B extends A {
5    void f(A a) { write("B.f(A)"); }
6    void f(B b) { write("B.f(B)"); }
7   }
8   A a1 = new A();
9   A a2 = new B();
10  B b1 = new B();
11  a1.f(a1);
12  a2.f(a2);
13  b1.f(a2);
```

1



2

Var.	a1	a2	b1
st.	A	A	B
dy.	A	B	B

3

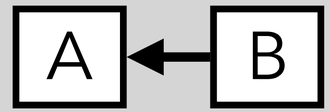
st. Signatur: B.f(A)

komp. Methode:
- Zeile 2
- Zeile 5

Polymorphie

```
1   public class A {
2    void f(A a) { write("A.f(A)"); }
3   }
4   public class B extends A {
5    void f(A a) { write("B.f(A)"); }
6    void f(B b) { write("B.f(B)"); }
7   }
8   A a1 = new A();
9   A a2 = new B();
10  B b1 = new B();
11  a1.f(a1);
12  a2.f(a2);
13  b1.f(a2);
```

1



2

Var.	a1	a2	b1
st.	A	A	B
dy.	A	B	B

3

st. Signatur: B.f(A)

4

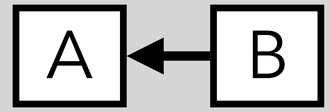
komp. Methode:
- Zeile 2
- Zeile 5

Polymorphie

```
1   public class A {
2    void f(A a) { write("A.f(A)"); }
3   }
4   public class B extends A {
5    void f(A a) { write("B.f(A)"); }
6    void f(B b) { write("B.f(B)"); }
7   }
8   A a1 = new A();
9   A a2 = new B();
10  B b1 = new B();
11  a1.f(a1);
12  a2.f(a2);
13  b1.f(a2);
```

```
B.f(A)
```

1



2

Var.	a1	a2	b1
st.	A	A	B
dy.	A	B	B

3

st. Signatur: B.f(A)

4

komp. Methode:
- Zeile 2
- Zeile 5

5

s:✗

d:B

komp. Meth.:
- Zeile 2
- Zeile 5

Polymorphie

Polymorphie mit Generics

- Java ersetzt alle (gewöhnlichen) Generics durch `Object`, wir können die Typparameter also ignorieren

Was ist zusätzlich zu beachten?

- Werden Attribute mit dem generischen Typen erstellt?
- Sind alle Methoden auf den generischen Typen definiert?
  > der generische Typ ist standardmäßig immer `Object`, kann aber durch `extends` eingeschränkt werden
  > nach Initialisierung gilt der übergebene statische Typ

Parameter
Garbage Collection
RegEx
Generics
Datenstrukturen
Streams
Lambda
Polymorphie
Server
Nebenläufigkeit

Polymorphie

Polymorphie mit Generics

```
class A<U> {
  U u;
  public A (U u) { this.u = u; }
  void foo() { write("A.foo()"); }
}
class B {
  void foo() { write("B.foo()"); }
}

A<B> aB = new A<B>(new B());
aB.u.foo();  // printet "B.foo()"
```

Polymorphie

Polymorphie mit Generics

```
class A<U> {
  U u;
  public A (U u) { this.u = u; }
  void foo() { write("A.foo()"); }
}
class B {
  void foo() { write("B.foo()"); }
}
A<B> aB = new A<B>(new B());
aB.u.foo();  // printet "B.foo()"
```

statisch

dynamisch

stat: B, dyn: B

Polymorphie

Polymorphie mit Generics

statisch

```
class A<U> {
  U u;
  public A (U u) { this.u = u; }
  void foo() { write("A.foo()"); }
}
class B {
  void foo() { write("B.foo()"); }
}

A<B> aB = new A<B>(new B());
aB.u.foo();  // printet "B.foo()"
```

stat: Object (kann durch extends eingeschränkt werden)

dynamischer Typ wird durch Parameter im Konstruktor bestimmt!

Polymorphie

Polymorphie mit Generics

```
class A<U> {
  U u;
  public A (U u) { this.u = u; }
  void foo() { this.u.foo(); }  // Compilerfehler
  // foo() ist nicht auf Object definiert
}
class B {
  void foo() { write("B.foo()"); }
}

A<B> aB = new A<B>(new B());
aB.u.foo();
```

Parameter
Garbage Collection
RegEx
Generics
Datenstrukturen
Streams
Lambda
Polymorphie
Server
Nebenläufigkeit

Polymorphie

Polymorphie mit Generics

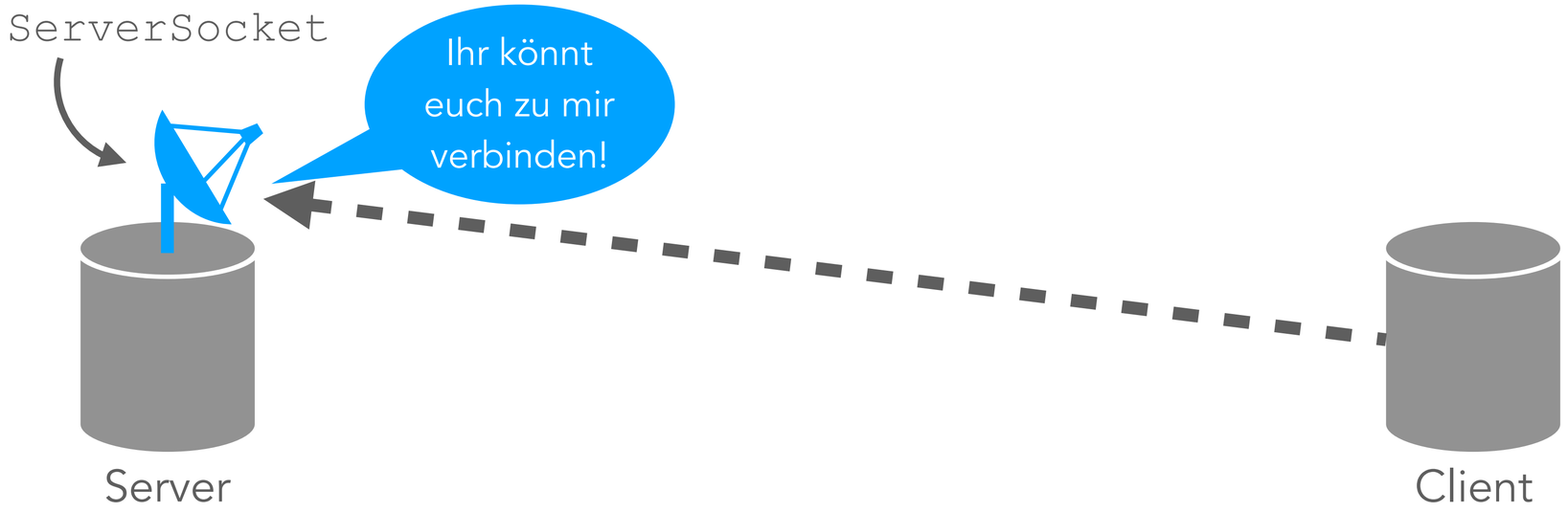
```
class A<U extends B> {
  U u;
  public A (U u) { this.u = u; }
  void foo() { this.u.foo(); }
  // foo() ist auf B definiert!
}
class B {
  void foo() { write("B.foo()"); }
}

A<B> aB = new A<B>(new B());
aB.u.foo();  // printet "B.foo()"
aB.foo();    // printet "B.foo()"
```

Parameter
Garbage Collection
RegEx
Generics
Datenstrukturen
Streams
Lambda
Polymorphie
Server
Nebenläufigkeit

Server

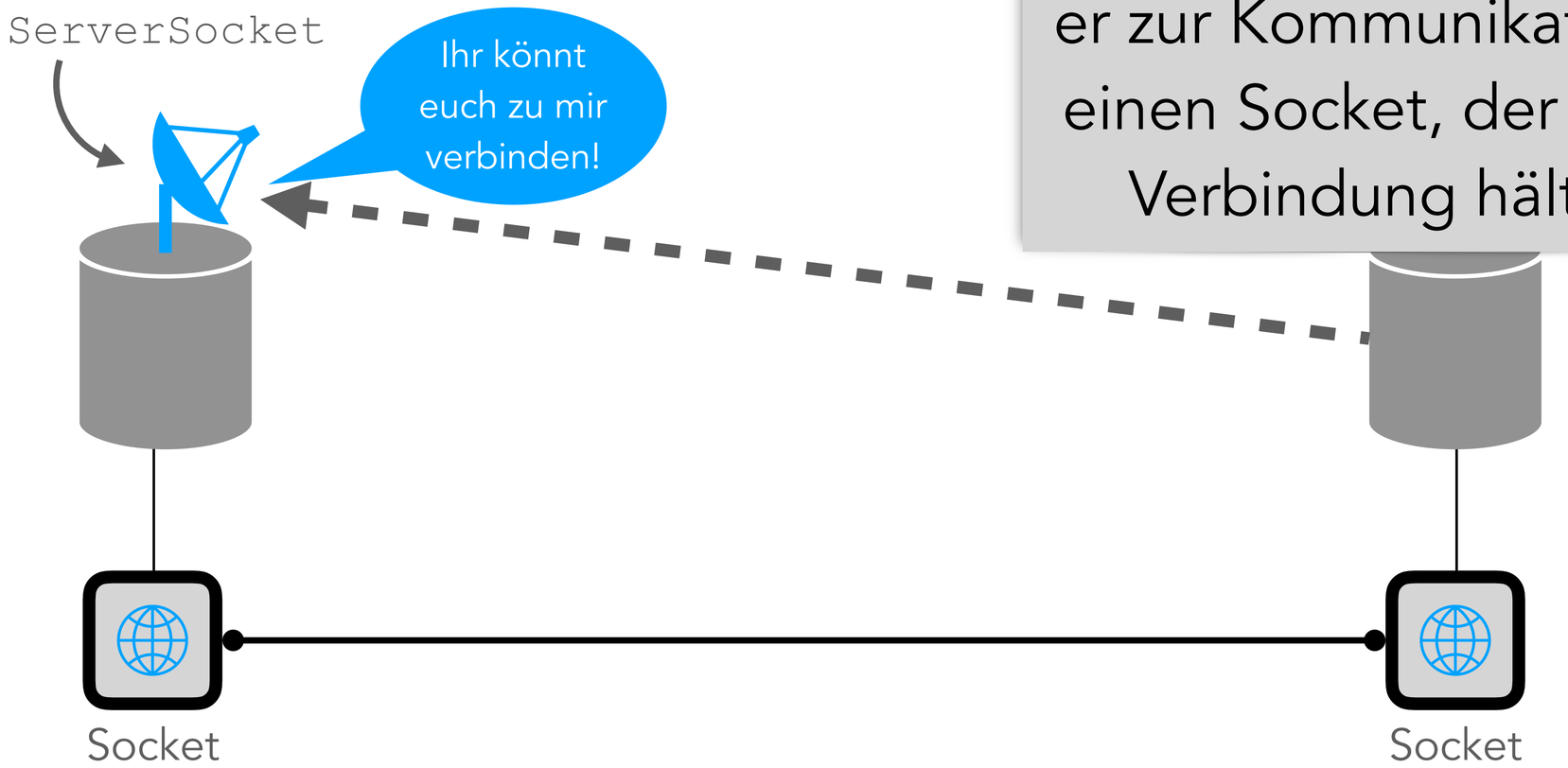
Socket & ServerSocket zur Netzwerkkommunikation



Parameter
Garbage Collection
RegEx
Generics
Datenstrukturen
Streams
Lambda
Polymorphie
Server
Nebenläufigkeit

Server

Socket & ServerSocket zur Netzwerkkor

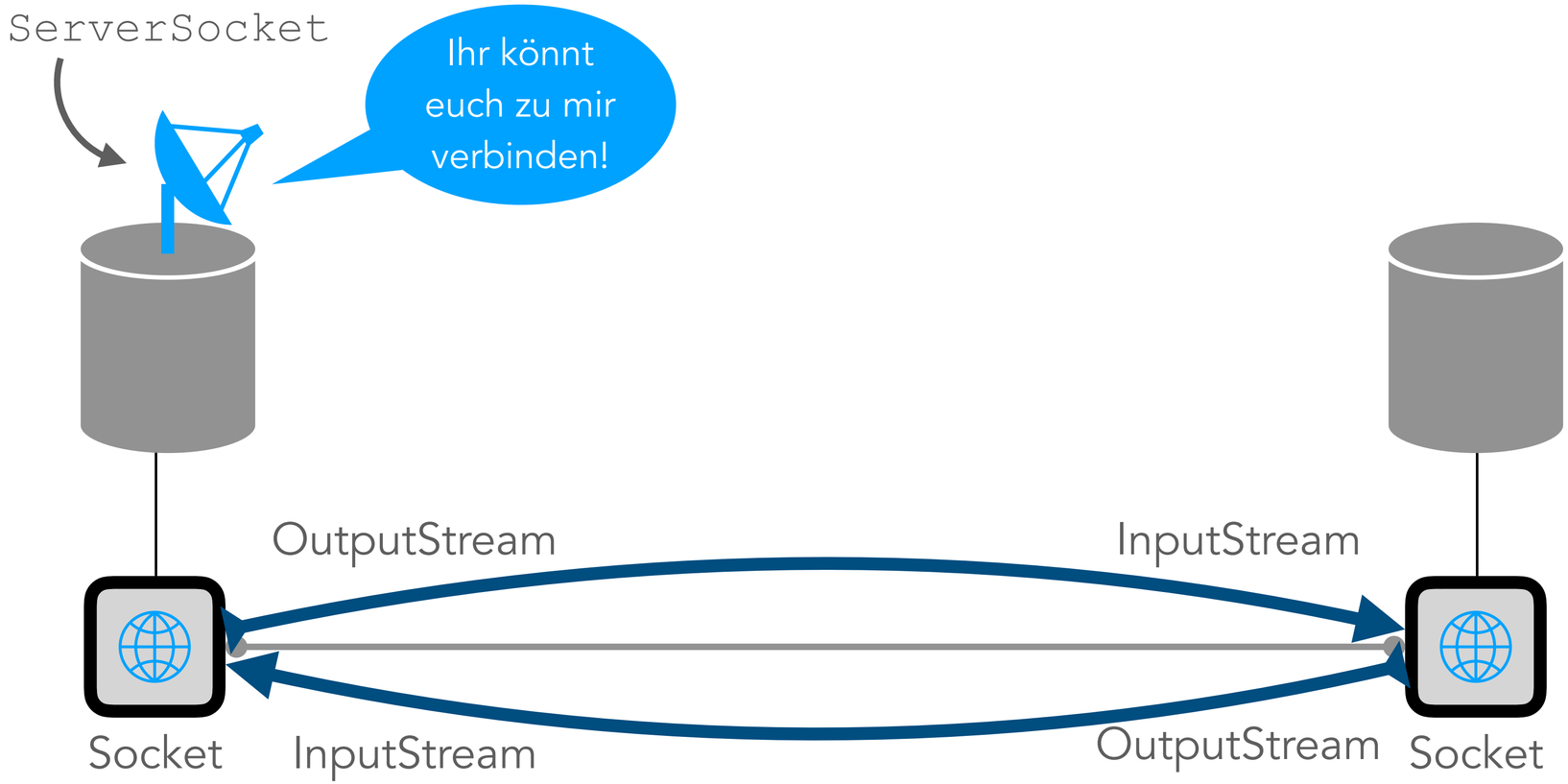


Nimmt der Server eine Verbindung an, erstellt er zur Kommunikation einen Socket, der die Verbindung hält.

Parameter
Garbage Collection
RegEx
Generics
Datenstrukturen
Streams
Lambda
Polymorphie
Server
Nebenläufigkeit

Server

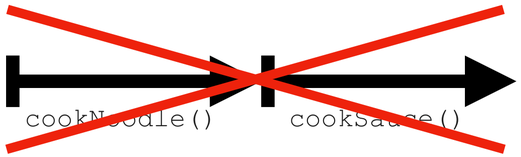
Socket & ServerSocket zur Netzwerkkommunikation

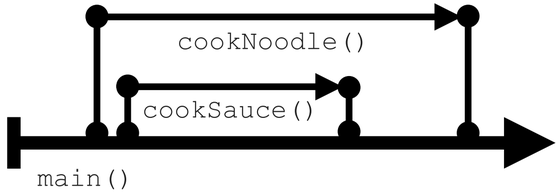


Parameter
Garbage Collection
RegEx
Generics
Datenstrukturen
Streams
Lambda
Polymorphie
Server
Nebenläufigkeit

Nebenläufigkeit

Generelle Idee: Paralleler Programmablauf auf mehreren Prozessorkernen.






```
public class PastaMaker extends Thread {
  public void run() { write("Noodles"); }
}
```

Lambda-Expression für die `run()`-Methode

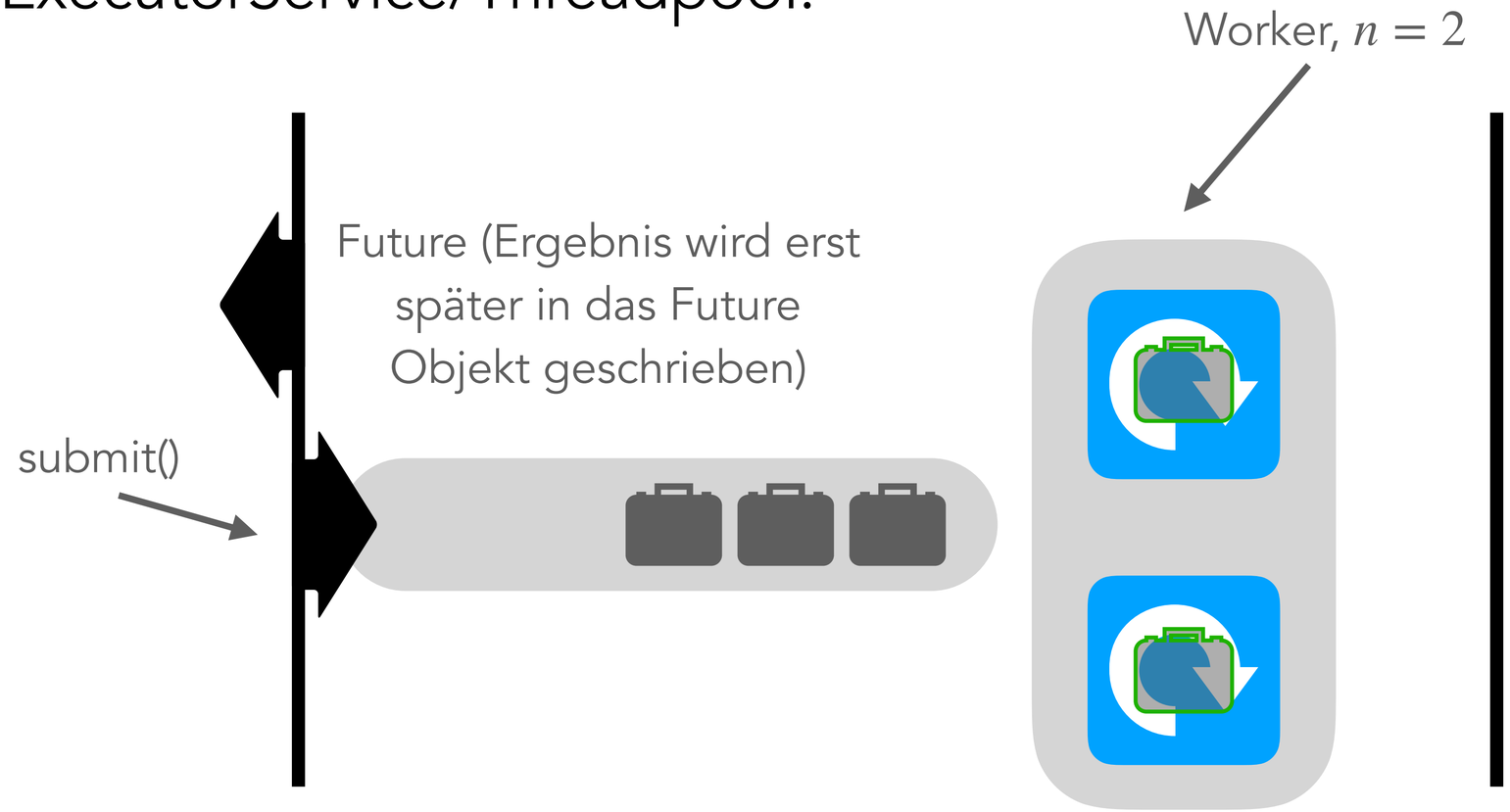
```
Runnable sauceMaker = () -> { write("Sauce"); }

(new Thread(sauceMaker)).start();
(new PastaMaker()).start();
```

Parameter
Garbage Collection
RegEx
Generics
Datenstrukturen
Streams
Lambda
Polymorphie
Server
Nebenläufigkeit

Nebenläufigkeit

ExecutorService/Threadpool:



Nebenläufigkeit

ExecutorService/Threadpool:

```
import java.util.concurrent.*;

ExecutorService es = new ExecutorService(2);
// created ExecutorService with 2 workers

es.submit(new PastaMaker());

es.submit(() -> {
  write("Sauce");
});

es.shutdown();
// interrupted workers
```

Der ExecutorService kümmert sich um die Verteilung der Aufgaben auf die Worker und startet sie.

Parameter
Garbage Collection
RegEx
Generics
Datenstrukturen
Streams
Lambda
Polymorphie
Server
Nebenläufigkeit

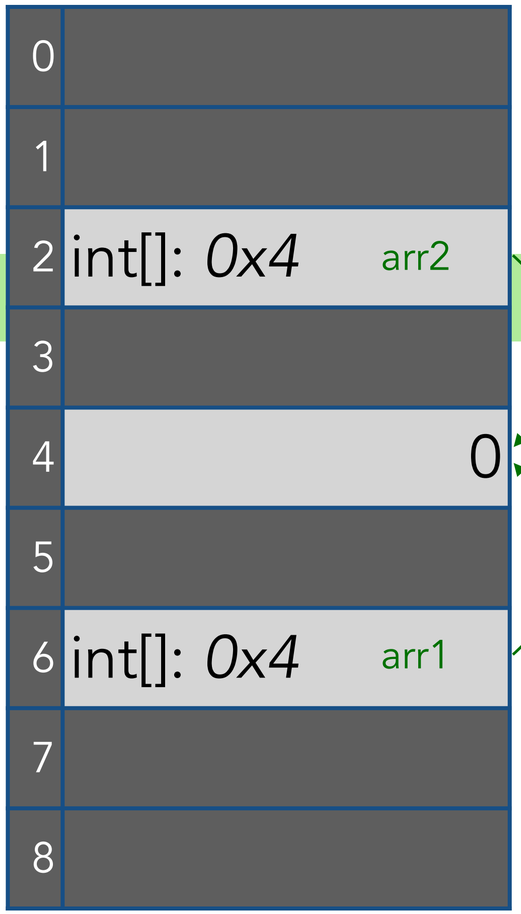
Nebenläufigkeit

Race Conditions bei geteilten Variablen.

```
void run() {
 arr1[0]++;
}
```

```
LOAD 0x6[0]
CONST 1
ADD
STORE 0x6[0]
```

0




```
void run() {
 arr2[0]++;
}
```

```
LOAD 0x2[0]
CONST 1
ADD
STORE 0x2[0]
```

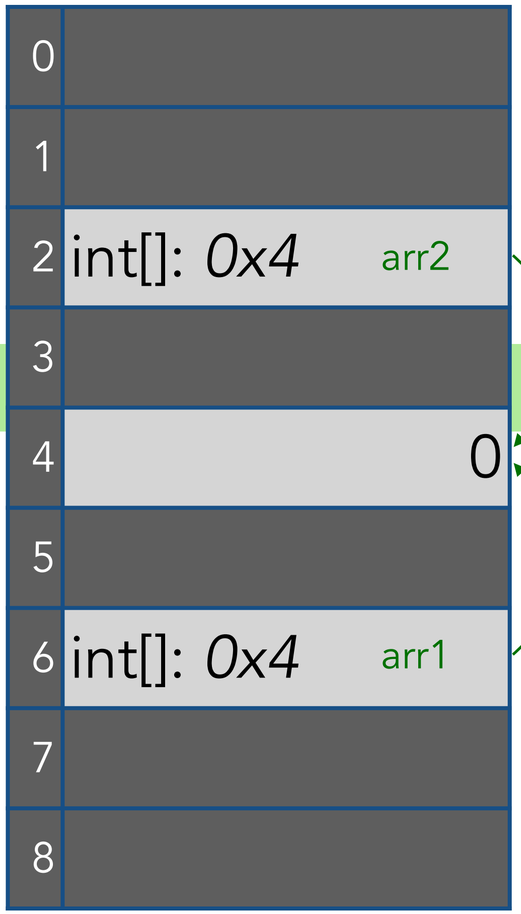
Parameter
Garbage Collection
RegEx
Generics
Datenstrukturen
Streams
Lambda
Polymorphie
Server
Nebenläufigkeit

Nebenläufigkeit

Race Conditions bei geteilten Variablen.

```
void run() {
 arr1[0]++;
}
```

```
LOAD 0x6[0]
CONST 1
ADD
STORE 0x6[0]
```




```
void run() {
 arr2[0]++;
}
```

```
LOAD 0x2[0]
CONST 1
ADD
STORE 0x2[0]
```

Parameter
Garbage Collection
RegEx
Generics
Datenstrukturen
Streams
Lambda
Polymorphie
Server
Nebenläufigkeit

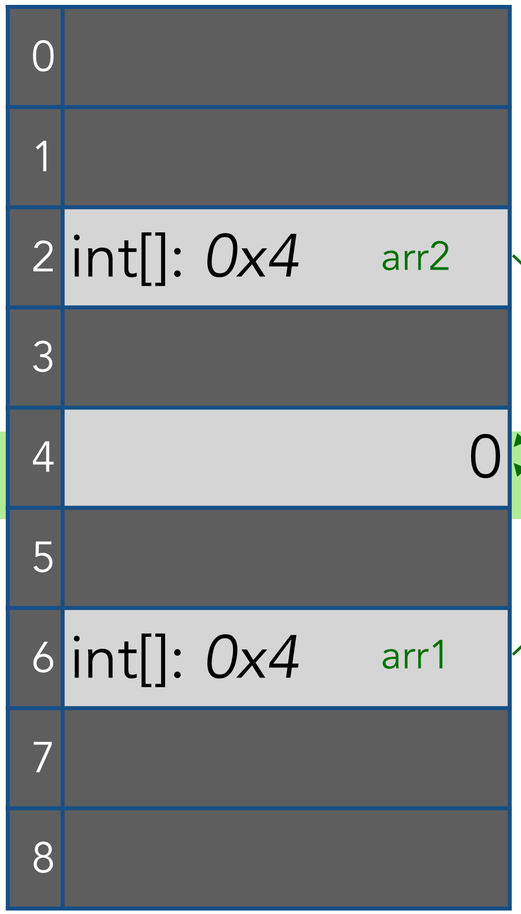
Nebenläufigkeit

Race Conditions bei geteilten Variablen.

```
void run() {
 arr1[0]++;
}
```

```
LOAD 0x6[0]
CONST 1
ADD
STORE 0x6[0]
```

1




```
void run() {
 arr2[0]++;
}
```

```
LOAD 0x2[0]
CONST 1
ADD
STORE 0x2[0]
```

0

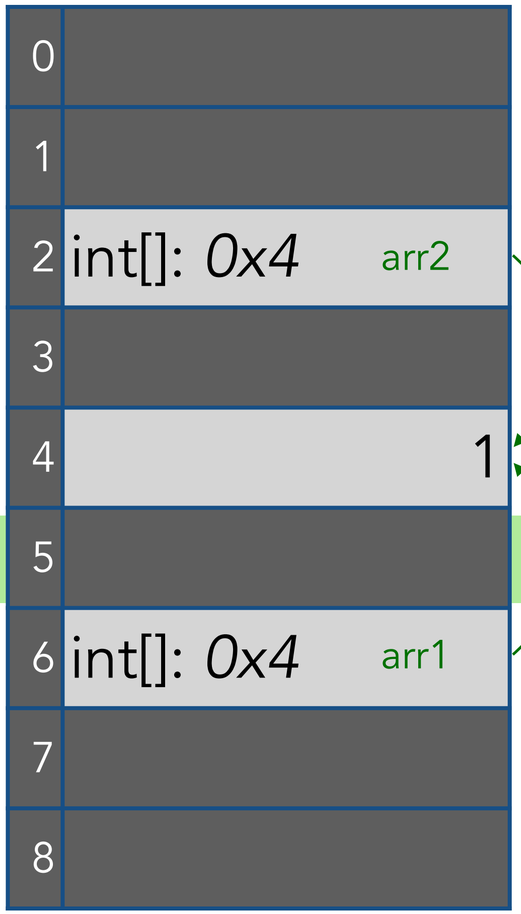
Parameter
Garbage Collection
RegEx
Generics
Datenstrukturen
Streams
Lambda
Polymorphie
Server
Nebenläufigkeit

Nebenläufigkeit

Race Conditions bei geteilten Variablen.

```
void run() {
 arr1[0]++;
}
```

```
LOAD 0x6[0]
CONST 1
ADD
STORE 0x6[0]
```




```
void run() {
 arr2[0]++;
}
```

```
LOAD 0x2[0]
CONST 1
ADD
STORE 0x2[0]
```

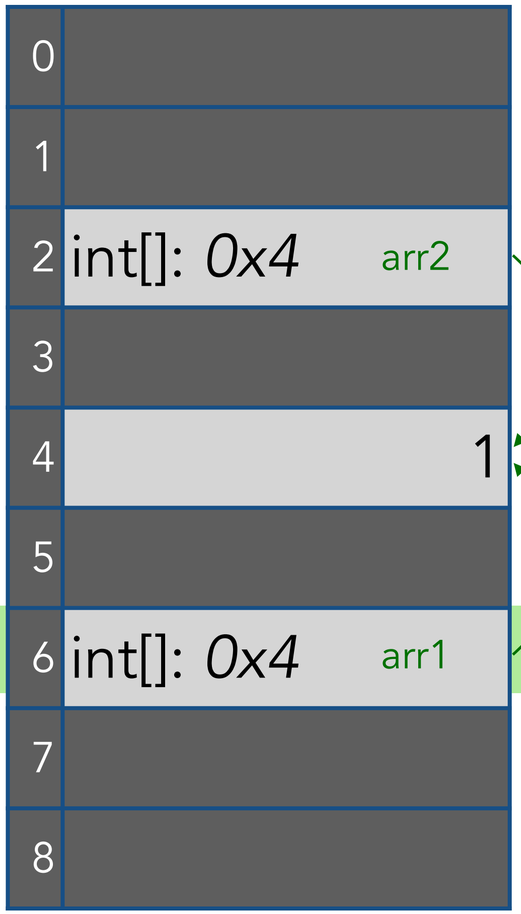
Parameter
Garbage Collection
RegEx
Generics
Datenstrukturen
Streams
Lambda
Polymorphie
Server
Nebenläufigkeit

Nebenläufigkeit

Race Conditions bei geteilten Variablen.

```
void run() {
 arr1[0]++;
}
```

```
LOAD 0x6[0]
CONST 1
ADD
STORE 0x6[0]
```




```
void run() {
 arr2[0]++;
}
```

```
LOAD 0x2[0]
CONST 1
ADD
STORE 0x2[0]
```

1

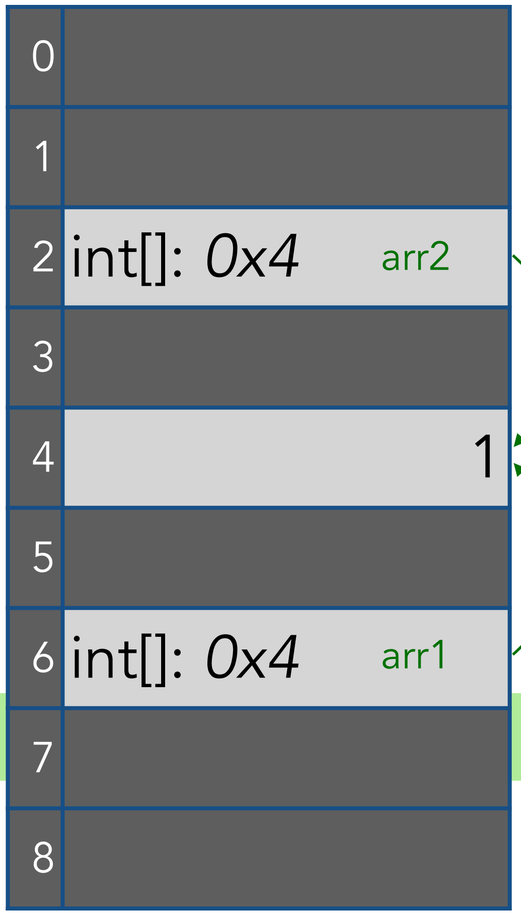
Parameter
Garbage Collection
RegEx
Generics
Datenstrukturen
Streams
Lambda
Polymorphie
Server
Nebenläufigkeit

Nebenläufigkeit

Race Conditions bei geteilten Variablen.

```
void run() {
 arr1[0]++;
}
```

```
LOAD 0x6[0]
CONST 1
ADD
STORE 0x6[0]
```




```
void run() {
 arr2[0]++;
}
```

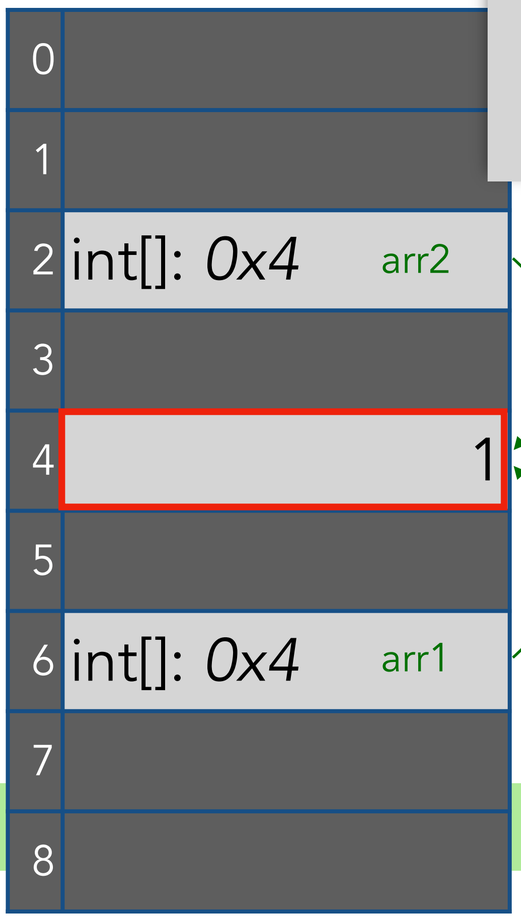
```
LOAD 0x2[0]
CONST 1
ADD
STORE 0x2[0]
```

Nebenläufigkeit

Race Conditions bei geteilten Variablen

```
void run() {
 arr1[0]++;
}
```

```
LOAD 0x6[0]
CONST 1
ADD
STORE 0x6[0]
```




```
}
```

```
LOAD 0x2[0]
CONST 1
ADD
STORE 0x2[0]
```

Zwei Inkrementationen von 0 ergeben 1?

$$0 + 1 + 1 \overset{!}{=} 1$$

Nebenläufigkeit

Lösung: Sperre/Lock auf Objekte, die zu jedem Zeitpunkt nur einen Thread Zugriff auf das Objekt gewährt.

```
void inc(int[] arr) {
  synchronized (arr) {
    arr[0]++;
  }
}
```

Führt zur sequentiellen Ausführung der Inkrementationen, da nur ein Thread zur gleichen Zeit auf `arr` zugreifen darf.

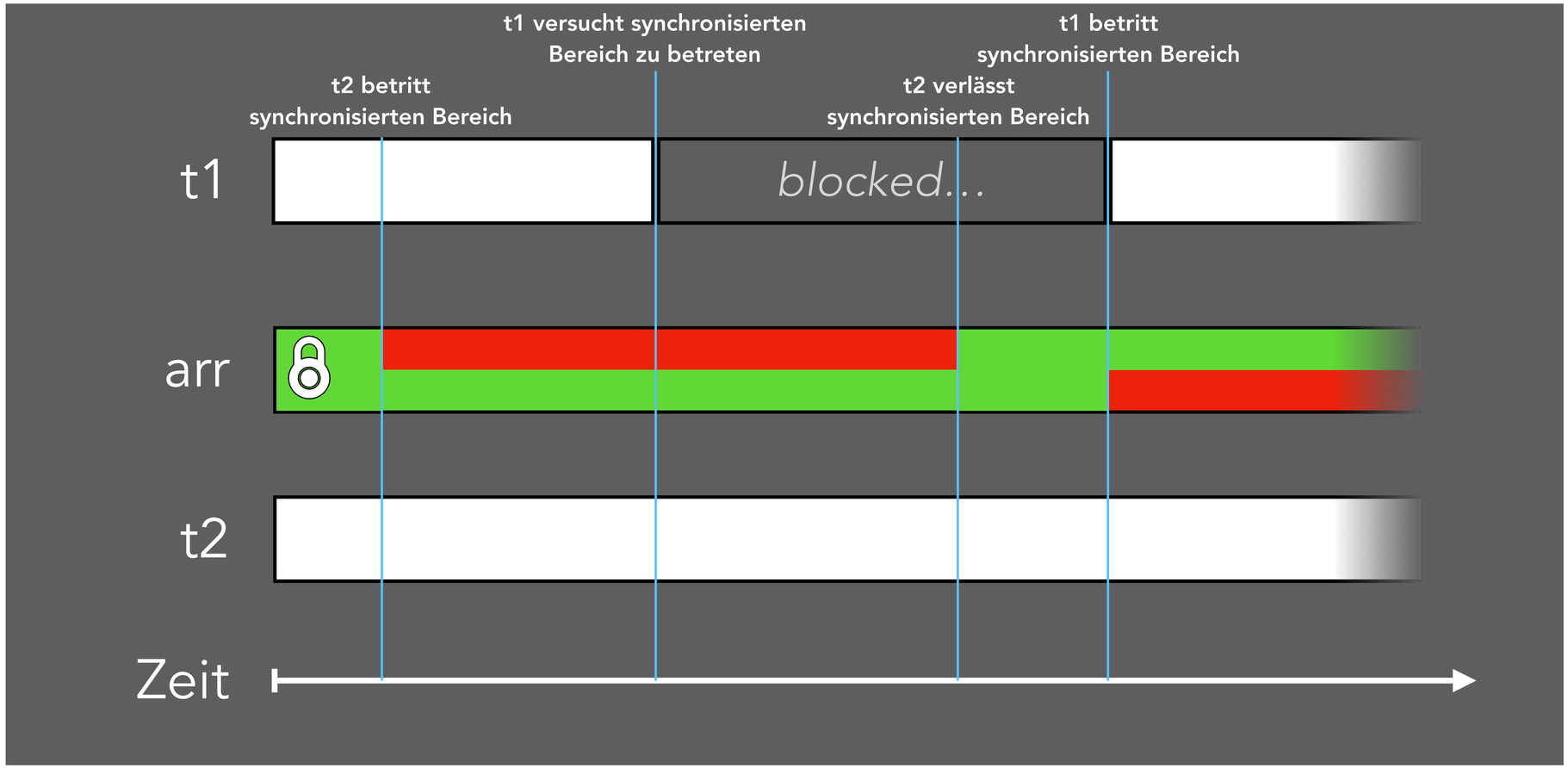
```
LOAD 0x2[0]
CONST 1
ADD
STORE 0x2[0]

LOAD 0x6[0]
CONST 1
ADD
STORE 0x6[0]
```

Parameter
Garbage Collection
RegEx
Generics
Datenstrukturen
Streams
Lambda
Polymorphie
Server
Nebenläufigkeit

Nebenläufigkeit

Lösung: Sperre/Lock auf Objekte



Parameter
Garbage Collection
RegEx
Generics
Datenstrukturen
Streams
Lambda
Polymorphie
Server
Nebenläufigkeit

Nebenläufigkeit

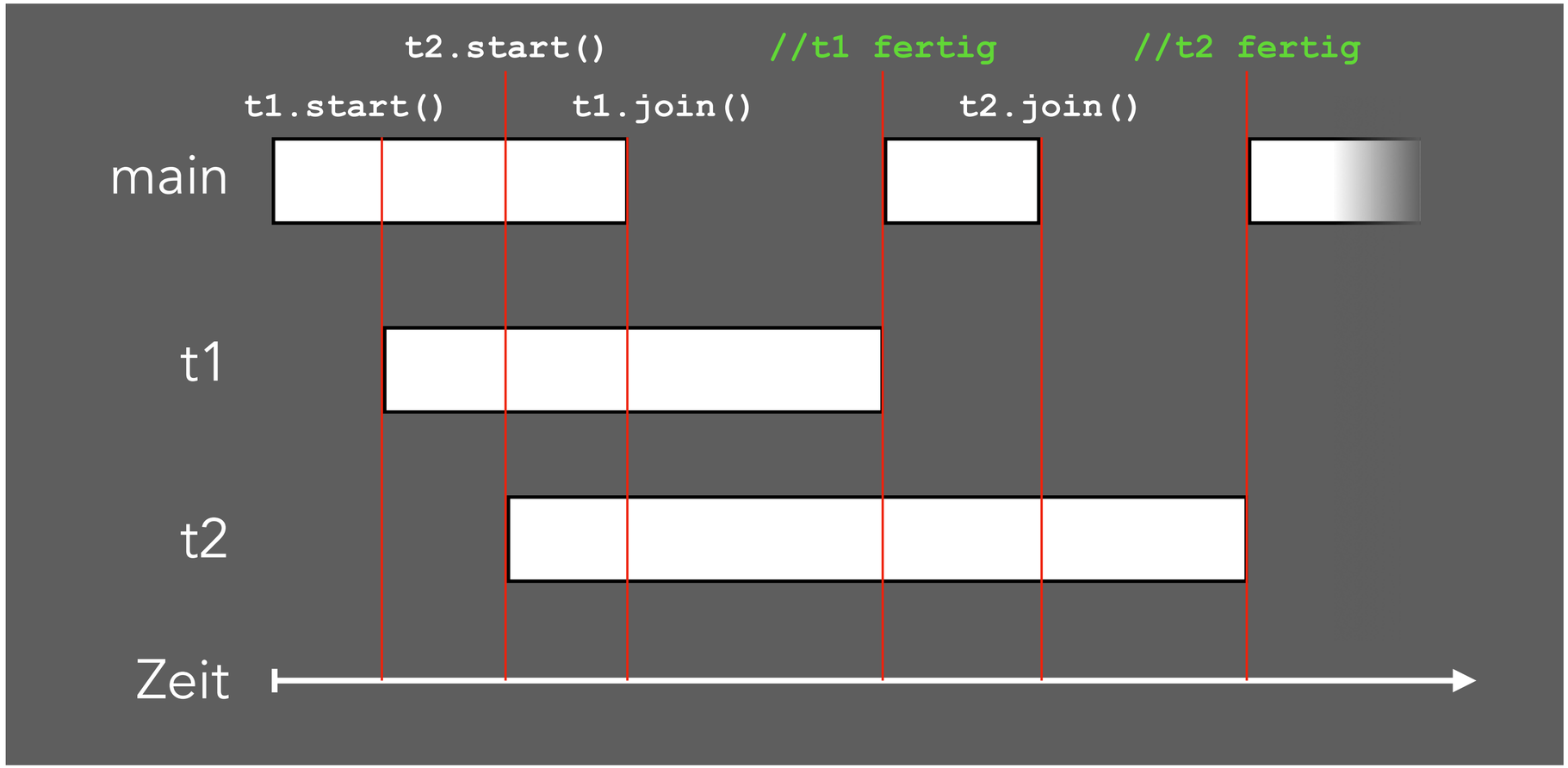
Strategien, Monitore

- **`synchronized`** Blöcke
- **`synchronized`** Methoden (≡ **`synchronized`** Blöcke auf **`this`**)
- Read-Write Lock
- Semaphore

Parameter
Garbage Collection
RegEx
Generics
Datenstrukturen
Streams
Lambda
Polymorphie
Server
Nebenläufigkeit

Nebenläufigkeit

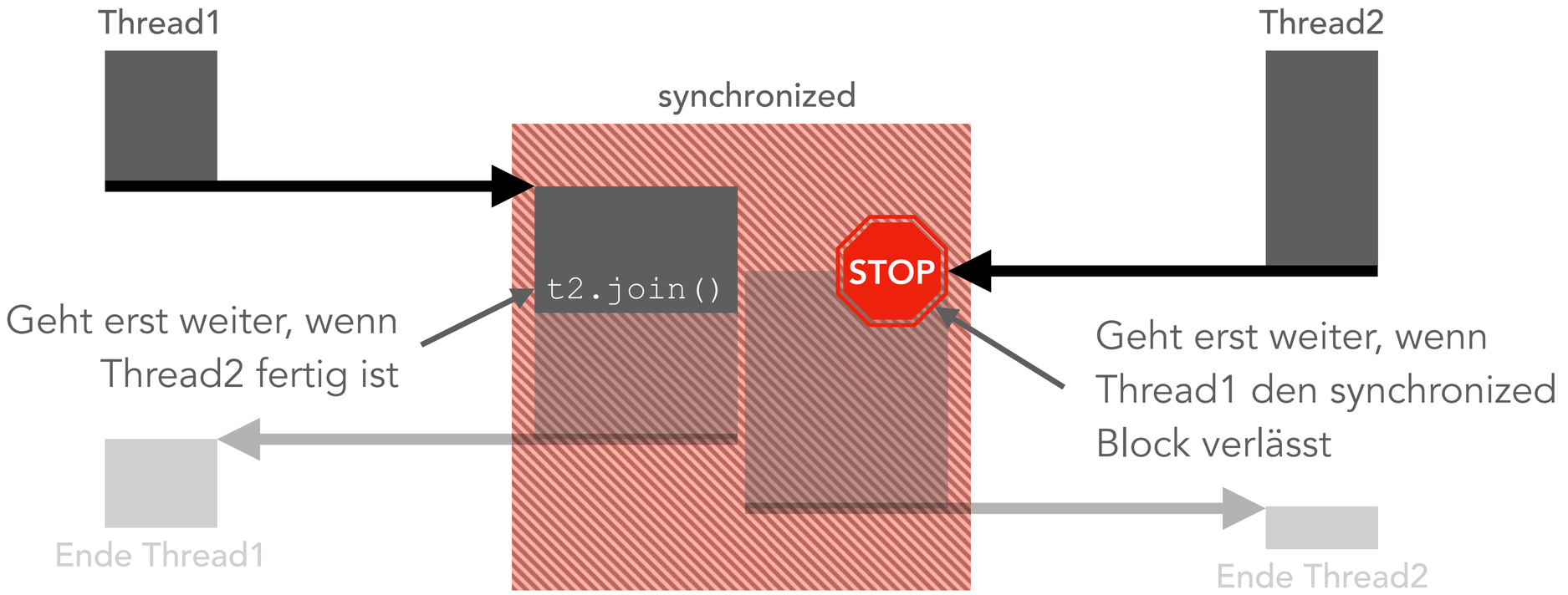
Threads Joinen:



Parameter
Garbage Collection
RegEx
Generics
Datenstrukturen
Streams
Lambda
Polymorphie
Server
Nebenläufigkeit

Nebenläufigkeit

Vorsicht vor Deadlocks:



Parameter
Garbage Collection
RegEx
Generics
Datenstrukturen
Streams
Lambda
Polymorphie
Server
Nebenläufigkeit

Nebenläufigkeit

Vorsicht vor Deadlocks:

Ein Deadlock entsteht immer dann, wenn es zyklische Warteabhängigkeiten gibt.