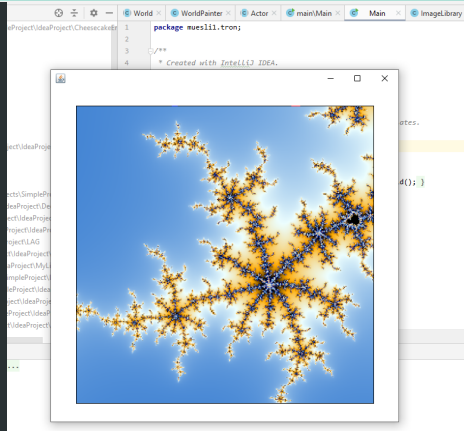


FOP Recap #7



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Rekursion und fortgeschrittene Konzepte mit Racket



Das steht heute auf dem Plan



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Structs

Deklaration und Verwendung
Java Äquivalent

Listen

Rekursiv vs Iterativ

Funktionen höherer Ordnung

Lambda-Ausdrücke

Closures



- Haben mehrere Felder
- Felder sind konstant
- Können beliebig erstellt werden
- Auf jedes Feld kann einzeln zugegriffen werden

Structs

Deklaration und Verwendung



TECHNISCHE
UNIVERSITÄT
DARMSTADT

```
1 (define-struct my-struct-name (field-one field-two field-three))
```

```
1 (make-my-struct-name expression-one expression-two  
  ↪ expression-three)
```

```
1 (my-struct-name-field-one instance)
```

Structs

Deklaration und Verwendung



TECHNISCHE
UNIVERSITÄT
DARMSTADT

```
1 (define-struct person (first-name last-name age))
```

```
1 (make-person "Konrad" "Zuse" 85)
```

```
1 (define ada-lovelace (make-person "Ada" "Lovelace" 36))
```

```
1 (person-first-name ada-lovelace)  
2 (person-age ada-lovelace)
```



```
1 public record Person(String firstName, String lastName, int age)
2 {
3     // ...
4 }
```

- Haben einen impliziten Konstruktor
- gleichnamige Methoden wie die einzelnen Elemente als getter-Methoden
- die Attribute sind private und final

Das steht heute auf dem Plan



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Structs

Listen

Rekursiv vs Iterativ

Funktionen höherer Ordnung

Lambda-Ausdrücke

Closures



- Arrays haben eine feste Länge
- Arrays haben einen festen statischen Typen
- Man kann auf einen beliebigen Index zugreifen



- Listen haben eine variable Länge
- Können heterogen sein



- Listen haben eine variable Länge
- Können heterogen sein
- Man erstellt sie über `(list element1 element2 elementN)`
- Man kann nur auf
 - ▣ Das erste Element mit `(first)` zugreifen
 - ▣ Den Rest mit `(rest)` zugreifen
- Der Ausdruck `empty` ist eine leere Liste
- Man kann über `(empty?)` prüfen, ob ein Ausdruck eine leere Liste ist
- Mit `(append lst1 lst2)` konkateniert man zwei Listen
- Mit `(cons element1 lst)` fügt man vorne an eine Liste ein neues Element an

Das steht heute auf dem Plan



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Structs

Listen

Rekursiv vs Iterativ

Funktionen höherer Ordnung

Lambda-Ausdrücke

Closures



```
1  int[] result = new int[5];  
2  
3  for(int i = 0; i < 5; i++) {  
4      result[i] = i;  
5  }
```



```
1 public void recursiveStep(int[] result, int currentIndex) {  
2     // Rekursionsanker  
3     if(currentIndex == result.length) {  
4         return;  
5     }  
6     result[currentIndex] = currentIndex;  
7     // Rekursiver Aufruf  
8     recursiveStep(array, currentIndex + 1);  
9 }
```

```
1 // Start der Rekursion  
2 int[] result = new int[5];  
3 recursiveStep(result, 0);
```



```
1 (define (recursive-function lst)
2   (cond
3     ; Rekursionsanker
4     [(empty? lst) empty]
5     [else (cons
6             (+ 1 (first lst))
7             ; Rekursiver Aufruf
8             (recursive-function (rest lst))
9             )
10    ]
11  )
12 )
```

Typische Rekursion in Racket



TECHNISCHE
UNIVERSITÄT
DARMSTADT

```
1 (define (recursive-function lst)
2   (cond
3     [(empty? lst) empty]
4     [else (cons
5              (+ 1 (first lst))
6              (recursive-function (rest lst))
7            )]
8   )
9 )
10 )
```

```
1 ; Start der Rekursion
2 (recursive-function (list 1 2 3 4 5))
```

Das steht heute auf dem Plan



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Structs

Listen

Rekursiv vs Iterativ

Funktionen höherer Ordnung

`filter`

`map`

`foldl` und `foldr`

Lambda-Ausdrücke

Closures



- Sind Funktionen, die als Parameter Funktionen erhalten
- Zum Beispiel
 - ▣ Zum Graphenzeichnen
 - ▣ Zur Nullstellenberechnung
 - ▣ Für Listenoperationen und Streams
 - ▣



- In Racket nutzen wir:
 - ▣ `filter` zum Filtern von Listen
 - ▣ `map` um eine Funktion auf jedes Element anzuwenden
 - ▣ `foldl` und `foldr` als terminale Listenoperation

Funktionen höherer Ordnung

filter



TECHNISCHE
UNIVERSITÄT
DARMSTADT

```
1 (filter filter-function the-list)
```

```
1 (define (is-big x) (> x 5))  
2  
3 (filter is-big (list 1 4 6 10))
```

```
$ (list 6 10)
```

Funktionen höherer Ordnung

map



TECHNISCHE
UNIVERSITÄT
DARMSTADT

```
1 (map map-function the-list)
```

```
1 (define (add-some x) (+ x 5))
```

```
2
```

```
3 (map add-some (list 1 4 6 10))
```

Funktionen höherer Ordnung

map



TECHNISCHE
UNIVERSITÄT
DARMSTADT

```
1 (map map-function the-list)
```

```
1 (define (add-some x) (+ x 5))
```

```
2
```

```
3 (map add-some (list 1 4 6 10))
```

```
$ (list 6 9 11 15)
```

Funktionen höherer Ordnung

map



TECHNISCHE
UNIVERSITÄT
DARMSTADT

```
1 (map map-function the-list)
```

```
1 (define (add-some x) (+ x 5))
```

```
2
```

```
3 (map add-some (list 1 4 6 10))
```

```
$ (list 6 9 11 15)
```

```
1 (map sqrt (list 1 4 16 121))
```

```
$ (list 1 2 4 11)
```

Funktionen höherer Ordnung

foldl und foldr



TECHNISCHE
UNIVERSITÄT
DARMSTADT

```
1 (foldl fold-function initial-value the-list)
2 (foldr fold-function initial-value the-list)
```

```
1 (define (my-fold-function elem acc) (+ elem acc))
2
3 (foldl my-fold-function 0 (list 1 -1 5 -5 2))
4 (foldl my-fold-function 5 (list 1 -1 5 -5 2))
```

```
$ 2
$ 7
```

- Hier: Identisches Ergebnis mit foldr

Funktionen höherer Ordnung

foldl und foldr



TECHNISCHE
UNIVERSITÄT
DARMSTADT

```
1  (define (my-fold-function elem acc)
2      (+ elem
3          (if (> acc 0)
4              acc
5              (- 0 acc)
6          )
7      )
8  )
9  (foldl my-fold-function 0 (list 1 -1 5 -5 2))
10 (foldr my-fold-function 0 (list 1 -1 5 -5 2))
```


Funktionen höherer Ordnung

foldl und foldr



TECHNISCHE
UNIVERSITÄT
DARMSTADT

```
1 (define (my-fold-function elem acc)
2   (+ elem
3     (if (> acc 0)
4         acc
5         (- 0 acc)
6     )
7   )
8 )
9 (foldl my-fold-function 0 (list 1 -1 5 -5 2))
10 (foldr my-fold-function 0 (list 1 -1 5 -5 2))
```

\$ 2

\$ 8

Das steht heute auf dem Plan



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Structs

Listen

Rekursiv vs Iterativ

Funktionen höherer Ordnung

Lambda-Ausdrücke

 In Racket

 In Java

 Methodenreferenzen

Closures



- Sind "Funktionen ohne Namen"
- Haben auch Parameter und Methodenrumpf

Lambda-Ausdrücke

In Racket



TECHNISCHE
UNIVERSITÄT
DARMSTADT

```
1 (lambda (parameter-one parameter-two ....)  
   ↪ function-body-expression)
```

```
1 (lambda (x) (* (+ x 5) 2))
```

```
1 ((lambda (x) (* (+ x 5) 2)) 2)
```

\$ 14

Lambda-Ausdrücke

In Racket



TECHNISCHE
UNIVERSITÄT
DARMSTADT

```
1 (lambda (parameter-one parameter-two ....)  
   ↪ function-body-expression)
```

```
1 (map (lambda (x) (* (+ x 5) 2)) (list 1 2 3))
```

```
$ (list 12 14 16)
```

```
1 (define my-const (lambda (x) (* (+ x 5) 2)))
```



- Funktionieren über Funktionale Interfaces
- Funktionale Interfaces
 - ▣ Haben genau eine (nicht default oder static) Methode
 - ▣ Haben optional die Annotation `FunctionalInterface`

Lambda-Ausdrücke

In Java



TECHNISCHE
UNIVERSITÄT
DARMSTADT

```
1 @FunctionalInterface
2 public interface MyIntPredicate {
3     boolean test(int number);
4
5     // .... Additional default or static methods ....
6 }
```

```
1 public class BiggerThanFive implements MyIntPredicate {
2     public boolean test(int number) {
3         return number > 5;
4     }
5 }
```



```
1 public static void higherFunction(MyIntPredicate predicate) {  
2     boolean testOne = predicate.test(4);  
3     boolean testTwo = predicate.test(6);  
4     // ....  
5 }
```

```
1 higherFunction(new BiggerThanFive());  
2  
3 higherFunction(new SmallerThanOne());
```


Lambda-Ausdrücke

In Java



TECHNISCHE
UNIVERSITÄT
DARMSTADT

```
1 public class BiggerThanFive implements MyIntPredicate {  
2     public boolean test(int number) {  
3         return number > 5;  
4     }  
5 }
```

```
1 MyIntPredicate pred = new BiggerThanFive();
```

```
1 MyIntPredicate pred = (int number) -> {  
2     return number > 5;  
3 };
```

Lambda-Ausdrücke

In Java



TECHNISCHE
UNIVERSITÄT
DARMSTADT

```
1 MyIntPredicate pred = (int number) -> {  
2     return number > 5;  
3 };
```

```
1 MyIntPredicate pred = (number) -> {  
2     return number > 5;  
3 };
```

Lambda-Ausdrücke

In Java



TECHNISCHE
UNIVERSITÄT
DARMSTADT

```
1 MyIntPredicate pred = number -> {  
2     return number > 5;  
3 };
```

```
1 MyIntPredicate pred = number -> number > 5;
```

Lambda-Ausdrücke

In Java



TECHNISCHE
UNIVERSITÄT
DARMSTADT

```
1 MyIntPredicate pred = number -> {  
2     return number > 5;  
3 };
```

```
1 MyIntPredicate pred = number -> number > 5;
```

```
1 System.out.println(pred.test(42));
```

```
$ true
```



```
1 @FunctionalInterface
2 public interface MyIntPredicate {
3     boolean test(int number);
4 }
```

```
1 MyIntPredicate pred = number -> number > 5;
```



```
1 @FunctionalInterface
2 public interface MyIntPredicate {
3     boolean test(int number);
4 }
```

```
1 public class X {
2     public static boolean anyName(int num) {
3         return num > 5;
4     }
5 }
```

```
1 MyIntPredicate pred = X::anyName;
```



```
1 @FunctionalInterface
2 public interface MyIntPredicate {
3     boolean test(int number);
4 }
```

```
1 public class X {
2     public boolean anyName(int num) {
3         return num > 5;
4     }
5 }
```

```
1 X anObject = new X();
2 MyIntPredicate pred = anObject::anyName;
```

Das steht heute auf dem Plan



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Structs

Listen

Rekursiv vs Iterativ

Funktionen höherer Ordnung

Lambda-Ausdrücke

Closures

In Racket

In Java



- Einbeziehung von Entstehungskontext in Lambda-Ausdruck
- Dieser wird dann im Lambda-Ausdruck zwischengespeichert
- Lässt sich dann nicht mehr verändern

Closures

In Racket



TECHNISCHE
UNIVERSITÄT
DARMSTADT

```
1 (define (create-func a b)  
2   (lambda (x) (* (+ x a) b)))
```

```
1 ((create-func 0 1) 5)
```

\$ 5

```
1 ((create-func 1 2) 5)
```

\$ 12

Closures

In Java



TECHNISCHE
UNIVERSITÄT
DARMSTADT

```
1 @FunctionalInterface
2 public interface MyDoubleFunction {
3     public double apply(double number);
4 }
```

```
1 public static MyDoubleFunction createFunc(double a, double b) {
2     return x -> (x + a) * b;
3 }
```

```
1 double result = createFunc(0, 1).apply(5); // = 5
```

```
1 double result = createFunc(1, 2).apply(5); // = 12
```



Live-Coding!