

# FOP Tutorium #8

## Fortgeschrittene Konzepte Racket





---

# Guten Tag!

# Das steht heute auf dem Plan



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Boxing

Erinnerung

Switch

Structs

Funktionen höherer Ordnung

Lambda Ausdrücke



- Für jeden primitiven Datentypen gibt es eine Wrapper-Klasse
- Später bei Generics sehr wichtig!

Primitiver Datentyp	Wrapper-Klasse
boolean	<code>java.lang.Boolean</code>
byte	<code>java.lang.Byte</code>
short	<code>java.lang.Short</code>
int	<code>java.lang.Integer</code>
double	<code>java.lang.Double</code>
....	....



- Konvertierung in Wrapper-Klasse: Boxing
- Konvertierung zu primitiven Wert: Unboxing
- Automatisch: Auto -> Auto-Boxing und Auto-Unboxing

```
1 Integer a = ....;
2 int aPrimitive = a; // Auto-Unboxing
3
4 double bPrimitive = ....;
5 Double b = bPrimitive; // Auto-Boxing
```



```
1 Integer a = Integer.valueOf(25); // Boxing
2 int aPrimitive = a.intValue(); // Unboxing
3
4 double bPrimitive = 5;
5 Double b = Double.valueOf(bPrimitive); // Boxing
```



- ACHTUNG!
- Keine Widening oder Narrowing Casts bei Wrapper-Klassen
- Weder explizit noch implizit

```
1 Integer myInteger = 5; // Auto-Boxing
2 Long myLong = myInteger; // ERROR!
3 Long myOtherLong = (Long) myInteger; // ERROR!
4 Byte smallByte = myInteger; // ERROR!
5 Byte smallOtherByte = (Byte) myInteger; // ERROR!
```

# Das steht heute auf dem Plan



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Boxing

Erinnerung

- Klassen richtig benennen

- Strings auf Gleichheit prüfen

- Casting

Switch

Structs

Funktionen höherer Ordnung



# Erinnerung

## Klassen richtig benennen



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

- Keine zu langen Namen wählen
- Leicht wiedererkennbar benennen

# Erinnerung

## Strings auf Gleichheit prüfen



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

- **IMMER** mit `.equals` oder `.equalsIgnoreCase`
- **NICHT** mit `==`



- Funktioniert nur ohne Fehler bei passenden dynamischen Typen!

```
1 class A { .... }
2 class B extends A { .... }
3 class C extends B { .... }
4
5 A a = ....;
6 C castedC = (C) a; // Nur möglich, wenn a dynamischen Typen von C
   ↳ oder Subtypen hat
```



- Man kann mit `instanceof` eingrenzen, welcher dynamischer Typ vorliegt

```
1  Tree happyTree = ....
2  happyTree.grow();
3
4  if(happyTree instanceof LemonTree) {
5      // happyTree ist vom Typ LemonTree oder Unterklasse
6  }
7  if(happyTree instanceof Tree) {
8      // Gilt immer
9  }
10 if(happyTree instanceof AppleTree) {
11     // Könnte gelten, wenn AppleTree Unterklasse von Tree ist
12 }
```

# Erinnerung

## Casting – Bei Objekttypen



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

- Man kann nur Methoden aufrufen und auf Attribute zugreifen, die im statischen Typen definiert sind
- Mit Casting kann man den statischen Typen ändern

```
1 Tree happyTree = ...
2 happyTree.grow();
3
4 if(happyTree instanceof LemonTree) {
5     // happyTree ist vom Typ LemonTree oder Unterklasse
6     LemonTree lemonTree = (LemonTree) happyTree;
7     lemonTree.countLemons();
8 }
```

# Das steht heute auf dem Plan



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Boxing

Erinnerung

Switch

Statement  
Expression

Structs

Funktionen höherer Ordnung



- Bessere Alternative zu gigantischen `if-else`-Konstrukten
- Funktionieren mit
  - ▣ Ganzzahligen Zahlen: `byte`, `short`, `char`, `int`
  - ▣ Strings
  - ▣ Enums
  - ▣ (Auch Boxed: `Byte`, `Short`, `Character`, `Integer`)

# Switch

Statement — Simuliert mit if-else



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1  int j = 5;
```

```
1  if(j == 0) {  
2      // ....  
3  }  
4  else if(j == 1) {  
5      // ....  
6  }  
7  else if(j == 2) {  
8      // ....  
9  }  
10 else ....
```



# Switch

## Statement — Mit break



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1  switch(j) {  
2      case 0:  
3          System.out.println("ZERO!");  
4          break;  
5      case 1:  
6          System.out.println("ONE!");  
7          break;  
8  
9      ....  
10  
11     default:  
12         System.out.println("DEFAULT!");  
13 }
```

- $j = 0 \rightarrow \text{ZERO!}$
- $j = 1 \rightarrow \text{ONE!}$
- $j = -1 \rightarrow \text{DEFAULT!}$

# Switch

## Statement — Mit break



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1  switch(j) {  
2      case 0 -> {  
3          System.out.println("ZERO!");  
4          break;  
5      }  
6      case 1 -> {  
7          System.out.println("ONE!");  
8          break;  
9      }  
10     ....  
11  
12     default ->  
13         ↪ System.out.println("DEFAULT!");  
14 }
```

- $j = 0 \rightarrow \text{ZERO!}$
- $j = 1 \rightarrow \text{ONE!}$
- $j = -1 \rightarrow \text{DEFAULT!}$

# Switch

## Statement — Ohne break



```
1  switch(j) {  
2      case 0:  
3          System.out.println("ZERO!");  
4          // break;  
5      case 1:  
6          System.out.println("ONE!");  
7          // break;  
8  
9      ....  
10  
11     default:  
12         System.out.println("DEFAULT!");  
13 }
```

- $j = 0 \rightarrow$ 
  - ▣ ZERO!
  - ▣ ONE!
  - ▣ ....
  - ▣ DEFAULT!
- $j = 1 \rightarrow$ 
  - ▣ ONE!
  - ▣ ....
  - ▣ DEFAULT!
- $j = -1 \rightarrow$ 
  - ▣ DEFAULT!



```
if(j == 0) {  
    // ....  
}  
else if(j == 1) {  
    // ....  
} else ....
```



```
switch(j) {  
    case 0:  
        // ....  
    case ....  
}
```

# Switch

## Expression – Warum?



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1 String result;  
2  
3 switch(j) {  
4     case 0:  
5         result = "YES";  
6         break;  
7     case 1:  
8         result = "OK";  
9         break;  
10    default:  
11        result = "MAYBE";  
12 }
```

# Switch

Expression – Mit ->



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1 String result;  
2  
3 result = switch(j) {  
4     case 0 -> "YES";  
5     case 1 -> "OK";  
6     default -> "MAYBE";  
7 };
```

# Switch

## Expression – Mit yield



```
1 String result;  
2  
3 result = switch(j) {  
4     case 0 -> {  
5         yield "YES";  
6     }  
7     case 1 -> {  
8         // ....  
9         yield "OK";  
10    }  
11    default -> {  
12        yield "MAYBE";  
13    }  
14 };
```

# Switch

## Expression – Mit yield



```
1 String result;  
2  
3 result = switch(j) {  
4     case 0:  
5         // ....  
6         yield "YES";  
7     case 1:  
8         // ....  
9         yield "OK";  
10    default:  
11        // ....  
12        yield "MAYBE";  
13    };
```



# Switch

Expression — Achtung: Nicht -> und : vermischen



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1 String result;  
2  
3 result = switch(j) {  
4     case 0 -> {  
5         yield "YES";  
6     }  
7     default: // Error: Different case kinds used in the switch  
8         yield "MAYBE";  
9 };
```

# Switch

Expression – Achtung: break vergessen



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1 String result = switch(new Random().nextInt(3)) {  
2     case 0:  
3         result = "YES";  
4     case 1:  
5         result = "OK";  
6     default:  
7         result = "MAYBE";  
8 }  
9 System.out.println(result);
```

```
$ "MAYBE"
```

- ist dann sinnvoll, wenn man den gleichen Code für mehrere Fälle ausführen möchte.
- Bei -> Syntax kann man auch mehrere Fälle angeben, diese werden dann mit , getrennt.

# Das steht heute auf dem Plan



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Boxing

Erinnerung

Switch

Structs

Deklaration und Verwendung  
Java Äquivalent

Funktionen höherer Ordnung



- Haben mehrere Felder
- 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 variable)
```

# Structs

## Deklaration und Verwendung



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1 (define-struct car (door-amount weight length door-window-speed))
```

```
1 (make-car 5 25 200 6)
```

```
1 (define my-car (make-car 5 25 200 6))
```

```
1 (car-door-amount my-car)  
2 (car-weight my-car)
```

# Structs

Java Äquivalent



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1 public class Car {  
2     private final int doorAmount;  
3     private final double weight;  
4     private final int length;  
5     private final double doorWindowSpeed;  
6  
7     // ...  
8 }
```

# Structs

## Java Äquivalent



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1  public class Car {  
2      // ...  
3  
4      public Car(int doorAmount, double weight,  
5                  int length, double doorWindowSpeed) {  
6          // ...  
7      }  
8      public double getDoorWindowSpeed() {  
9          return doorWindowSpeed;  
10     }  
11  
12     // ...  
13 }
```



# Das steht heute auf dem Plan



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

Boxing

Erinnerung

Switch

Structs

Funktionen höherer Ordnung

`filter`

`map`

`foldl` und `foldr`



- Sind Funktionen die als Parameter Funktionen benötigen
- 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))  
4  
5 |->| (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))
```

```
4
```

```
5 |->| (list 6 9 11 15)
```

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

```
2
```

```
3 |->| (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

Boxing

Erinnerung

Switch

Structs

Funktionen höherer Ordnung

Lambda Ausdrücke  
In Racket



- 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)
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))  
2 |->| (list 12 14 16)
```

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



- Funktionieren über Funktionale Interfaces
- Funktionale Interfaces
  - ▣ Haben nur 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 };
```



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

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



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

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

# Lambda Ausdrücke

## Methodenreferenzen – Klassen-Methode



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
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;
```

# Lambda Ausdrücke

## Methodenreferenzen – Objekt-Methode



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
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

Boxing

Erinnerung

Switch

Structs

Funktionen höherer Ordnung

Lambda Ausdrücke



- Wenn man sich im Lambda Ausdruck auf den Context bezieht
- Wird im Lambda Ausdruck zwischengespeichert

# Closures

## In Racket



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT

```
1 (define (create-func a b)
2   (lambda (x) (* (+ x a) b))
3 )
```

```
1 ((create-func 0 1) 5)
2 |->| 5
```

```
1 ((create-func 1 2) 5)
2 |->| 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!