

# Web Programming

## Woche 4

*"Developers seem to love those languages most, in which they understood the value of higher-order functions."*

@ProfDKoenig

# Retrospective

JS Goodie

Last Week Refresher

Open Questions

# Agenda

Applied Map/Filter/Reduce

Snake and Tuple(n)

Quiz

# $(a, b)$ vs. $a \Rightarrow b \Rightarrow$

*// multiple arguments*

```
const times = (a, b) => a * b;
```

```
times(2) // ???
```

*// argument chain*

```
const times = a => b => a * b;
```

```
times(2) // ???
```

*error message?*

*useful?*

# Partial Application

Is particularly elegant in combination with higher-order functions like in

`map`, `filter`, and `reduce`

map

1

2

3

$x \Rightarrow x * 2$

map

1 2 3

$x \Rightarrow x * 2$

2 4 6

# "partial" application: map

```
const times      = a => b => a * b;
```

```
const twoTimes = times(2);
```

```
[1, 2, 3].map(x => times(2)(x));  
[1, 2, 3].map(times(2));  
[1, 2, 3].map(twoTimes);
```



filter

1

2

3

$$x \Rightarrow x \% 2 == 1$$

filter

1

2

3

$$x \Rightarrow x \% 2 == 1$$

1

3

# "partial" filter

```
const odd      = x => x % 2 === 1;
```

```
[1, 2, 3].filter(x => x % 2 === 1);  
[1, 2, 3].filter(x => odd(x));  
[1, 2, 3].filter(odd);
```

reduce((acc, cur) => acc + cur)

1

2

3

reduce((acc, cur) => acc + cur)



reduce((acc, cur) => acc + cur)



reduce((acc, cur) => acc + cur)

1

2

3

6

# "un-partial" reduce

```
const plus = (accu, cur) => accu + cur;
```

```
[1, 2, 3].reduce((accu, cur) => accu + cur);  
[1, 2, 3].reduce(plus);
```

```
// variant with initial accu value as 2nd argument  
// then cur starts at first element
```

```
[1, 2, 3].reduce(plus, 0);
```



# Functions everywhere

Literal scope (IIFE)

Capturing scope (closures)

Higher-order functions

Constructors (returning functions)

# Pair, Product Type

```
const pair = x => y => f => f(x)(y);  
const fst  = p => p(T);  
const snd  = p => p(F);
```

the basic product type

# Either, Co-Product, Sum

```
const Left    = x => f => g => f(x);           // ctor 1  
const Right   = x => f => g => g(x);           // ctor 2  
const either  = e => f => g => e(f)(g);        // accessor
```

the basic sum type

# Special Case: Maybe

```
const Nothing = Left ();  
const Just    = Right  ;  
const maybe  = either ;
```

```
maybe (expressionThatMightGoWrong)  
      (handleBad)  
      (handleGood);
```

*go around null / undefined*

# Lambdify Snake

Use pairs and either where possible

Follow the todos

# Neue Konzepte in Snake

`plus + pair == pair`      `// monoid`

`map (f) (pair) == pair`    `// functor`

# To Do at Home

Complete lambdaified snake.

Make the following work:

```
[1,2,3].reduce(preOrder, []) === [3,2,1]
```