



# LIGHTNING TALKS

- share anything cool/fun/interesting  
(blog posts, new RFCs, funny bugs, open-source projects)
- <=5 minute slots
- max. 3-4 "talks"
- slides, no slides, anything goes
- (not intended for recruiting)

# Functional Programming

## for Pragmatic PHP Developers



PHP User Group Munich - 2017-09-27

Michael Haeuslmann for [TNG Technology Consulting GmbH](#) / [@michaelhaeu](#)

# Disclaimer

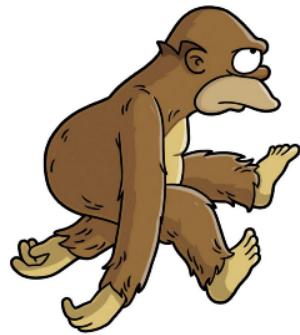
*I am not an expert at functional programming. These opinions are my own and do not reflect the opinions of proper functional programmers ;)*

*This talk is from a pragmatic developer for other pragmatic developers.*

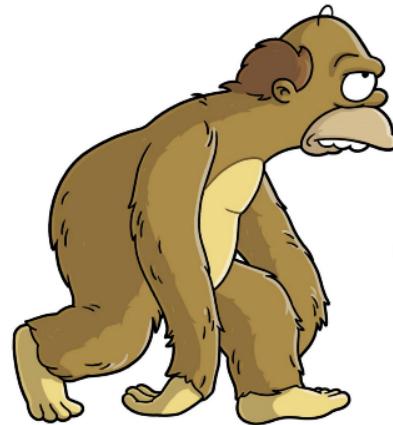
*We want to get stuff done!*



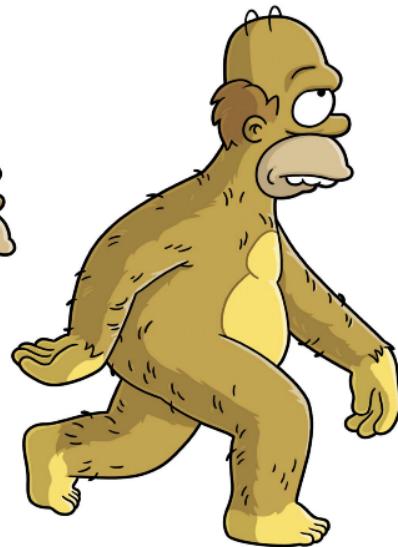
MACHINE



ASSEMBLY



PROCEDURAL

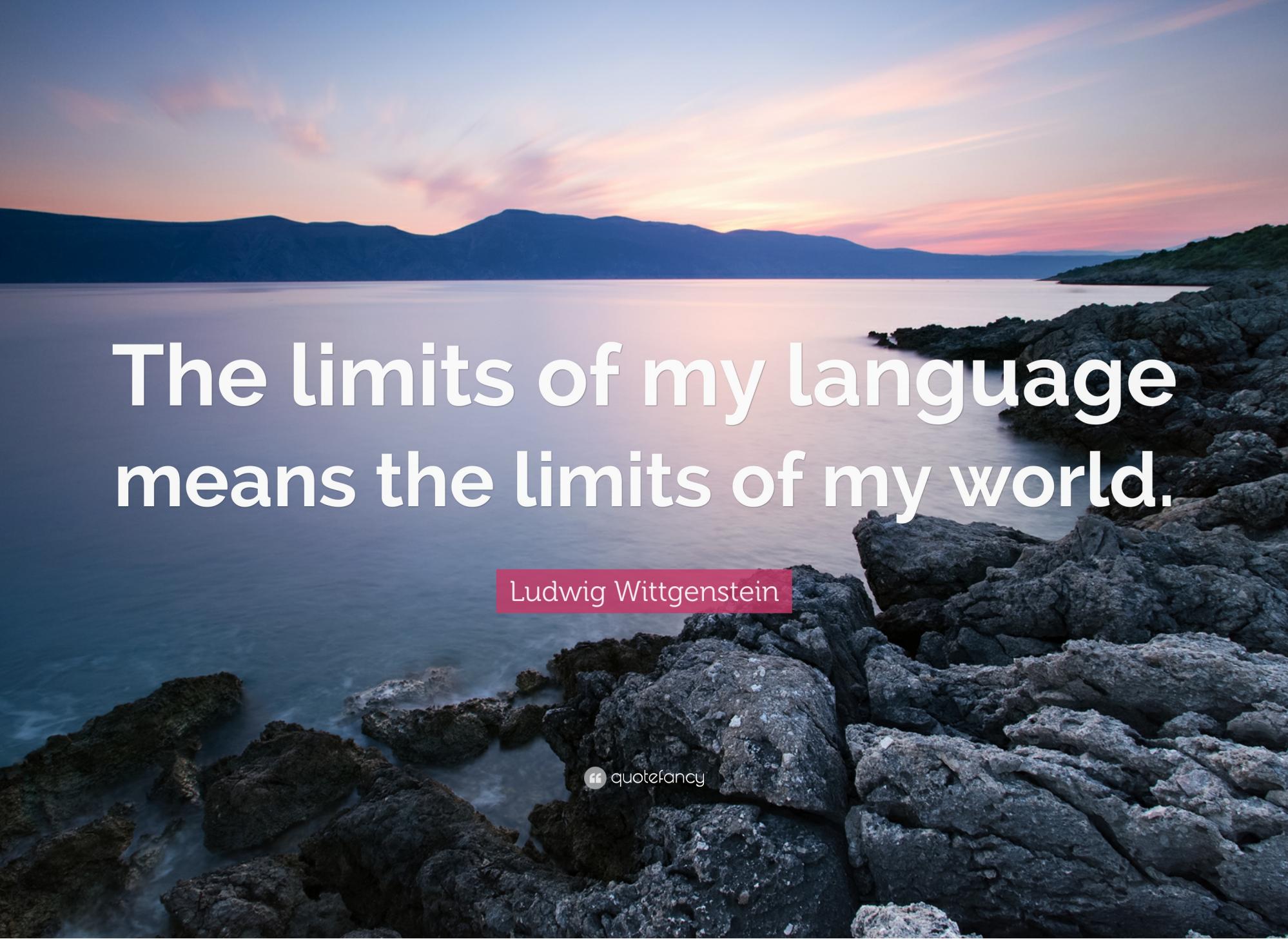


OBJECT ORIENTED



FUNCTIONAL

MATT GROENING

A wide-angle photograph of a coastal landscape at sunset. The sky is filled with soft, pastel-colored clouds transitioning from blue to orange and yellow. In the distance, dark silhouettes of mountains are visible across the water. The foreground consists of dark, rugged rocks and a small, rocky peninsula extending into the water.

The limits of my language  
means the limits of my world.

Ludwig Wittgenstein



quotefancy

# Why functional programming?

- Declarative vs. Imperative (what vs how?)
- More concise
- More generic and reusable
- ~~Mathematically proven~~
- ~~Parallel Programming~~
  - ⇒ A different way of looking at problems

# Declarative vs Imperative

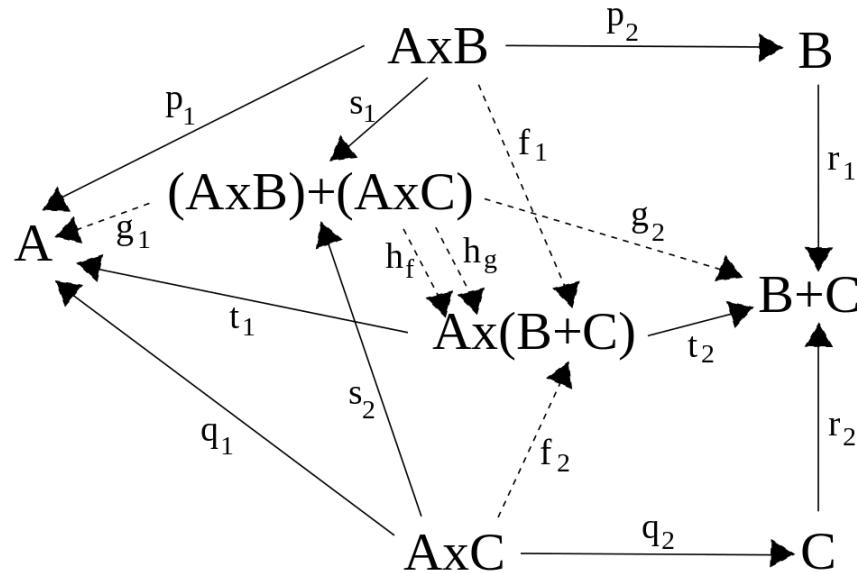
SQL is declarative,  
we only say **what** we want,  
and not **how**.

```
SELECT *
FROM users
WHERE age="42"
```

That doesn't mean we have to like SQL  
:)

# ~~Mathematically proven~~

## Category Theory

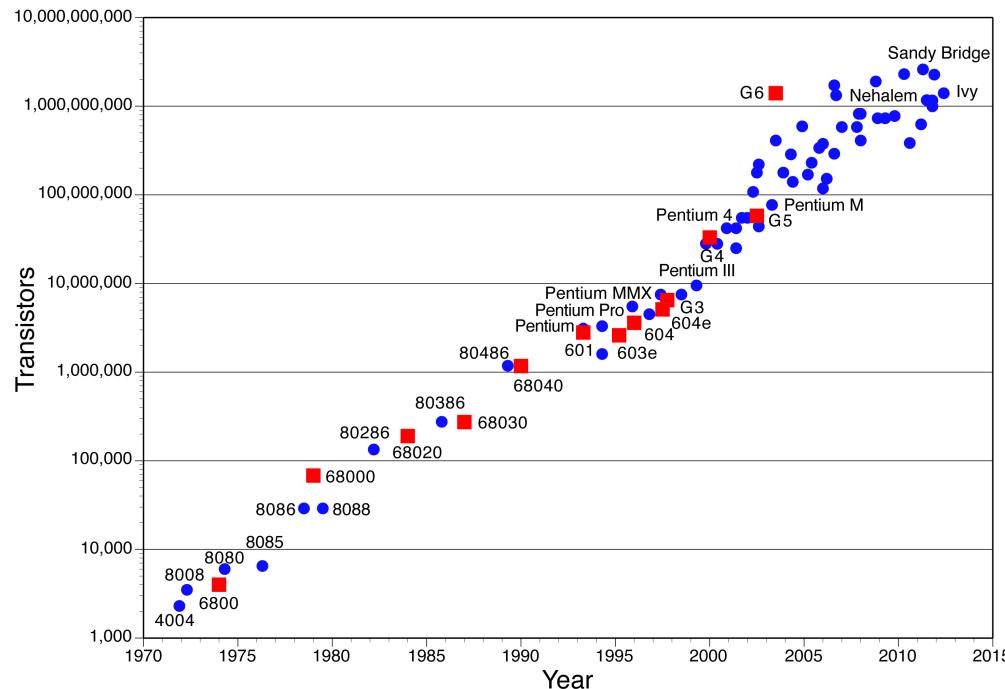


<http://ism.uqam.ca/~ism/academics/category-theory-and-applications/>

# Parallel Programming

## Moore's Law

Processor's aren't getting any faster,  
we can only expect to get more cores.



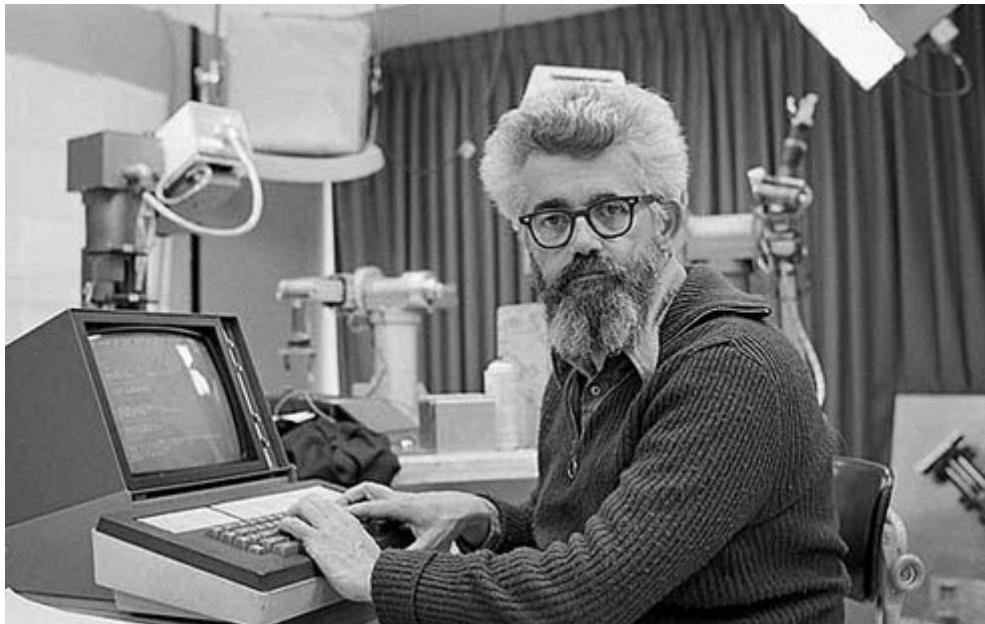
# Functional Programming is not a Hype



# 1958 - LISP

First functional language

```
(defun factorial (n &optional (acc 1))
  (if (= n 0) acc
      (factorial (- n 1) (* acc n))))
```



# 1962 - Simula

## First object-oriented language

```
Simulation Begin
  Class FittingRoom; Begin
    Ref (Head) door;
    Boolean inUse;
    Procedure request; Begin
      If inUse Then Begin
        Wait (door);
        door.First.Out;
      End;
      inUse:= True;
    End;
    Procedure leave; Begin
      inUse:= False;
      Activate door.First;
    End;
    door:- New Head;
  End;

  Integer u;
  Ref (FittingRoom) fittingRoom1;

  fittingRoom1:- New FittingRoom;
  Hold (100);
End;
```

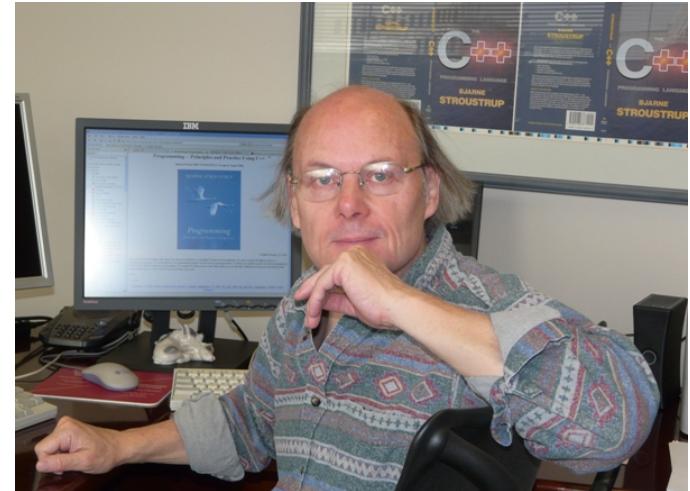
# 1980 - C++

## C-with-classes

```
#include <iostream>

using namespace std;

class Box {
public:
    double length;    // Length of a box
    double breadth;   // Breadth of a box
    double height;    // Height of a box
};
```



# 1984 - Common Lisp

- modern Lisp dialect
- still too many brackets

# 1990 - Haskell

Purely functional and lazy

```
factorial n
| n < 2      = 1
| otherwise = n * factorial (n - 1)
```

# 1995 - JavaScript

Not functional, but functions always have been first-class citizens

```
class Pseudo00 {
  method() {
    return
      `I won't return what you
      think I should.`;
  }
}

const tax = R.curry((tax, price) => {
  if (!_.isNumber(price)) return Left(new Error());

  return Right(price + (tax * price));
});
```



# 1995 - Java

"Modern" OO, support for a more functional style since  
Java 8

```
List<String> myList =  
    Arrays.asList("a1", "a2", "b1", "c2", "c1");  
  
myList  
    .stream()  
    .filter(s -> s.startsWith("c"))  
    .map(String::toUpperCase)  
    .sorted()  
    .forEach(System.out::println);  
  
// C1  
// C2
```

# 2003 - Scala

Modern Java with many functional features

```
class Point(  
    val x: Double, val y: Double,  
    addToGrid: Boolean = false  
) {  
    // ...  
}  
  
val a = List(1, 2, 3, 4, 5)  
val b = a.map(x => x * x)  
val c = b.reduce((x, y) => x + y)
```

# 2005 - F#

## Microsoft's answer to Scala

```
type Person(name : string, age : int) =  
    member x.Name = name  
    member x.Age = age
```

# 2007 - Clojure

- Lisp with objects
- now runs on the JVM
- still too many brackets

```
(def bit-bucket-writer
  (proxy [java.io.Writer] []
    (write [buf] nil)
    (close [] nil)
    (flush [] nil)))
```

# 1995 - PHP

- 4.0 - `array_map`, `array_filter`, `array_reduce`
- < 5 - procedural, painful, "C as a scripting language with convenience functions"
- 5.4 - lambdas and closures, callable
- 5.5 - generators
- 5.6 - varargs
- 7.0 - anonymous classes, scalar return types, direct call
- 7.2 - `Closure :: fromCallable`
- 7.3 - ??? Fat arrow functions, pipes, short closures

# Arrow functions

- concise
- easier to use
- automatic binding of scope to closure

```
array_map( fn($x) => $x * 2, [1, 2, 3] );
```

by Levi Morrison, Bob Weinand

Status: Under Discussion

# Pipe Operator (Hack)

Before:

```
$ret =  
    array_merge(  
        $ret,  
        getFileArg(  
            array_map(  
                function ($x) use ($arg) { return $arg . '/' . $x; },  
                array_filter(  
                    scandir($arg),  
                    function ($x) { return $x !== '.' && $x !== '..'); }  
                )  
            )  
        );
```

by Sara Golemon  
Status: Under Discussion

# Pipe Operator (Hack)

After:

```
$ret = scandir($arg)
|> array_filter($$, function($x) { return $x !== '.' && $x != '..'; })
|> array_map(function ($x) use ($arg) { return $arg . '/' . $x; }, $$)
|> getFileArg($$)
|> array_merge($ret, $$);
```

by Sara Golemon

Status: Under Discussion

# Short Closures

Before:

```
$action = CustomerController::class . '::delete';
```

After:

```
$action = {CustomerController::delete};  
  
// which is a simplification for  
$action = Closure::fromCallable('CustomerController::delete');
```

by Michał Brzuchalski  
Status: Draft

# externals.io

#externals

Opening PHP's #internals to the outside.

Search with  algolia

Search...

- 
-  **[VOTE] JSON\_THROW\_ON\_ERROR**  
2   
① 21 hours ago  Andrea Faulds  2

- 
-  **[RFC] [Discussion] Implement SQLite "openBlob" feature in PDO**  
0   
① 16 hours ago  BohwaZ/PHP

- 
-  **PHP extension - Saxon/C**  
0   
① 4 days ago  O'Neil Delpratt  9

# Core principles of functional programming

- Functions as first-class citizens
- Pure functions (i.e. no side-effects)
- Higher order functions
- Immutability
- Stateless
- Currying and composition

# Functions as first-class citizens

- Functions can be passed as arguments
- Functions can be returned from other functions
- Functions can be assigned to variables

```
$times = function ($x) {  
    return function ($y) use ($x) {  
        return $x * $y;  
    };  
};  
$times(7)(6); // 42
```

# Pure functions

(i.e. no side-effects)

The output of a function, depends only on the input arguments.

```
function getUsers($db) {  
    return $db->query('SELECT * FROM users');  
}  
  
public function getUsers() {  
    return $this->db->query('SELECT * FROM users');  
}
```

*You wanted a banana but what you got  
was a gorilla holding the banana and the  
entire jungle.*

*- Joe Armstrong, Creator of Erlang*



# Stateless

- the more state, the harder to test
- more reasons to change
- unpredictable when state is shared and mutable
- (hard to parallelize)

```
function bumpCounter() {  
    static $counter = 0;  
    ++$counter;  
    echo $counter;  
}  
bumpCounter(); // 1  
bumpCounter(); // 2
```

# Immutability

- the hardest thing when debugging is to figure out where a variable went wrong
- encourages pure functions (you always want to return something)
- allows checking for references instead of inner state

```
class Counter{  
    private $counter = 0;  
  
    public function increment() {  
        $clone = clone $this;  
        ++$clone->counter;  
        return $clone;  
    }  
  
    public function __toString() { /** ... */}  
}  
  
$c1 = (new Counter)->increment();  
$c2 = $c1->increment();  
echo "$c1 < $c2"; // 1 < 2
```

# Higher-order functions

- functions which receive other functions as arguments
- and/or return a function

```
$xs = [4, 5, 1, 2, 3];
usort($xs, function ($x, $y) {
    return $x <= $y;
});
```

# Currying and composition

Describing addition in "Hindley-Milner Form" (Haskell):

```
add :: a → b → c
```

- Add is a function which takes an **a**
- ... and returns a function which takes a **b** ...
- ... and returns a **c**.

So why not simply

```
add(int $a, int $b) : int?
```

# Let's say we want an increment function

We could do it the normal way:

```
function increment($x) {  
    return $x + 1;  
}
```

But we already have everything we need:

```
$increment = add(1);
```

# Another one:

```
map :: (a → b) → [a] → [b]
```

Every function takes only one argument.

- Map is a function which takes an function ...
- ... which maps an **a** to a **b** ...
- ... and returns a function which takes a list of **a** ...
- ... and returns a list of **b**.

# Examples of functional style

- Route definition in micro frameworks
- Promises
- Filter, map, reduce

# Filter, map, reduce

- `array array_filter ( array $array [, callable $callback [, int $flag = 0 ]] )`
- `array array_map ( callable $callback , array $array1 [, array $... ] )`
- `mixed array_reduce ( array $array , callable $callback [, mixed $initial = NULL ] )`
- `bool array_walk ( array &$array , callable $callback [, mixed $userdata = NULL ] )`
- `array array_udiff ( array $array1 , array $array2 [, array $... ], callable $value_compare_func )`
- ...

# Type-safe "iteration"

```
foreach ($xs as $x) {  
    /** @var Object $x */  
    $x->sideEffect();  
}  
  
array_walk($xs, function (Object $x) {  
    $x->sideEffect();  
});  
  
each($xs, fn($x) => $x->sideEffect());
```

# Route definition in micro frameworks

```
$app = new App();
$app->get(
    'customers',
    function (RequestInterface $request, ResponseInterface $response) : ResponseInterface {
        return $response;
    }
);
```

# Promises

- Async promises (e.g. from ReactPHP) use functional concepts
- Wrap side-effects
- higher order functions for result handling
- map, reduce etc. for results
- not lazy

(see continuation monad)

# reactphp / promise

```
$deferred = new React\Promise\Deferred();
$deferred->promise()
    ->then(function ($x) {
        return $x + 1;
    })
    ->then(function ($x) {
        throw new \Exception($x + 1);
    })
    ->otherwise(function (\Exception $x) {
        return $x->getMessage() + 1;
    })
    ->then(function ($x) {
        echo 'Mixed ' . $x;
    });
$deferred->resolve(1);
```

# Functional PHP libraries

- functional-php
  - small functions for everyday use
  - use what you need
  - not invasive
- php-functional
  - more complicated functional features
  - low coverage (~60%)
  - interesting implementation of various monads
- <https://preprocess.io/>
  - not a library
  - pre-processor (duh!)
  - makes functional programming more enjoyable

# functional-php

```
function filter($collection, callable $callback) : array {}

function map($collection, callable $callback) : array {}

function reduce_left($collection, callable $callback, $initial = null) : array {}

function reduce_right($collection, callable $callback, $initial = null) : array {}

function each($collection, callable $callback) : null {}

// ...
```

# functional-php

by Lars Strojny (@lstrojny)

Imperative:

```
foreach ($xs as $x) {  
    if ($x % 2 === 0) {  
        return true;  
    }  
}  
return false;
```

Declarative:

```
Functional\some($xs, function ($x) {  
    return $x % 2 === 0;  
});
```

# Let's take this further

## A curried mod function

```
function mod($mod) {  
    return function ($x) use ($mod) {  
        return $x % $mod;  
    };  
}  
some($xs, mod(2));
```

## Or: if our operators were functions

```
$even = curry('%')(2);      // operators as functions RFC pending  
some($xs, $even);
```



(<https://github.com/preprocess>)



(<https://twitter.com/assertchris>)

---

code.pre

code.php

output

```
1 <?php  
2  
3 |
```

# Drawbacks

- takes getting used to
- not as intuitive (think first, debug less later)
- knowledge not yet widespread (yet)
- PHP is not made for purely functional style (yet)
- performance?

# Performance ...

- Premature optimization is the root of all evil
- copy on write helps
- don't worry about collections with less than 10000 entries



# Lessons learned

- Functional concepts can help us write better OO code
- Pure functions and immutability will save you from debugging
- Higher order functions and composition help simplify complex OO patterns
- Libraries provide common functions (functional-php)
- Document your callables using HMF (Hindley-Milner Form)

# Thank you! Any Questions?

Become vocal about  
functional programming in  
PHP.

# Sources

- <https://github.com/Istrojny/functional-php>
- <https://github.com/widmogrod/php-functional>
- <https://www.slideshare.net/boerdedavid/being-functional-in-php>
- ~~"Functional Programming in PHP" by Simon Holywell~~

# Figures

- Haskell Comic <https://xkcd.com/1312/>
- Lightning Talks <https://www.meetup.com/ReactJS-B>
- Wittgenstein <https://quotefancy.com/quote/100807-means-the-limits-of-my-world>
- McCarthy <http://history-computer.com/ModernCon>
- Stroustrup <http://www.stroustrup.com/Bjarne.jpg>
- Gorilla  
<http://media.tumblr.com/97691c212d7d07c2b6409>
- Simpsons <https://medium.com/@cscalfani/so-you-want-to-be-a-simpson-1f15e387e536>
- Moore's Law <http://www.overclock.net/t/1542835/reviews/7nm-chips/30>
- Category Theory <http://ism.uqam.ca/~ism/academic.html>

