

ELIXIR

a talk for college students

CodeWeek 15
Informatics Dept. , UMinho

STORYTIME

23/09/2011



23/09/2011

ONE MILLION TCP CONNECTIONS



23/09/2011

**ONE MILLION TCP CONNECTIONS
ON A SINGLE NODE**



23/09/2011

**ONE MILLION TCP CONNECTIONS
ON A SINGLE NODE
WITH RESOURCES TO SPARE**

23/09/2

ONE M

ON A S

WITH R



13:04

7%



- FUNCTIONAL
- FAULT TOLERANT
- DISTRIBUTED & SCALABLE
- PARALLEL
- HOT CODE SWAPPING



- **LIGHTWEIGHT PROCESSES**
- **ACTOR MODEL**
- **TRIED AND TESTED VM**
- **GARBAGE COLLECTION**
- **NETWORK PROTOCOLS**

YAHOO!

facebook®



MOTOROLA



WhatsApp



ERICSSON



heroku

amazon.com®

T-Mobile®

THE
HUFFINGTON
POST

ERLANG

IS

AWESOME



ERLANG

IS

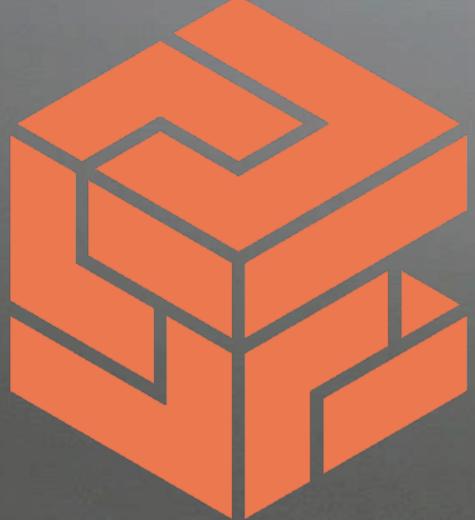
AWESOME

... but why Elixir, then?

my awesome sister who
doesn't code but it's all good
'cause she doesn't like Java



github: @frmendes
everywhere else: @fribmendes



cesium







lol, not my cat



love me, please



Elixir

everything erlang has to offer



Elixir

beautiful syntax

A photograph of a person working at a wooden desk. On the desk are a laptop, a notebook with a pen, a glass of coffee, and a pair of glasses. The background is blurred.

ELIXIR

powerful libraries



Elixir

metaprogramming \m/

A photograph of a person working at a wooden desk. On the desk is a laptop, a notebook with handwritten notes, a pen, a pair of glasses, and a coffee cup. The scene is lit from above, creating a warm atmosphere.

ELIXIR

awesome DSLs



ELIXIR

mix ecosystem

An aerial photograph showing a series of agricultural fields on a hillside. The fields are rectangular and appear to be planted in different crops, creating a pattern of varying shades of green and brown. The terrain is rugged and uneven, with deep furrows between the fields.

ELIXIR

ASA LANGUAGE

ELIXIR

```
iex> 1                      # integer
iex> 0x1F                   # integer
iex> 1.0                     # float
iex> true                    # boolean
iex> :atom                   # atom / symbol
iex> "elixir"                # string
iex> [1, 2, 3]               # list
iex> {1, 2, 3}               # tuple
```

LISTS



ELIXIR

```
iex> [1, 2, true, 3]
```

```
[1, 2, true, 3] ← linked list
```

```
iex> tuple = {:ok, "hello"}
```

```
{:ok, “hello”} ← tuple - contiguous memory
```

ELIXIR

```
iex> tuple = {:ok, "hello"}  
{:ok, “hello”}
```

```
iex> put_elem(tuple, 1, "world")  
{:ok, “world”}
```

```
iex> tuple  
{:ok, “hello”}
```

ELIXIR

```
iex> tuple = {:ok, "hello"}  
{:ok, “hello”}
```

```
iex> put_elem(tuple, 1, "world")  
{:ok, “world”}
```

```
iex> tuple ← immutability  
{:ok, “hello”}                                           aka “lolno.”
```

PATTERN MATCHING



ELIXIR

```
iex> x = 1
```

```
1
```

```
iex> x
```

```
1
```

```
iex> 1 = x
```

```
1
```

ELIXIR

```
iex> x = 1
```



$x = l?$

1

what is l?

l is l

```
iex> x
```

x is l

1

```
iex> 1 = x
```

1

ELIXIR

```
iex> x = 1
```

```
1
```

```
iex> x
```

```
1
```

```
iex> 1 = x
```



I know what's 1! it's x!

```
1
```

ELIXIR

```
iex> x = 1
```

```
1
```

```
iex> 2 = x
```

```
** (MatchError) no match of right hand side value: 1
```

```
iex> 1 = y ← What is l?
```

```
** (RuntimeError) undefined function: y/0
```

ELIXIR

```
iex> x = 1
```

```
1
```

```
iex> 2 = x
```

```
** (MatchError) no match of right hand side value: 1
```

```
iex> 1 = y ← I don't know y so y can't be 1
```

```
** (RuntimeError) undefined function: y/0
```

ELIXIR

```
iex> [head | tail] = [1, 2, 3]  
[1, 2, 3]
```

```
iex> head  
1
```

```
iex> tail  
[2, 3]
```

ELIXIR

```
iex> x = 1
```

```
1
```

```
iex> ^x = 2
```

```
** (MatchError) no match of right hand side value: 2
```

JAVA

```
int x = 1;  
if (x == 1)  
    y = 2;
```

JAVA

```
int x = 1;  
if (x == 1)  
    y = 2;
```

RUBY

```
x = 1  
y = 2 if x = 1
```

JAVA

```
int x = 1;  
if (x == 1)  
    y = 2;
```

RUBY

```
x = 1  
y = 2 if x = 1
```

HASKELL

```
x = 1  
y = f x  
where f 1 = 2
```

JAVA

```
int x = 1;  
if (x == 1)  
    y = 2;
```

RUBY

```
x = 1  
y = 2 if x = 1
```

HASKELL

```
x = 1  
y = f x  
where f 1 = 2
```

ELIXIR

```
x = 1  
{y, ^x} = {2, 1}
```

ELIXIR

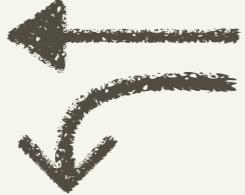
```
iex> 'hello' == "hello"  
false
```

```
iex> [104, 101, 108, 108, 111]  
'hello'
```

ELIXIR

```
iex> 'hello' == "hello"
```

```
false
```



wait, what?

```
iex> [104, 101, 108, 108, 111]
```

```
'hello'
```



STRINGS

A black and white photograph of a skein of dark brown, textured yarn. The skein is coiled in a loose circle, with several strands fanning out from the center. In the lower right foreground, a single, straight knitting needle lies diagonally across the frame.

UTF-8 STRINGS



UTF-8 STRINGS

“   ” .length

?

“   ” .size

?

“   ” .size

4 bytes each emoji

“baffle”.length

?

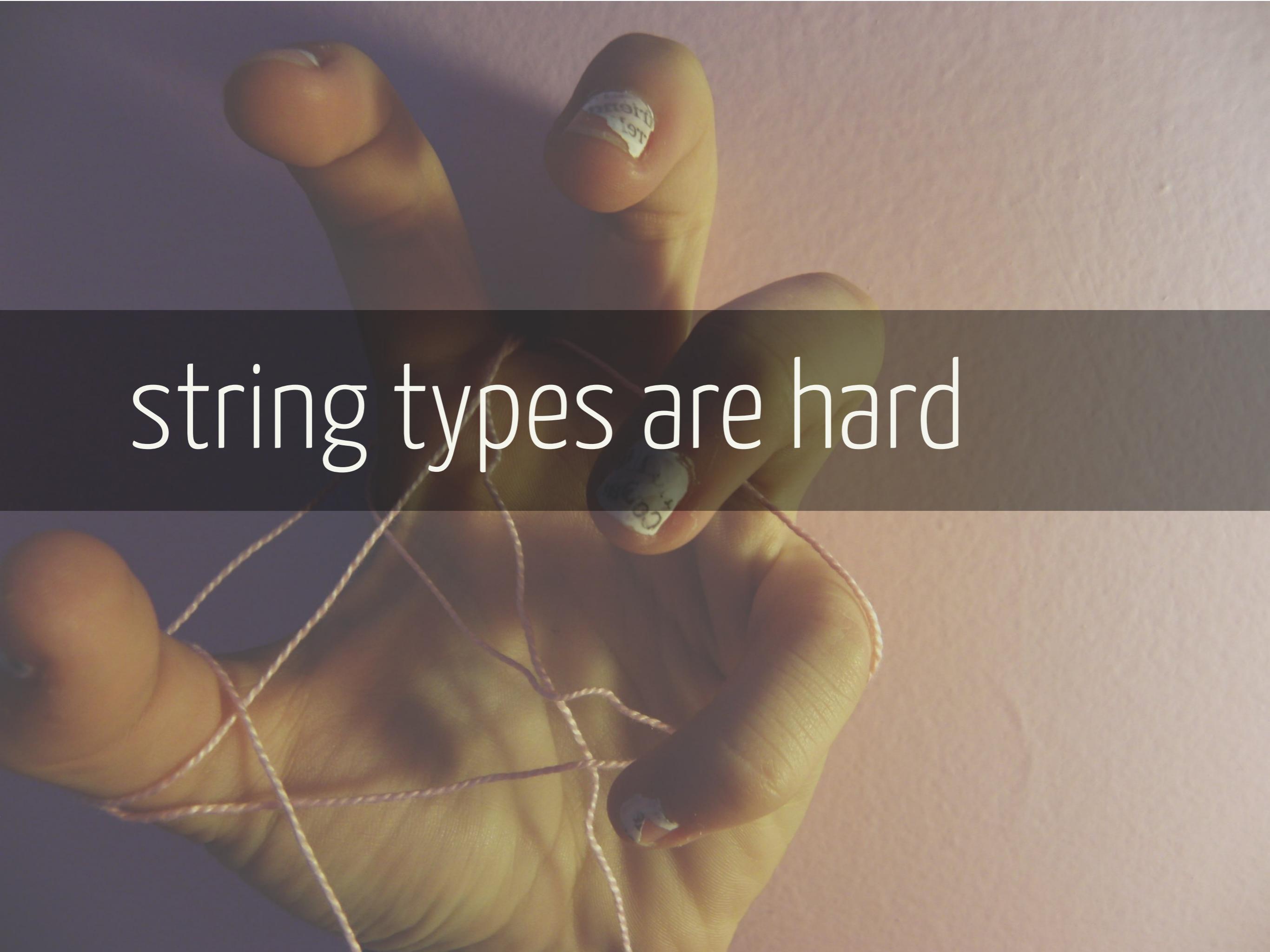
“baffle”.size

?

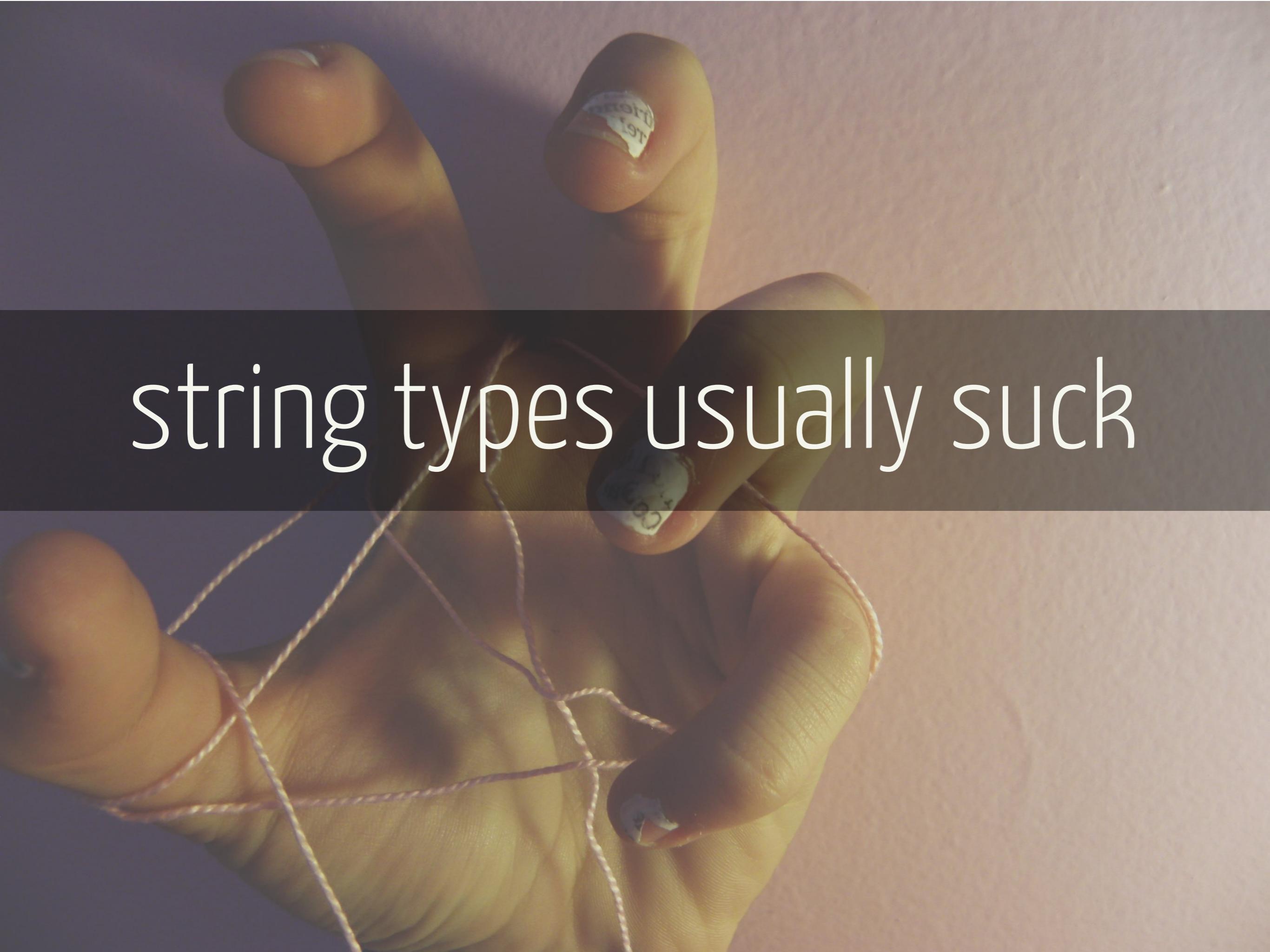
“baffle”.size

length 4

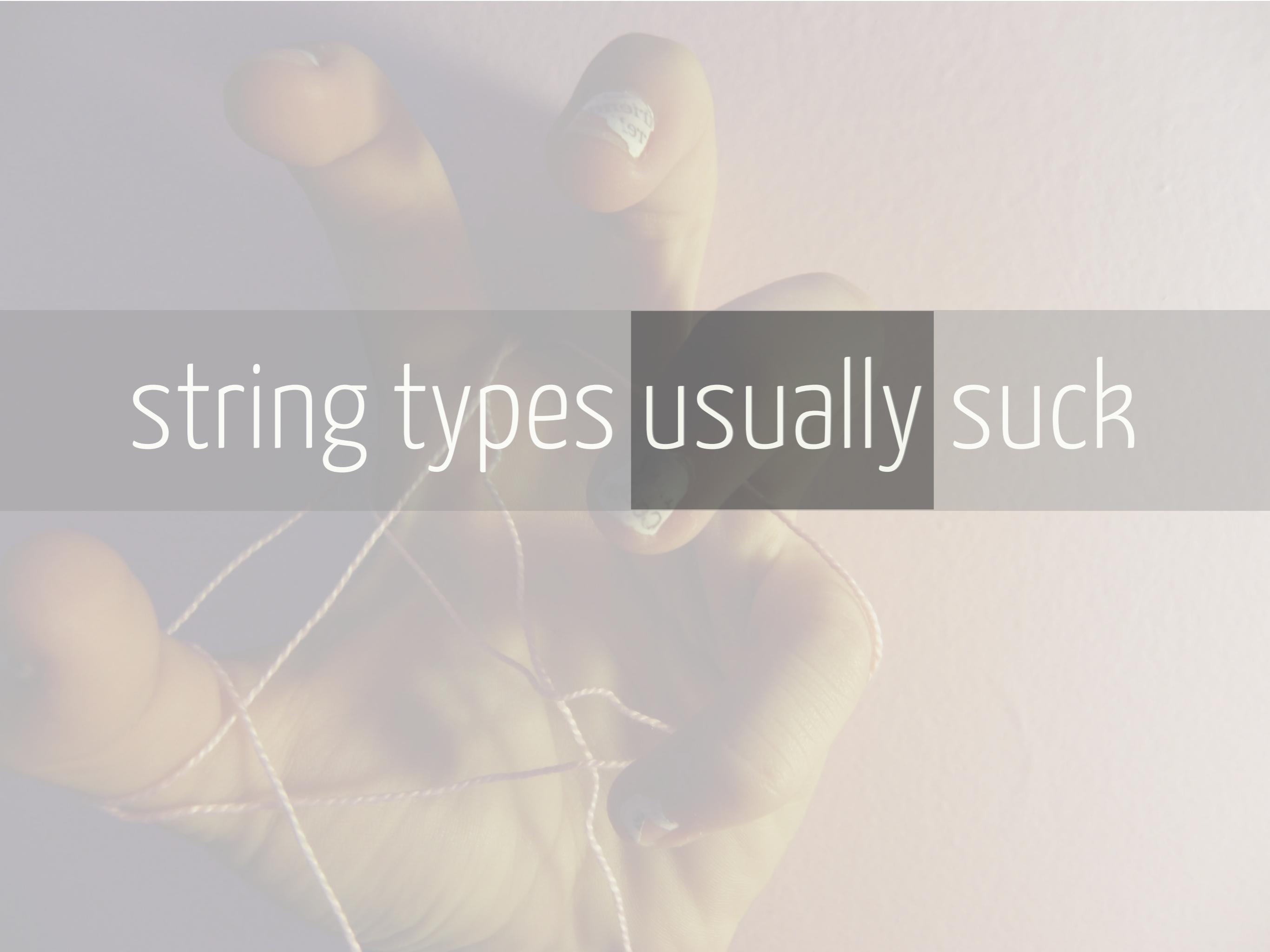
size 6

A close-up photograph of a person's hand wearing a yellow nitrile glove. The hand is holding several bright orange, oval-shaped objects, likely eggs, which are secured together by a light-colored string tied in a knot. The background is a plain, light-colored surface.

string types are hard

A close-up photograph of a person's hand wearing a yellow glove, holding a string of yellow beads. Each bead has a small white tag with handwritten text on it. The background is dark.

string types usually suck

A close-up photograph of a person's hand holding a guitar neck. A capo is attached to the neck, and the strings are visible. The background is blurred.

string types | usually suck

JAVA

```
“🐱🐱🐱🐱”.length();
```

> 8



ffl is a single unicode character

```
“baffle”.length();
```

> 4

JAVA

```
new StringBuilder("baffle").reverse();
```

```
> "efflab"
```

```
"baffle".toUpperCase();
```

```
> "BAFFLE"
```

RUBY

“   ”.length

> 4

“baffle”.length

> 4

RUBY

“baffle”.reverse

> “efflab”

“baffle”.upcase

> “BAfflE”

RUBY

```
def 😺  
  puts "it works!"  
end
```



```
> "it works!"
```

} UTF-8 file encoding
means we get to write
awesome,
readable code!

ELIXIR

```
iex> byte_size “🐱🐱🐱🐱”
```

16

```
iex> String.length “🐱🐱🐱🐱”
```

4

ELIXIR

```
iex> byte_size “baffle”
```

6

```
iex> String.length “baffle”
```

4

ELIXIR

```
iex> String.codepoints “🐱🐱🐱🐱”  
[“🐱”, “🐱”, “🐱”, “🐱”]
```

```
iex> String.codepoints “baffle”  
[“b”, “a”, “f”, “f”, “e”]
```

```
iex> to_char_list “baffle”  
[98, 97, 64260, 101]
```

ELIXIR

```
iex> String.reverse "baffle"  
"efflab"
```

```
iex> String.upcase "baffle"  
"BAFFLE"
```

ELIXIR

```
iex> "he" <> "llo"  
"hello"
```

```
iex> "he" <> rest = "hello"  
"hello"  ↑ Concatenation operator for  
          pattern matching
```

```
iex> rest  
"llo"
```

A large-scale chessboard made of stone or concrete is set up on a city street. The board is a dark grey color with light grey squares. People are standing around the board, some watching the game. A young boy in a green shirt is sitting on the ground to the right of the board, looking at it.

BINARIES

```
iex> <<0, 1, 2, 3>>
```

```
<<0, 1, 2, 3>>
```



“basically” a byte array

```
iex> byte_size <<0, 1, 2, 3>>
```

4

ELIXIR

```
iex> <<256>>
```

```
<<0>>      ↪ truncation
```

```
iex> <<256 :: size(16)>>
```

```
<<1, 0>>    ↪ “yo, use 2 bytes!”
```

ELIXIR

```
iex> <<256 :: utf8>>
```

```
"Ā"
```



treat as codepoint

ELIXIR

use a single bit

```
iex> <<1 :: size(1)>>  
<<1::size(1)>>
```

```
iex> <<2 :: size(1)>>  
<<0::size(1)>>           ← truncation
```

ELIXIR

```
iex> is_binary(<< 1 :: size(1)>>)  
false
```

```
iex> is_bitstring(<< 1 :: size(1)>>)  
true
```

```
iex> bit_size(<< 1 :: size(1)>>)
```

```
1
```

```
iex> byte_size(<< 1 :: size(1)>>)
```

```
1
```



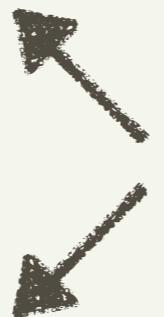
Erlang VM still allocates 1 byte

ELIXIR

```
iex> byte_size(<< 1 :: size(8)>>)
```

1

byte allocation



```
iex> byte_size(<< 1 :: size(9)>>)
```

2

KEYWORD LISTS



```
iex> list = [{:a, 1}, {:b, 2}]  
[a: 1, b: 2]
```

```
iex> list == [a: 1, b: 2]
```

```
true
```



special keyword list syntax

```
iex> list[:a]
```

```
1
```

ELIXIR

```
iex> new_list = [a: 0] ++ list  
[a: 0, a: 1, b: 2] ← Allows  
                           repeated keys  
iex> new_list[:a]  
0
```

```
iex> new_list = [a: 0] ++ list  
[a: 0, a: 1, b: 2]
```

```
iex> new_list[:a]
```

0



first value of the lookup

ELIXIR

```
iex> if(false, [do: :this, else: :that])  
:that
```

ELIXIR

```
query = from w in Weather,  
         where: w.prcp > 0,  
         where: w.temp < 20,  
         select: w
```

MAPS



JAVA

```
Map<String, Integer> map = new HashMap<>();
```

```
map.put("a", 1);
```

```
map.put("b", 2);
```

JAVA



Short version is long

```
Map<String, Integer> map = new HashMap<>();
```

```
map.put("a", 1);
```

```
map.put("b", 2);
```

JAVA

```
Map<String, Integer> map = new HashMap<>();
```

```
map.put("a", 1);
```

```
map.put("b", 2);
```

Auto-boxing

RUBY

```
map = {a: 1, b: 2}
```

```
other_map = {a: 1, 2 => :b}
```

ELIXIR

```
map = %{a: 1, b: 2}
```

```
other_map = %{a: 1, 2 => :b}
```

```
iex> other_map[2]
```

```
:b
```



Elegant pattern matching

```
iex> %{a: a} = other_map
```

```
%{a: 1, 2 => :b}
```

```
iex> a
```

```
1
```

↓ Elegant pattern matching failing

```
iex> %{b: b} = other_map  
** (MatchError) no match of right hand side value:  
%{2 => :b, :a => 1}
```

← No error

```
iex> other_map[:b]  
nil
```

FACTORY

Schiffbrücke

6

LOGICAL OPERATORS

ERLANG

1> true and false.

false

2> false or true.

true

3> not (true and true)

false

ERLANG

1> true and false.

false



Always evaluate on each side

2> false or true.

true

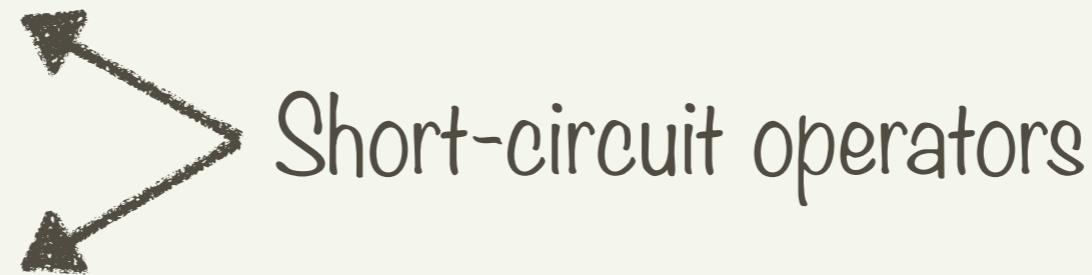
3> not (true and true)

false

ERLANG

```
1> true andalso false.
```

```
false
```



```
2> false orelse true.
```

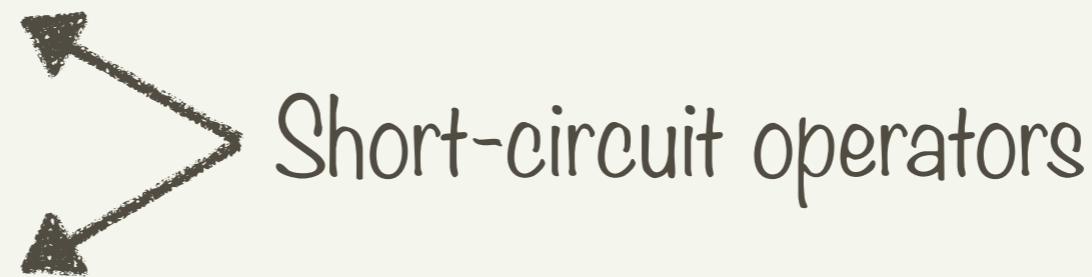
```
true
```

Short-circuit operators

ELIXIR

```
iex> true and true
```

```
true
```



```
iex> false or is_atom(:example)
```

```
true
```

ELIXIR

```
iex> false and raise("This error will  
never be raised")
```

```
false
```

```
iex> true or raise("This error will  
never be raised")
```

```
true
```

A wide-angle photograph of a sunset over a calm sea. The sky is filled with warm orange, yellow, and pink hues, transitioning into darker blues at the top. A large flock of birds is captured in flight across the horizon. In the foreground, dark, foamy waves break onto a sandy beach. The overall atmosphere is serene and peaceful.

CONTROL FLOW

JAVA

```
int x = 1;  
switch(x) {  
    case 1:  
        System.out.println("Will match");  
        break;  
    default:  
        System.out.println("Doesn't match");  
}
```

JAVA

```
int x = 1;  
switch(x) {  
    case 1: ← No extra conditions  
        System.out.println("Will match");  
        break;  
    default:  
        System.out.println("Doesn't match");  
}
```

ELIXIR

```
iex> case {1, 2, 3} do
...>   {4, 5, 6} ->
...>     "This clause won't match"
...>   {1, x, 3} ->
...>     "This clause will match and bind x to 2"
...>   _ ->
...>     "This clause would match any value"
...> end
"This clause will match and bind x to 2"
```

ELIXIR

```
iex> case {1, 2, 3} do
...>   {4, 5, 6} ->
...>     "This clause won't match"
...>   {1, x, 3} -> ← binds x to 2
...>     "This clause will match and bind x to 2"
...>   _ ->
...>     "This clause would match any value"
...> end
"This clause will match and bind x to 2"
```

ELIXIR

```
iex> case {1, 2, 3} do
...>   {4, 5, 6} ->
...>     "This clause won't match"
...>   {1, x, 3} -> ← but x is only valid in this scope
...>     "This clause will match and bind x to 2"
...>   _ ->
...>     "This clause would match any value"
...> end
"This clause will match and bind x to 2"
```

ELIXIR

```
iex> x = 1  
1
```

```
iex> case 10 do  
...>     ^x -> "Won't match"  
...>     _ -> "Will match"  
...> end  
"Will match"
```

Pin operator allows comparison



ELIXIR

```
iex> case {1, 2, 3} do
```

```
...>   {1, x, 3} when x > 0 ->
```

```
...>   "Will match"
```

```
...>   _ ->
```

```
...>   "Won't match"
```

```
...> end
```

```
"Will match"
```



Scoped assignment allows guards

ELIXIR

```
iex> if true do
...>   "This works!"
...> end
"This works!"
```

ELIXIR

```
iex> unless false do
...>   "This works!"
...> end
"This works!"
```

ELIXIR

```
iex> if nil do
...>   "This won't be seen"
...> else
...>   "This will"
...> end
"This will"
```

ELIXIR

```
iex> if nil do
...>   "This won't be seen"
...> else if
...>   "This is an error"
...> else
...>   "This will"
...> end
"This will"
```

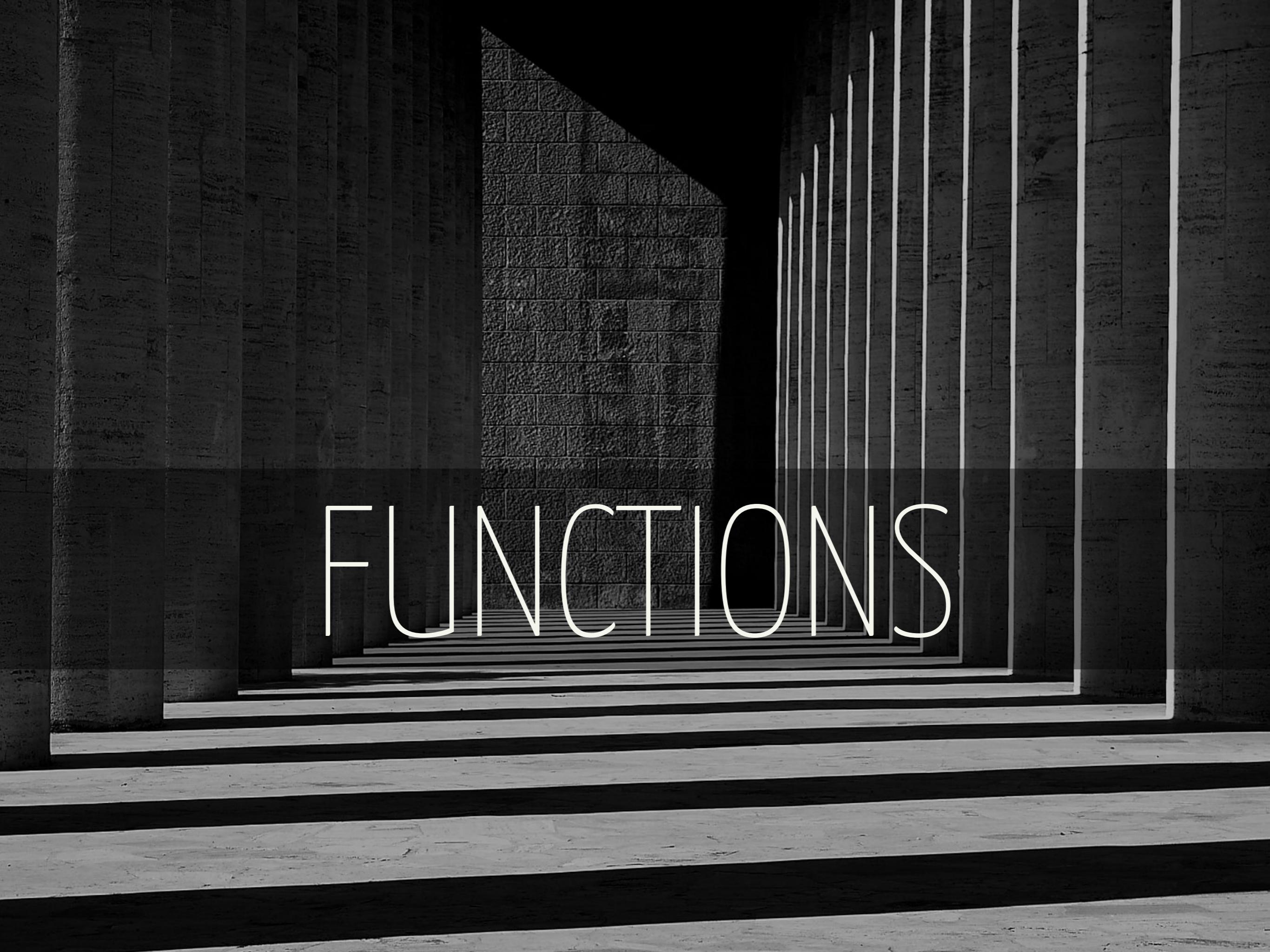
ELIXIR

```
iex> if nil do
...>   "This won't be seen"
...> else if
...>   "This is an error"
...> else
...>   "This will"
...> end
"This will"
```

ELIXIR

```
iex> if(false, [do: :this, else: :that])  
:that
```

```
iex> cond do
...>   2 + 2 == 5 ->
...>     "This is never true"
...>   2 * 2 == 3 ->
...>     "Nor this"
...>   true ->
...>     "This is always true (equivalent
to else)"
...> end
```



A black and white photograph of a modern building's entrance. The entrance consists of a glass door set into a dark brick wall. Above the door, the word "FUNCTIONS" is written in large, white, sans-serif capital letters. The building has a flat roof and is surrounded by a paved area.

FUNCTIONS

ELIXIR



Needs to be inside a module

```
def add(x, y) do  
    x + y  
end
```

ELIXIR

Arguments can do pattern matching



```
def add(x, 0) do
  x
end
```

ELIXIR



And we can use guards

```
def add(x, y) when x == 0 do
  y
end
```

ELIXIR

↙ Anonymous function

```
add2 = fn
```

```
  x, y when x == 0 -> y
```

```
  x, 0 -> x
```

```
  x, y -> x + y
```

```
end
```



Module name

```
iex> Arithmetic.add(1, 3)
```

4

```
iex> add2.(-1, 0)
```

-1



arity identifies functions

ELIXIR

```
def add(x, y) do ← add/2  
  x + y  
end
```

add/1

```
def add(x) do  
  x + 1  
end
```

```
add2 = fn  
  x, y -> x + y  
  x -> 0  
end
```

** (SyntaxError) cannot mix clauses with different arities in function definition



anonymous functions are closures

ELIXIR

```
iex> x = 1
```

```
iex> add_x = fn y -> x + y end
```

```
iex> add_x.(2)
```

3

ELIXIR

```
x = 1
```

```
def add_x(y) do
  x + y
end
```

ELIXIR

```
x = 1
```

```
def add_x(y) do  
  x + y  
end
```

compile error

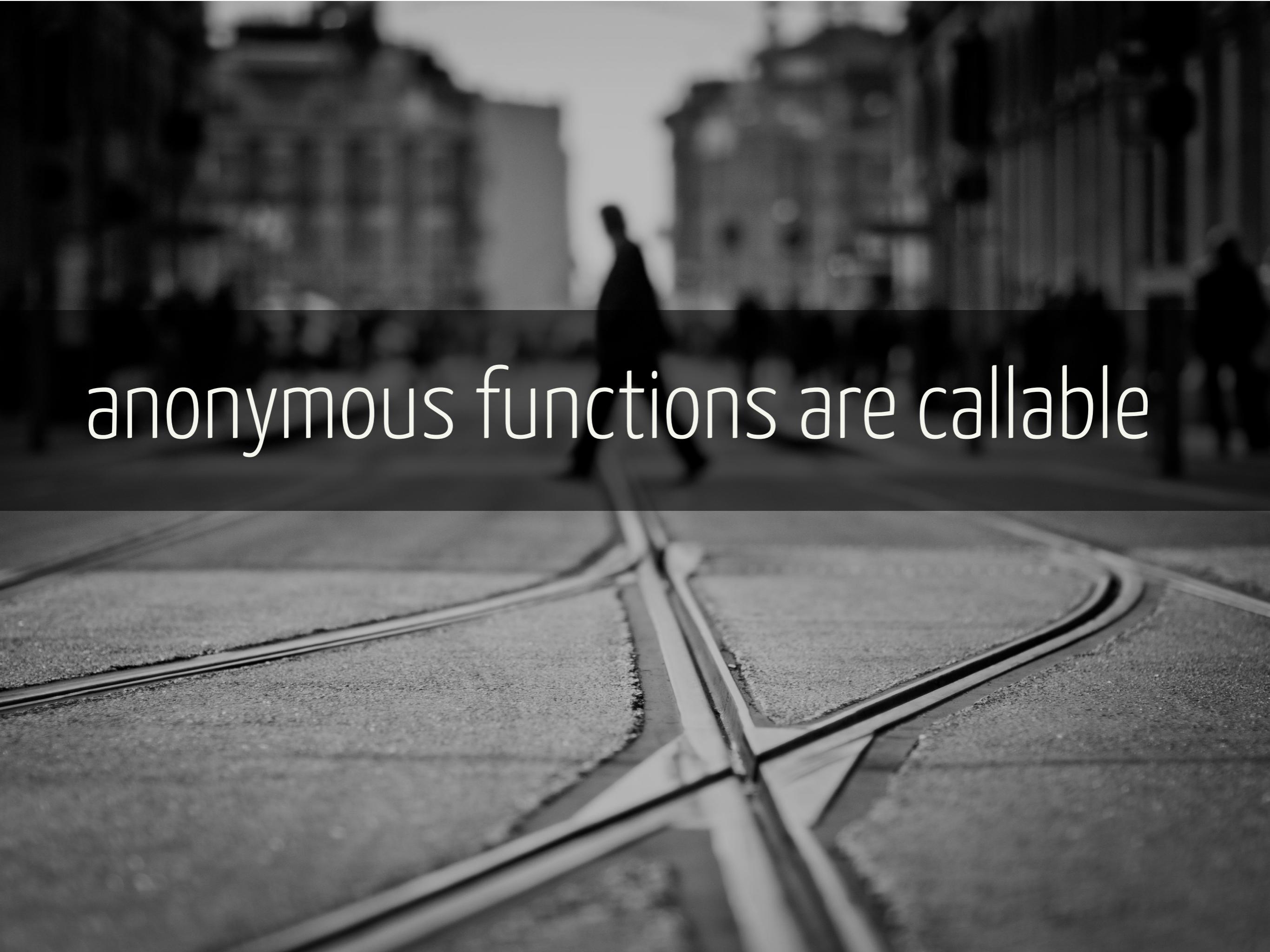
ELIXIR

```
x = 1
```



```
def add_x(y) do  
  x + y  
end
```

each function definition has a blank scope



anonymous functions are callable

ELIXIR

```
iex> add_one = fn x -> x + 1 end  
  
iex> Enum.map([1, 2, 3, 4], add_one)  
[2, 3, 4, 5]
```

ELIXIR

```
iex> defmodule Arith do  
...>   def add_two(x), do: x + 2  
...> end
```

```
iex> Enum.map([1, 2, 3, 4],  
             &Arith.add_two/1)  
[2, 3, 4, 5]
```



capture syntax

A photograph of a winter landscape. A dark, rocky stream flows from the bottom left towards the center of the frame. The banks of the stream are covered in white snow and several large, fallen tree trunks. The background is filled with tall evergreen trees, their branches heavily laden with snow. The overall atmosphere is cold and serene.

ENUMS & STREAMS

ELIXIR

```
iex> add_one = fn x -> x + 1 end
```

Enum module


```
iex> Enum.map([1, 2, 3, 4], add_one)  
[2, 3, 4, 5]
```

ELIXIR

```
iex> 0..99 |> Enum.map(&(&1 + 1))  
|> Enum.filter(&Integer.is_odd/1)  
|> Enum.sum
```

ELIXIR



pipe operator - similar to unix pipe

```
iex> 0..99 |> Enum.map(&(&1 + 1))  
|> Enum.filter(&Integer.is_odd/1)  
|> Enum.sum
```

```
iex> 0..99 |> Enum.map(&(&1 + 1))  
|> Enum.filter(&Integer.is_odd/1)  
|> Enum.sum
```



capturing a function

ELIXIR

first argument passed to the function



```
iex> 0..99 |> Enum.map(&(&1 + 1))  
|> Enum.filter(&Integer.is_odd/1)  
|> Enum.sum
```

ELIXIR

```
iex> 0..99 |> Enum.map(&(&1 + 1))  
|> Enum.filter(&Integer.is_odd/1)  
|> Enum.sum
```



capture module function

```
iex> 0..99 |> Enum.map(&(&1 + 1))  
|> Enum.filter(&Integer.is_odd/1)  
|> Enum.sum
```



Enum is eager: every operation generates an intermediate list

```
iex> 0..99 |> Stream.map(&(&1 + 1))  
|> Stream.filter(&Integer.is_odd/1)  
|> Enum.sum
```



Stream is lazy: every operation returns another stream

ELIXIR

```
iex> 0..99 |> Stream.map(&(&1 + 1))  
|> Stream.filter(&Integer.is_odd/1)  
|> Enum.sum
```



Enum is used to make the calculation

ELIXIR

```
iex> 0..99 |> Stream.map(&(&1 + 1))  
|> Stream.filter(&Integer.is_odd/1)  
|> Enum.sum
```



Enum is used to make the calculation

```
iex> 0..99 |> Stream.map(&(&1 + 1))  
|> Stream.filter(&Integer.is_odd/1)  
|> Enum.sum
```

Stream only carries a set of operations
to make somewhere in the future

ELIXIR

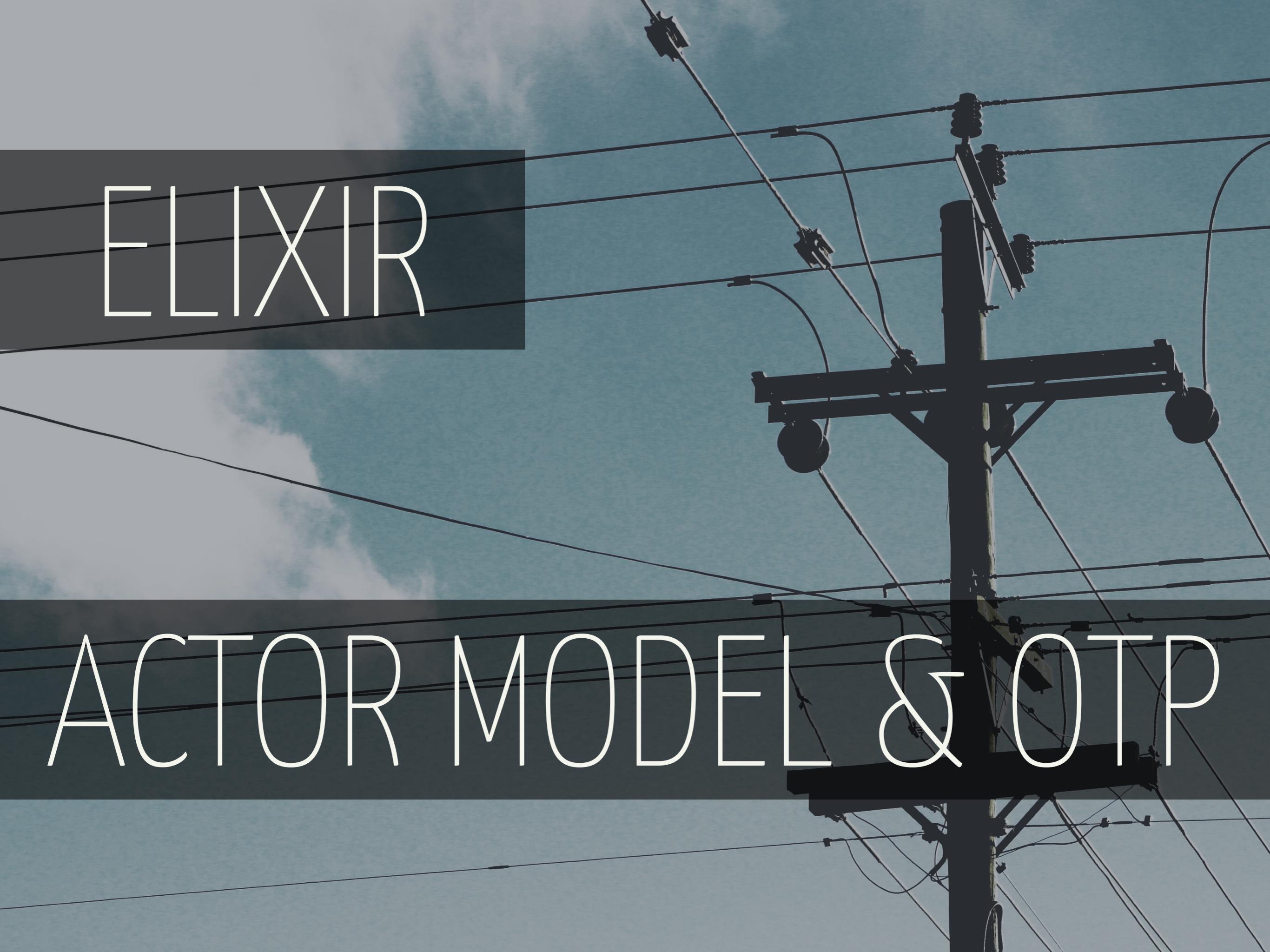
iex> stream = Stream.cycle([1, 2, 3])

infinite set



iex> Enum.take(stream, 10)

[1, 2, 3, 1, 2, 3, 1, 2, 3, 1]



The background image shows a utility pole with several black power lines against a light blue sky with white clouds.

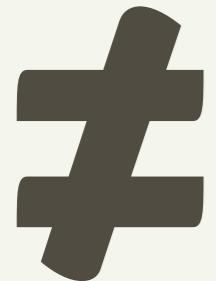
ELIXIR

ACTOR MODELS & OTP



PROCESSES

ELIXIR PROCESSES



OS PROCESSES

“An actor is an entity that receives messages and, according to those messages, can send more messages, create a finite number of actors and choose behaviour for processing the next message”

- Carl Hewitt, 1973

Actors are **processes**.

Processes act upon **messages**.

Each process has a **mailbox**.

Messages are **asynchronous** and there are no guarantees regarding delivery.

Each process has its **own state**, which is not shared.

```
parent = self

spawn fn -> send(parent, {:hello, self}) end

receive do
  {:hello, pid} -> IO.puts "Hello #{inspect pid}"
end
```

what if...

no one is listening?

Actors are **processes**.

Processes act upon **messages**.

Each process has a **mailbox**.

Messages are **asynchronous** and there are no guarantees regarding delivery.

Each process has its **own state**, which is not shared.

what if...

we received an unexpected message?

**everything
is immutable**

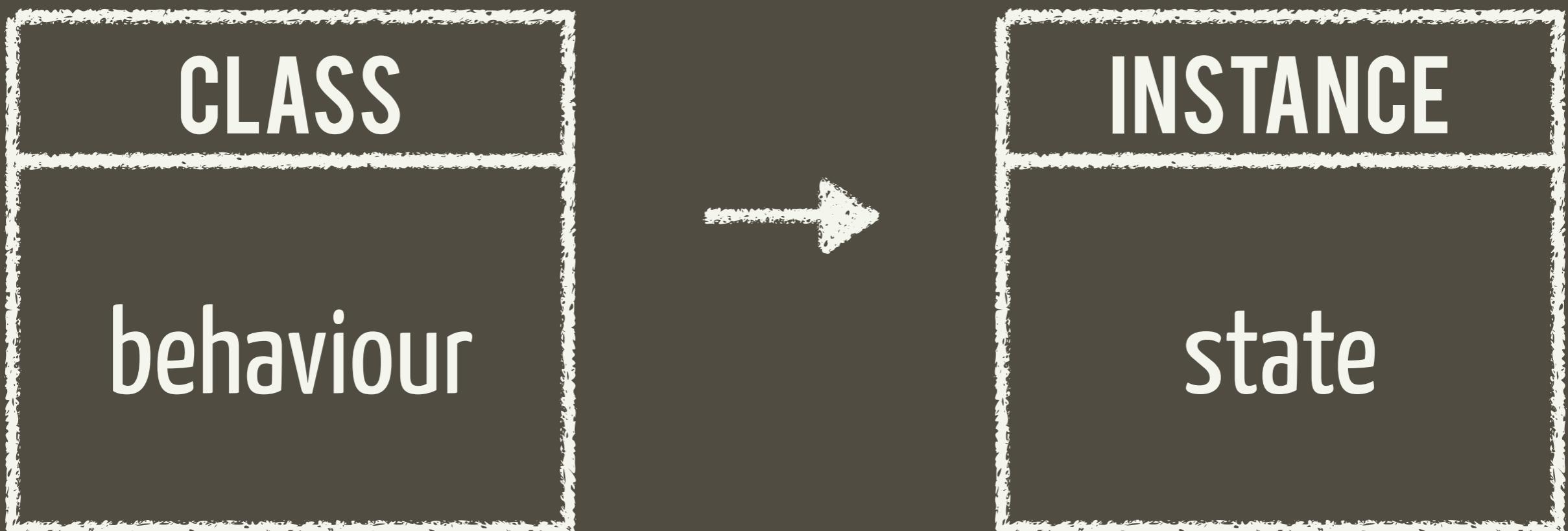
FUNCTIONAL PARADIGM™

immutable

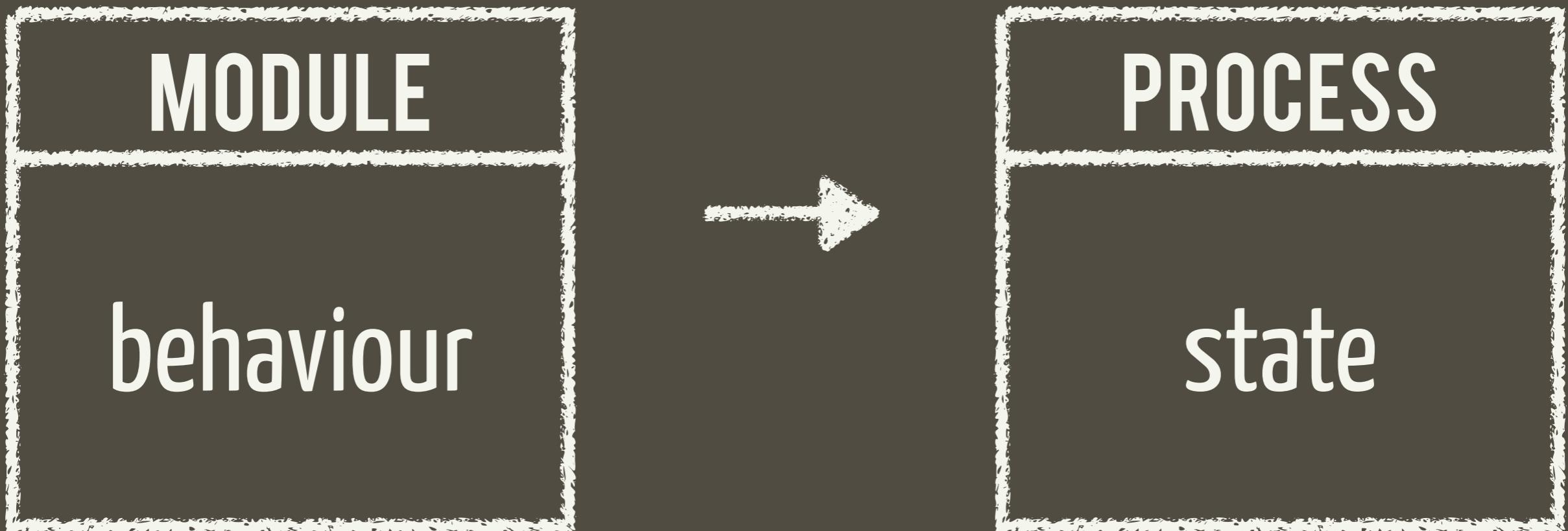
stateless

FUNCTIONAL PARADIGM™

OOP



FP



A SAMPLE QUEUE

```
defmodule Queue do
  use GenServer

  def new do
    GenServer.start_link(Queue, [], [])
  end
end
```

```
def handle_cast({:put, value}, state) do
  {:noreply, state ++ [value]}
end
```

Server callbacks

```
def handle_call(:poll, _from, []) do
  {:reply, nil, []}
end

def handle_call(:poll, _from, state) do
  [head|tail] = state
  {:reply, head, tail}
end
```

Server callbacks

```
iex> {:ok, q} = Queue.new
```

```
iex> GenServer.cast(q, {:put, 1})
```

```
iex> GenServer.call(q, :poll)
```

```
1
```

```
def put(queue, value) do
  GenServer.cast(queue, {:put, value})
end

def poll(queue) do
  GenServer.call(queue, :poll)
end
```

Client callbacks

```
iex> {:ok, q} = Queue.new
```

```
iex> Queue.put(q, 1)
```

```
iex> Queue.poll(q)
```

```
1
```

TASK



“Tasks are processes meant to execute one particular action throughout their life-cycle, often with little or no communication with other processes.”

- Elixir Docs, 2015

A photograph of a street at night. In the foreground, a stop sign is mounted on a pole. Above it is a rectangular sign with a white arrow pointing right and the words "ONE WAY" in black. Below the stop sign is a smaller rectangular sign that reads "3-WAY". The background shows a two-story brick building with several lit windows. Bare trees are visible against a dark sky.

FAILURES

“The greatest advantage actor-based systems give us is accepting programmers make mistakes all the time and not all scenarios are testable.”

- Me, right now

fail

fast

let it
crash

```
try do
    raise "oops"
rescue
    e in RuntimeError -> e
end
```

```
try do
    raise "oops"
rescue
    e in RuntimeError -> e
end
```

don't care about failures

```
try do
    raise "oops"
rescue
    e in RuntimeError -> e
end
```

use Supervisors instead

```
try do
    raise "oops"
rescue
    e in RuntimeError -> e
end
```

keep your code clean

```
try do
    raise "oops"
rescue
    e in RuntimeError -> e
end
```

keep your flow clean

A SAMPLE CHAT



```
def listen(port) do
  {:ok, socket} = :gen_tcp.listen(port,
    [:list, packet: :line,
     active: false,
     reuseaddr: true])
  acceptor(socket)
end
```

Server Module

```
defp acceptor(socket) do
  {:ok, client} = :gen_tcp.accept(socket)
  handle_client(client)
  acceptor(socket)
end
```

Server Module

```
defp handle_client(client) do
  client |> read() |> write(client)

  handle_client(client)
end
```

Server Module

```
defp read(socket) do
  # 2nd param is byte length
  # 0 means receive all available bytes
  {:ok, data} = :gen_tcp.recv(socket, 0)
  data
end
```

Server Module

```
defp write(msg, socket) do
  :gen_tcp.send(socket, msg)
end
```

Server Module

what if...

we want to echo multiple clients?

```
def start do
  import Supervisor.Spec

  children = [
    supervisor(Task.Supervisor, [[name: Server.TaskSupervisor]]),
    worker(Task, [Server, :listen, [3000]])
  ]

  opts = [strategy: :one_for_one, name: ServerSupervisor]
  Supervisor.start_link(children, opts)
end
```

Server Module

```
defp acceptor(socket) do
  {:ok, client} = :gen_tcp.accept(socket)
  handle_client(client)
  acceptor(socket)
end
```

Old Server Module

```
defp acceptor(socket) do
  {:ok, client} = :gen_tcp.accept(socket)
  {:ok, pid} = Task.Supervisor.start_child(
    Server.TaskSupervisor, fn -> handle_client(client) end)

  # transferring the socket control to the new child
  :ok = :gen_tcp.controlling_process(client, pid)
  acceptor(socket)
end
```

New Server Module

ELIXIR

a talk for college students

@frmendes
@fribmendes