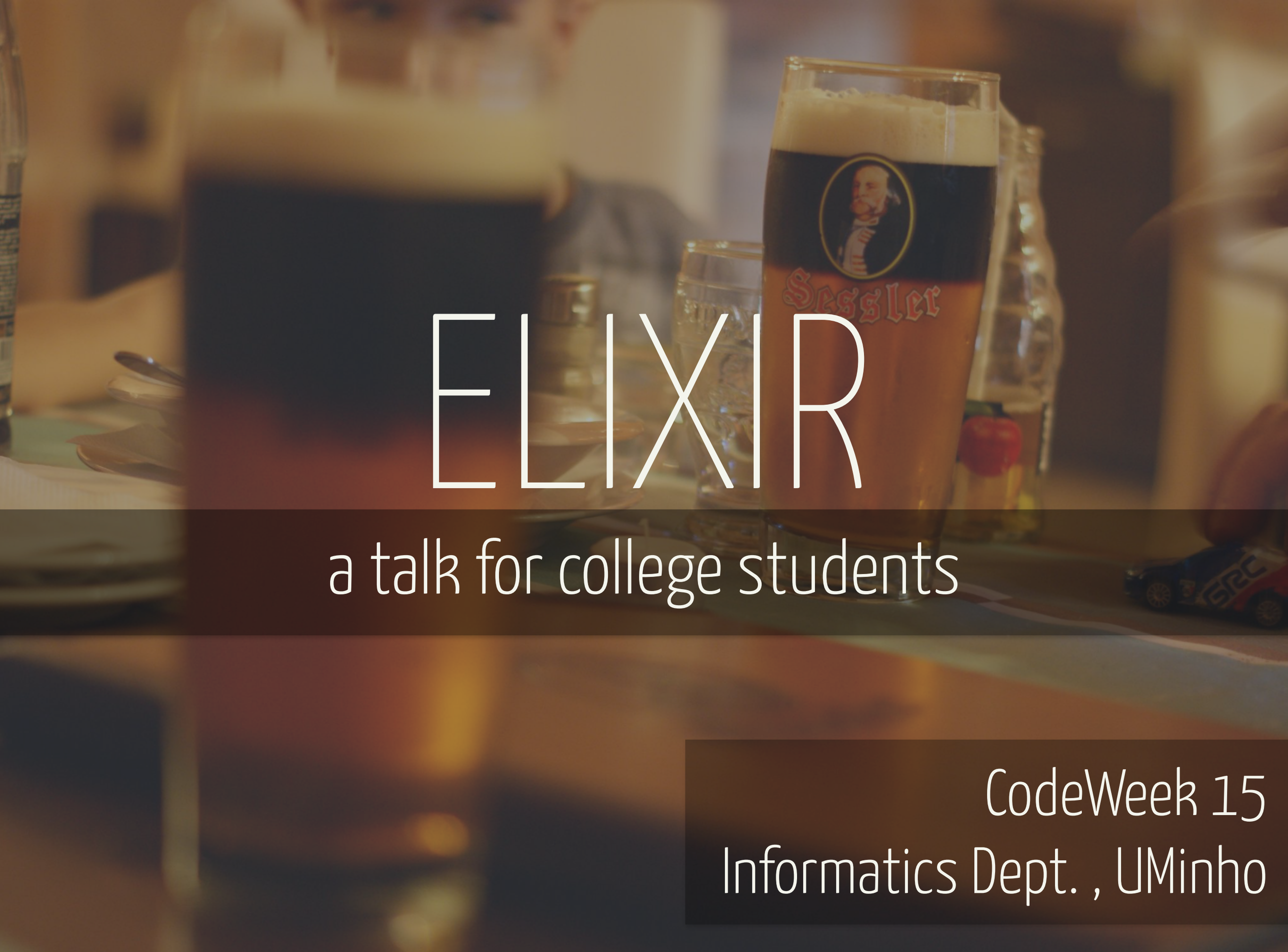


ELIXIR

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 MI IS

ON A S

WITH R

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

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

YAHOO!

facebook

MOTOROLA

heroku

WhatsApp

ERICSSON

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
CESIUM
RVCA


lol, not my cat
love me, please


ELIXIR
everything erlang has to offer

ELIXIR

beautiful syntax

ELIXIR

powerful libraries

ELIXIR

metaprogramming \m/

ELIXIR

awesome DSLs

ELIXIR

mix ecosystem


ELIXIR
AS A 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
{:ok, “hello”}
```

immutability

aka “lolno.”

PATTERN MATCHING

ELIXIR

```
iex> x = 1
1


iex> x
1


iex> 1 = x
1
```

ELIXIR

```
iex> x = 1
1

iex> x
1

iex> 1 = x
1
```

x = 1?

what is 1?

1 is 1

x is 1

ELIXIR

```
iex> x = 1
1


iex> x
1


iex> 1 = x
1
```

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

ELIXIR

```
iex> x = 1
1


iex> 2 = x
** (MatchError) no match of right hand side value: 1


iex> 1 = y
** (RuntimeError) undefined function: y/0
```

What is 1?

ELIXIR

```
iex> x = 1
1


iex> 2 = x
** (MatchError) no match of right hand side value: 1


iex> 1 = y
** (RuntimeError) undefined function: y/0
```

I don't know y so y can't be 1

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

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

JAVA

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

RUBY

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

JAVA

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

RUBY

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

HASKELL

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

JAVA

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

RUBY

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

HASKELL

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

ELIXIR

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

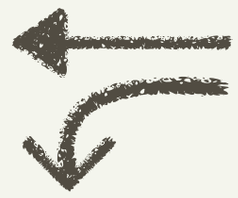
ELIXIR

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

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

ELIXIR

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




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

STRINGS

UTF-8 STRINGS

UTF-8 STRINGS

“😸😂😺😻”.length

?

“😸😹😺😻”.size

?

“😸😹😺😻”.size

4 bytes each emoji

```
“baffle”.length
```

?

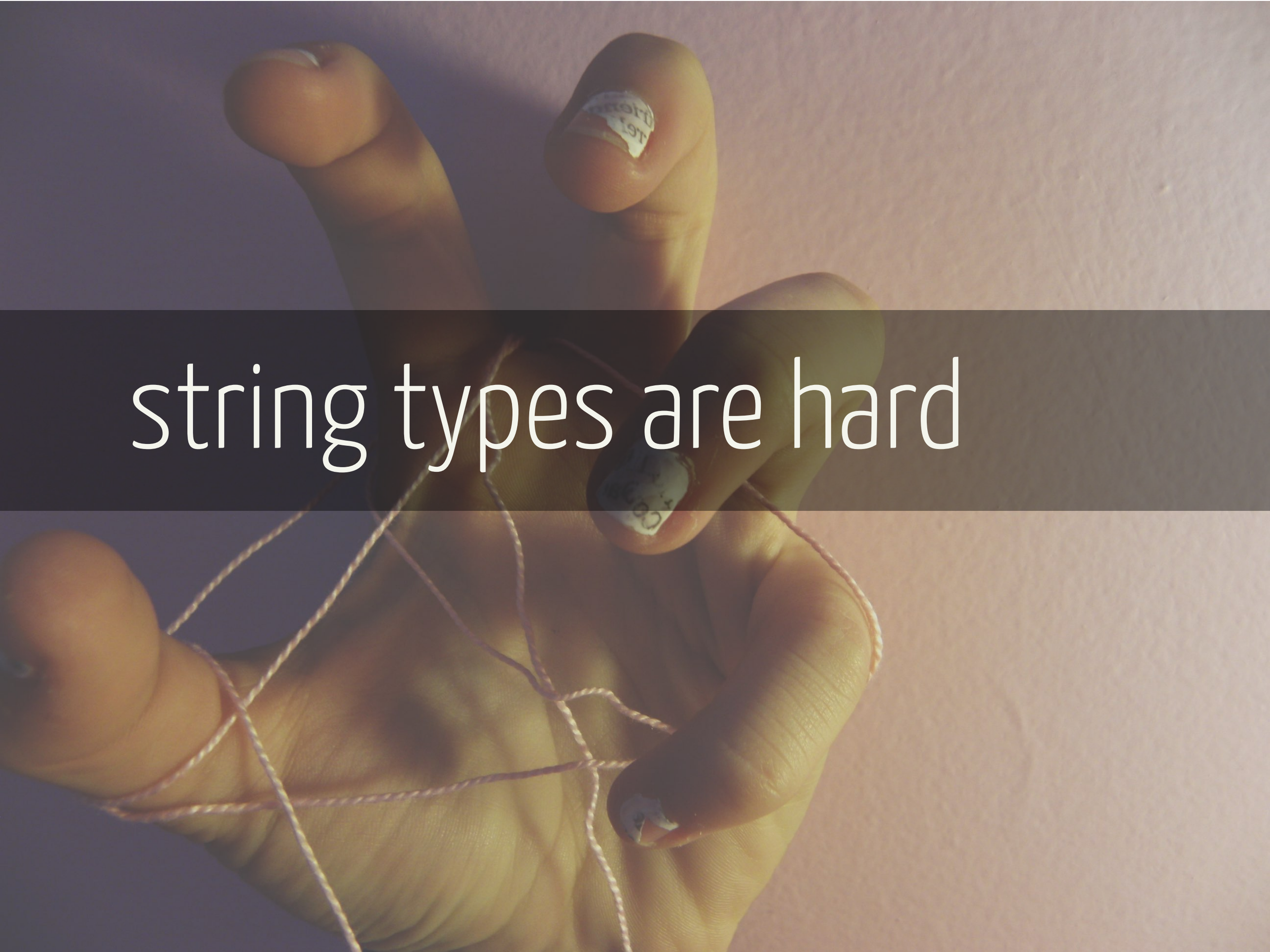
“baffle”.size

?

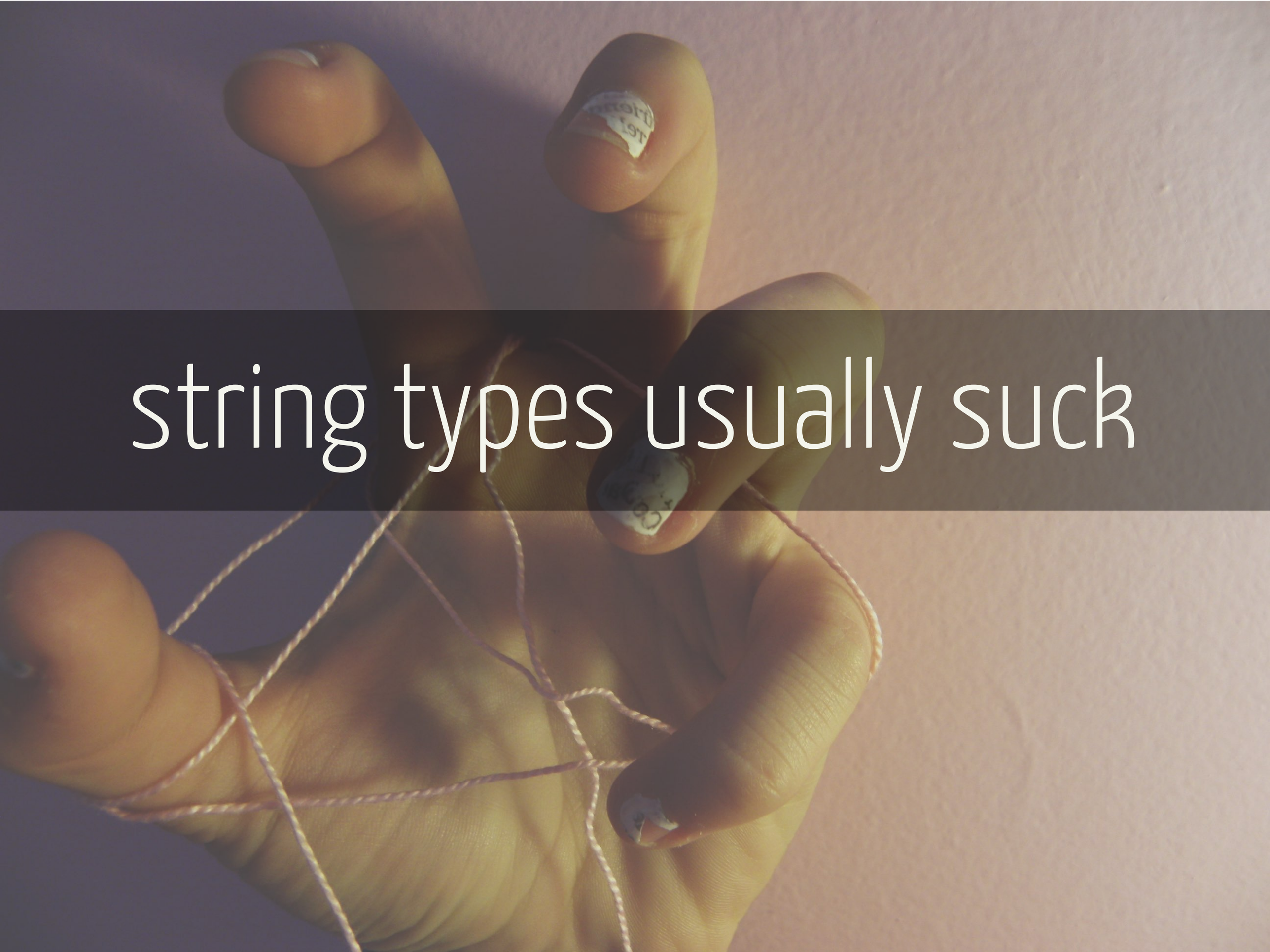
“baffle”.size

length 4

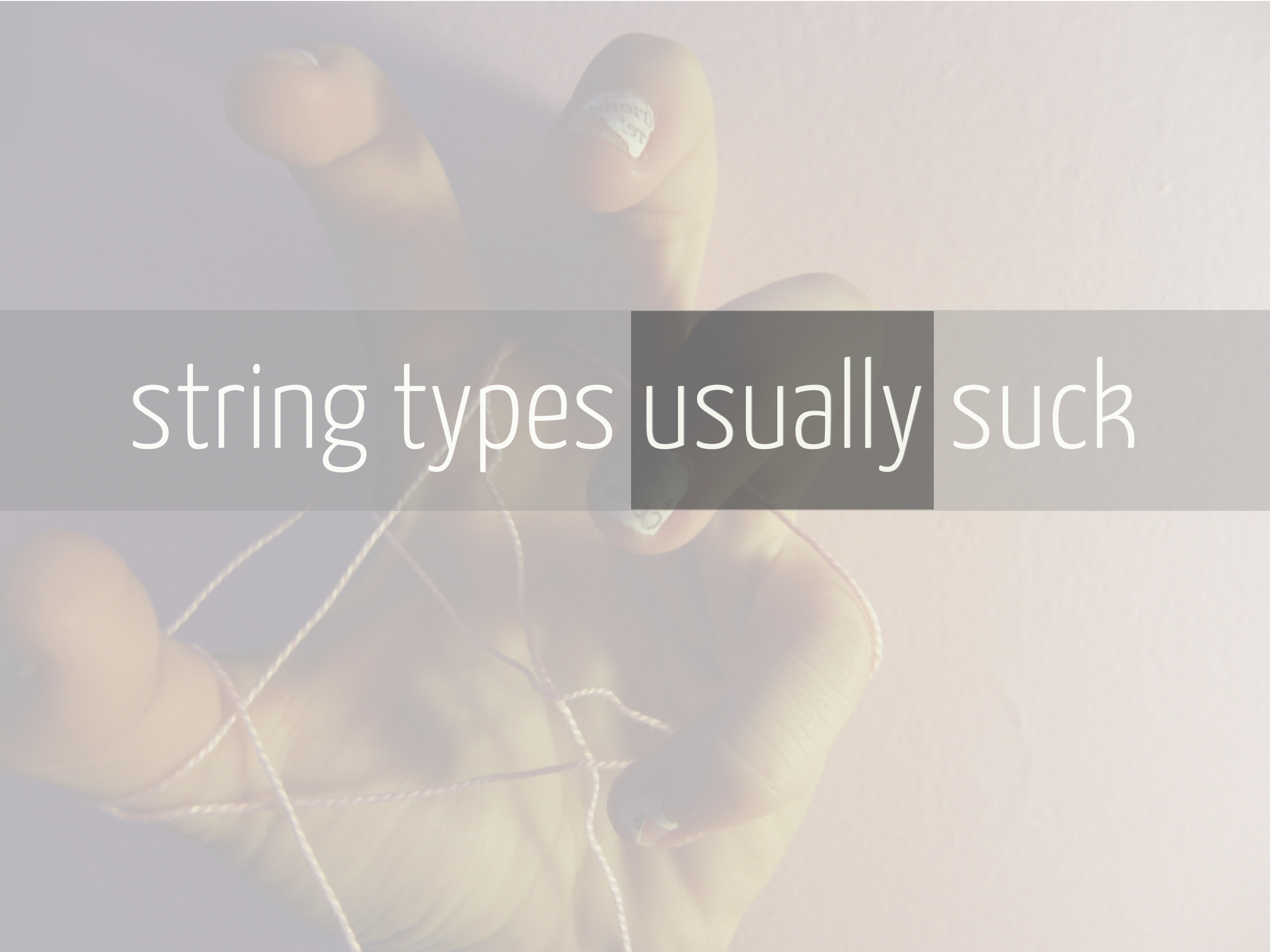
size 6



string types are hard



string types usually suck



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”, “ffl”, “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"

iex> rest
"llo"
```

Concatenation operator for pattern matching

BINARIES

ELIXIR

```
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
```

ELIXIR

```
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

iex> byte_size(<< 1 :: size(9)>>)
2
```

KEYWORD LISTS

ELIXIR

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

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

iex> list[:a]
1
```

special keyword list syntax

ELIXIR

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

Allows repeated keys

```
iex> new_list[:a]
0
```

ELIXIR

```
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

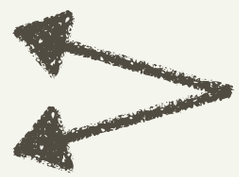
```java
Map<String, Integer> map = new HashMap<>();

map.put(“a”, 1);

map.put(“b”, 2);
```

JAVA


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

map.put(“a”, 1);

map.put(“b”, 2);
```

JAVA

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

map.put(“a”, 1);

map.put(“b”, 2);
```



Auto-boxing

RUBY

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

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

ELIXIR

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

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

ELIXIR

```
iex> other_map[2]
:b
```

Elegant pattern matching

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

iex> a
1
```

ELIXIR

Elegant pattern matching failing

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

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

No error

LOGICAL OPERATORS

ERLANG

```
1> true and false.
false

2> false or true.
true

3> not (true and true)
false
```

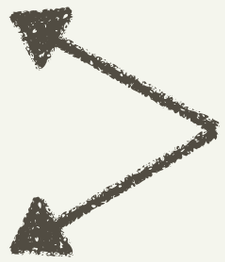
ERLANG

```
1> true and false.
false

2> false or true.
true

3> not (true and true)
false
```

Always evaluate on each side

ERLANG

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

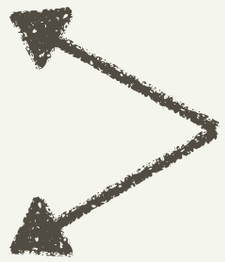


Short-circuit operators

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

ELIXIR

```
iex> true and true
true
```



Short-circuit operators

```
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
```

CONTROL FLOW

JAVA

```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:
    System.out.println(“Will match”);
    break;
  default:
    System.out.println(“Doesn’t match”);
}
```

No extra conditions

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} ->
...>     "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} ->
...>     "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"
```

but x is only valid in this scope

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
```

ELIXIR

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

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
```

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
```

ELIXIR


```
iex> Arithmetic.add(1, 3)
4


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

arity identifies functions

ELIXIR

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

add/2

add/1

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

ELIXIR

```
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
```

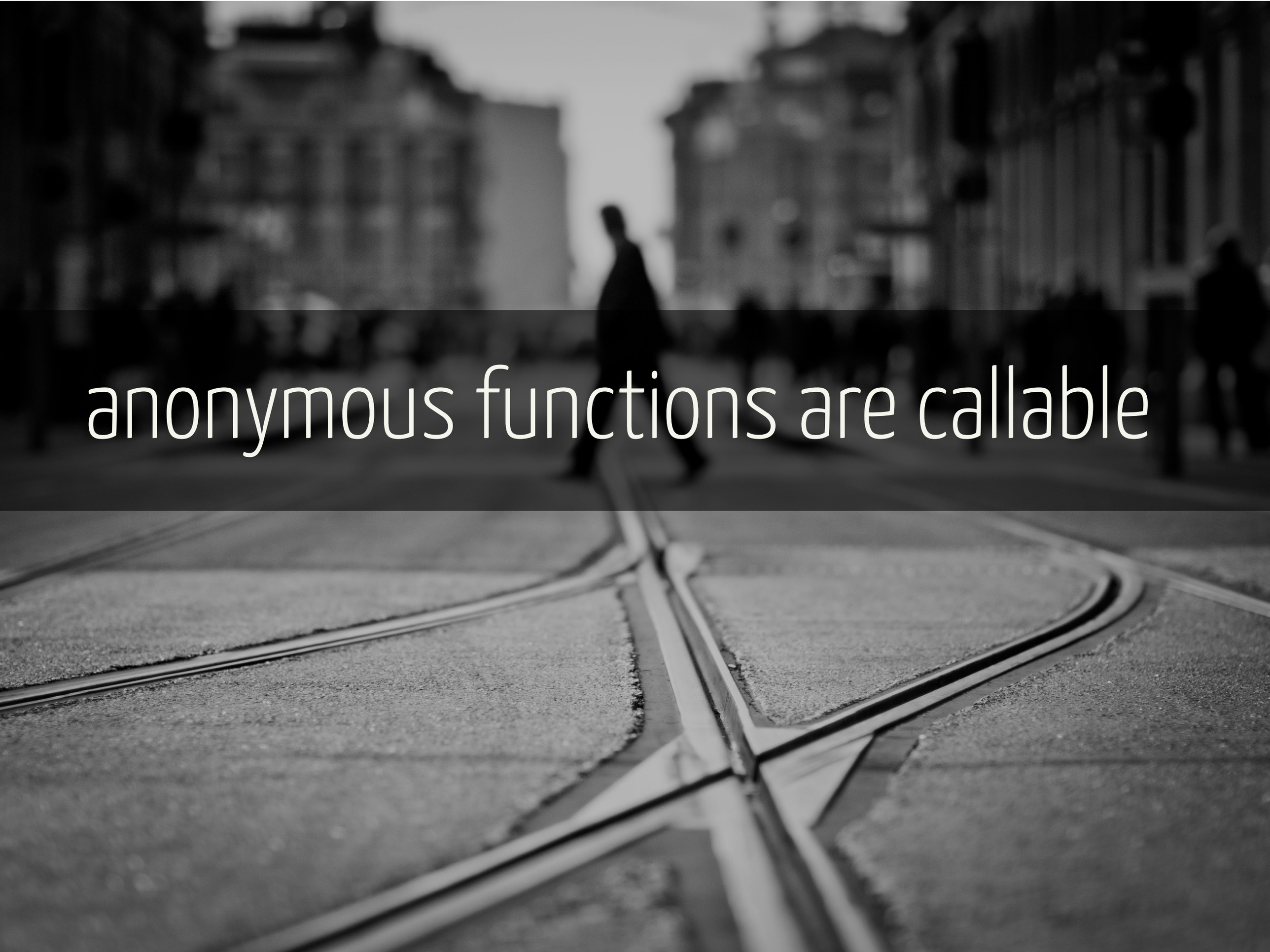
compile error

ELIXIR

```
x = 1
```

each function definition has a blank scope

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



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
```

capture syntax

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

ENUMS & STREAMS

ELIXIR

```
iex> add_one = fn x -> x + 1 end
#Function<6.90072148/1 in :erl_eval.expr/5>
```

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
```

ELIXIR

capturing a function

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

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

ELIXIR

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

Enum is eager: every operation generates an intermediate list

ELIXIR

```
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

ELIXIR

```
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

infinite set

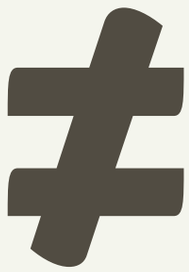
```
iex> stream = Stream.cycle([1, 2, 3])

iex> Enum.take(stream, 10)
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1]
```

ELIXIR

ACTOR MODEL & 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.

```elixir
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

CLASS	INSTANCE
behaviour	state

FP

MODULE	PROCESS
behaviour	state

A SAMPLE QUEUE

```elixir
defmodule Queue do
  use GenServer

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

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

Server callbacks

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

```elixir
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
```

```elixir
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

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

```elixir
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

```elixir
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

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

Server Module

what if...

we want to echo multiple clients?

```elixir
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

```elixir
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