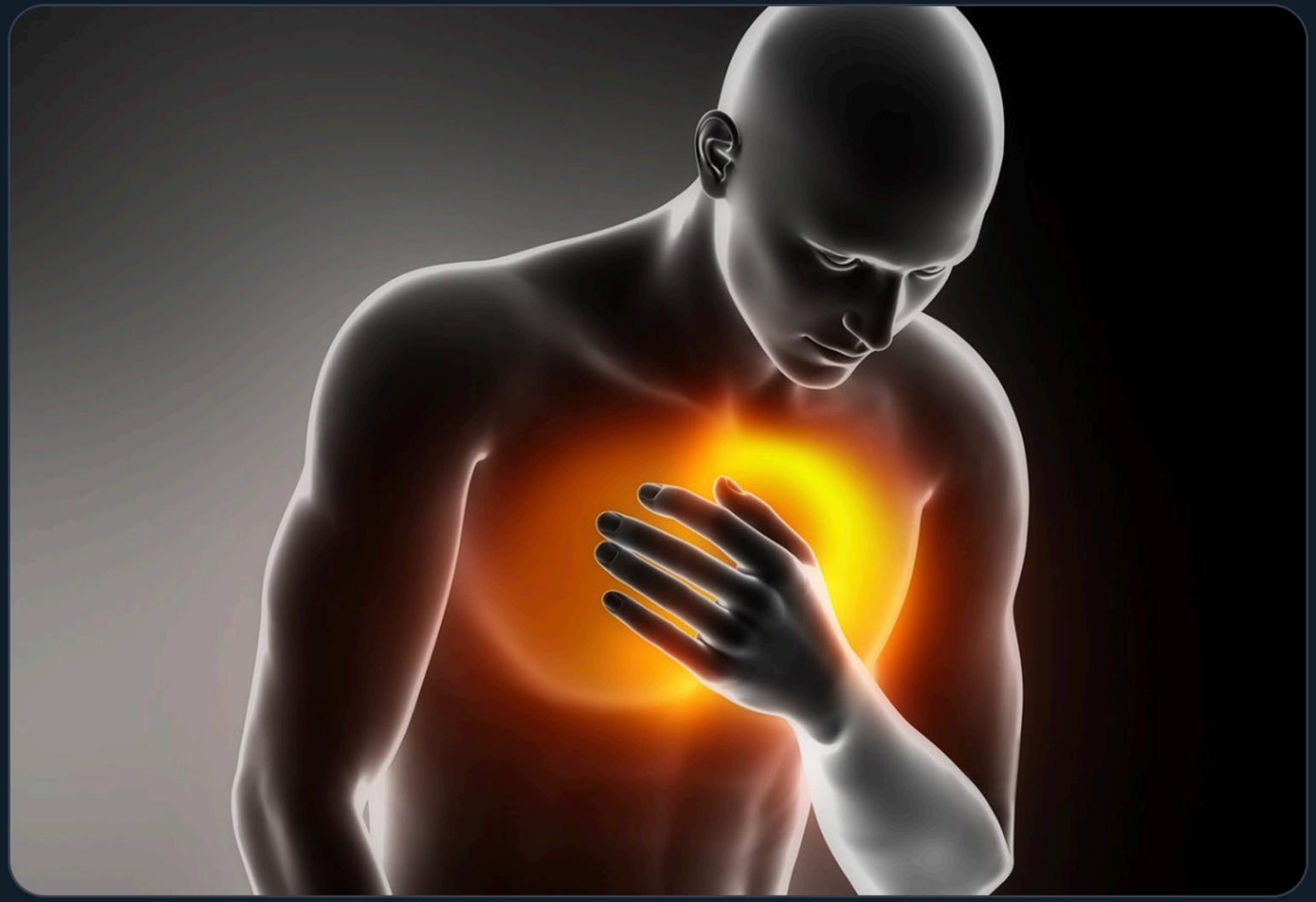
Dillon Mulroy λ
@dillon_mulroy

when f1 qualifying overlaps with your speaking slot at @CodesCarolina



Lets talk about programming languages

But first, who am I?

Dillon Mulroy

Software Engineer @ Vercel ▲

- twitch.tv/dmmulroy
- x.com/dillon_mulroy
- github.com/dmmulroy
- dillonis.online

https://github.com/dmmulroy/carolina_code_conf_24

Okay, back to programming languages

We’ve never had it better
or had more choices

New languages are taking the best features and ideas from those that came before them

And adding features & capabilities from cutting edge PhD PL research

Older languages are evolving and modernizing to add these capabilities as well

What are some of these features & capabilities?

- Linear Types
- Algebraic Effects
- Gradual Type Systems
- Algebraic Data Types (Specifically Sum Types)
- Pattern Matching
- Immutable/Persistent Data Structures
- Actor Based Concurrency Models

We’ve never had it better
or had more ***choices***

We’ve never had it better
or had more ***choices***

We’ve never had it better
or had more ***choices***

Choices

So.

So. Many.

So. Many. Choices.

TS

TS

GO

TS

GO

Java

TS

GO

Java

TS

GO

Java

TS

GO

php

Java

TS

GO

php

Java

TS
GO
R
php
Java

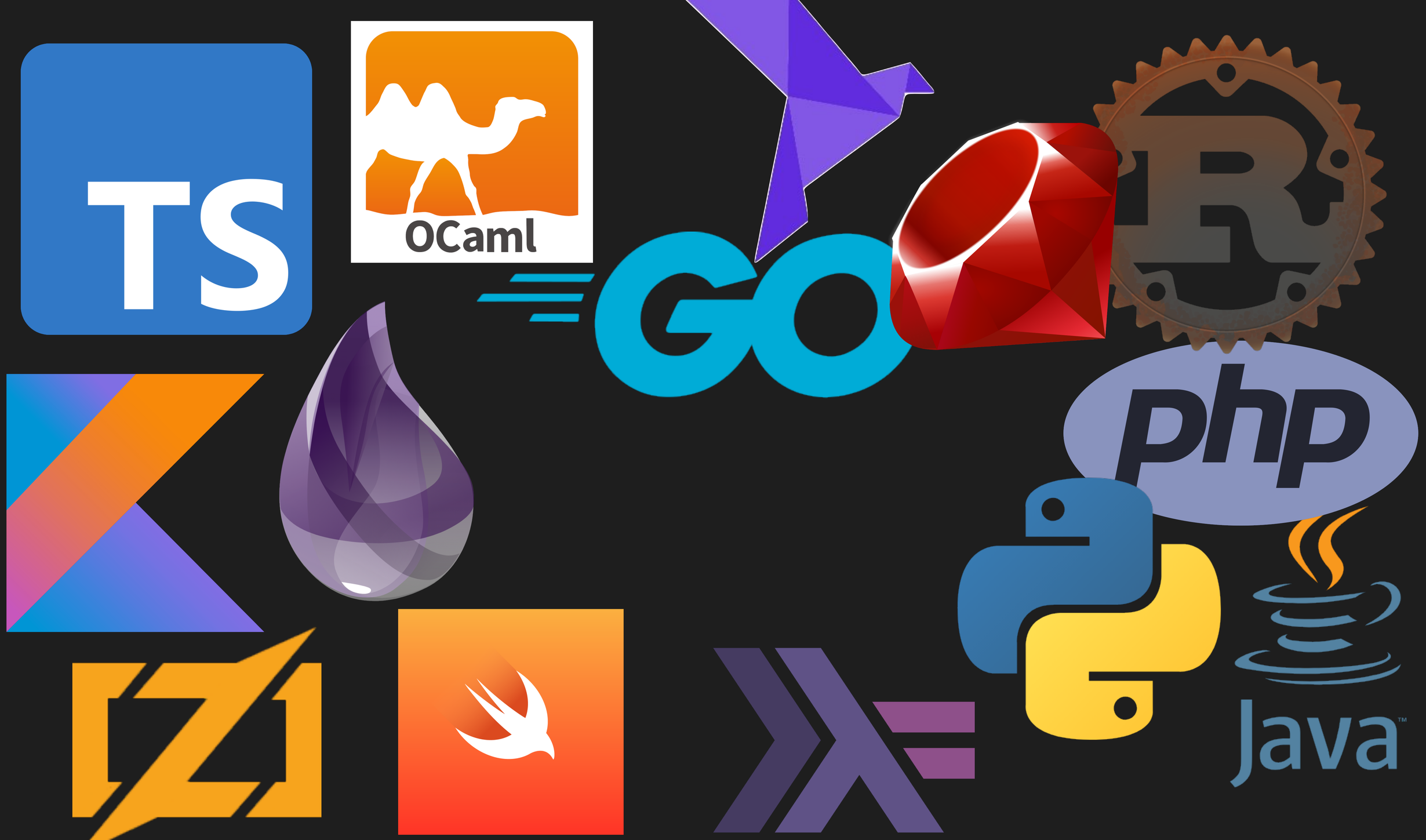
TS
GO
R
php
Java

TS
GO
R
php
Java

TS
GO
R
php
Java

TS
GO
R
php
Java

TS
GO
R
php
Java


TS
OCaml
GO
R
php
Java

TS
OCaml
GO
R
php
Java

TS
OCaml
Go
R
C
php
Java

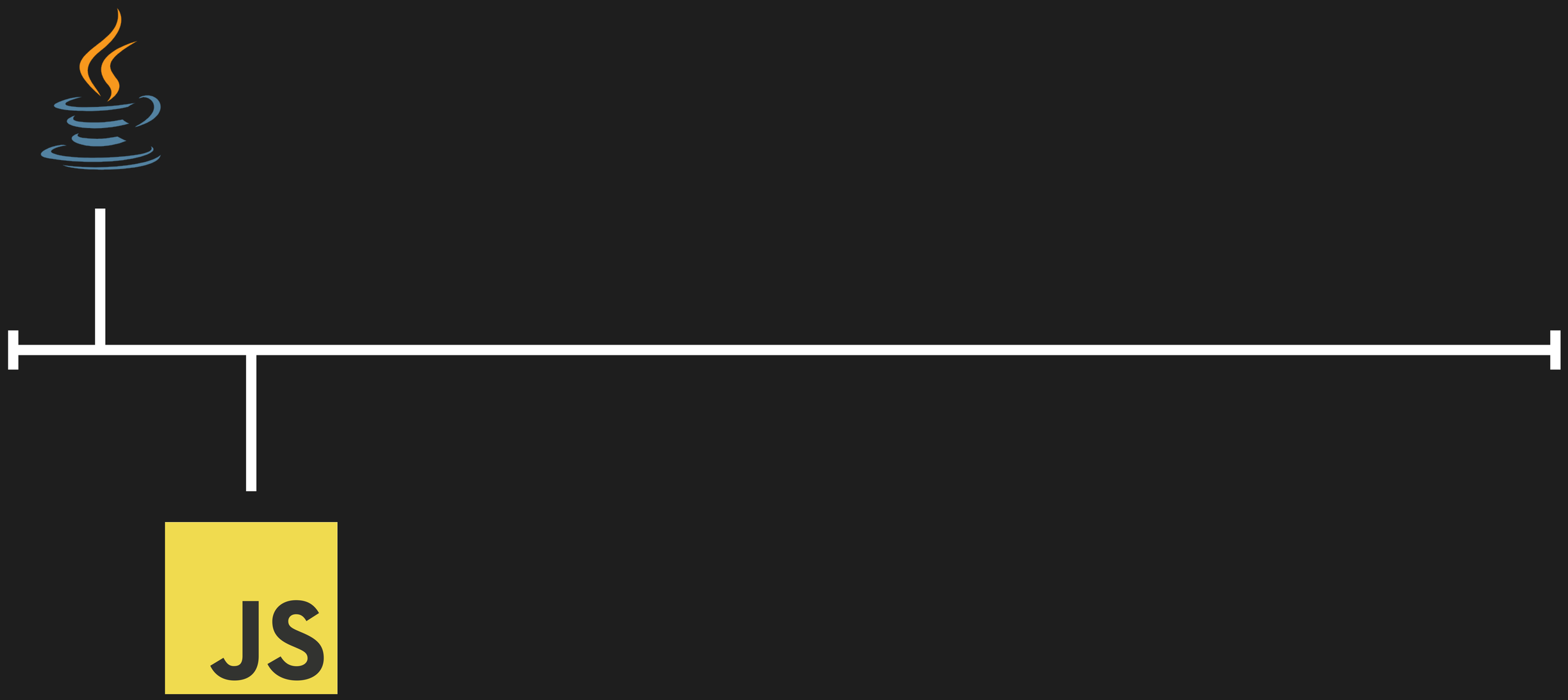
TS
OCaml
C++
R
php
Java

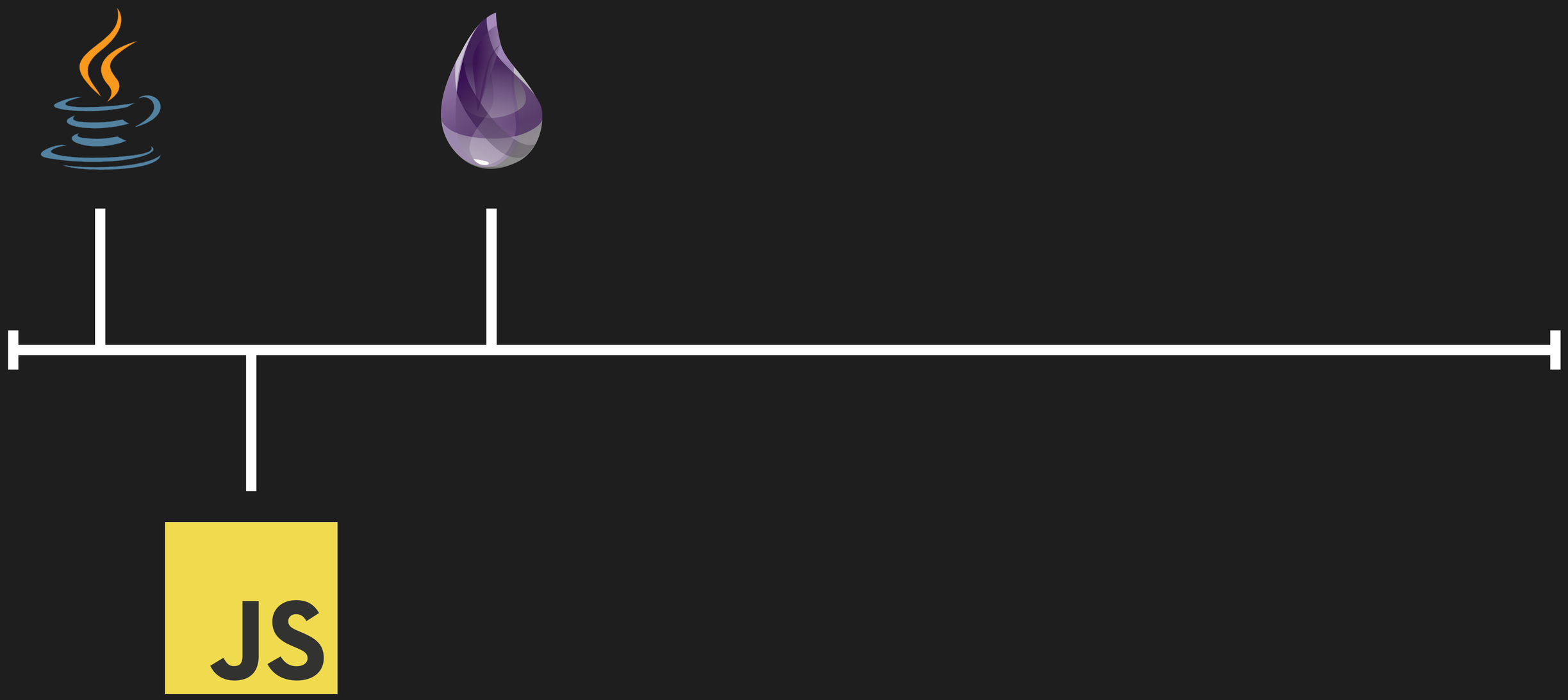
TS
OCaml
C#
R
php
Java

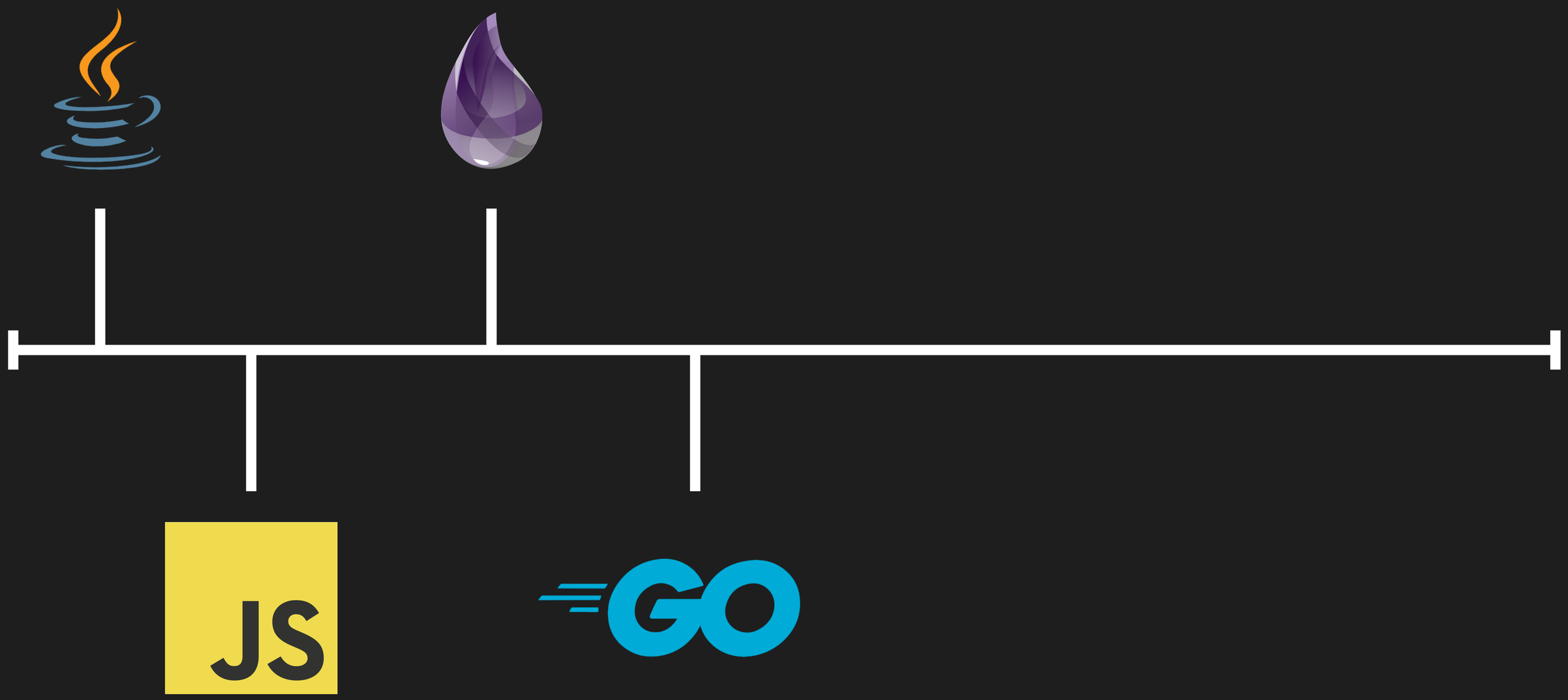
TS
OCaml
R
C#
php
Java

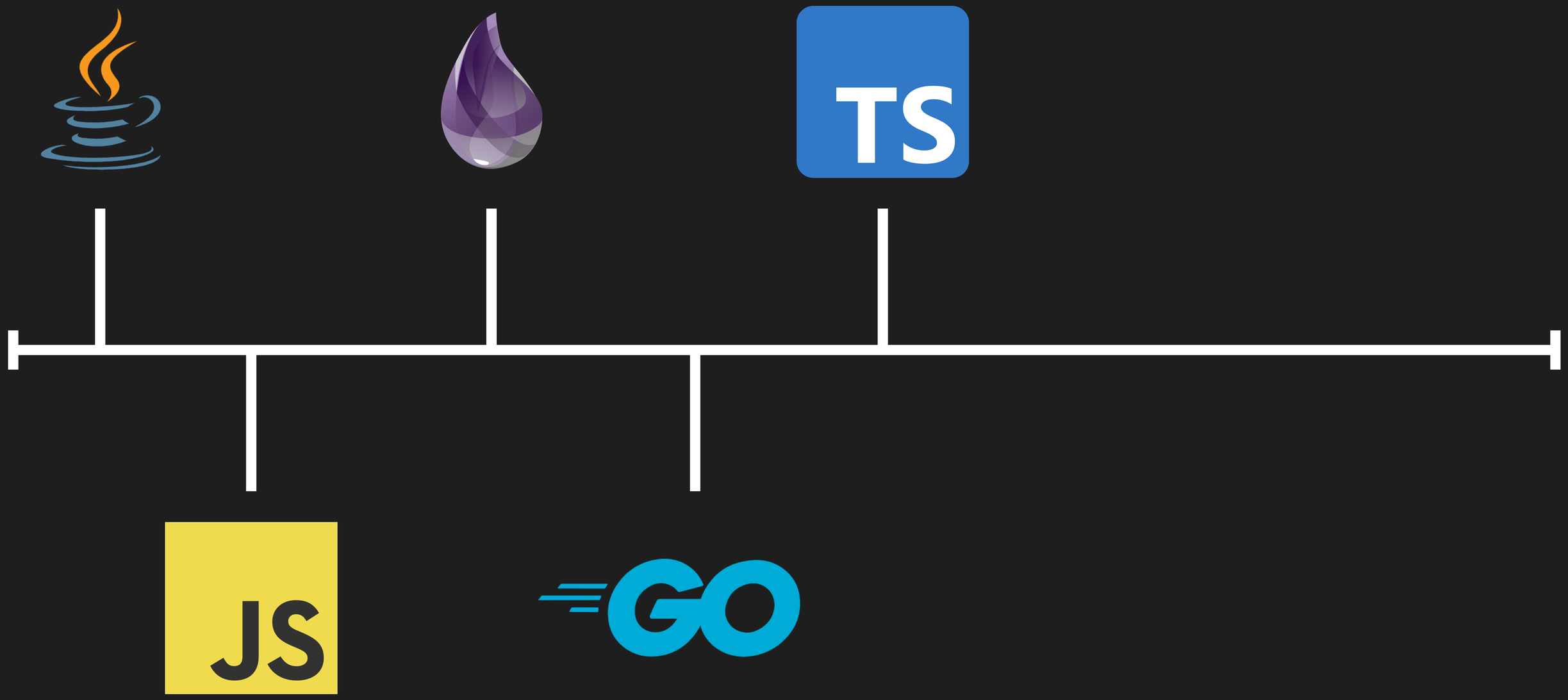
My Language Journey

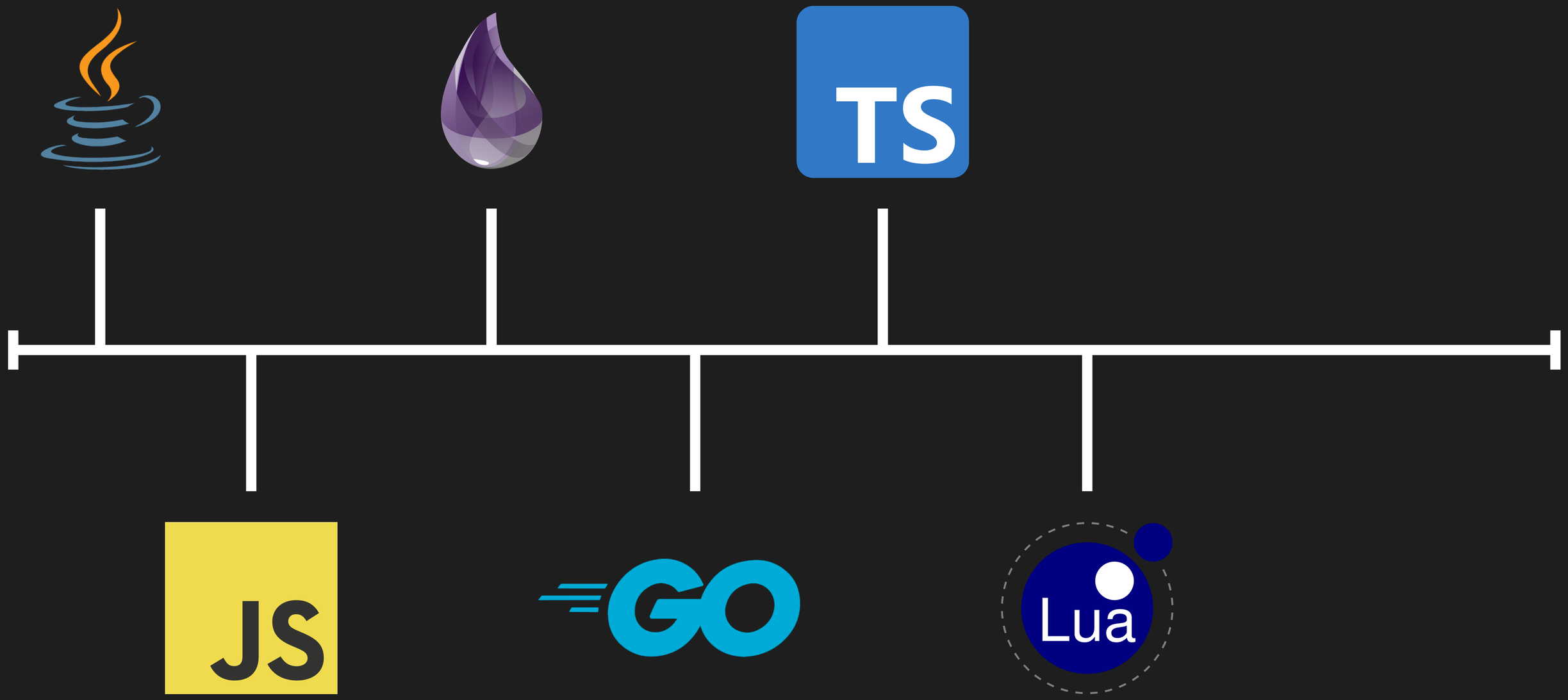
A timeline & my takeaway(s)

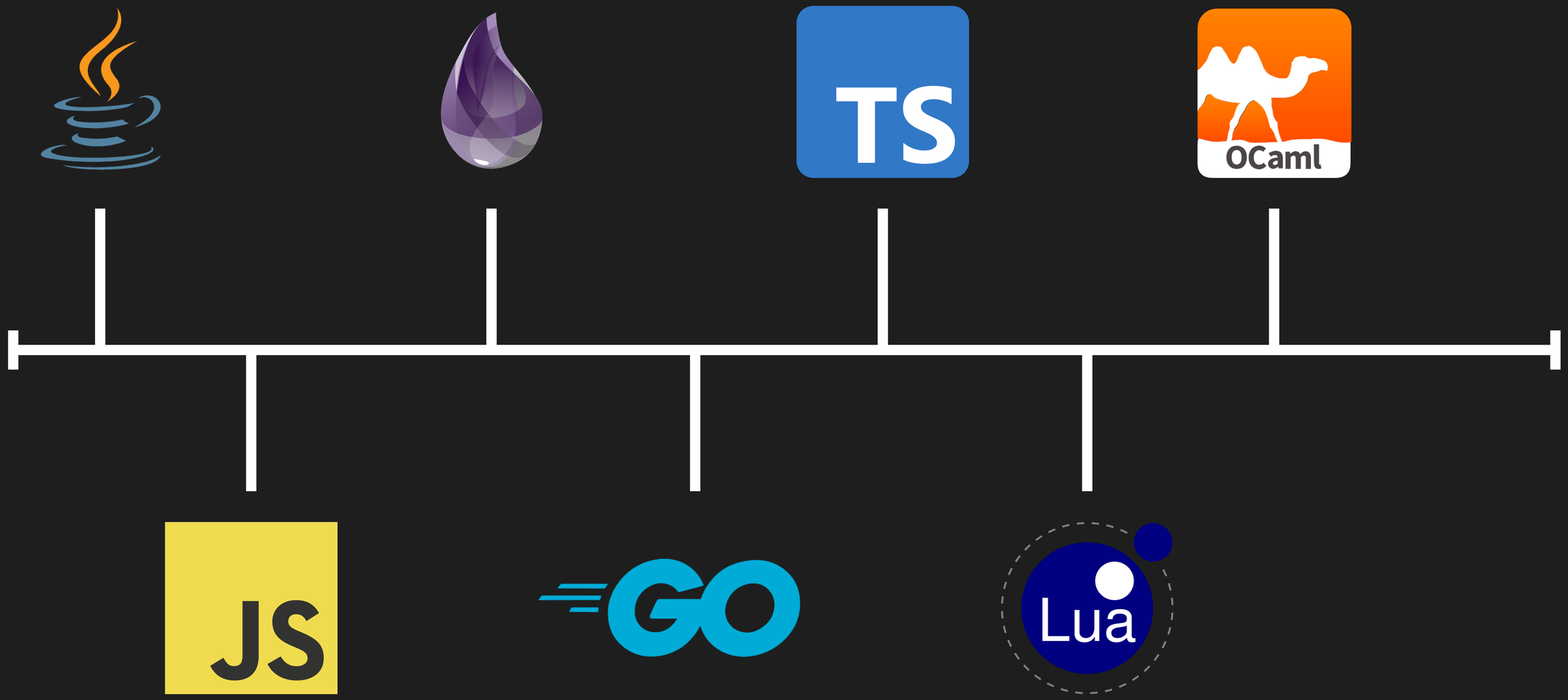

JS

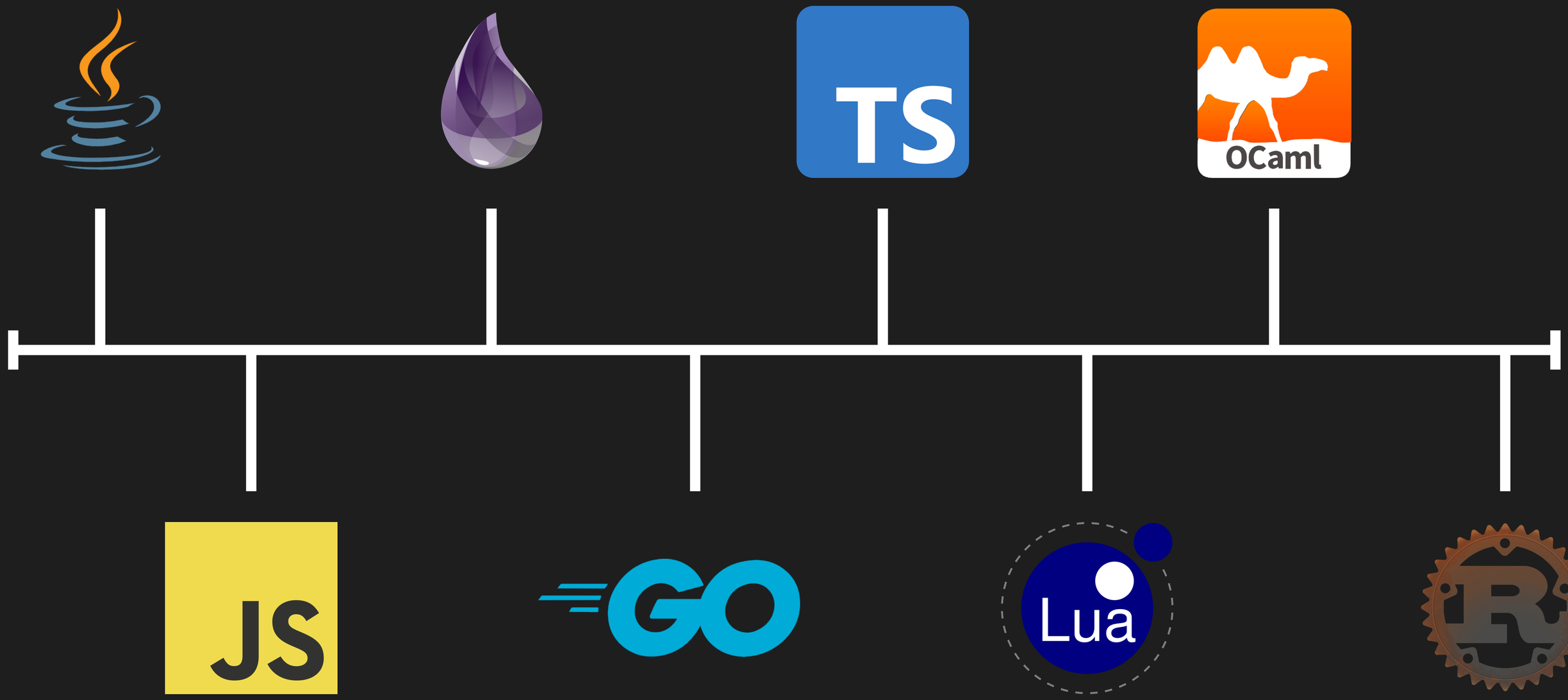

JS


JS
GO


TS
JS
GO


TS
JS
GO
Lua


TS
OCaml
JS
GO
Lua


TS
OCaml
JS
GO
Lua

So… where does that leave me?

Unsatisfied

There are so many languages that get close to being ideal yet *fall short…*

What features would my *ideal* language have?

A Sound Type System

Inspired by Hindley-Milner

- A type system that works for you, instead of against you
- Static typing's safety, dynamic typing's simplicity
- If it compiles, it runs
- Simple polymorphism & generic programming
- Powerful compile time optimizations

Errors as Values

"Throw" Try/Catch in the 🗑️

Errors as Values

“Throw” Try/Catch in the 🗑️

- Ideally using a composable type like Result<Ok, Error>
- Enforce explicit handling of the “unhappy path”
- Use of strongly typed errors, distinguishing between expected and unexpected exceptions
- Avoid control flow interruptions for error management
- Promotes reliable, stable, and maintainable software

Sum Types & Pattern Matching

Enums on steroids

- Encode meaning along with our data
- Express, model, and safely handle complex data structures & control flow with ease
- Make illegal states unrepresentable
- Exhaustive case analysis

First Class Concurrency

Embrace the actor model

- Lightweight, isolated units of computation (actors) powered by green threads
- Simple, cross process, and type-safe communication via message passing
- Fault tolerance through supervision hierarchies
- Easily scale from single-core to distributed systems
- Reduced need for low-level synchronization primitives such as mutexes, semaphores, and RWLocks

DX as a Core Value

Simple, Friendly, & Opinionated

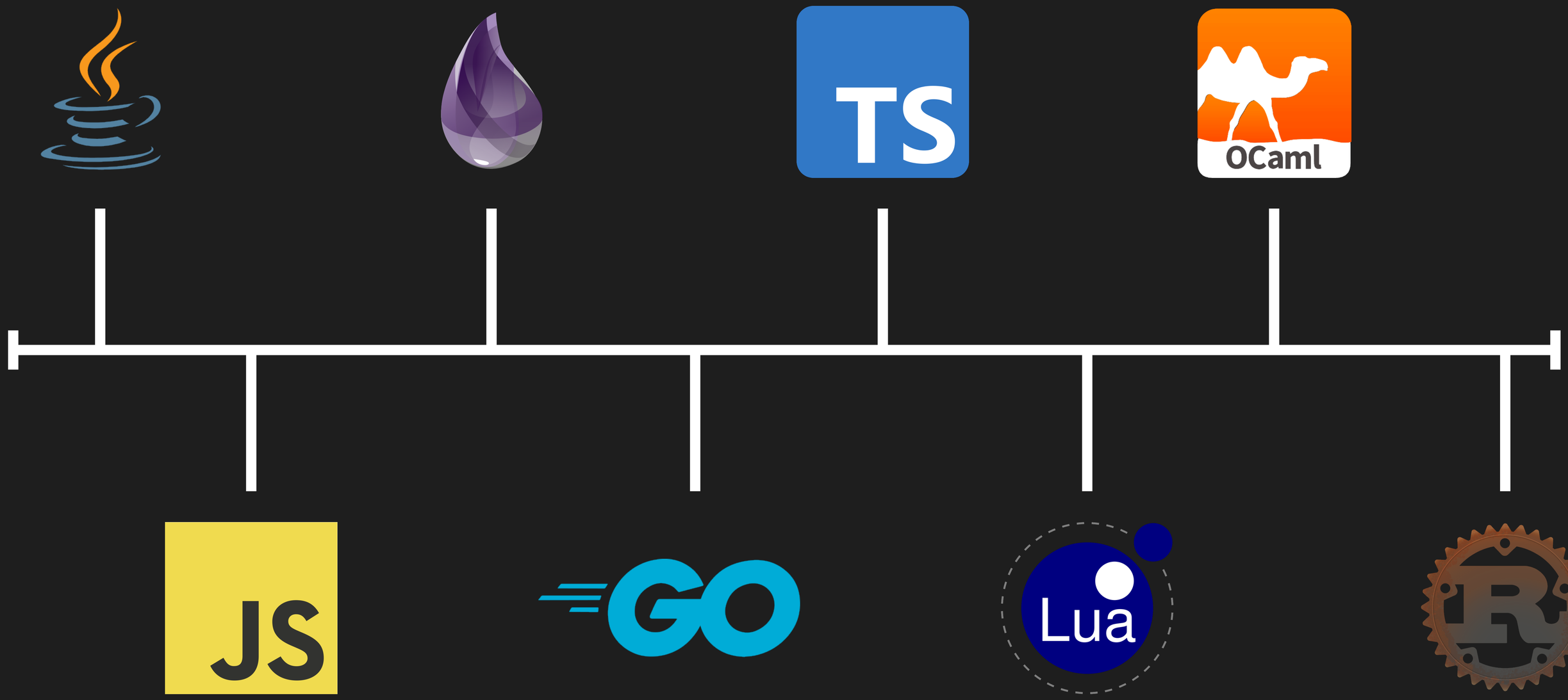
- “One obvious way” to solve common problems
- Unified & modern toolchain (pkg mgmt, build tool, formatting)
- Approachable, searchable, and thoughtful documentation
- Friendly & informative error messages
- Fast compilations times
- Favor backwards compatibility & language simplicity over excessive flexibility & features

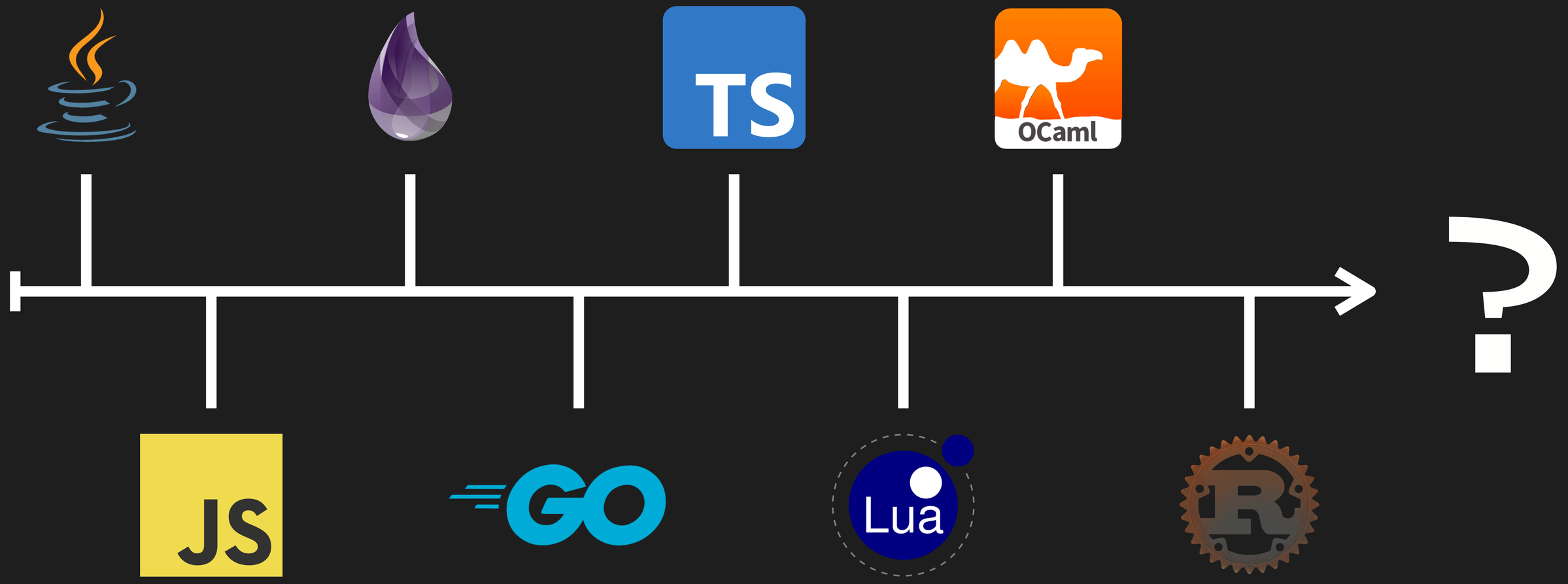
Code Once, Run Everywhere

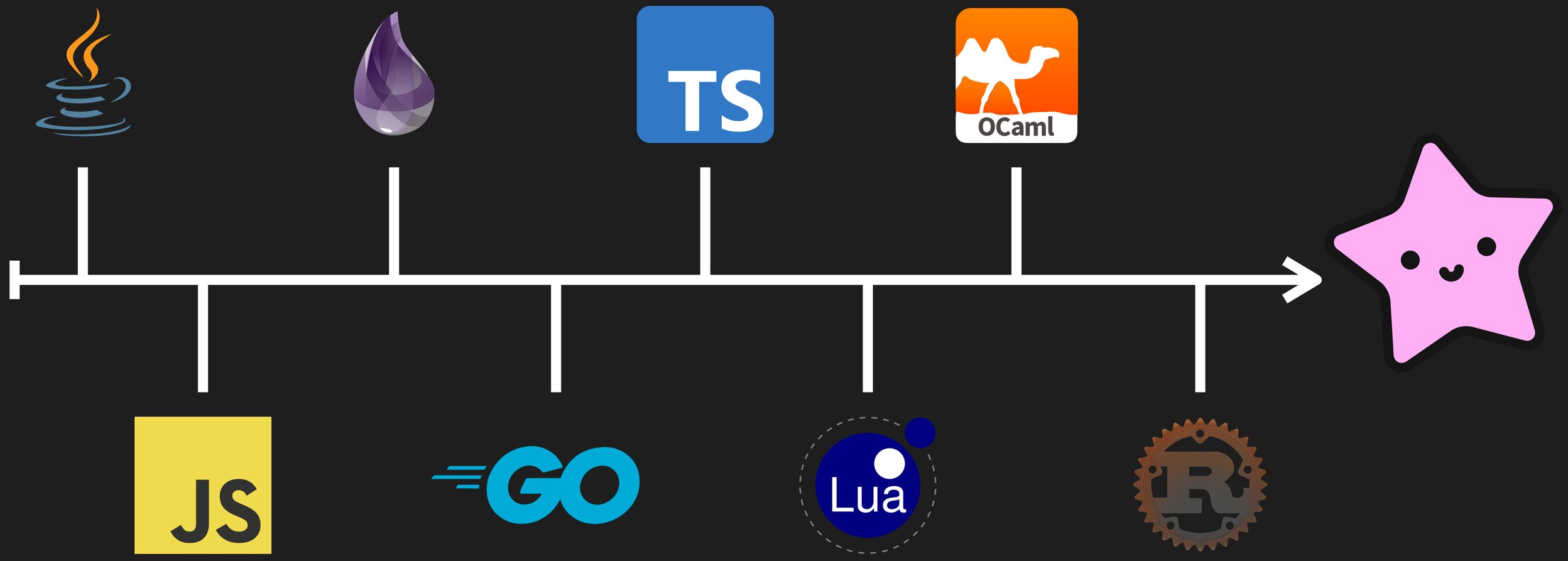
Embracing versatile compilation targets

- Support for compilation to JavaScript and a simple binding/FFI API
- Shared types across the application stack (frontend & back)
- Support for serverless & edge environments
- Simple cross platform support
- Native + WASM targets would be a huge +

I haven’t been completely honest…


TS
OCaml
JS
GO
Lua


TS
OCaml
?
JS
GO
Lua


TS
OCaml
JS
GO
Lua

Gleam

Your next favorite programming language

Gleam is
a *friendly* language
for building
type-safe systems
that *scale*!

Gleam is a sibling language to *Elixir* & *Erlang* that runs on the *BEAM* (Erlang VM)

Gleam uses a simplified implementation of the type system found in *Haskell* & *OCaml*

OXI CLEAN
OXI CLEAN
OXI CLEAN
OXI CLEAN
OXI CLEAN
OXI CLEAN
OXI CLEAN

Gleam can compile to *JavaScript* & generate *TypeScript* types (.d.ts files)

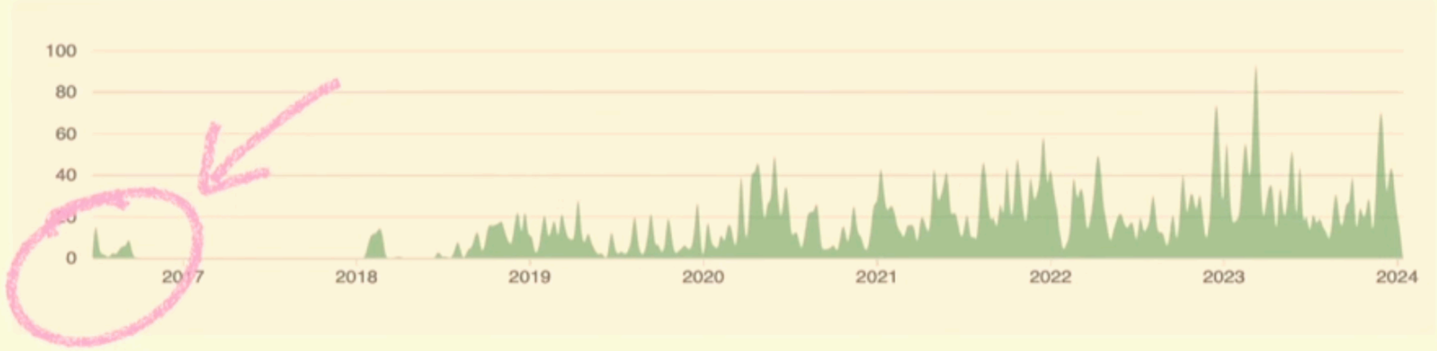
YouTube
Search

A history of Gleam

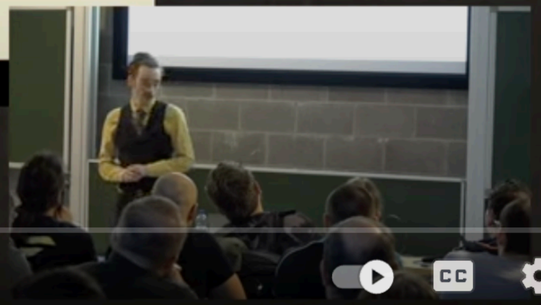
Jun 26, 2016 – Jan 15, 2024

Contributions: Commits

Contributions to main, excluding merge commits



5



2:59 / 40:32

Gleam: Past, Present, Future! • Louis Pilfold @ FOSDEM 2024

Louis Pilfold
2.85K subscribers
Join
Subscribe
800
Share
Download
Thanks
All
From your search
From Louis
Gleam: Typed FP
Is Gleam your n

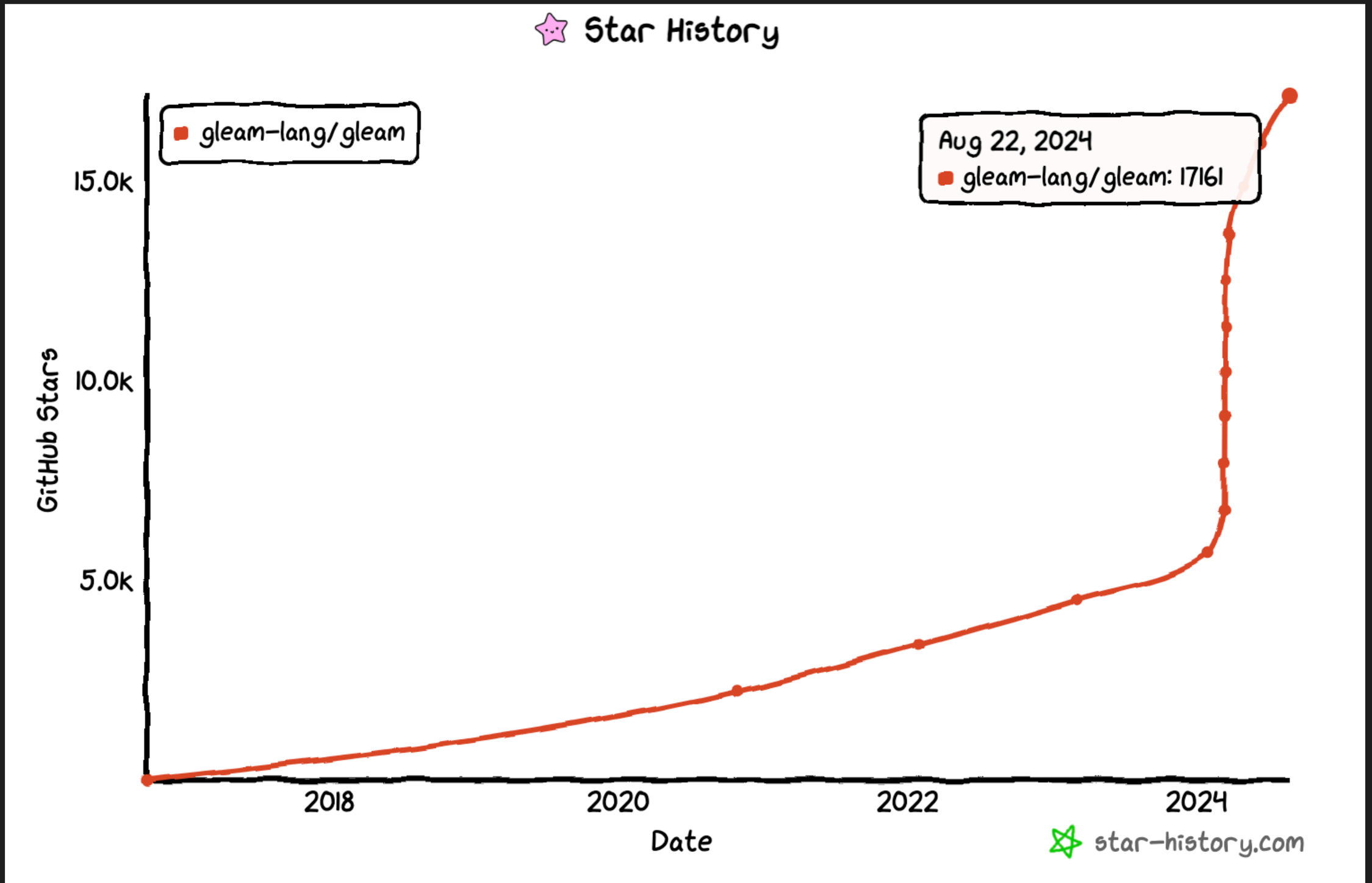
Gleam version 1

It's finally here! 🎉

Published 04 Mar, 2024 by Louis Pilfold

Today Gleam v1.0.0 has been released! This is the first "stable" release of the language (more on what that means later) and a big milestone for the ecosystem.

It has been a long journey and I'm proud of both the technology and the community that we've built here with Gleam. Thank you so much to everyone who has taken part in any way, you've all contributed to the project getting here in meaningful ways of all sizes.


Star History
gleam-lang/gleam
Aug 22, 2024
gleam-lang/gleam: 17161
15.0k
10.0k
5.0k
GitHub Stars
2018
2020
2022
2024
Date
star-history.com

What does *Gleam* look like?

What does *Gleam* look like?

```
pub type User {
  User(
    id: String,
    login: String,
    display_name: String,
    user_type: Type,
    broadcaster_type: BroadcasterType,
    description: String,
    profile_image_url: Uri,
    email: Option(String),
    created_at: String,
  )
}
```

What does *Gleam* look like?

```
pub type User {
  User(
    id: String,
    login: String,
    display_name: String,
    user_type: Type,
    broadcaster_type: BroadcasterType,
    description: String,
    profile_image_url: Uri,
    email: Option(String),
    created_at: String,
  )
}
```

What does *Gleam* look like?

```
pub type User {
  User(
    id: String,
    login: String,
    display_name: String,
    user_type: Type,
    broadcaster_type: BroadcasterType,
    description: String,
    profile_image_url: Uri,
    email: Option(String),
    created_at: String,
  )
}
```

```
pub type Option(a) {
  Some(a)
  None
}
```

What does *Gleam* look like?

```
pub fn get_users(
  client: Client,
  request: GetUsersRequest,
) → Result(List(User), TwitchError) {
  let request =
    api_request.new_helix_request()
    ▷ api_request.set_query(query_params_from_request(request))
    ▷ api_request.set_path("helix/users")

  use response ← result.try(client.get(client, request))

  api_response.get_list_data(response, decoder())
}
```

What does *Gleam* look like?

```
pub fn get_users(
  client: Client,
  request: GetUsersRequest,
) → Result(List(User), TwitchError) {
  let request =
    api_request.new_helix_request()
    ▷ api_request.set_query(query_params_from_request(request))
    ▷ api_request.set_path("helix/users")

  use response ← result.try(client.get(client, request))

  api_response.get_list_data(response, decoder())
}
```

What does *Gleam* look like?

```
pub fn get_users(
  client,
  request,
) {
  let request =
    api_request.new_helix_request()
    ▷ api_request.set_query(query_params_from_request(request))
    ▷ api_request.set_path("helix/users")

  use response ← result.try(client.get(client, request))

  api_response.get_list_data(response, decoder())
}
```

What does *Gleam* look like?

```
pub fn get_users(
  client,
  request,
) {
  let request =
    api_request.new_helix_request()
    ▷ api_request.set_query(query_params_from_request(request))
    ▷ api_request.set_path("helix/users")

  use response ← result.try(client.get(client, request))

  api_response.get_list_data(response, decoder())
}
```

What does *Gleam* look like?

```
pub type MessageType {
  Notification
  SessionWelcome
  SessionKeepalive
  SessionReconnect
  Revocation
}

pub fn message_type_to_string(message_type: MessageType) → String {
  case message_type {
    Notification → "notification"
    SessionWelcome → "session_welcome"
    SessionKeepalive → "session_keepalive"
    SessionReconnect → "sessions_reconnect"
    Revocation → "revocation"
  }
}
```

What does *Gleam* look like?

```
pub type MessageType {
  Notification
  SessionWelcome
  SessionKeepalive
  SessionReconnect
  Revocation
}

pub fn message_type_to_string(message_type) {
  case message_type {
    Notification → "notification"
    SessionWelcome → "session_welcome"
    SessionKeepalive → "session_keepalive"
    SessionReconnect → "sessions_reconnect"
    Revocation → "revocation"
  }
}
```

How would I describe *Gleam*?

Gleam is a language with the simplicity of *Go*

Gleam is a language with the simplicity of *Go*, the type system of *OCaml*

Gleam is a language with the simplicity of *Go*, the type system of *OCaml*, the concurrency model of *Elixir*

Gleam is a language with the simplicity of *Go*, the type system of *OCaml*, the concurrency model of *Elixir*, and the syntax of *Rust* (without the hard parts)

But… does it meet the requirements of my *ideal* language? 🧐

My ideal language requirements

Does Gleam meet them?

My ideal language requirements

Does Gleam meet them?

- A sound type system

A sound type system

- *Gleam* uses an implementation of *Hindley-Milner!*
- This means we have a *sound* type system with powerful *type inference*

A sound type system

```
pub fn add(a, b) {
  a +. b +. 6.9
}
```

A sound type system

```
pub fn add(a, b) {
  a +. b +. 6.9
}
```

A sound type system

```
pub fn add(a: Float, b: Float) → Float {
  a +. b +. 6.9
}
```

A sound type system

```
pub fn add(a, b) {
  a +. b +. 6.9
}
```

A sound type system

```
// fn(Float, Float) → Float
pub fn add(a, b) {
  a +. b +. 6.9
}
```

A sound type system

```
// fn(Float, Float) → Float
pub fn add(a, b) {
  a +. b +. 6.9
}

pub fn add2(a, b) {
  a + b + 69
}
```

A sound type system

```
// fn(Float, Float) → Float
pub fn add(a, b) {
  a +. b +. 6.9
}
// fn(Int, Int) → Int
pub fn add2(a, b) {
  a + b + 69
}
```

A sound type system

```
// fn(Float, Float) → Float
pub fn add(a, b) {
  a +. b +. 6.9
}
// fn(Int, Int) → Int
pub fn add2(a, b) {
  a + b + 69
}

pub fn apply(x, y, apply_fn) {
  apply_fn(x, y)
}
```

A sound type system

```
// fn(Float, Float) → Float
pub fn add(a, b) {
  a +. b +. 6.9
}
// fn(Int, Int) → Int
pub fn add2(a, b) {
  a + b + 69
}
// fn(?, ?, ?) → ?
pub fn apply(x, y, apply_fn) {
  apply_fn(x, y)
}
```

A sound type system

```
// fn(Float, Float) → Float
pub fn add(a, b) {
  a +. b +. 6.9
}
// fn(Int, Int) → Int
pub fn add2(a, b) {
  a + b + 69
}
// fn(?, ?, ?) → ?
pub fn apply(x, y, apply_fn) {
  apply_fn(x, y) // fn(a, b) → c
}
```

A sound type system

```
// fn(Float, Float) → Float
pub fn add(a, b) {
  a +. b +. 6.9
}
// fn(Int, Int) → Int
pub fn add2(a, b) {
  a + b + 69
}
// fn(?, ?, fn(a, b) → c) → ?
pub fn apply(x, y, apply_fn) {
  apply_fn(x, y)
}
```

A sound type system

```
// fn(Float, Float) → Float
pub fn add(a, b) {
  a +. b +. 6.9
}
// fn(Int, Int) → Int
pub fn add2(a, b) {
  a + b + 69
}
// fn(a, b, fn(a, b) → c) → c
pub fn apply(x, y, apply_fn) {
  apply_fn(x, y)
}
```

A sound type system

```
fn main() {
  let a = apply(10.0, 4.0, add)
  let b = apply(10, 4, add2)
}
```

A sound type system

```
fn main() {
  // fn(?, ?, fn(?, ?) → ?) → ?
  let a = apply(10.0, 4.0, add)
  let b = apply(10, 4, add2)
}
```

A sound type system

```
fn main() {
  // fn(Float, Float, fn(Float, Float) → Float) → Float
  let a = apply(10.0, 4.0, add)
  let b = apply(10, 4, add2)
}
```

A sound type system

```
fn main() {
  // fn(Float, Float, fn(Float, Float) → Float) → Float
  let a: Float = apply(10.0, 4.0, add)
  let b = apply(10, 4, add2)
}
```

A sound type system

```
fn main() {
  // fn(Float, Float, fn(Float, Float) → Float) → Float
  let a: Float = apply(10.0, 4.0, add)
  // fn(Int, Int, fn(Int, Int) → Int) → Int
  let b: Int = apply(10, 4, add2)
}
```

My ideal language requirements

Does Gleam meet them?

- A sound type system ✅

My ideal language requirements

Does Gleam meet them?

- A sound type system ✅
- Errors as values

Errors as values

Errors as values

```
pub type Result(ok, error) {
  Ok(ok)
  Error(error)
}
```

Errors as values

```
pub fn fetch_data() → Result(data, RequestError) {
  todo as "impl left to the curious reader"
}

fn main() {
  let data = fetch_data()
}
```

Errors as values

```
pub fn fetch_data() → Result(data, RequestError) {
  todo as "impl left to the curious reader"
}

fn main() {
  let data: Result(data, RequestError) = fetch_data()
}
```

Errors as values

```
pub fn fetch_data() → Result(data, RequestError) {
  todo as "impl left to the curious reader"
}

fn main() {
  // well, we can’t really _do_ anything w/ the Result
  // How can we get the data???
  let data: Result(data, RequestError) = fetch_data()
}
```

Errors as values

```
pub fn fetch_data() → Result(data, RequestError) {
  todo as "impl left to the curious reader"
}

fn main() {
  // we can “assert” that the result matches a pattern
  // note: if it does not match your program will panic
  let assert Ok(data) = fetch_data()
  // now we can do something w/ data
  process(data)
}
```

Errors as values

```
pub fn fetch_data() → Result(data, RequestError) {
  todo as "impl left to the curious reader"
}

fn main() {
  let fetch_result = fetch_data()
  // we can also pattern match on Results!
  case fetch_result {
    Ok(data) → {
      // now we can do something w/ data
      process(data)
    }
    Error(error) {
      // explicitly handle the error case
    }
  }
}
```

Errors as values

```
pub fn fetch_data() → Result(data, RequestError) {
  todo as "impl left to the curious reader"
}

fn main() {
  let fetch_result = fetch_data()
  // we can use the result modules `unwrap` fn
  let data = result.unwrap(fetch_result, “boom 💥")
  process(data)
}
```

Errors as values

```
pub fn fetch_data() → Result(data, RequestError) {
  todo as "impl left to the curious reader"
}

fn main() {
  let fetch_result = fetch_data()
  // We can use `use` to "unbox" an Ok value
  // or immediately return from the scope
  // with the Error value
  use data ← result.try(fetch_result)

  process(data)
}
```

My ideal language requirements

Does Gleam meet them?

- A sound type system ✅
- Errors as values ✅

My ideal language requirements

Does Gleam meet them?

- A sound type system ✅
- Errors as values ✅
- Sum types & pattern matching

Sum types & pattern matching

Sum types & pattern matching

Lets implement Rock, Paper, Scissors

Sum types & pattern matching

Lets implement Rock, Paper, Scissors

```
pub type Move {
  Rock
  Paper
  Scissors
}
```

Sum types & pattern matching

Lets implement Rock, Paper, Scissors

```
pub type Move {
  Rock
  Paper
  Scissors
}

pub type GameResult {
  Player1Wins(threw: Move, beat: Move)
  Player2Wins(threw: Move, beat: Move)
  Draw(Move)
}
```

Sum types & pattern matching

Lets implement Rock, Paper, Scissors

```
pub fn move_to_string(move) {
  case move {
    Rock → "Rock"
    Paper → "Paper"
    Scissors → "Scissors"
  }
}
```

Sum types & pattern matching

Lets implement Rock, Paper, Scissors

```
pub fn game_result_to_string(result) {
  case result {
    Player1Wins(threw, beat) →
      "Player 1 won by throwing "
      <> move_to_string(threw)
      <> "against "
      <> move_to_string(beat)
    Player2Wins(threw, beat) →
      "Player 2 won by throwing "
      <> move_to_string(threw)
      <> "against "
      <> move_to_string(beat)
    Draw(move) →
      "Draw! Both players threw " <> move_to_string(move)
  }
}
```

Sum types & pattern matching

Lets implement Rock, Paper, Scissors

```
pub fn game_result_to_string(result) {
  case result {
    Player1Wins(threw, beat) →
      "Player 1 won by throwing "
      <> move_to_string(threw)
      <> "against "
      <> move_to_string(beat)
    Player2Wins(threw, beat) →
      "Player 2 won by throwing "
      <> move_to_string(threw)
      <> "against "
      <> move_to_string(beat)
    Draw(move) →
      "Draw! Both players threw " <> move_to_string(move)
  }
}
```

Sum types & pattern matching

Lets implement Rock, Paper, Scissors

```
pub fn game_result_to_string(result) {
  case result {
    Player1Wins(threw, beat) →
      "Player 1 won by throwing "
      <> move_to_string(threw)
      <> "against "
      <> move_to_string(beat)
    Player2Wins(threw, beat) →
      "Player 2 won by throwing "
      <> move_to_string(threw)
      <> "against "
      <> move_to_string(beat)
    Draw(move) →
      "Draw! Both players threw " <> move_to_string(move)
  }
}
```

Sum types & pattern matching

Lets implement Rock, Paper, Scissors

```
pub fn game_result_to_string(result) {
  case result {
    Player1Wins(threw, beat) →
      "Player 1 won by throwing "
      <> move_to_string(threw)
      <> "against "
      <> move_to_string(beat)
    Player2Wins(threw, beat) →
      "Player 2 won by throwing "
      <> move_to_string(threw)
      <> "against "
      <> move_to_string(beat)
    Draw(move) →
      "Draw! Both players threw " <> move_to_string(move)
  }
}
```

Sum types & pattern matching

Lets implement Rock, Paper, Scissors

```
pub fn game_result_to_string(result) {
  case result {
    Player1Wins(threw, beat) →
      "Player 1 won by throwing "
      <> move_to_string(threw)
      <> "against "
      <> move_to_string(beat)
    Player2Wins(threw, beat) →
      "Player 2 won by throwing "
      <> move_to_string(threw)
      <> "against "
      <> move_to_string(beat)
    Draw(move) →
      "Draw! Both players threw " <> move_to_string(move)
  }
}
```

Sum types & pattern matching

Lets implement Rock, Paper, Scissors

```
pub fn play(player_1_move, player_2_move) {
  let game_result = case player_1_move, player_2_move {
    Rock, Scissors → Player1Wins(Rock, Scissors)
    Rock, Paper → Player2Wins(Paper, Rock)
    Paper, Rock → Player1Wins(Paper, Rock)
    Paper, Scissors → Player2Wins(Scissors, Paper)
    Scissors, Rock → Player2Wins(Rock, Scissors)
    Scissors, Paper → Player1Wins(Scissors, Paper)
    _, _ → Draw(player_1_move)
  }

  game_result
  ▷ game_result_to_string
  ▷ io.println
}
```

Sum types & pattern matching

Lets implement Rock, Paper, Scissors

```
pub fn play(player_1_move, player_2_move) {
  let game_result = case player_1_move, player_2_move {
    Rock, Scissors → Player1Wins(Rock, Scissors)
    Rock, Paper → Player2Wins(Paper, Rock)
    Paper, Rock → Player1Wins(Paper, Rock)
    Paper, Scissors → Player2Wins(Scissors, Paper)
    Scissors, Rock → Player2Wins(Rock, Scissors)
    Scissors, Paper → Player1Wins(Scissors, Paper)
    _, _ → Draw(player_1_move)
  }

  game_result
  ▷ game_result_to_string
  ▷ io.println
}
```

Sum types & pattern matching

Lets implement Rock, Paper, Scissors

```
pub fn play(player_1_move, player_2_move) {
  let game_result = case player_1_move, player_2_move {
    Rock, Scissors → Player1Wins(Rock, Scissors)
    Rock, Paper → Player2Wins(Paper, Rock)
    Paper, Rock → Player1Wins(Paper, Rock)
    Paper, Scissors → Player2Wins(Scissors, Paper)
    Scissors, Rock → Player2Wins(Rock, Scissors)
    Scissors, Paper → Player1Wins(Scissors, Paper)
    _, _ → Draw(player_1_move)
  }

  game_result
  ▷ game_result_to_string
  ▷ io.println
}
```

Sum types & pattern matching

Lets implement Rock, Paper, Scissors

```
pub fn play(player_1_move, player_2_move) {
  let game_result = case player_1_move, player_2_move {
    Rock, Scissors → Player1Wins(Rock, Scissors)
    Rock, Paper → Player2Wins(Paper, Rock)
    Paper, Rock → Player1Wins(Paper, Rock)
    Paper, Scissors → Player2Wins(Scissors, Paper)
    Scissors, Rock → Player2Wins(Rock, Scissors)
    Scissors, Paper → Player1Wins(Scissors, Paper)
    _, _ → Draw(player_1_move)
  }

  game_result
  ▷ game_result_to_string
  ▷ io.println
}
```

Sum types & pattern matching

Lets implement Rock, Paper, Scissors

```
pub fn play(player_1_move, player_2_move) {
  let game_result = case player_1_move, player_2_move {
    Rock, Scissors → Player1Wins(Rock, Scissors)
    Rock, Paper → Player2Wins(Paper, Rock)
    Paper, Rock → Player1Wins(Paper, Rock)
    Paper, Scissors → Player2Wins(Scissors, Paper)
    Scissors, Rock → Player2Wins(Rock, Scissors)
    Scissors, Paper → Player1Wins(Scissors, Paper)
    _, _ → Draw(player_1_move)
  }

  game_result
  ▷ game_result_to_string
  ▷ io.println
}
```

Sum types & pattern matching

Lets implement Rock, Paper, Scissors

```
pub fn play(player_1_move, player_2_move) {
  let game_result = case player_1_move, player_2_move {
    Rock, Scissors → Player1Wins(Rock, Scissors)
    Rock, Paper → Player2Wins(Paper, Rock)
    Paper, Rock → Player1Wins(Paper, Rock)
    Paper, Scissors → Player2Wins(Scissors, Paper)
    Scissors, Rock → Player2Wins(Rock, Scissors)
    Scissors, Paper → Player1Wins(Scissors, Paper)
    _, _ → Draw(player_1_move)
  }

  game_result
  ▷ game_result_to_string
  ▷ io.println
}
```

Sum types & pattern matching

Lets implement Rock, Paper, Scissors

```
pub fn play(player_1_move, player_2_move) {
  let game_result = case player_1_move, player_2_move {
    Rock, Scissors → Player1Wins(Rock, Scissors)
    Rock, Paper → Player2Wins(Paper, Rock)
    Paper, Rock → Player1Wins(Paper, Rock)
    Paper, Scissors → Player2Wins(Scissors, Paper)
    Scissors, Rock → Player2Wins(Rock, Scissors)
    Scissors, Paper → Player1Wins(Scissors, Paper)
    _, _ → Draw(player_1_move)
  }

  game_result
  ▷ game_result_to_string
  ▷ io.println
}
```

Sum types & pattern matching

Lets implement Rock, Paper, Scissors

```
pub fn play(player_1_move, player_2_move) {
  let game_result = case player_1_move, player_2_move {
    Rock, Scissors → Player1Wins(Rock, Scissors)
    Rock, Paper → Player2Wins(Paper, Rock)
    Paper, Rock → Player1Wins(Paper, Rock)
    Paper, Scissors → Player2Wins(Scissors, Paper)
    Scissors, Rock → Player2Wins(Rock, Scissors)
    Scissors, Paper → Player1Wins(Scissors, Paper)
    _, _ → Draw(player_1_move)
  }

  game_result
  ▷ game_result_to_string
  ▷ io.println
}
```

Sum types & pattern matching

Lets implement Rock, Paper, Scissors

```
pub fn play(player_1_move, player_2_move) {
  let game_result = case player_1_move, player_2_move {
    Rock, Scissors → Player1Wins(Rock, Scissors)
    Rock, Paper → Player2Wins(Paper, Rock)
    Paper, Rock → Player1Wins(Paper, Rock)
    Paper, Scissors → Player2Wins(Scissors, Paper)
    Scissors, Rock → Player2Wins(Rock, Scissors)
    Scissors, Paper → Player1Wins(Scissors, Paper)
    _, _ → Draw(player_1_move)
  }

  game_result
  ▷ game_result_to_string
  ▷ io.println
}
```

My ideal language requirements

Does Gleam meet them?

- A sound type system ✅
- Errors as values ✅
- Sum types & pattern matching ✅

My ideal language requirements

Does Gleam meet them?

- A sound type system ✅
- Errors as values ✅
- Sum types & pattern matching ✅
- First class concurrency

First class concurrency

Scale fearlessly with OTP

First class concurrency

Scale fearlessly with OTP

```
pub fn main() {
  let tasks =
    list.range(1, 100_000)
    ▷ list.map(fn(i) {
      task.async(fn() {
        io.println("Task " <> int.to_string(i) <> " completed")
      })
    })

  let results = list.map(tasks, task.await_forever)

  io.println(
    "All tasks completed. Total: "
    <> results ▷ int.sum ▷ int.to_string
  )
}
```

First class concurrency

Scale fearlessly with OTP

```
pub fn main() {
  let tasks =
    list.range(1, 100_000)
    ▷ list.map(fn(i) {
      task.async(fn() {
        io.println("Task " <> int.to_string(i) <> " completed")
      })
    })

  let results = list.map(tasks, task.await_forever)

  io.println(
    "All tasks completed. Total: "
    <> results ▷ int.sum ▷ int.to_string
  )
}
```

First class concurrency

Scale fearlessly with OTP

```
pub fn main() {
  let tasks =
    list.range(1, 100_000)
    ▷ list.map(fn(i) {
      task.async(fn() {
        io.println("Task " <> int.to_string(i) <> " completed")
      })
    })

  let results = list.map(tasks, task.await_forever)

  io.println(
    "All tasks completed. Total: "
    <> results ▷ int.sum ▷ int.to_string
  )
}
```

First class concurrency

Scale fearlessly with OTP

```
pub fn main() {
  let tasks =
    list.range(1, 100_000)
    ▷ list.map(fn(i) {
      task.async(fn() {
        io.println("Task " <> int.to_string(i) <> " completed")
      })
    })

  let results = list.map(tasks, task.await_forever)

  io.println(
    "All tasks completed. Total: "
    <> results ▷ int.sum ▷ int.to_string
  )
}
```

First class concurrency

Scale fearlessly with OTP

```
pub fn main() {
  let tasks =
    list.range(1, 100_000)
    ▷ list.map(fn(i) {
      task.async(fn() {
        io.println("Task " <> int.to_string(i) <> " completed")
      })
    })

  let results = list.map(tasks, task.await_forever)

  io.println(
    "All tasks completed. Total: "
    <> results ▷ int.sum ▷ int.to_string
  )
}
```

First class concurrency

Scale fearlessly with OTP

```
pub fn main() {
  let tasks =
    list.range(1, 100_000)
    ▷ list.map(fn(i) {
      task.async(fn() {
        io.println("Task " <> int.to_string(i) <> " completed")
      })
    })

  let results = list.map(tasks, task.await_forever)

  io.println(
    "All tasks completed. Total: "
    <> results ▷ int.sum ▷ int.to_string
  )
}
```

First class concurrency

Scale fearlessly with OTP

First class concurrency

Scale fearlessly with OTP

```
pub type State {
  Count(Int)
}
```

First class concurrency

Scale fearlessly with OTP

```
pub type State {
  Count(Int)
}

pub type Message {
  Increment
  GetCount(reply_with: Subject(Result(State, Nil)))
  Shutdown
}
```

First class concurrency

Scale fearlessly with OTP

```
pub type State {
  Count(Int)
}

pub type Message {
  Increment
  GetCount(reply_with: Subject(Result(State, Nil)))
  Shutdown
}

pub fn get_count(state) {
  case state {
    Count(value) → value
  }
}
```

First class concurrency

Scale fearlessly with OTP

```
pub fn handler(message, state) → actor.Next(Message, State) {
  case message {
    Increment → actor.continue(Count(get_count(state) + 1))
    GetCount(mailbox) → {
      process.send(mailbox, Ok(state))
      actor.continue(state)
    }
    Shutdown → actor.Stop(process.Normal)
  }
}
```

First class concurrency

Scale fearlessly with OTP

```
pub fn handler(message, state) → actor.Next(Message, State) {
  case message {
    Increment → actor.continue(Count(get_count(state) + 1))
    GetCount(mailbox) → {
      process.send(mailbox, Ok(state))
      actor.continue(state)
    }
    Shutdown → actor.Stop(process.Normal)
  }
}
```

First class concurrency

Scale fearlessly with OTP

```
pub fn handler(message, state) → actor.Next(Message, State) {
  case message {
    Increment → actor.continue(Count(get_count(state) + 1))
    GetCount(mailbox) → {
      process.send(mailbox, Ok(state))
      actor.continue(state)
    }
    Shutdown → actor.Stop(process.Normal)
  }
}
```

First class concurrency

Scale fearlessly with OTP

```
pub type Next(message, state) {
  Continue(state: state, selector: Option(Selector(message)))
  Stop(ExitReason)
}
```

First class concurrency

Scale fearlessly with OTP

```
pub fn handler(message, state) → actor.Next(Message, State) {
  case message {
    Increment → actor.continue(Count(get_count(state) + 1))
    GetCount(mailbox) → {
      process.send(mailbox, Ok(state))
      actor.continue(state)
    }
    Shutdown → actor.Stop(process.Normal)
  }
}
```

First class concurrency

Scale fearlessly with OTP

```
pub fn handler(message, state) → actor.Next(Message, State) {
  case message {
    Increment → actor.continue(Count(get_count(state) + 1))
    GetCount(mailbox) → {
      process.send(mailbox, Ok(state))
      actor.continue(state)
    }
    Shutdown → actor.Stop(process.Normal)
  }
}
```

First class concurrency

Scale fearlessly with OTP

```
pub fn handler(message, state) → actor.Next(Message, State) {
  case message {
    Increment → actor.continue(Count(get_count(state) + 1))
    GetCount(mailbox) → {
      process.send(mailbox, Ok(state))
      actor.continue(state)
    }
    Shutdown → actor.Stop(process.Normal)
  }
}
```

First class concurrency

Scale fearlessly with OTP

```
pub type Next(message, state) {
  Continue(state: state, selector: Option(Selector(message)))
  Stop(ExitReason)
}
```

First class concurrency

Scale fearlessly with OTP

```
pub fn handler(message, state) → actor.Next(Message, State) {
  case message {
    // Count(State, None)
    Increment → actor.continue(Count(get_count(state) + 1))
    GetCount(mailbox) → {
      process.send(mailbox, Ok(state))
      actor.continue(state)
    }
    Shutdown → actor.Stop(process.Normal)
  }
}
```

First class concurrency

Scale fearlessly with OTP

```
pub fn handler(message, state) → actor.Next(Message, State) {
  case message {
    Increment → actor.continue(Count(get_count(state) + 1))
    GetCount(mailbox) → {
      process.send(mailbox, Ok(state))
      actor.continue(state)
    }
    Shutdown → actor.Stop(process.Normal)
  }
}
```

First class concurrency

Scale fearlessly with OTP

```
pub fn handler(message, state) → actor.Next(Message, State) {
  case message {
    Increment → actor.continue(Count(get_count(state) + 1))
    GetCount(mailbox) → {
      process.send(mailbox, Ok(state))
      actor.continue(state)
    }
    Shutdown → actor.Stop(process.Normal)
  }
}
```

First class concurrency

Scale fearlessly with OTP

```
pub fn handler(message, state) → actor.Next(Message, State) {
  case message {
    Increment → actor.continue(Count(get_count(state) + 1))
    GetCount(mailbox) → {
      process.send(mailbox, Ok(state))
      actor.continue(state)
    }
    Shutdown → actor.Stop(process.Normal)
  }
}
```

First class concurrency

Scale fearlessly with OTP

```
pub fn handler(message, state) → actor.Next(Message, State) {
  case message {
    Increment → actor.continue(Count(get_count(state) + 1))
    GetCount(mailbox) → {
      process.send(mailbox, Ok(state))
      actor.continue(state)
    }
    Shutdown → actor.Stop(process.Normal)
  }
}
```

First class concurrency

Scale fearlessly with OTP

```
pub fn main() {
  let assert Ok(subject) = actor.start(Count(0), handler)

  actor.send(subject, Increment)
  actor.send(subject, Increment)
  let assert Ok(state) = actor.call(subject, GetCount, 1000)
  // prints: Current count: 2
  io.println("Current count: " <> int.to_string(get_count(state)))

  actor.send(subject, Increment)
  let state = actor.call(subject, function.constant(GetCount), 1000)
  // prints: Current count: 2
  io.println("Current count: " <> int.to_string(get_count(state)))

 actor.send(subject, Shutdown)
}
```

First class concurrency

Scale fearlessly with OTP

```
pub fn main() {
  let assert Ok(subject) = actor.start(Count(0), handler)

  actor.send(subject, Increment)
  actor.send(subject, Increment)
  let assert Ok(state) = actor.call(subject, GetCount, 1000)
  // prints: Current count: 2
  io.println("Current count: " <> int.to_string(get_count(state)))

  actor.send(subject, Increment)
  let state = actor.call(subject, function.constant(GetCount), 1000)
  // prints: Current count: 2
  io.println("Current count: " <> int.to_string(get_count(state)))

 actor.send(subject, Shutdown)
}
```

First class concurrency

Scale fearlessly with OTP

```
pub fn main() {
  let assert Ok(subject) = actor.start(Count(0), handler)

  actor.send(subject, Increment)
  actor.send(subject, Increment)
  let assert Ok(state) = actor.call(subject, GetCount, 1000)
  // prints: Current count: 2
  io.println("Current count: " <> int.to_string(get_count(state)))

  actor.send(subject, Increment)
  let state = actor.call(subject, function.constant(GetCount), 1000)
  // prints: Current count: 2
  io.println("Current count: " <> int.to_string(get_count(state)))

 actor.send(subject, Shutdown)
}
```

First class concurrency

Scale fearlessly with OTP

```
pub fn main() {
  let assert Ok(subject) = actor.start(Count(0), handler)

  actor.send(subject, Increment)
  actor.send(subject, Increment)
  let assert Ok(state) = actor.call(subject, GetCount, 1000)
  // prints: Current count: 2
  io.println("Current count: " <> int.to_string(get_count(state)))

  actor.send(subject, Increment)
  let state = actor.call(subject, function.constant(GetCount), 1000)
  // prints: Current count: 2
  io.println("Current count: " <> int.to_string(get_count(state)))

 actor.send(subject, Shutdown)
}
```

First class concurrency

Scale fearlessly with OTP

```
pub fn main() {
  let assert Ok(subject) = actor.start(Count(0), handler)

  actor.send(subject, Increment)
  actor.send(subject, Increment)
  let assert Ok(state) = actor.call(subject, GetCount, 1000)
  // prints: Current count: 2
  io.println("Current count: " <> int.to_string(get_count(state)))

  actor.send(subject, Increment)
  let state = actor.call(subject, function.constant(GetCount), 1000)
  // prints: Current count: 2
  io.println("Current count: " <> int.to_string(get_count(state)))

 actor.send(subject, Shutdown)
}
```

First class concurrency

Scale fearlessly with OTP

```
pub fn main() {
  let assert Ok(subject) = actor.start(Count(0), handler)

  actor.send(subject, Increment)
  actor.send(subject, Increment)
  let assert Ok(state) = actor.call(subject, GetCount, 1000)
  // prints: Current count: 2
  io.println("Current count: " <> int.to_string(get_count(state)))

  actor.send(subject, Increment)
  let state = actor.call(subject, function.constant(GetCount), 1000)
  // prints: Current count: 2
  io.println("Current count: " <> int.to_string(get_count(state)))

  actor.send(subject, Shutdown)
}
```

First class concurrency

Scale fearlessly with OTP

```
pub fn main() {
  let assert Ok(subject) = actor.start(Count(0), handler)

  actor.send(subject, Increment)
  actor.send(subject, Increment)
  let assert Ok(state) = actor.call(subject, GetCount, 1000)
  // prints: Current count: 2
  io.println("Current count: " <> int.to_string(get_count(state)))

  actor.send(subject, Increment)
  let state = actor.call(subject, function.constant(GetCount), 1000)
  // prints: Current count: 2
  io.println("Current count: " <> int.to_string(get_count(state)))

 actor.send(subject, Shutdown)
}
```

My ideal language requirements

Does Gleam meet them?

- A sound type system ✅
- Errors as values ✅
- Sum types & pattern matching ✅
- First class concurrency ✅

My ideal language requirements

Does Gleam meet them?

- A sound type system ✅
- Errors as values ✅
- Sum types & pattern matching ✅
- First class concurrency ✅
- DX as a core value

DX as a core value

Don't make me think

- Louis and the core team are ***incredibly diligent*** and ***thoughtful*** about keeping the language simple, small, and practical
- Like Go, you'll often find that there is ***one way*** of doing something and the language will naturally guide you there
- New features & syntax must meet a ***very high bar*** and strictly improve the language while maintaining its ***simplicity***

DX as a core value

Modern tooling

```
ghostty
1 glitch                                                    glitch session gleam
Code/personal/glitch on ᚠ main [!?] via ⭐ v1.4.1 via ❄ impure (nix-shell-env)   22:21:55 [5/35]
λ gleam --help
gleam 1.4.1

Usage: gleam <COMMAND>

Commands:
  add      Add new project dependencies
  build    Build the project
  check    Type check the project
  clean    Clean build artifacts
  deps     Work with dependency packages
  docs     Render HTML documentation
  export   Export something useful from the Gleam project
  fix      Rewrite deprecated Gleam code
  format   Format source code
  help     Print this message or the help of the given subcommand(s)
  hex      Work with the Hex package manager
  lsp      Run the language server, to be used by editors
  new      Create a new project
  publish  Publish the project to the Hex package manager
  remove   Remove project dependencies
  run      Run the project
  shell    Start an Erlang shell
  test     Run the project tests
  update   Update dependency packages to their latest versions

Options:
  -h, --help     Print help
  -V, --version  Print version
```

DX as a core value

Stellar documentation built on HexDocs

gleam/result · gleam_stdlib

hexdocs.pm/gleam_stdlib/gleam/result.html

vercel | dev resources | recipes | other | All Bookmarks

gleam_st... v0.40.0

Pages

README

Links

Website
Sponsor
Repository
Hex

Modules

gleam/bit_array
gleam/bool
gleam/bytes_builder
gleam/dict
gleam/dynamic
gleam/float
gleam/function
gleam/int
gleam/io
gleam/iterator
gleam/list
gleam/option
gleam/order
gleam/pair
gleam/queue
gleam/regex
gleam/result
gleam/set
gleam/string
gleam/string_builder

try

```
pub fn try(
  result: Result(a, b),
  apply fun: fn(a) -> Result(c, b),
) -> Result(c, b)
```

"Updates" an `Ok` result by passing its value to a function that yields a result, and returning the yielded result. (This may "replace" the `Ok` with an `Error`.)

If the input is an `Error` rather than an `Ok`, the function is not called and the original `Error` is returned.

This function is the equivalent of calling `map` followed by `flatten`, and it is useful for chaining together multiple functions that may fail.

Examples

```
try(Ok(1), fn(x) { Ok(x + 1) })
// -> Ok(2)
```

```
try(Ok(1), fn(x) { Ok(#("a", x)) })
// -> Ok(#("a", 1))
```

```
try(Ok(1), fn(_) { Error("Oh no") })
// -> Error("Oh no")
```

DX as a core value

Helpful & friendly error messages

DX as a core value

Helpful & friendly error messages

```
error: Unknown record field

   ┌─ ./src/app.gleam:8:16
   │
8  │ user.alias
   │     ^^^^^^ Did you mean `name`?

The value being accessed has this type:
    User

It has these fields:
    .name
```

DX as a core value

Helpful & friendly error messages

```
error: Incorrect arity
   ┌─ /src/one/two.gleam:3:8
   │
3  │ use <- f(1, 2)
   │        ^^^^^^^ Expected 2 arguments, got 3

The function on the right of `<-` here takes 2 arguments.
All the arguments have already been supplied, so it cannot take the
`use` callback function as a final argument.

See: https://tour.gleam.run/advanced-features/use/
```

DX as a core value

Helpful & friendly error messages

```
warning: Redundant list
   ┌─ /src/warning/wrn.gleam:2:14
   │
2  │         case [1, 2] {
   │              ^^^^^^ You can remove this list wrapper

Instead of building a list and matching on it, you can match on its
contents directly.
A case expression can take multiple subjects separated by commas like this:

    case one_subject, another_subject {
      _, _ -> todo
    }

See: https://tour.gleam.run/flow-control/multiple-subjects/
```

My ideal language requirements

Does Gleam meet them?

- A sound type system ✅
- Errors as values ✅
- Sum types & pattern matching ✅
- First class concurrency ✅
- DX as a core value ✅

My ideal language requirements

Does Gleam meet them?

- A sound type system ✅
- Errors as values ✅
- Sum types & pattern matching ✅
- First class concurrency ✅
- DX as a core value ✅
- Write once, run anywhere

Write once, run anywhere

Two compilation targets, every runtime environment

- Long running processe(s) & servers - *Erlang/BEAM*
- Browser - *JavaScript*
- Edge & Serverless - *JavaScript*
- CLI & Dev Tooling - *Erlang/BEAM* or *JavaScript*

Write once, run anywhere

TypeScript++

- Easily ***bind*** to & use your ***favorite*** JS/TS libraries
- ***Generate TS types*** and ES Modules enabling use in your existing JS/TS code
- All the ***benefits*** of the JS/TS ecosystem without the ***pain***

My ideal language requirements

Does Gleam meet them?

- A sound type system ✅
- Errors as values ✅
- Sum types & pattern matching ✅
- First class concurrency ✅
- DX as a core value ✅
- Write once, run anywhere ✅

Is Gleam *perfect*?

No.

But it is ***ideal***, for ***me***, and hopefully for ***you***!

Thank you