

Gleam v1

Robert Ellen

2024/05/14

Summary

- > Gleam is a statically-typed, functional programming language for the Erlang/BEAM and JavaScript ecosystems.
- > It just went 1.0 in March 2024, and is considered stable for industrial use.
- > Gleam makes some interesting language design choices, opting for simplicity and ergonomics.
- > It's further removed from its Erlang roots more than other popular BEAM languages such as Elixir.
- > Gleam is simple without being patronising, and seems like a great choice for building small services where you want to roll up your sleeves and build things from scratch

Gleam

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

<https://gleam.run>

A functional programming language in the Erlang/BEAM ecosystem with a static type-system designed for simplicity and ergonomics. Oh, and it happens to also target JavaScript.



Gleam

```
// https://gleam.run
import gleam/io

pub fn main() {
  io.println("hello, friend!")
}
```

Gleam

- > created by Louis Pilfold (<https://github.com/lpil>)
- > functional: immutable data and HoF, but impure
- > statically-typed: Hindley-Milner type system
- > strongly-typed
- > strict semantics

Gleam

```
import gleam/io
import gleam/list

fn sum(xs: List(Int)) → Int {
  list.fold(xs, 0, fn(acc, x) { acc + x })
}

pub fn main() {
  io.debug(sum([1, 2, 3])) // prints "6"
}
```

Gleam

- > compiles to BEAM or JavaScript
- > FFI to JS and Erlang/Elixir - both at the same time with fallback
 - TS type annotations
 - can't call Elixir macros - need to wrap
- > all tooling provided by gleam
- > gleam is written in Rust
- > went v1.0 in March 2024
- > has a fantastic interactive language tour
 - <https://tour.gleam.run/>

Gleam philosophy

“Gleam has no null, no implicit conversions, no exceptions, and always performs full type checking. If the code compiles you can be reasonably confident it does not have any inconsistencies that may cause bugs or crashes.”

“Gleam lacks exceptions, macros, type classes, early returns, and a variety of other features, instead going all-in with just first-class-functions and pattern matching.”

(<https://gleam.run>)

Type-inference

```
import gleam/list
```

```
fn sum(xs) {  
  list.fold(xs, 0, fn(acc, x) { acc + x })  
}
```

```
pub fn main() {  
  sum(["a", "b", "c"])  
}
```

Type-inference

```
import gleam/list
```

```
fn sum(xs) {  
  list.fold(xs, 0, fn(acc, x) { acc + x })  
}
```

```
pub fn main() {  
  sum(["a", "b", "c"])  
}
```

```
// Produces the compile-time error ...
```

```
// 8 |      sum(["a", "b", "c"])
```

```
//   |      ^^^^^^^^^^^^^^^^^
```

```
// Expected type:
```

```
//      List(Int)
```

```
// Found type:
```

```
//      List(String)
```

Type aliases...are not a new type

```
import gleam/io

pub type SpecialString = String

pub fn main() {
  let normal: String = "I'm a string"
  let special: SpecialString = "I'm a special string"

  io.debug(normal == special) // False
}
```

Opaque types with smart constructors

```
import gleam/io

pub opaque type OpaqueString {
  OpaqueString(inner: String)
}

pub fn make(str: String) → OpaqueString {
  OpaqueString(str ◇ " (totally a string)")
}

pub fn main() {
  let normal: String = "I'm a string"
  let special: OpaqueString = make("I'm not a string")

  io.debug(normal == special) // Compile error
}
```

Data types

```
import gleam/io
import gleam/string

pub type Person {
  Person(name: String, age: Int)
}

pub fn main() {
  let ashley = Person("Ashley", 42)
  let message =
    ashley.name < " is " < string.inspect(ashley.age) < " years old"
  io.println(message)
  // "Ashley is 42 years old"
}
```

Data types

```
pub type Shape {  
    Square(side: Float)  
    Rectangle(length: Float, width: Float)  
    Circle(radius: Float)  
}  
  
pub fn area(shape: Shape) → Float {  
    case shape {  
        // note the *. operator  
        Square(side: s) → s *. s  
        Rectangle(length: l, width: w) → l *. w  
        Circle(radius: r) → 3.14 *. r *. r  
    }  
}
```

Data types - no pattern matching in function head

```
pub type Shape {  
    Square(side: Float)  
    Rectangle(length: Float, width: Float)  
    Circle(radius: Float)  
}
```

// something like this adapted from Erlang/Elixir won't compile

```
pub fn area(Square(side: s)) {  
    s *. s  
}
```

```
pub fn area(Rectangle(length: l, width: w)) {  
    l *. w  
}
```


Type variables - parametric polymorphism but not HKT

```
import gleam/io
import gleam/string

fn my_fold(collection: List(a), accumulator: b, reducer: fn(a, b) → b)
  case collection {
    [] → accumulator
    [x, ..xs] → { // expression block
      let new_acc = reducer(x, accumulator)
      my_fold(xs, new_acc, reducer)
    }
  }

pub fn main() {
  io.debug(my_fold([1, 2, 3], "", fn(a, b) { string.inspect(a) ◇ b }))
  // "321"
}
```

Type variables - in type definitions too

```
...  
pub type Option(a) {  
    Some(a)  
    None  
}  
  
...  
pub type MyEither(a, b){  
    Left(a)  
    Right(b)  
}  
  
...
```

Nil - the unit type

```
// gleam_stdlib/src/gleam/io.gleam
...
pub fn println(string: String) → Nil {
  do_println(string)
}

@external(erlang, "gleam_stdlib", "println")
@external(javascript, "../gleam_stdlib.mjs", "console_log")
fn do_println(string string: String) → Nil
...
```

todo - “top” type

```
import gleam/io
```

```
// prints "one" then errors at two()
```

```
pub fn main() {  
  one()  
  two()  
  three()  
}
```

```
pub fn one() {  
  io.println("one")  
}
```

```
pub fn two() {  
  todo as "two() is not implemented yet!"  
}
```

use - a monad if you squint

```
import gleam/io
import gleam/list
import gleam/string
```

```
fn upcase_duplicate(strings) {
  use str <- list.map(strings)
  let dup = string.append(to: str, suffix: str)
  string.uppercase(dup)
}
```

```
pub fn main() {
  ["hello", "world!! "]
  ▷ upcase_duplicate()
  ▷ io.debug()
}
```

use - a monad if you squint

```
import gleam/io
import gleam/result
import gleam/string

pub fn main() {
  let res = {
    use data <- result.try(read_data())
    use record <- result.map(find_record(data))
    format(record)
  }

  case res {
    Ok(formatted) → io.println(formatted)
    Error(error) → io.println("ERROR: " ◇ error)
  }
}
```

use - a monad if you squint

```
fn read_data() {  
    Ok(("#Ashley", 42))  
}  
  
fn find_record(record) {  
    Ok(record)  
}
```

use - a monad if you squint

```
import gleam/io
import gleam/result
import gleam/string

// prints "#("Ashley", 42)"
pub fn main() {
  let res = {
    use data <- result.try(read_data())
    use record <- result.map(find_record(data))
    format(record)
  }

  case res {
    Ok(formatted) → io.println(formatted)
    Error(error) → io.println("ERROR: " ◇ error)
  }
}
```


use - a monad if you squint

```
fn read_data() {  
    Error("couldn't read data")  
}
```

```
fn find_record(record) {  
    Error("not found")  
}
```

```
fn format(record) {  
    string.inspect(record)  
}
```

use - a monad if you squint

```
import gleam/io
import gleam/result
import gleam/string

// prints "ERROR: couldn't read data"
pub fn main() {
  let res = {
    use data <- result.try(read_data())
    use record <- result.map(find_record(data))
    format(record)
  }

  case res {
    Ok(formatted) → io.println(formatted)
    Error(error) → io.println("ERROR: " ◇ error)
  }
}
```

No type classes, traits, or interfaces

<https://mckayla.blog/posts/all-you-need-is-data-and-functions.html>

```
import gleam/io
```

```
pub type MyDebug(a) {  
  MyDebug(debug: fn(a) → String)  
}
```

```
pub fn main() {  
  let my_dbg = MyDebug(debug: fn(str) { io.debug("The string is: " <math>\diamond</math> st  
  my_dbg.debug("Hello, World!!!")  
}
```

Some gleam libraries - no magic

- > Lustre: Elm-inspired frontend framework
 - <https://github.com/lustre-labs/lustre>
- > cgi: CGI in gleam
 - <https://github.com/lpil/cgi>
- > wisp: web framework
 - <https://github.com/gleam-wisp/wisp>

OTP concurrency

Gleam being statically-typed does not wrap OTP's concurrency entities wholesale like Elixir. Instead providing an OTP-interoperable library built using basic BEAM primitives.

- > <https://github.com/gleam-lang/otp>
- > Actor hierarchy
 - Process - a wrapper around BEAM processes, all other actors based on process
 - Actor - like a `gen_server`, receives messages, updates state
 - Task - run a function and quit
 - Supervisor - manage other processes, provides fault-tolerance

OTP and BEAM ecosystem

- > there are several other libraries that wrap OTP features, like ETS
- > if something from the OTP libraries can be made type-safe, it's an FFI away

Where to use Gleam

- > Gleam is simple without being patronising, no magic
- > seems like a great choice for building small services
- > you'll have to roll up your sleeves and build more things from scratch
- > small-to-medium web apps with less CRUD
- > isomorphic apps (share common types across FE and BE packages)
- > learning static-typing, polymorphism, immutable data structures

Thank you

I'll post slides soon.