Functional programming with Gleam

What if best practices were actually the norm?

Who am I

👨🏻‍💻 Computer Science graduate at Unibo

Who am I

🧑🏻‍💻 Computer Science graduate at Unibo

💖 Functional Programming enthusiast

Who am I

👨🏻‍💻 Computer Science graduate at Unibo

💖 Functional Programming enthusiast

✨ Lately doing a bunch of Gleam

Functional programming with Gleam

What if best practices were actually the norm?

Language shapes the way we think, and determines what we can think about

- Benjamin Lee Whorf

Simply Reliable

After 2 years and 200’000 lines of production Elm code, we got our first production runtime exception.

In that period, our legacy JavaScript code has crashed a mere 60’000 times.

- Richard Feldman, Head of Technology at noredink

Fearless Refactoring

Messenger used to receive bugs reports on a daily basis; since the introduction of Reason, there have been a total of 10 bugs (that's during the whole year, not per week)!

Refactoring speed went from days to hours to dozens of ***minutes****.*

- From the REason language blog

RE

Pits of Success

- Simplicity
- No runtime exceptions
- No null values
- Structural equality
- Immutable data
- Great developer experience

Pits of Success

- Simplicity
- No runtime exceptions
- No null values
- Structural equality
- Immutable data
- Great developer experience

Simplicity lets you spend less time thinking about **how to approach a problem** and more time focused on **what the solution is.**

Your first Gleam program

```
import gleam/io

pub type Pet {
  Cat
  Dog
}

pub fn speak(pet: Pet) -> String {
  case pet {
    Cat -> “meow”
    Dog -> “woof”
   }
}

pub fn main() {
  let my_pet = Dog
  io.println(speak(my_pet))
}
```

Your first Gleam program

```
import gleam/io

pub type Pet {
  Cat
  Dog
}

pub fn speak(pet: Pet) -> String {
  case pet {
    Cat -> “meow”
    Dog -> “woof”
  }
}

pub fn main() {
  let my_pet = Dog
  io.println(speak(my_pet))
}
```

Here we enumerate all the possible variants of a Pet: in our program a pet can only be a Cat or a Dog

Your first Gleam program

```
import gleam/io

pub type Pet {
  Cat
  Dog
}

pub fn speak(pet: Pet) -> String {
  case pet {
    Cat -> “meow”
    Dog -> “woof”
   }
}

pub fn main() {
  let my_pet = Dog
  io.println(speak(my_pet))
}
```

The speak function takes a Pet as input and returns a String

Your first Gleam program

```
import gleam/io

pub type Pet {
  Cat
  Dog
}

pub fn speak(pet: Pet) -> String {
  case pet {
    Cat -> “meow”
    Dog -> “woof”
  }
}

pub fn main() {
  let my_pet = Dog
  io.println(speak(my_pet))
}
```

Thanks to **pattern matching** we can tell if a Pet is a Cat or a Dog

Your first Gleam program

```
import gleam/io

pub type Pet {
  Cat
  Dog
}

pub fn speak(pet: Pet) -> String {
  case pet {
    Cat -> “meow”
    Dog -> “woof”
  }
}

pub fn main() {
  let my_pet = Dog
  io.println(speak(my_pet))
}
```

No need to add type annotations: the compiler will **always infer the correct types** for your program

Your first Gleam program

```
import gleam/io

pub type Pet {
  Cat
  Dog
}

pub fn speak(pet: Pet) -> String {
  case pet {
    Cat -> “meow”
    Dog -> “woof”
  }
}

pub fn main() {
  let my_pet = Dog
  io.println(speak(my_pet))
}
```

```java
import java.util.Objects;

interface Pet {
    String speak();
}

class Cat implements Pet {
    @Override
    public String speak() {
        return “woof”;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null ||
            this.getClass() != obj.getClass())
            return false;
        return true;
    }

    @Override
    public int hashCode() {
        return Objects.hash(this.toString());
    }
}
```

```java
class Dog implements Pet {
    @Override
    public String speak() {
        return “meow”;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null &&
            this.getClass() != obj.getClass())
            return false;
        return true;
    }

    @Override
    public int hashCode() {
        return Objects.hash(this.toString());
    }
}

public class Main {
    public static void main(String[] args) {
        Pet myPet = new Dog();
        System.out.println(myPet.speak());
    }
}
```

```
public interface Pet {
  String speak();
}

record Cat() implements Pet {
    @Override
    public String speak() {
        return “meow”;
    }
}

record Dog() implements Pet {
    @Override
    public String speak() {
        return “woof”;
    }
}

void main() {
    val myPet = new Dog();
    System.out.println(myPet.speak());
}
```

```
public interface Pet {
  String speak();
}

record Cat() implements Pet {
    @Override
    public String speak() {
        return “meow”;
    }
}

record Dog() implements Pet {
    @Override
    public String speak() {
        return “woof”;
    }
}

void main() {
    val myPet = new Dog();
    System.out.println(myPet.speak());
}
```

Released in Java 16, **record classes** greatly reduce the pain of defining new data structures and make the code more readable

```java
public interface Pet {
  String speak();
}

record Cat() implements Pet {
    @Override
    public String speak() {
        return “meow”;
    }
}

record Dog() implements Pet {
    @Override
    public String speak() {
        return “woof”;
    }
}

void main() {
    val myPet = new Dog();
    System.out.println(myPet.speak());
}
```

Starting from Java 21, **unnamed classes** reduce the ceremonies needed to define the program’s entry point

Pits of Success

- Simplicity
- **No runtime exceptions**
- **No null values**
- Structural equality
- Immutable data
- Great developer experience

The compiler **forces you to be explicit** about the behaviour of your functions: if something can fail you must handle it

It’s all about honesty

```
class User {
  public final int id;
  public final String name;
  // …
}

class Users {
  static User load(int id) { … }
}

val user = Users.load(1)
System.out.println(user.name)
```

It’s all about honesty

```
class Users {
  static User load(int id) { … }
}

val user = Users.load(1)
System.out.println(user.name)
```

It’s all about honesty

```
class Users {
  static User load(int id) { … }
}

val user = Users.load(1)
// User(id: 1, name: “Rob”)
System.out.println(user.name)
// “Rob”
```

We got what we expected, so far so good...

It’s all about honesty

```
class Users {
  static User load(int id) { … }
}

val user = Users.load(2)
// null
System.out.println(user.name)
// 💥 NullPointerException
```

Null pointer references: **the billion dollar mistake**

It’s all about honesty

```
class Users {
  static User load(int id) { … }
}

val user = Users.load(2)
if (user ≠ null) {
  System.out.println(user.name)
} else {
  System.out.println(“user not found?”)
}
```

Defensive programming! We always have to be on the lookout

It’s all about honesty

```
class Users {
  static User load(int id) { … }
}

val user = Users.load(2)
// null
if (user ≠ null) {
  System.out.println(user.name)
} else {
  System.out.println(“user not found?”)
// “user not found?”
}
```

It’s all about honesty

```
class Users {
  static User load(int id) { … }
}

val user = Users.load(2)
// 💥 Runtime exception: no user found
if (user ≠ null) {
  System.out.println(user.name)
} else {
  System.out.println(“user not found?”)
}
```

I give up!

It’s all about honesty

```
class Users {
  static User load(int id) { … }
}

try {
  val user = Users.load(2)
  if (user ≠ null) {
    System.out.println(user.name)
  } else {
    System.out.println(“user not found?”)
  }
} catch (final UserNotFoundException e) {
  System.out.println(e.toString())
}
```

We want errors to happen at compile time, in front of a developer, instead of runtime, in front of a user

It’s all about honesty

```
pub type User {
  User(id: Int, name: String)
}

pub fn load_user(id: Int) { … }

let user = load_user(1)
io.println(user.name)
```

It’s all about honesty

```
pub type User {
  User(id: Int, name: String)
}

pub fn load_user(id: Int) { … }

let user = load_user(1)
io.println(user.name)
//             ^^^^^
// This field does not exist.
// The value being accessed has this type:
//
//     Result(User, LoadError)
//
// It does not have any fields.
```

The compiler won’t let us do this because it knows it is unsafe

It’s all about honesty

```
pub fn load_user(id: Int) { … }

case load_user(1) {
  Ok(user) ->
    io.println(user.name)

  Error(UserNotFound) ->
    io.println(“No user with id 1”)

  Error(ConnectionError) ->
    io.println(“Connection error”)
}
```

It’s all about honesty

```
pub fn load_user(id: Int) { … }

case load_user(1) {
  Ok(user) →
    io.println(user.name)

  Error(UserNotFound) →
    io.println(“No user with id 1”)

  Error(ConnectionError) →
    io.println(“Connection error”)
}
```

If everything went well we can access the user’s name

It’s all about honesty

```
pub fn load_user(id: Int) { … }

case load_user(1) {
  Ok(user) ->
    io.println(user.name)

  Error(UserNotFound) ->
    io.println(“No user with id 1”)

  Error(ConnectionError) ->
    io.println(“Connection error”)
}
```

The compiler forces us to **explicitly deal** with the errors that may have occurred

```
pub type LoadError {
  UserNotFound
  ConnectionError
}
```

Pits of Success

- Simplicity
- No runtime exceptions
- No null values
- **Structural equality**
- Immutable data
- Great developer experience

Complete and utter chaos

```
1 == 1
// -> true
```

Complete and utter chaos

```
1 == 1
// -> true

Integer.valueOf(“1024”) == Integer.valueOf(“1024”)
// -> false
```

This is not what one would expect intuitively. It disregards the **principle of least astonishment**

Complete and utter chaos

```
1 == 1
// → true

Integer.valueOf(“1024”) == Integer.valueOf(“1024”)
// → false

Integer.valueOf(“1”) == Integer.valueOf(“1”)
```

Complete and utter chaos

```
1 == 1
// → true

Integer.valueOf(“1024”) == Integer.valueOf(“1024”)
// → false

Integer.valueOf(“1”) == Integer.valueOf(“1”)
// → true
```

Complete and utter chaos

```
1 == 1
// -> true

Integer.valueOf(“1024”) == Integer.valueOf(“1024”)
// -> false

Integer.valueOf(“1”) == Integer.valueOf(“1”)
// -> true

“I’m an object” == “I’m an object”
```

Complete and utter chaos

```
1 == 1
// -> true

Integer.valueOf(“1024”) == Integer.valueOf(“1024”)
// -> false

Integer.valueOf(“1”) == Integer.valueOf(“1”)
// -> true

“I’m an object” == “I’m an object”
// -> true
```

Two things are equal when…

Two things are equal when…
they have the same structure!

Structural equality

```
1 == 1
// -> true
```

Structural equality

```
1 == 1
// -> true

int.parse(“1024”) == int.parse(“1024”)
// -> true
```

Structural equality

```
1 == 1
// -> true

int.parse(“1024”) == int.parse(“1024”)
// -> true

Dog == Dog
// -> true
```

Structural equality

```
1 == 1
// -> true

int.parse(“1024”) == int.parse(“1024”)
// -> true

Dog == Dog
// -> true

[1, 2, 3] == [1, 2, 3]
// -> true
```

Pits of Success

- Simplicity
- No runtime exceptions
- No null values
- Structural equality
- **Immutable data**
- Great developer experience

Always on the lookout!

```
public final class User {
    private final String name;
    private final Date birthday;

    public User(String name, Date birthday) {
        this.name = Objects.requireNonNull(name);
        this.birthday = new Date(birthday.getTime());
    }

    public Date getBirthday() {
        return new Date(this.birthday);
    }
}
```

Always on the lookout!

```
public final class User {
    private final String name;
    private final Date birthday;

    public User(String name, Date birthday) {
        this.name = Objects.requireNonNull(name);
        this.birthday = new Date(birthday.getTime());
    }

    public Date getBirthday() {
        return new Date(this.birthday);
    }
}
```

We can't **trust** that an object will never change or be changed, so we have to remember to make **defensive copies!**

Always on the lookout!

```
var birthday = new Date();

var ben = new User(“Ben”, birthday);
var rob = new User(“Rob”, birthday);
```

Can we really trust that User is never going to change that date? **Is it safe to share it?**

Immutability gives us peace of mind that **things are not going to change unexpectedly** under our feet!

Some peace of mind

```
let birthday = date.new()

let ben = User(“Ben”, birthday)
let rob = User(“Rob”, birthday)
```

Can we really trust that User is never going to change that date? **Is it safe to share it?**

Some peace of mind

```
let birthday = date.new()

let ben = User(“Ben”, birthday)
let rob = User(“Rob”, birthday)
```

Can we really trust that User is never going to change that date? **Is it safe to share it?**

Yes!

***The Free Lunch Is Over.
The biggest sea change in software development since the OO revolution is knocking at the door, and its name is Concurrency***

- Herb Sutter

Pits of Success

- Simplicity
- No runtime exceptions
- No null values
- Structural equality
- Immutable data
- **Great developer experience**

The compiler is here to help

```
error: Unknown record field

  ┌─ ./src/app.gleam:4:16
  │
4 │ io.println(user.nam)
  │                ^^^^ Did you mean `name`?

The value being accessed has this type:
    User

It has these fields:
    .name
    .status
```

The compiler is here to help

```
error: Type mismatch

  ┌─ ./src/app.gleam:8:22
  │
8 │ let numbers = [1, 2, “3”]
  │                      ^^^
  │

All elements of a list must be the same type,
but this one doesn’t match the one before it.

Expected type:
    Int

Found type:
    String
```

Time for a live demo!

There’s a lot more to it!

⚙️ Compiles both to **Erlang** and **JavaScript**

There’s a lot more to it!

⚙️ Compiles both to **Erlang** and **JavaScript**

✨ Great fit for building **rich and interactive front-end applications**

There’s a lot more to it!

⚙️ Compiles both to **Erlang** and **JavaScript**

✨ Great fit for building **rich and interactive front-end applications**

📈 Multi-core actor based concurrency system that can run **millions of lightweight, concurrent tasks**

There's a lot more to it!

⚙️ Compiles both to **Erlang** and **JavaScript**

✨ Great fit for building **rich and interactive front-end applications**

📈 Multi-core actor based concurrency system that can run **millions of lightweight, concurrent tasks**

💜 **Great community** full of lovely people

Get in touch!

Join the Gleam community on Discord

discord.gg/Fm8Pwmy

Learn Gleam on Exercism

The best way to start your journey with Functional Programming

exercism.org/tracks/gleam

Ping me anytime!

✉️ *giacomo.cavalieri@icloud.com*

@giacomo_cava

giacomocavalieri

Questions?

Thanks for listening!

And a huge thank you to Hayleigh Thompson for letting me use her slides template!