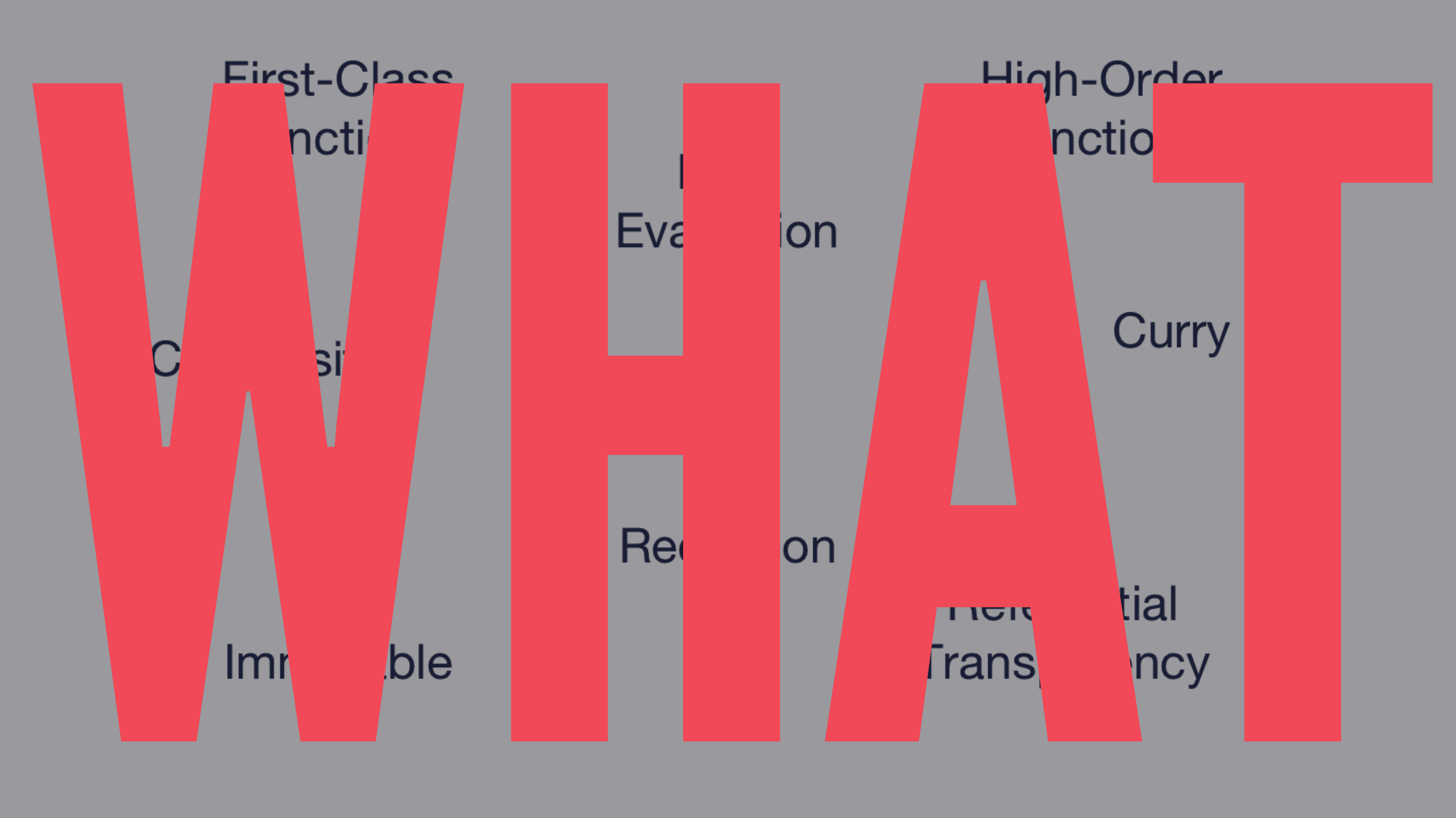
SWIFT
函数式编程
实践

@HANKBAO

瀑布IM

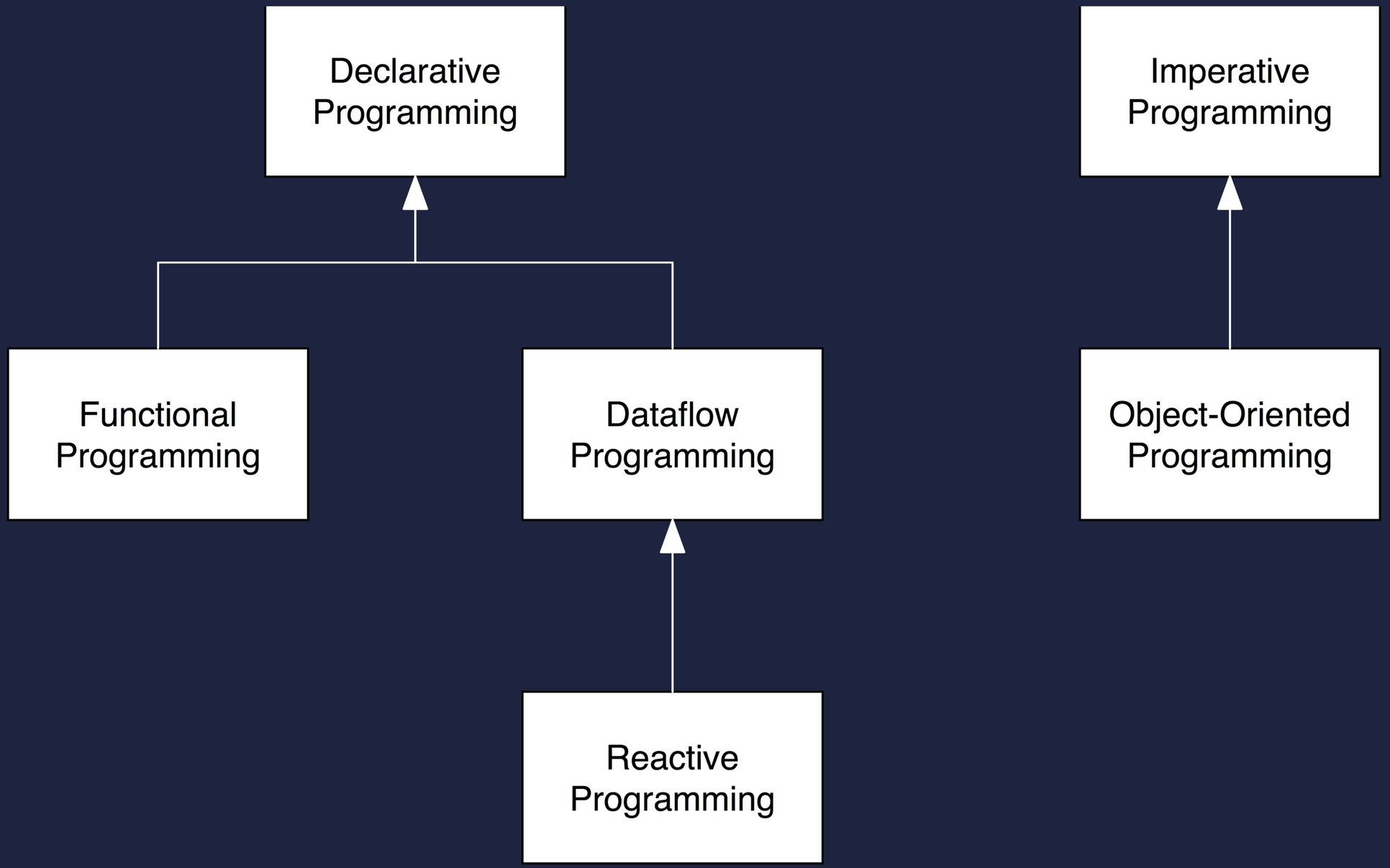

First-Class
nctio
High-Order
nctio
Eva ion
Curry
C si
Re on
tial
Trans ncy
Imm ble
WHAT

WHAT

Functional programming is a programming *paradigm*
1. treats computation as the evaluation of mathematical functions
2. avoids changing-state and mutable data

– Wikipedia

PARADIGM


Declarative Programming
Imperative Programming
Functional Programming
Dataflow Programming
Object-Oriented Programming
Reactive Programming

THINK

IMPERATIVE: MACHINE

```
let nums = [1, 2, 3, 4, 5, 6, 7]

var strs = [String]()
for var i = 0; i < nums.count; ++i {
    strs.append(String(nums[i]))
}
```

DECLARATIVE: MATHEMATICS

```
let nums = [1, 2, 3, 4, 5, 6, 7]
let strs = nums.map(String.init)
```

WHY


CURRY

CURRY

```
func x(a: A, b: B, c: C) -> E

func x(a: A) -> (b: B) -> (c: C) -> E
```

CURRY

```
struct User {
    func login(password: String)
}

let passwd = "@Swift"
let usr = User()

usr.login(passwd)
```

CURRY

```
struct User {
    func login(password: String)
}

let passwd = "@Swift"
let usr = User()

User.login(usr)(passwd)
```

CURRY

usr.login(passwd)

||

User.login(usr)(passwd)

CURRY IN PRACTICE

```
struct User {
    func name() -> String
}

let collation: UILocalizedIndexedCollation = ...

let sorted = collation.sortedArrayFromArray(users,
                collationStringSelector: "name")
```

CURRY IN PRACTICE

```
class Wrapper<T>: NSObject {
    let payload: T
    let localization: (T) -> () -> String

    @objc func localizable() -> NSString {
        return localization(payload)()
    }

    static var selector: Selector {
        return "localizable"
    }
}
```

CURRY IN PRACTICE

```
let wrappers = users.map {
    Wrapper(payload: $0, localization: User.name)
}

let sorted = collation.sortedArrayFromArray(wrappers,
      collationStringSelector: Wrapper<User>.selector)
```

FUNCTIONAL ABSTRACTION

OPTIONAL

```
enum Optional<T> {
   case None
   case Some(T)
}
```

OPTIONAL

```
func map<U>(f: T -> U) -> U?
func flatMap<U>(f: T -> U?) -> U?

let date: NSDate? = ...
let formatter: NSDateFormatter = ...

let dateString = date.map(formatter.stringFromDate)
```

ARRAY

```
func map<T>(t: Self.Generator.Element -> T) -> [T]
func flatMap<S: SequenceType>
        (t: Self.Generator.Element -> S) -> [S.Generator.Element]
```

MONAD<?>

MONAD<ASYNC>

PROMISE

```
class Promise<T> {
    func then<U>(body: T -> U) -> Promise<U>
    func then<U>(body: T -> Promise<U>) -> Promise<U>
}
```

PROMISE

```
class Promise<T> {
    func map<U>(body: T -> U) -> Promise<U>
    func flatMap<U>(body: T -> Promise<U>) -> Promise<U>
}
```

OBSERVABLE

```
class Observable<T> {
    func map<U>(body: T -> U) -> Observable<U>
    func flatMap<U>(body: T -> Observable<U>) -> Observable<U>
}
```

MONAD IN PRACTICE

ASYNC CALLBACK

(value: T?, error: ErrorType?) -> Void

ASYNC CALLBACK

```
(value: T?, error: ErrorType?) -> Void

if let error = error {
    // handle error
} else if let value = value {
    // handle value
} else {
    // all nil?
}

// all non-nil?!
```

RESULT

```
enum Result<Value> {
    case Failure(ErrorType)
    case Success(Value)
}
```

RESULT

```
(result: Result<T>) -> Void

switch result {
case let .Error(error):
    // handle error
case let .Success(value):
    // handle value
}
```

RESULT

```
enum Result<Value> {
    func map<T>(...) -> Result<T> {
        ...
    }

    func flatMap<T>(...) -> Result<T> {
        ...
    }
}
```

RESULT

```
func flatMap<T>(@noescape transform: Value throws -> Result<T>) rethrows -> Result<T> {
    switch self {
    case let .Failure(error):
        return .Failure(error)

    case let .Success(value):
        return try transform(value)
    }
}

func map<T>(@noescape transform: Value throws -> T) rethrows -> Result<T> {
    return try flatMap { .Success(try transform($0)) }
}
```

RESULT

```
func toImage(data: NSData) -> Result<UIImage>
func addAlpha(image: UIImage) -> Result<UIImage>
func roundCorner(image: UIImage) -> Result<UIImage>
func applyBlur(image: UIImage) -> Result<UIImage>
```

RESULT

```
toImage(data)
   .flatMap(addAlpha)
   .flatMap(roundCorner)
   .flatMap(applyBlur)
```

REFERENCE

- Wikipedia
- Haskell Wiki
- Functional Programming in Swift
- objc.io

THANKS

Q & A