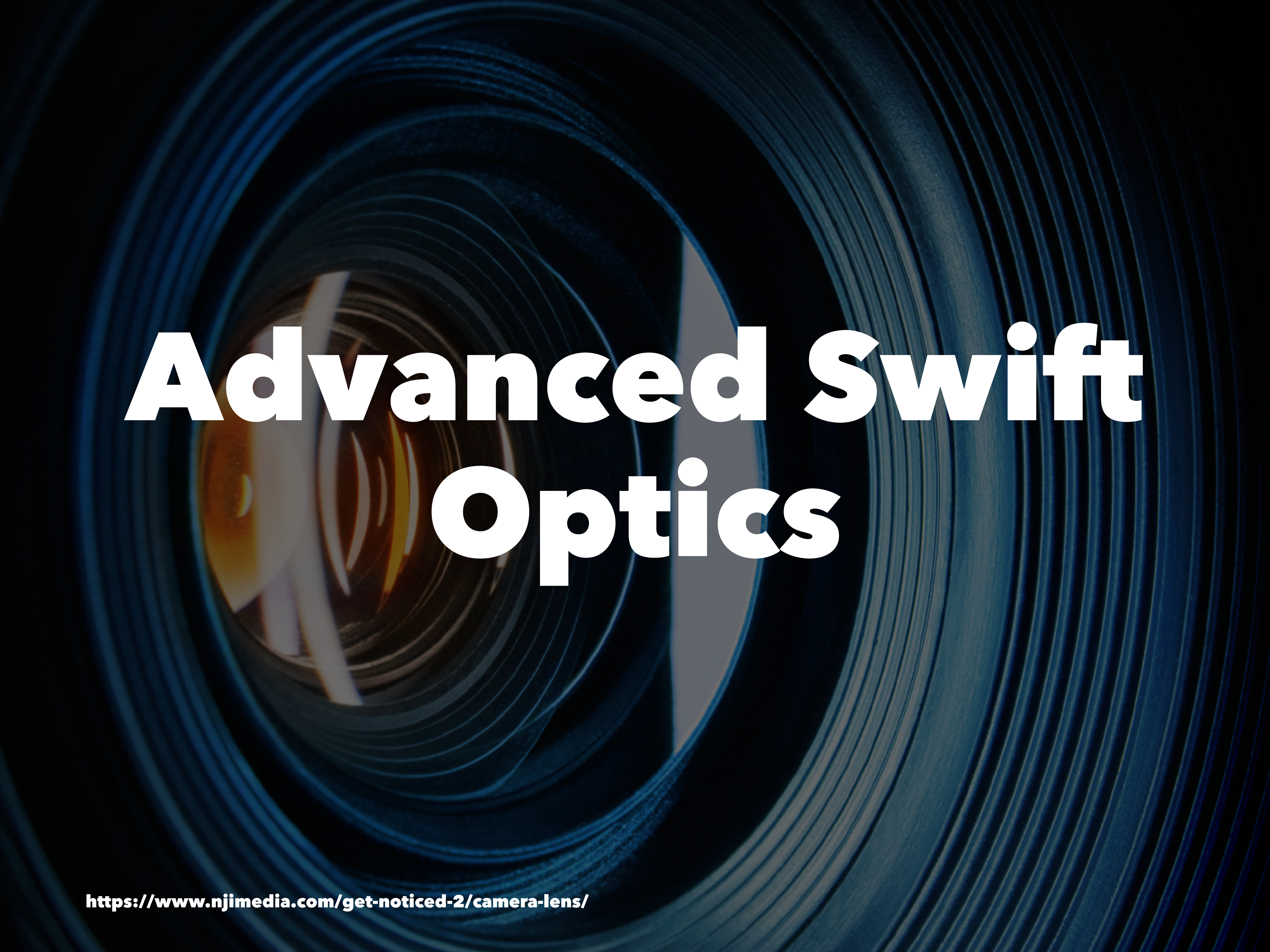

Advanced Swift
Optics
https://www.njimedia.com/get-noticed-2/camera-lens/

Today's menu

Today's menu

- Functional Programming

Today's menu

- Functional Programming
  - The value of data structures

Today's menu

- Functional Programming
  - The value of data structures
- Functional Optics

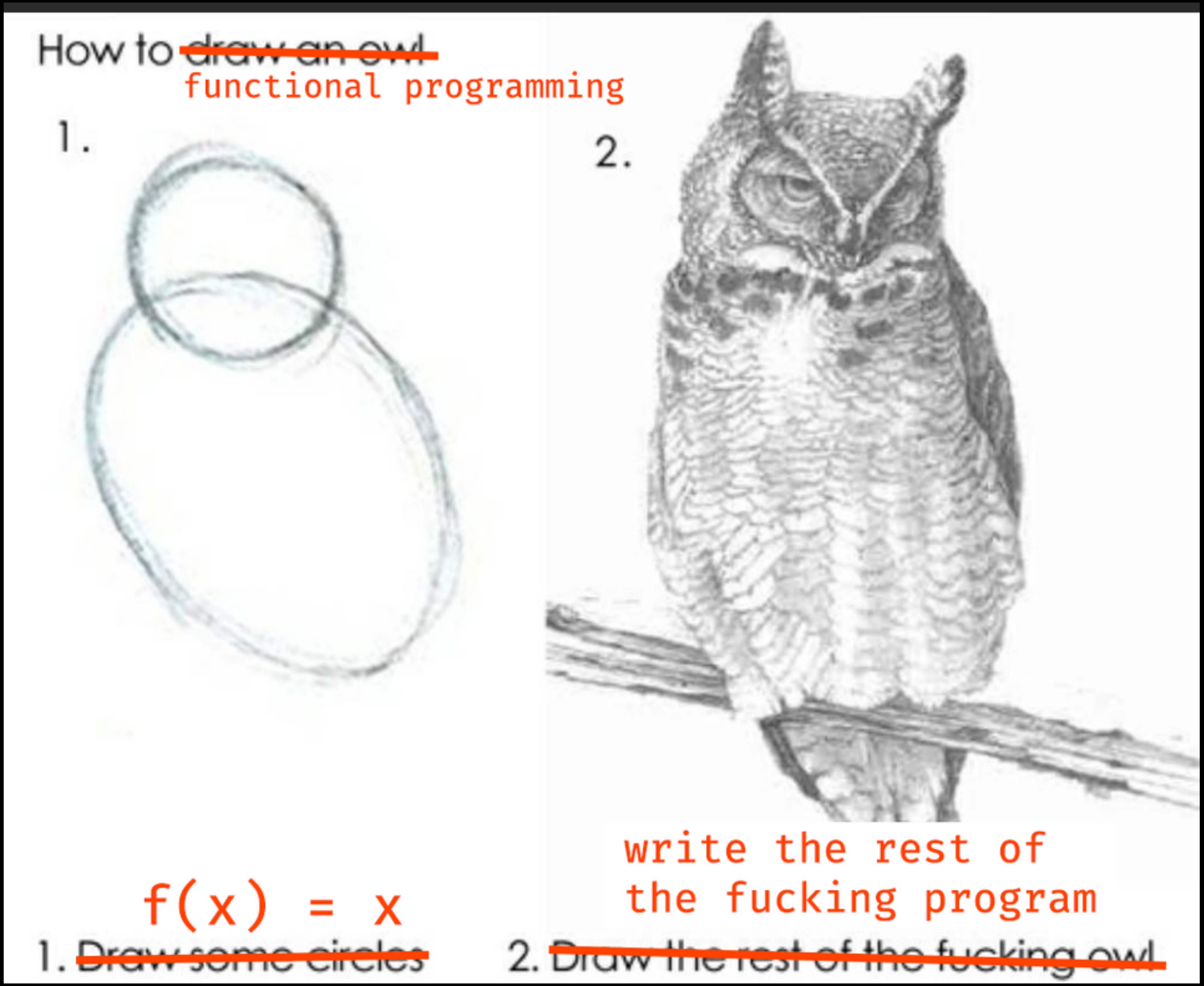
Today's menu

- Functional Programming
  - The value of data structures
- Functional Optics
  - Sane data manipulation

Today's menu

- Functional Programming
  - The value of data structures
- Functional Optics
  - Sane data manipulation
- Lots of code

Functional Programming



https://www.reddit.com/r/ProgrammerHumor/comments/6d3eqh/functional*programming*for_beginners/

John Ⓐ De Goes
@jdegoes

Segui

FP is just programming with functions. Functions are:

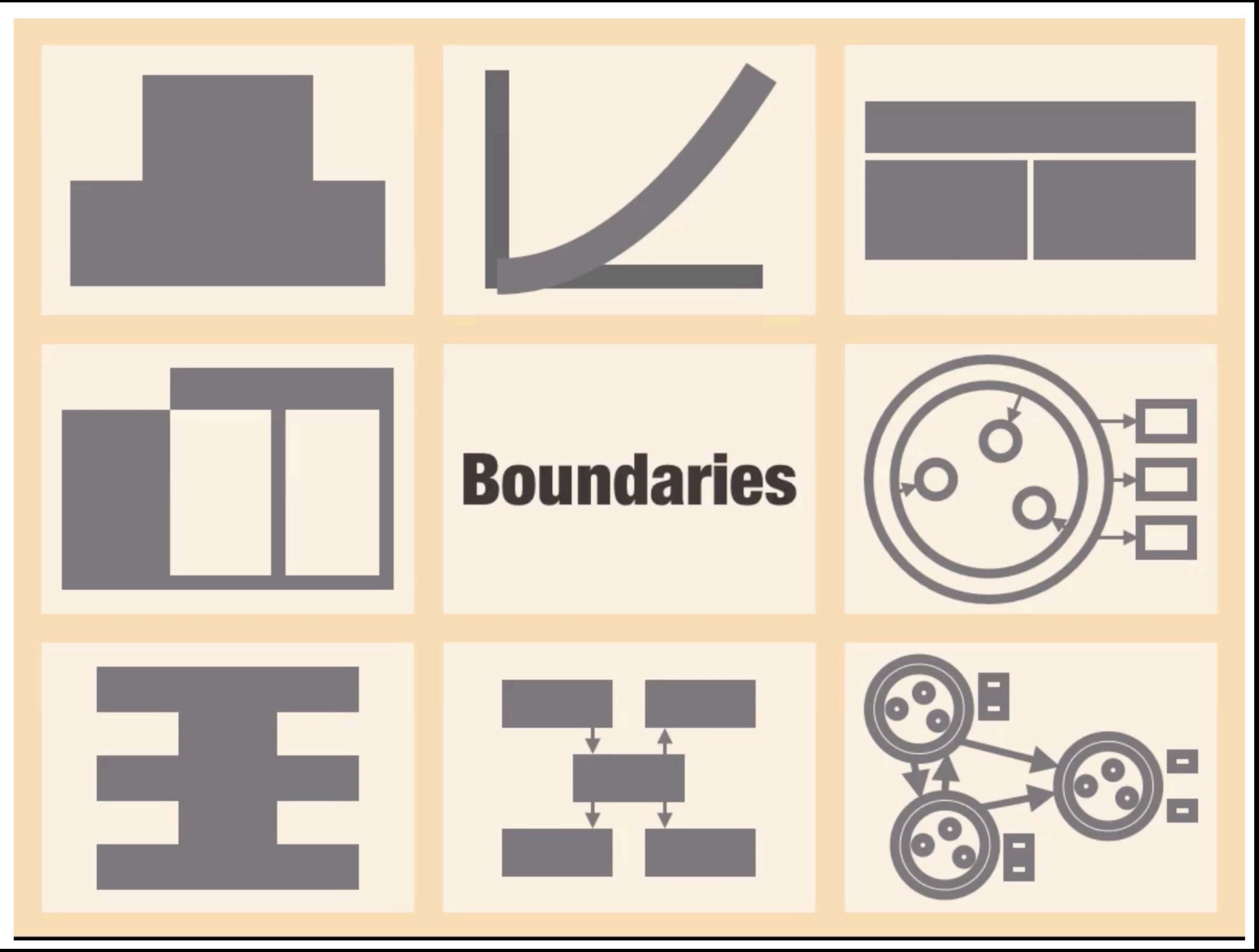
1. Total: They return an output for every input.
2. Deterministic: They return the same output for the same input.
3. Pure: Their only effect is computing the output.

The rest is just composition you can learn over time.

10:32 - 30 nov 2017

https://twitter.com/jdegoes/status/936301872066977792

Gary Bernhardt - "Boundaries" (2012)



https://www.destroyallsoftware.com/talks/boundaries

Pure Functions

Pure Functions

- they can only transform data

Pure Functions

- they can only transform data
- input: one of more instances of some types

Pure Functions

- they can only transform data
- input: one of more instances of some types
- output: the same

Pure Functions

- they can only transform data
- input: one of more instances of some types
- output: the same
- the data involved is as important as the functions themselves

Show me your flowcharts and conceal your tables, and I shall continue to be mystified. Show me your tables, and I won't usually need your flowcharts; they'll be obvious.

Fred Brooks, The Mythical Man-Month

Immutable Data Structures

Immutable Data Structures

- value types

Immutable Data Structures

- value types
- `struct`, `enum`

Immutable Data Structures

- value types
- `struct`, `enum`
- domain modeling

Immutable Data Structures

- value types
- `struct`, `enum`
- domain modeling
- UI modeling

Immutable Data Structures

- value types
- `struct`, enum
- domain modeling
- UI modeling
- transitional information

Immutable Data Structures

- value types
- `struct`, `enum`
- domain modeling
- UI modeling
- transitional information
- intent representation

```swift
func triggerAction(
    data: ActionData,
    listener: ListenerType)
{
    /// depending on the current state, the data value
    /// and some rules, call methods on the listener
}
```

```
func triggerAction(
    data: ActionData,
    listener: ListenerType)
{
    /// depending on the current state, the data value
    /// and some rules, call methods on the listener
}

enum ActionResult {
    case doThis(value: Int)
    case doThat(value: String)
}

func triggerAction(data: ActionData) -> ActionResult {
    /// depending on the current state, the data value
    /// and some rules, return a specific ActionResult
}
```

```swift
extension Int {
    var isPositive: Bool {
        get {
            return self >= 0
        }
        set(shouldBePositive) {
            if isPositive != shouldBePositive {
                self = -self
            }
        }
    }
}
```

Data structures bring us joy…

Data structures bring us joy…

- separation of *representation* from *interpretation*

Data structures bring us joy…

- separation of *representation* from *interpretation*
- the former at the core, the latter at the boundary

Data structures bring us joy…

- separation of *representation* from *interpretation*
- the former at the core, the latter at the boundary
- readable, testable, maintainable

Data structures bring us joy…

- separation of *representation* from *interpretation*
- the former at the core, the latter at the boundary
- readable, testable, maintainable
- not just for functional programming

…and pain

...and pain

- they can get messy and complex

…and pain

- they can get messy and complex
- lots of nesting

…and pain

- they can get messy and complex
- lots of nesting
- we usually operate on tiny parts…

…and pain

- they can get messy and complex
- lots of nesting
- we usually operate on tiny parts…
- …but we need to transform the whole

Optics

Optics

- types (usually structs)

Optics

- types (usually structs)
- encapsulate various kinds of relationships between data structures

Optics

- types (usually structs)
- encapsulate various kinds of relationships between data structures
- for example, "focusing":

Optics

- types (usually structs)
- encapsulate various kinds of relationships between data structures
- for example, "focusing":
    - relationship between a data structure and one of its parts

Optics

- types (usually structs)
- encapsulate various kinds of relationships between data structures
- for example, "focusing":
  - relationship between a data structure and one of its parts
- see data from a different point of view

A note on naming

http://www.cs.ox.ac.uk/people/jeremy.gibbons/publications/poptics.pdf | https://github.com/hablapps/DontFearTheProfunctorOptics

A note on naming

- different sources and different platforms use different names

http://www.cs.ox.ac.uk/people/jeremy.gibbons/publications/poptics.pdf | https://github.com/hablapps/DontFearTheProfunctorOptics

A note on naming

- different sources and different platforms use different names
- "Profunctor Optics - Modular Data Accessors" by Matthew Pickering, Jeremy Gibbons, and Nicolas Wu

http://www.cs.ox.ac.uk/people/jeremy.gibbons/publications/poptics.pdf | https://github.com/hablapps/DontFearTheProfunctorOptics

A note on naming

- different sources and different platforms use different names
- "Profunctor Optics - Modular Data Accessors" by Matthew Pickering, Jeremy Gibbons, and Nicolas Wu
- "Don't Fear the Profunctor Optics!" GitHub repo

http://www.cs.ox.ac.uk/people/jeremy.gibbons/publications/poptics.pdf | https://github.com/hablapps/DontFearTheProfunctorOptics

Optic<Root, Value>

Optic<Root, Value>

- like KeyPath (not a coincidence)

Optic<Root, Value>

- like KeyPath (not a coincidence)
- simplified representation

Optic<Root, Value>

- like KeyPath (not a coincidence)
- simplified representation
- full representations has 2 more parameters:

Optic<Root, Value>

- like `KeyPath` (not a coincidence)
- simplified representation
- full representations has 2 more parameters:
    - generic change on both `Root` and `Value`

Lens

```swift
struct Lens<Root, Value> {
    let view: (Root) -> Value
    let update: (Value, Root) -> Root
}
```

```swift
func makeLens<Root, Value>(
    _ wkp: WritableKeyPath<Root, Value>)
    -> Lens<Root, Value>
{
    return Lens<Root, Value>(
        view: { root in
            root[keyPath: wkp]
        },
        update: { newValue, root in
            var m_root = root
            m_root[keyPath: wkp] = newValue
            return m_root
    })
}
```

```swift
extension Lens {
    func modify (
        _ transformValue: @escaping (Value) -> Value)
        -> (Root) -> Root
    {
        return { root in
            self.update(
                transformValue(self.view(root)),
                root)
        }
    }
}
```

```swift
struct LoginPage {
    var username: String
    var password: String
    var isRememberMeActive: Bool
    var isLoginButtonActive: Bool
}
```

```swift
extension LoginPage {
    static func lens<Value>(
        _ wkp: WritableKeyPath<LoginPage, Value>)
        -> Lens<LoginPage, Value>
    {
        return makeLens(wkp)
    }
}

let passwordLens = LoginPage.lens(\.password)

/// trimPassword: (LoginPage) -> LoginPage
let trimPassword = passwordLens.modify {
    $0.trimmingCharacters(
        in: CharacterSet(charactersIn: " "))
}
```

```swift
func zip<Root, Value1, Value2>(
    _ lens1: Lens<Root, Value1>,
    _ lens2: Lens<Root, Value2>)
    -> Lens<Root, (Value1, Value2)>
{
    return Lens<Root, (Value1, Value2)>(
        view: { root in
            (lens1.view(root), lens2.view(root))
    },
        update: { tuple, root in
            lens2.update(
                tuple.1,
                lens1.update(
                    tuple.0,
                    root))
    })
}
```

```swift
struct Application {
    var loginPage: LoginPage
    var userSession: UserSession
}

struct UserSession {
    var token: String?
    var currentUsername: String?
}
```

```swift
/// storedUsernameLens:
/// Lens<Application, (String?, String, Bool)>
let storedUsernameLens = zip(
    Application.lens(\.userSession.currentUsername),
    Application.lens(\.loginPage.username),
    Application.lens(\.loginPage.isRememberMeActive)
)

/// restoreUsername: (Application) -> Application
let restoreUsername = storedUsernameLens.modify {
    current, _, rememberMe in
    guard let current = current, rememberMe else {
        return (nil, "", rememberMe)
    }

    return (current, current, rememberMe)
}
```

```swift
extension Dictionary {
    static func lens(
        at key: Key)
        -> Lens<Dictionary, Value?>
    {
        return Lens<Dictionary, Value?>(
            view: { $0[key] },
            update: { value, root in
                var m_root = root
                m_root[key] = value
                return m_root
        })
    }
}
```

Prism

```swift
struct Prism<Root, Value> {
    let match: (Root) -> Value?
    let build: (Value) -> Root
}
```

```swift
enum LoginState {
    case idle
    case processing(attempt: Int)
    case failed(error: Error)
    case success(message: String)
}
```

```swift
extension LoginState {
    typealias prism<Value> = Prism<LoginState, Value>
}

extension Prism where Root == LoginState, Value == String {
    static var success: LoginState.prism<String> {
        return .init(
            match: {
                switch $0 {
                case .success(let message):
                    return message
                default:
                    return nil
                }
        },
            build: { .success(message: $0) })
    }
}
```

```swift
let currentState = LoginState.idle /// any state
let successPrism = LoginState.prism.success

/// successMessage: String?
let successMessage = successPrism.match(currentState)
```

```swift
extension Prism {
    func tryModify (
        _ transformValue: @escaping (Value) -> Value)
        -> (Root) -> Root
    {
        return { root in
            guard let matched = self.match(root) else {
                return root
            }
            return self.build(transformValue(matched))
        }
    }
}
```

```swift
let processingPrism = LoginState.prism.processing

/// incrementAttemptsIfPossible: (LoginState) -> LoginState
let incrementAttemptsIfPossible = processingPrism
    .tryModify { $0 + 1 }
```

```swift
enum Event {
    case application(Application)
    case login(Login)

    enum Login {
        case tryLogin(outcome: LoginOutcome)
        case logout(motivation: LogoutMotivation)

        enum LoginOutcome {
            case success
            case failure(message: String)
        }
    }
}
```

```swift
/// Prism<A, B> + Prism<B, C> = Prism<A, C>

func pipe<A, B, C>(
    _ prism1: Prism<A, B>,
    _ prism2: Prism<B, C>)
    -> Prism<A, C>
{
    return Prism<A, C>(
        match: {
            prism1.match($0).flatMap(prism2.match)
        },
        build: {
            prism1.build(prism2.build($0))
        })
}
```

```swift
extension Prism where Value == Event.Login {
    var tryLogin: Prism<Root, Event.Login.LoginOutcome> {
        return pipe(self, .tryLogin)
    }
}

extension Prism where Value == Event.Login.LoginOutcome {
    var failure: Prism<Root, String> {
        return pipe(self, .failure)
    }
}
```

```swift
/// failureMessagePrism: Prism<Event, String>
let failureMessagePrism = Event.prism
    .login.tryLogin.failure

/// uppercasedMessageIfPossible: (Event) -> Event
let uppercasedMessageIfPossible = failureMessagePrism
    .tryModify { $0.uppercased() }
```

Affine

```
struct Affine<Root, Value> {
    let preview: (Root) -> Value?
    let tryUpdate: (Value, Root) -> Root?
}
```

```swift
extension Array {
    static func affineForFirst(
        where predicate: @escaping (Element) -> Bool)
        -> Affine<Array, Element>
    {
        return Affine<Array, Element>(
            preview: { array in
                array.first(where: predicate)
        },
            tryUpdate: { element, array in
                guard let index = array
                    .index(where: predicate)
                    else { return nil }

                var m_array = array
                m_array.remove(at: index)
                m_array.insert(element, at: index)
                return m_array
        })
    }
}
```

Affine generalizes both Lens and Prism

Affine generalizes both Lens and Prism

- `Lens` -> `Affine`

Affine generalizes both Lens and Prism

- `Lens -> Affine`
- `Prism -> Affine`

Affine generalizes both Lens and Prism

- `Lens` -> `Affine`
- `Prism` -> `Affine`
- `pipe` on `Affine`

Affine generalizes both Lens and Prism

- `Lens` -> `Affine`
- `Prism` -> `Affine`
- `pipe` on `Affine`
- `Lens` + `Prism` = `Affine`

```swift
enum TransactionState {
    case idle
    case failure(String)
    case success([String: TransactionResult])
}

struct TransactionResult {
    var completion: Date
    var outcomes: [TransactionOutcome]
}

struct TransactionOutcome {
    var user: String
    var balance: Double
}
```

```swift
let ultimateAffine = pipe(

    TransactionState.prism.success,

    Dictionary.lens(at: "IllRememberThis"),

    Optional.prism,

    TransactionResult.lens(\.outcomes),

    Array.affineForFirst {
        $0.user == "Siri McSirison"
    },

    TransactionOutcome.lens(\.balance)
)
```

Recap

Recap

- Functional Programming rocks

Recap

- Functional Programming rocks
  - even if just a bit

Recap

- Functional Programming rocks
  - even if just a bit
- Well-defined data structures are the key to clear, testable and maintainable code

Recap

- Functional Programming rocks
  - even if just a bit
- Well-defined data structures are the key to clear, testable and maintainable code
- Optics: a useful tool to operate on data structures in a generic way

Recap

- Functional Programming rocks
    - even if just a bit
- Well-defined data structures are the key to clear, testable and maintainable code
- Optics: a useful tool to operate on data structures in a generic way
    - Lens: structs and classes

Recap

- Functional Programming rocks
    - even if just a bit
- Well-defined data structures are the key to clear, testable and maintainable code
- Optics: a useful tool to operate on data structures in a generic way
    - Lens: structs and classes
    - Prisms: enums

Recap

- Functional Programming rocks
    - even if just a bit
- Well-defined data structures are the key to clear, testable and maintainable code
- Optics: a useful tool to operate on data structures in a generic way
    - Lens: structs and classes
    - Prisms: enums
    - Affine: optional access

Thank You

@_logicist

https://github.com/broomburgo/advanced-swift-optics