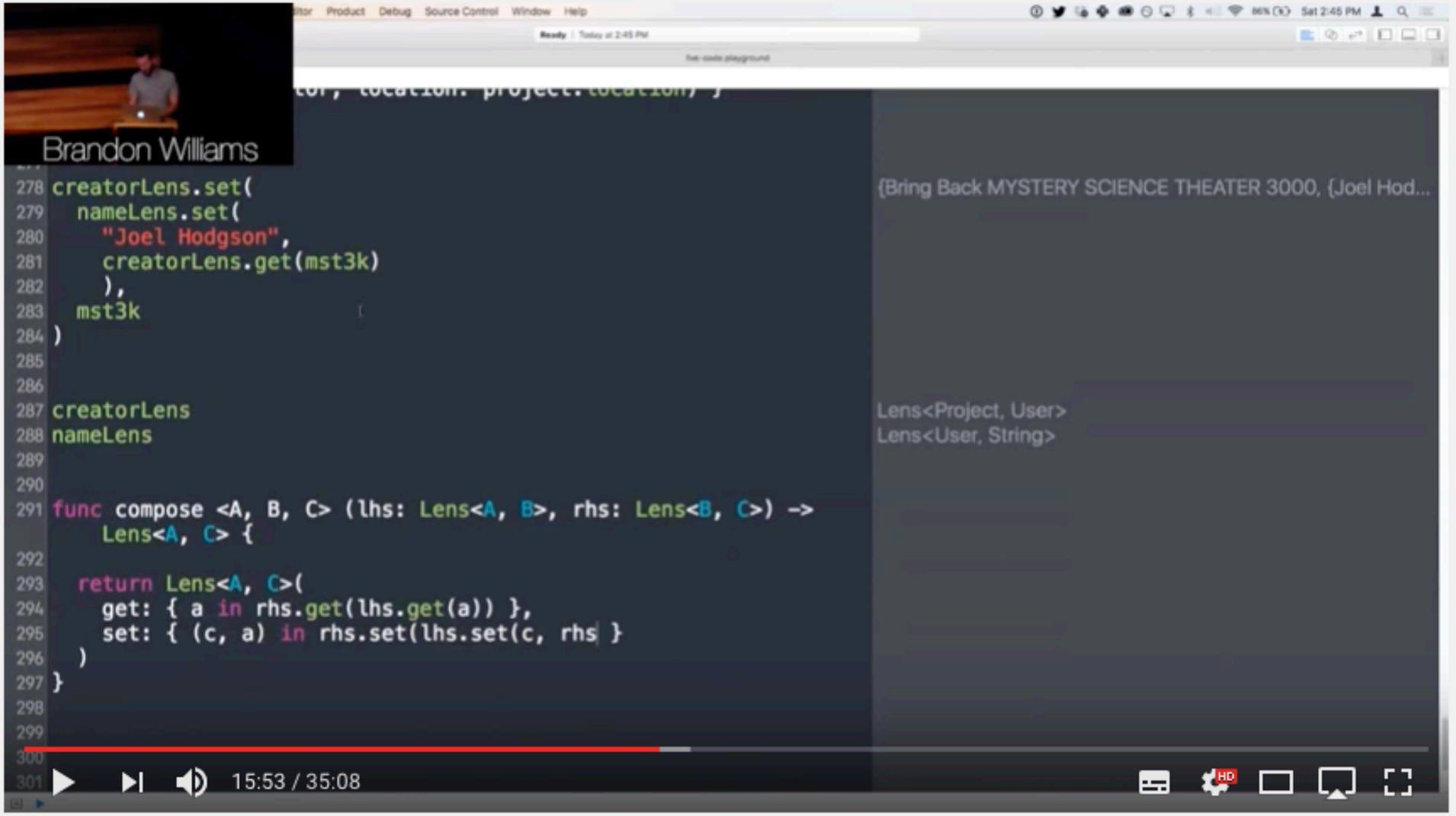
Lenses and Prisms in Swift



Brandon Williams - Lenses in Swift

F

Functional Swift

Iscritto | 2.087

6.332 visualizzazioni

+ Aggiungi a Condividi ••• Altro 57 2

Functional getters and setters

```
let repo = PersonRepository.init() // class

var p = Person.init(firstName: "Jane", lastName: "Doe") // struct

repo.add(p)

p.firstName = "John"

repo.add(p)
```

Lenses and Prisms

Lenses and Prisms

→ **encapsulate** data references and relations

Lenses and Prisms

→ **encapsulate** data references and relations

→ **simple** to define

Lenses and Prisms

→ **encapsulate** data references and relations

→ **simple** to define

→ **composition** tools

Lenses and Prisms

→ **encapsulate** data references and relations

→ **simple** to define

→ **composition** tools

→ build powerful **abstractions**

```
struct Lens<Whole,Part> {
    let get: (Whole) -> Part
    let set: (Part) -> (Whole) -> Whole
}

struct Prism<Whole,Part> {
    let tryGet: (Whole) -> Part?
    let inject: (Part) -> Whole
}
```

```
struct Lens<Whole,Part> {
    let get: (Whole) -> Part
    let set: (Part) -> (Whole) -> Whole
}

struct Prism<Whole,Part> {
    let tryGet: (Whole) -> Part?
    let inject: (Part) -> Whole
}
```

```
struct Lens<Whole,Part> {
    let get: (Whole) -> Part
    let set: (Part) -> (Whole) -> Whole
}

struct Prism<Whole,Part> {
    let tryGet: (Whole) -> Part?
    let inject: (Part) -> Whole
}
```

```
struct Lens<Whole,Part> {
    let get: (Whole) -> Part
    let set: (Part) -> (Whole) -> Whole
}

struct Prism<Whole,Part> {
    let tryGet: (Whole) -> Part?
    let inject: (Part) -> Whole
}
```

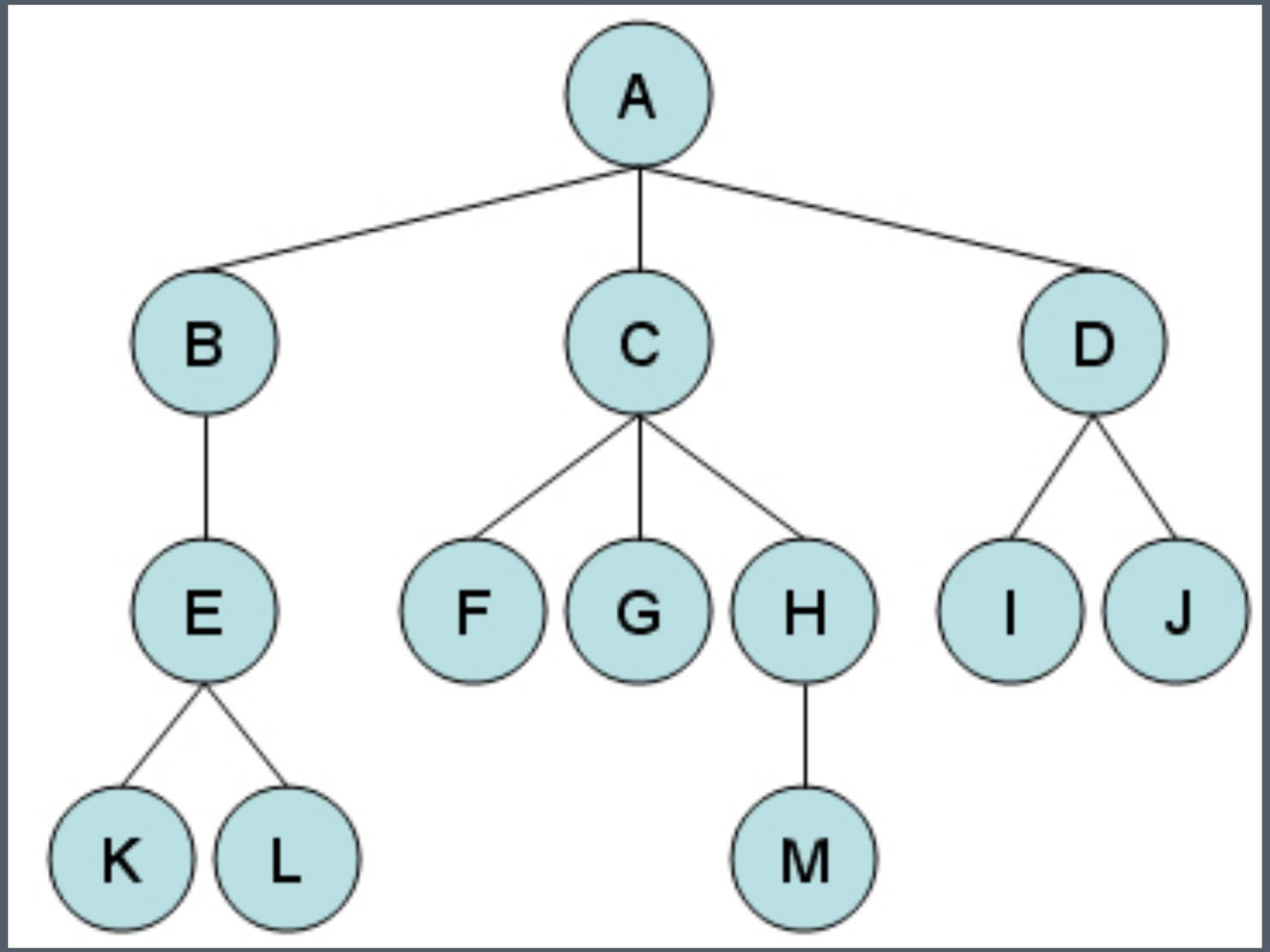
```
enum ViewState<T> {
    case empty
    case processing(String)
    case failed(Error)
    case completed(T)
}
```

```
enum ViewState<T> {
    case empty
    case processing(String)
    case failed(Error)
    case completed(T)
}
```

```
struct Lens<Whole,Part> {
    let get: (Whole) -> Part
    let set: (Part) -> (Whole) -> Whole
}

struct Prism<Whole,Part> {
    let tryGet: (Whole) -> Part?
    let inject: (Part) -> Whole
}
```

```
struct Lens<Whole,Part> {
    let get: (Whole) -> Part
    let set: (Part) -> (Whole) -> Whole
}

struct Prism<Whole,Part> {
    let tryGet: (Whole) -> Part?
    let inject: (Part) -> Whole
}
```

```
struct Lens<Whole,Part> {
    let get: (Whole) -> Part
    let set: (Part) -> (Whole) -> Whole
}

struct Prism<Whole,Part> {
    let tryGet: (Whole) -> Part?
    let inject: (Part) -> Whole
}
```


A
B
C
D
E
F
G
H
I
J
K
L
M

View

ViewModel

Functional ViewModel

```
struct LoginPage {
    var title: String
    var credendials: CredentialBox
    var buttonState: ViewState<Button>
}

struct CredentialBox {
    var usernameField: TextField
    var passwordField: TextField
}

struct TextField {
    var text: String
    var placeholder: String?
    var secureText: Bool
}

struct Button {
    var title: String
    var enabled: Bool
}
```

```
struct LoginPage {
    var title: String
    var credendials: CredentialBox
    var buttonState: ViewState<Button>
}

struct CredentialBox {
    var usernameField: TextField
    var passwordField: TextField
}

struct TextField {
    var text: String
    var placeholder: String?
    var secureText: Bool
}

struct Button {
    var title: String
    var enabled: Bool
}
```

```
extension CredentialBox {
    enum lens {
        static let usernameField = Lens<CredentialBox,TextField>.init(
            get: { $0.usernameField },
            set: { part in
                { whole in
                    var m = whole
                    m.usernameField = part
                    return m
                }
        })
    }
}
```

```
extension ViewState {
    enum prism {
        static var processing: Prism<ViewState,String> {
            return .init(
                tryGet: {
                    guard case .processing(let message) = $0 else {
                        return nil
                    }
                    return message
                },
                inject: { .processing($0) })
        }
    }
}
```

Manipulate the model

Manipulate the model

→ a previous model exists

Manipulate the model

→ a previous model exists

→ set a greeting title

Manipulate the model

→ a previous model exists

→ set a greeting title

→ set a stored username

Manipulate the model

→ a previous model exists

→ set a greeting title

→ set a stored username

→ button initially not enabled

```
let initialState = (
    title: "Welcome back!",
    username: savedUsername,
    buttonState: ViewState<Button>.completed(Button.init(
        title: "Login",
        enabled: false)))

var m_newModel = oldModel
m_newModel.title = initialState.title
m_newModel.credendials.usernameField.text = initialState.username
m_newModel.buttonState = initialState.buttonState
```

```
let initialState = (
    title: "Welcome back!",
    username: savedUsername,
    buttonState: ViewState<Button>.completed(Button.init(
        title: "Login",
        enabled: false)))

var m_newModel = oldModel
m_newModel.title = initialState.title
m_newModel.credendials.usernameField.text = initialState.username
m_newModel.buttonState = initialState.buttonState
```

```
Lens<A,B> + Lens<B,C> = Lens<A,C>
```

```
extension Lens {
    func compose<Subpart>(_ other: Lens<Part,Subpart>) -> Lens<Whole,Subpart> {
        /// some code
    }
}

extension Prism {
    func compose<Subpart>(_ other: Prism<Part,Subpart>) -> Prism<Whole,Subpart> {
        /// some code
    }
}
```

```
let titleLens = LoginPage.lens.title

let usernameTextLens = LoginPage.lens.credentials
    .compose(CredentialBox.lens.usernameField)
    .compose(TextField.lens.text)

let buttonStateLens = LoginPage.lens.buttonState
```

```
let newModel = titleLens.set(initialState.title)
    (usernameTextLens.set(initialState.username)
    (buttonStateLens.set(initialState.buttonState)
    (oldModel)))
```

```
Lens<A,B1> + Lens<A,B2> = Lens<A,(B1,B2)>
```

```
extension Lens {
    static func zip<Part1,Part2>(
        _ a: Lens<Whole,Part1>,
        _ b: Lens<Whole,Part2>)
        -> Lens<Whole,(Part1,Part2)>
        where Part == (Part1,Part2)
    {
        /// some code
    }
}
```

```
let initialStateLens = Lens.zip(
    titleLens,
    usernameTextLens,
    buttonStateLens)

let newModel = initialStateLens
    .set(initialState)(oldModel)
```

What about prisms?

```
func advanceProcessingMessage(_ previous: String) -> String {
    switch previous {
    case "Please wait":
        return "Almost there"
    case "Almost there":
        return "ALMOST THERE"
    default:
        return previous + "!"
    }
}
```

```
extension Lens {
    func over(_ transform: (Part) -> Part) -> (Whole) -> Whole {
        /// some code
    }
}

extension Prism {
    func tryOver(_ transform: (Part) -> Part) -> (Whole) -> Whole {
        /// some code
    }
}
```

```
Lens<LoginPage,ViewState<Button>>

+

Prism<ViewState<Button>,String>

=

?
```

What if lenses and prisms don't compose well?

Who cares? Under the hood it's just functions.

```
(A -> B) + (B -> C) = (A -> C)
```

```
infix operator •

func • <A,B,C> (
    _ left: @escaping (B) -> C,
    _ right: @escaping (A) -> B)
    -> (A) -> C
{
    return { left(right($0)) }
}
```

```
let processingPrism = ViewState<Button>.prism.processing

let onProcessing = buttonStateLens.over • processingPrism.tryOver

let newModel = onProcessing(advanceProcessingMessage)(oldModel)
```

```
let processingPrism = ViewState<Button>.prism.processing

let onProcessing = buttonStateLens.over • processingPrism.tryOver

let newModel = onProcessing(advanceProcessingMessage)(oldModel)
```

```
let processingPrism = ViewState<Button>.prism.processing

let onProcessing = buttonStateLens.over • processingPrism.tryOver

let newModel = onProcessing(advanceProcessingMessage)(oldModel)
```

```
let processingPrism = ViewState<Button>.prism.processing

let onProcessing = buttonStateLens.over • processingPrism.tryOver

let newModel = onProcessing(advanceProcessingMessage)(oldModel)
```

```
(B -> B) -> (A -> A)

        .

(C -> C) -> (B -> B)

        =

(C -> C) -> (A -> A)
```

(ViewState<Button> -> ViewState<Button>) -> (LoginPage -> LoginPage)

•

(String -> String) -> (ViewState<Button> -> ViewState<Button>)

=

(String -> String) -> (LoginPage -> LoginPage)

```
let processingPrism = ViewState<Button>.prism.processing

let onProcessing = buttonStateLens.over • processingPrism.tryOver

let newModel = onProcessing(advanceProcessingMessage)(oldModel)
```

Let's recap

Let's recap

→ **separated** logic for accessing and mutating data;

Let's recap

- → **separated** logic for accessing and mutating data;
  - → defined **simple**, composable atomic objects;

Let's recap

- → **separated** logic for accessing and mutating data;
  - → defined **simple**, composable atomic objects;
- → built more complex objects just by **combination**;

Let's recap

→ **separated** logic for accessing and mutating data;

→ defined **simple**, composable atomic objects;

→ built more complex objects just by **combination**;

→ transformed data with multiple, **independent** pieces;

Let's recap

→ **separated** logic for accessing and mutating data;

→ defined **simple**, composable atomic objects;

→ built more complex objects just by **combination**;

→ transformed data with multiple, **independent** pieces;

→ types are the only **interface**;

Let's recap

→ **separated** logic for accessing and mutating data;

→ defined **simple**, composable atomic objects;

→ built more complex objects just by **combination**;

→ transformed data with multiple, **independent** pieces;

→ types are the only **interface**;

→ nothing will break if types don't change;

Let's recap

→ **separated** logic for accessing and mutating data;

→ defined **simple**, composable atomic objects;

→ built more complex objects just by **combination**;

→ transformed data with multiple, **independent** pieces;

→ types are the only **interface**;

→ nothing will break if types don't change;

→ **are we sure?**

Buggy lenses?

AXIOMS

Axioms

Axioms

→ required when building large systems from **small pieces**

Axioms

→ required when building large systems from **small pieces**

→ we need to **trust** the pieces

Axioms

→ required when building large systems from **small pieces**

→ we need to **trust** the pieces

→ they are **implicit** in tests

Axioms

→ required when building large systems from **small pieces**

→ we need to **trust** the pieces

→ they are **implicit** in tests

→ let's define them **explicitly** for Lenses and Prisms

Laws: Lens

Laws: Lens

→ **getSet**: the whole is the same

Laws: Lens

→ **getSet**: the whole is the same

→ **setGet**: the part is the same

Laws: Lens

→ **getSet**: the whole is the same

→ **setGet**: the part is the same

→ **setSet**: idempotence

Laws: Prisms

Laws: Prisms

→ tryGetInject

Laws: Prisms

→ tryGetInject

→ injectTryGet

```
struct LensLaw {
    static func setGet<Whole, Part>(
        lens: Lens<Whole,Part>,
        whole: Whole,
        part: Part)
        -> Bool where Part: Equatable
    {
        return lens.get(lens.set(part)(whole)) == part
    }
}
```

Do we need to test ALL lenses and prisms?

Do we need to test ALL lenses and prisms?

→ **trivial** lenses have always the same structure;

Do we need to test ALL lenses and prisms?

→ **trivial** lenses have always the same structure;

→ if written **by hand**, tests can be useful;

Do we need to test ALL lenses and prisms?

→ **trivial** lenses have always the same structure;

→ if written **by hand**, tests can be useful;

→ lots of **boilerplate**;

Do we need to test ALL lenses and prisms?

→ **trivial** lenses have always the same structure;

→ if written **by hand**, tests can be useful;

→ lots of **boilerplate**;

→ a perfect fit for **code generation**.

Non-trivial lenses and prisms

Non-trivial lenses and prisms

→ outputs of combinators;

Non-trivial lenses and prisms

→ outputs of combinators;

→ particular data structures.

```
extension Dictionary {
    static func lens(at key: Key) -> Lens<Dictionary,Value?> {
        return Lens<Dictionary,Value?>(
            get: { $0[key] },
            set: { part in
                { whole in
                    var m_dict = whole
                    m_dict[key] = part
                    return m_dict
                }
        })
    }
}
```

```
extension Dictionary {
    static func lens(at key: Key) -> Lens<Dictionary,Value?> {
        return Lens<Dictionary,Value?>(
            get: { $0[key] },
            set: { part in
                { whole in
                    var m_dict = whole
                    m_dict[key] = part
                    return m_dict
                }
        })
    }
}
```

```
Lens<A,B?> + Lens<B,C> = ?
```

```
Lens<A,B?> + Lens<B,C> = Lens<A,C?>
```

```
func compose<A,B,C>(
    _ first: Lens<A,B?>,
    _ second: Lens<B,C>)
    -> Lens<A,C?> {
    /// some code
}
```

```
func compose<A,B,C>(
    _ first: Lens<A,B?>,
    _ second: Lens<B,C>,
    injecting: B)
    -> Lens<A,C?> {
    /// some code
}
```

Optics library

https://github.com/facile-it/Optics

Optics library

https://github.com/facile-it/Optics

→ Lens and Prism **definitions**

Optics library

https://github.com/facile-it/Optics

→ Lens and Prism **definitions**

→ some **combinators**

Optics **library**

https://github.com/facile-it/Optics

→ Lens and Prism **definitions**

→ some **combinators**

→ **Law** functions

Thanks

Links and Contacts

→ @_logicist

→ https://github.com/broomburgo/Lenses-and-Prisms-in-Swift

→ https://github.com/facile-it/Optics

→ B.W.'s talk: https://youtu.be/ofjehH9f-CU