

# Lenses and Prisms in Swift



Ready | Today at 2:45 PM

Two-code playground

Brandon Williams

```
278 creatorLens.set(  
279     nameLens.set(  
280         "Joel Hodgson",  
281         creatorLens.get(mst3k)  
282     ),  
283     mst3k  
284 )  
285  
286  
287 creatorLens  
288 nameLens  
289  
290  
291 func compose <A, B, C> (lhs: Lens<A, B>, rhs: Lens<B, C>) ->  
292     Lens<A, C> {  
293         return Lens<A, C>(  
294             get: { a in rhs.get(lhs.get(a)) },  
295             set: { (c, a) in rhs.set(lhs.set(c, rhs)  
296         )  
297     }  
298  
299  
300  
301 ► ▶ 🔍 15:53 / 35:08
```

Lens<Project, User>  
Lens<User, String>

## Brandon Williams - Lenses in Swift



Functional Swift

Iscritto



2.087

6.332 visualizzazioni

+

Aggiungi a



Condividi

••• Altro



57



2

# Functional getters and setters

# Lenses and Prisms

# Lenses and Prisms

→ **encapsulate** a relationship between a data structure and its parts

# Lenses and Prisms

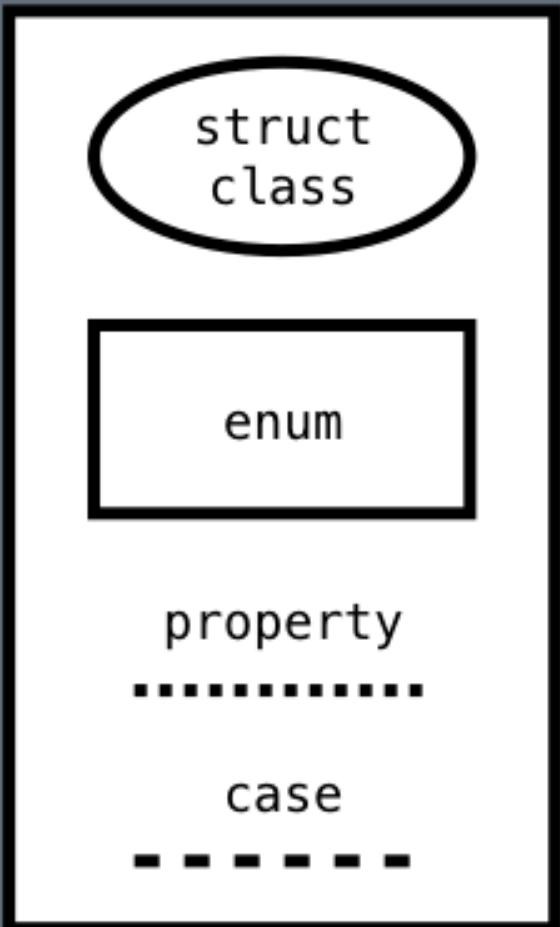
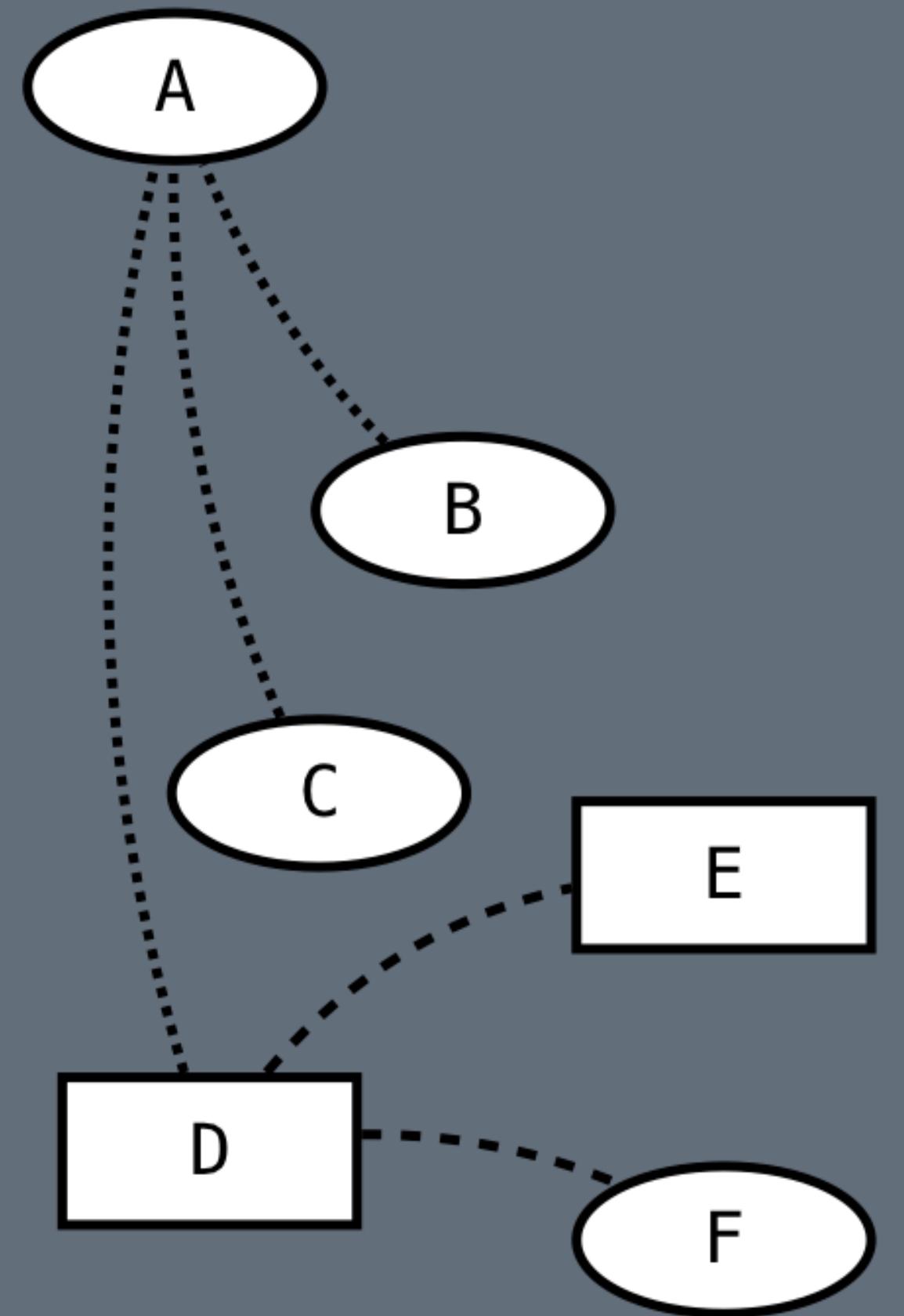
- **encapsulate** a relationship between a data structure and its parts
  - **simple** to define

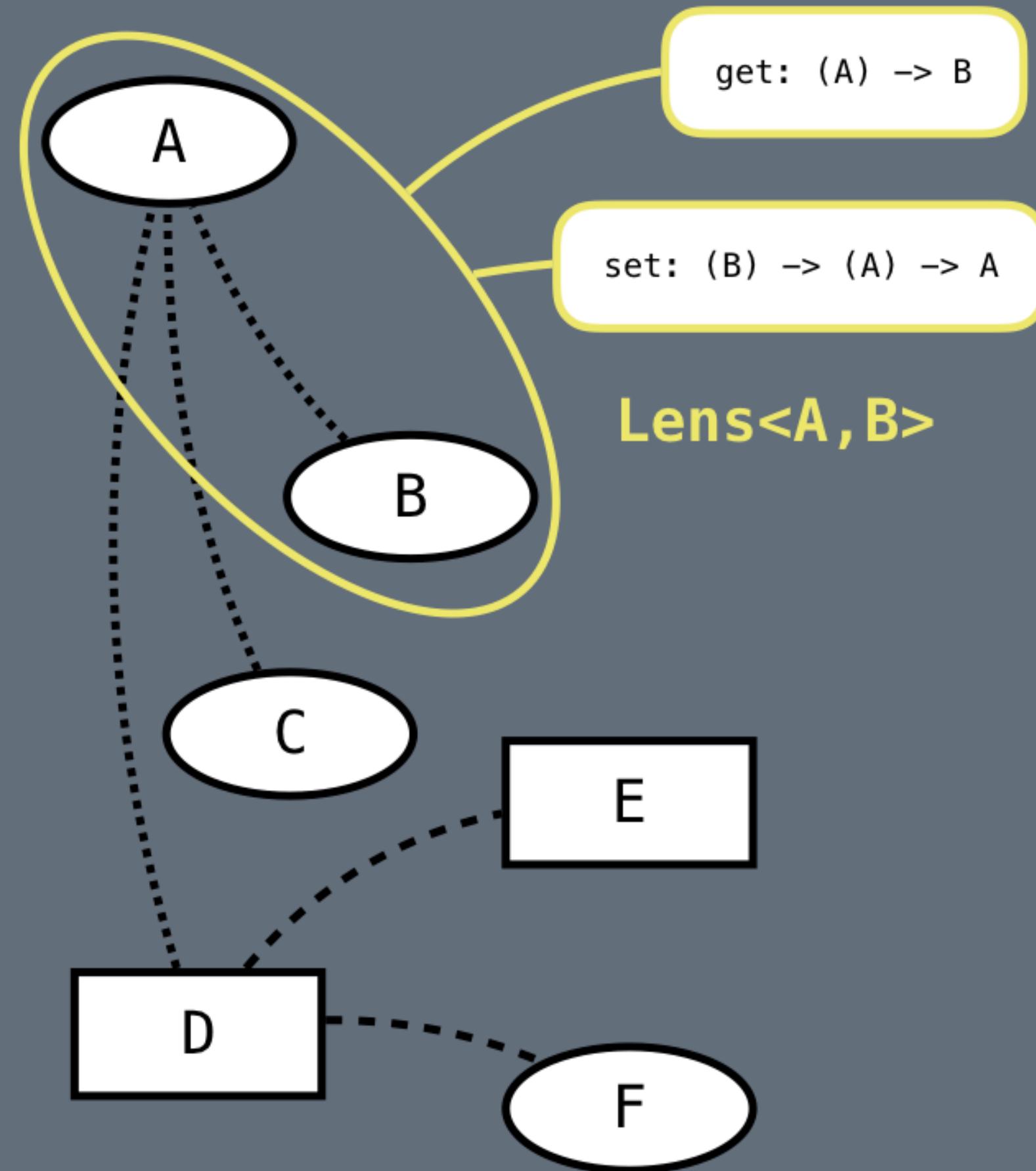
# Lenses and Prisms

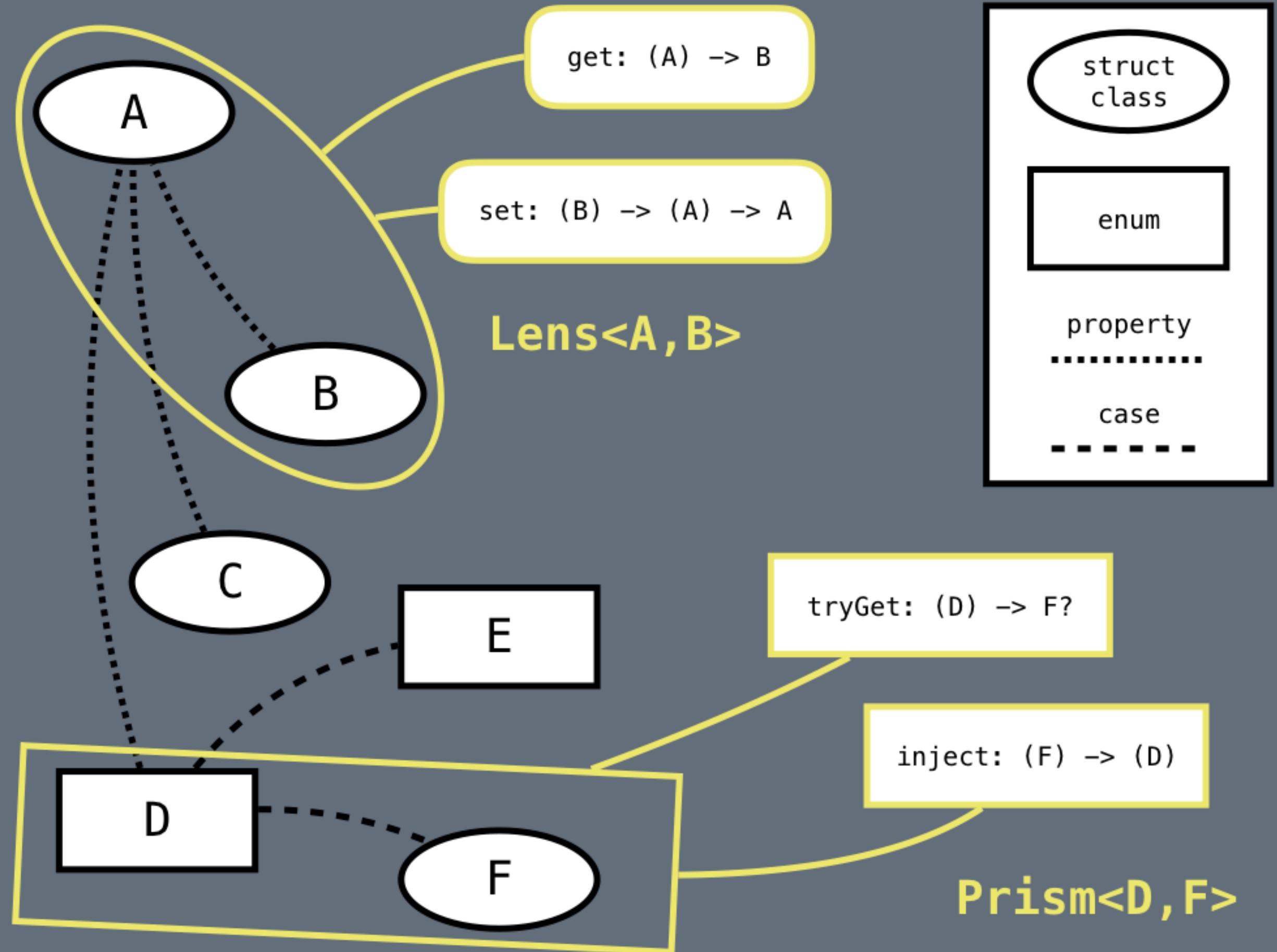
- **encapsulate** a relationship between a data structure and its parts
  - **simple** to define
  - **composition** tools

# Lenses and Prisms

- **encapsulate** a relationship between a data structure and its parts
  - **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
}
```







```
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  
}
```

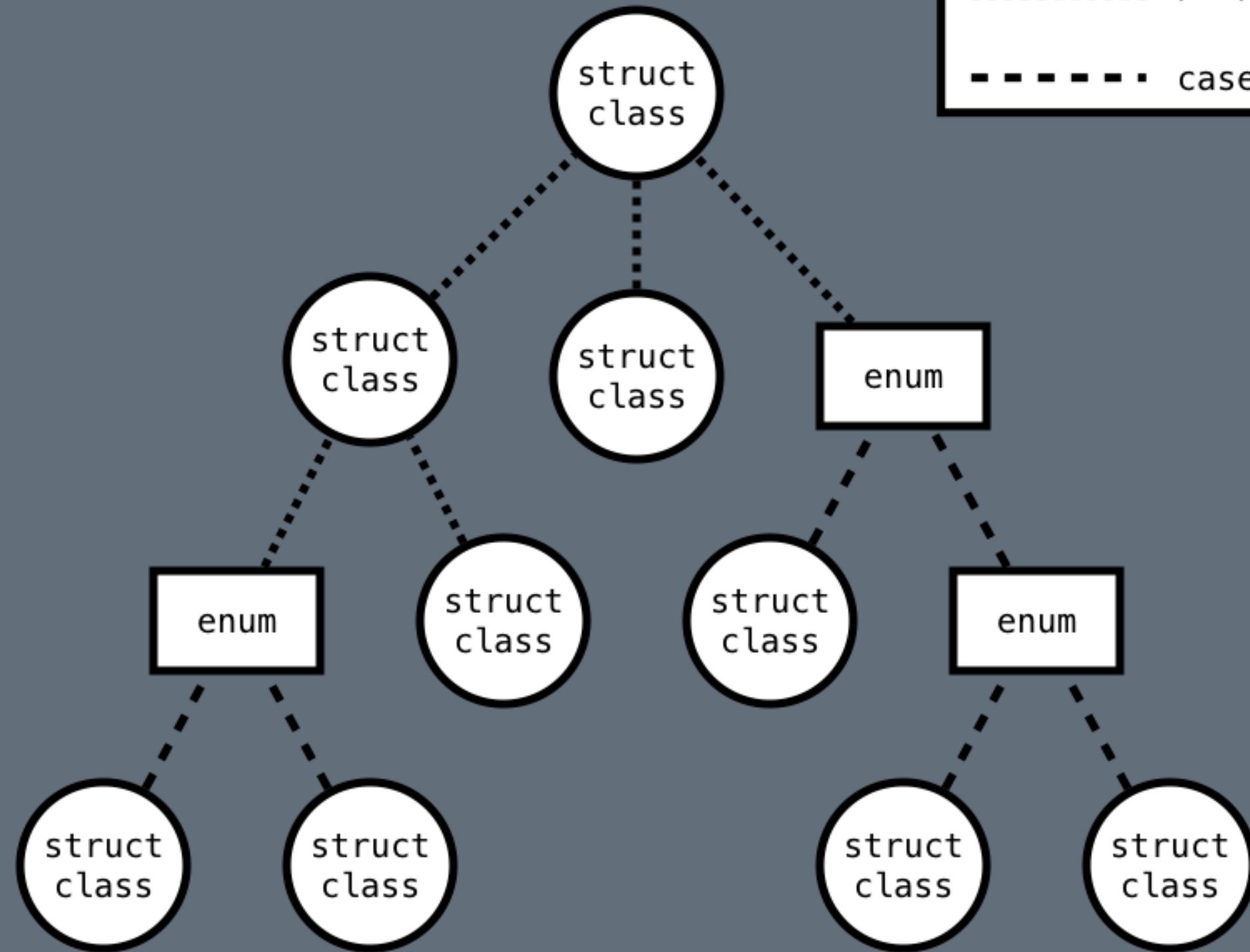
```
struct Lens<Whole, Part> {  
    let get: (Whole) -> Part  
    let set: (Part) -> (Whole) -> Whole  
}
```

```
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
}
```

..... property  
- - - - case



view

# ViewModel

# Functional viewModel

```
struct LoginPage {  
    var title: String  
    var credentials: 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 credentials: 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.credentials.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.credentials.usernameField.text = initialState.username
m_newModel.buttonState = initialState.buttonState
```

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

=====

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



```
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)>`

`=====`

`Prism<A,B1> + Prism<A,B2> = Prism<A,Either<B1,B2>>`



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

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

**what about  
prisms?**





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.**

$$(A \rightarrow B) + (B \rightarrow C) = (A \rightarrow C)$$

```
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)
```



$(B \rightarrow B) \rightarrow (A \rightarrow A)$

.

$(C \rightarrow C) \rightarrow (B \rightarrow B)$

=

$(C \rightarrow C) \rightarrow (A \rightarrow A)$

$(ViewState<Button> \rightarrow ViewState<Button>) \rightarrow (LoginPage \rightarrow$   
 $\quad LoginPage)$

•

$(String \rightarrow String) \rightarrow (ViewState<Button> \rightarrow ViewState<Button>)$

=

$(String \rightarrow String) \rightarrow (LoginPage \rightarrow LoginPage)$

```
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?

LAWS

# Laws

# Laws

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

# Laws

- required when building large systems from **small pieces**
- we need to **trust** the pieces

# Laws

- required when building large systems from **small pieces**
- we need to **trust** the pieces
- they are **implicit** in tests

# Laws

- 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 stays the same

# Laws: Lens

- **getSet**: the whole stays the same
- **setGet**: the part stays the same

# Laws: Lens

- **getSet**: the whole stays the same
- **setGet**: the part stays the same
- **setSet**: set is idempotent

# 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;

# 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;
- Sourcery

# Deriving Lens from KeyPath

```
extension WritableKeyPath {  
    var lens: Lens<Root, Value> {  
        return Lens<Root, Value>.init(  
            get: { whole in whole[keyPath: self] },  
            set: { part in  
                { whole in  
                    var m = whole  
                    m[keyPath: self] = part  
                    return m  
                }  
            } )  
    }  
}
```

```
let passwordLens = (\LoginPage.credentials.passwordField.text).lens
```

# 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  
                }  
            }  
        })  
    }  
}
```



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

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





# 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>