Lenses and Prisms in Swift

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

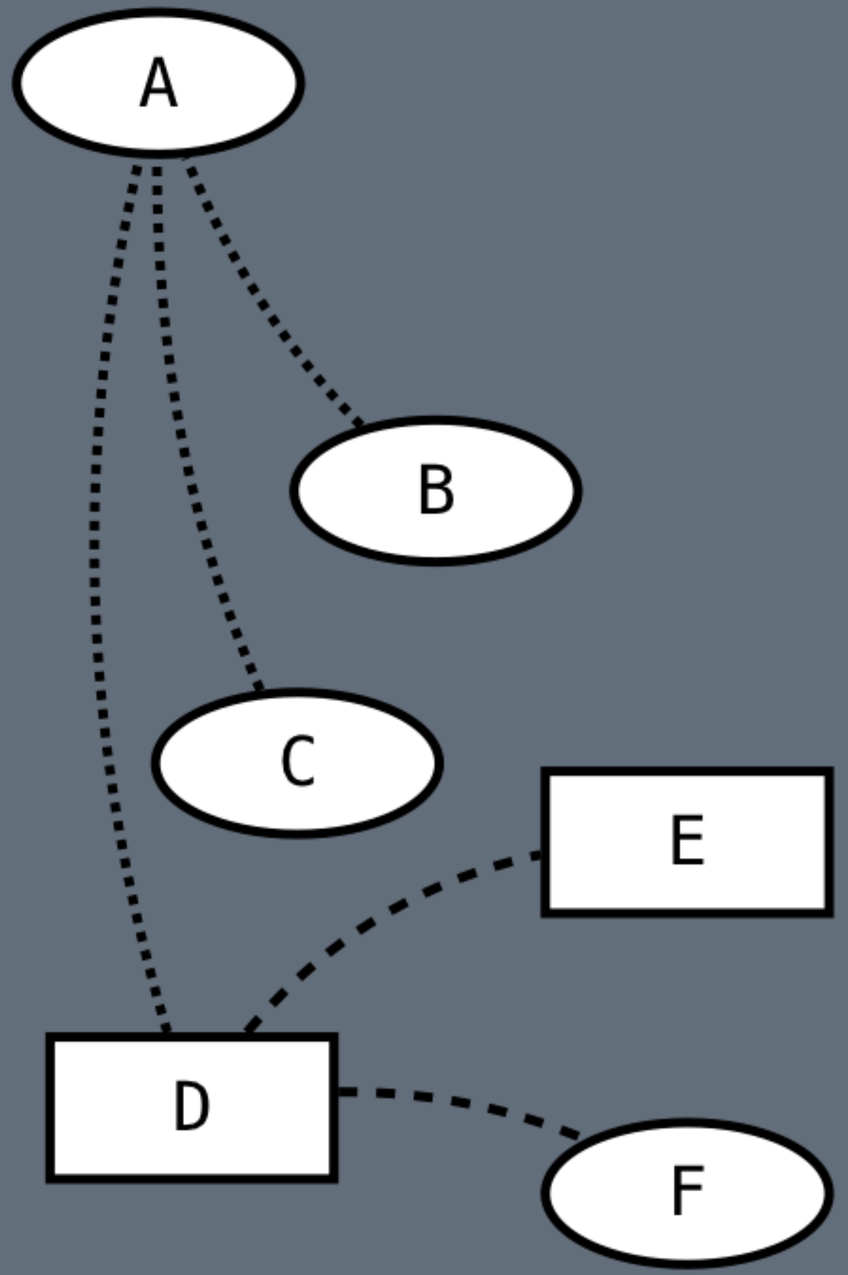
Lenses and Prisms

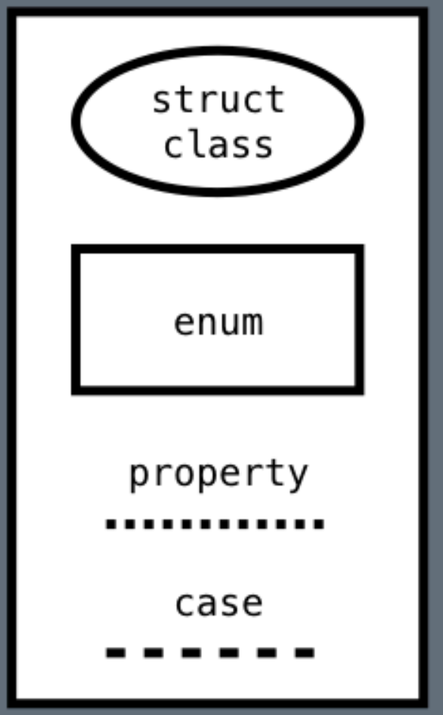
→ **encapsulate** a relationship between a data structure and its parts

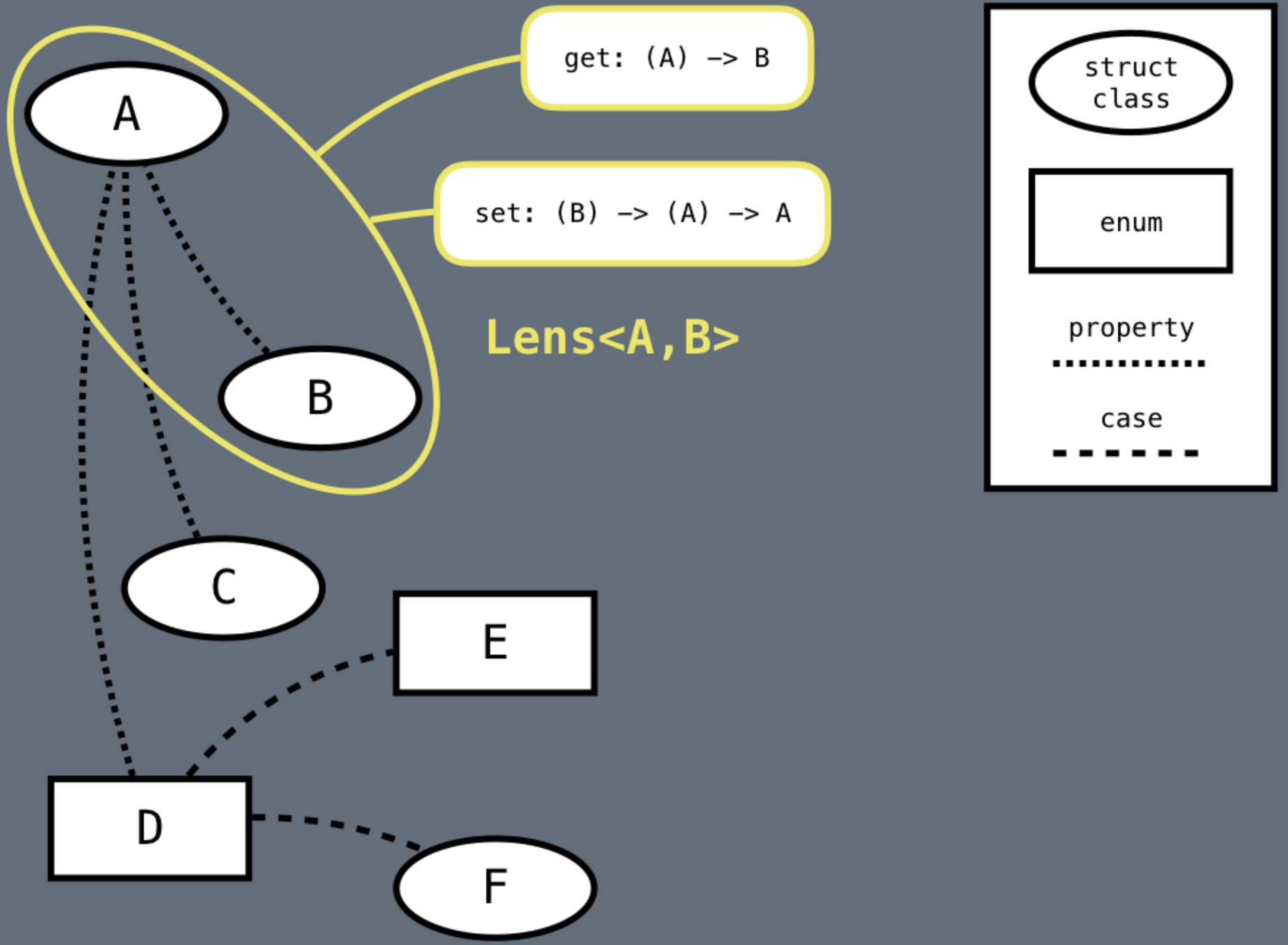
→ **simple** to define

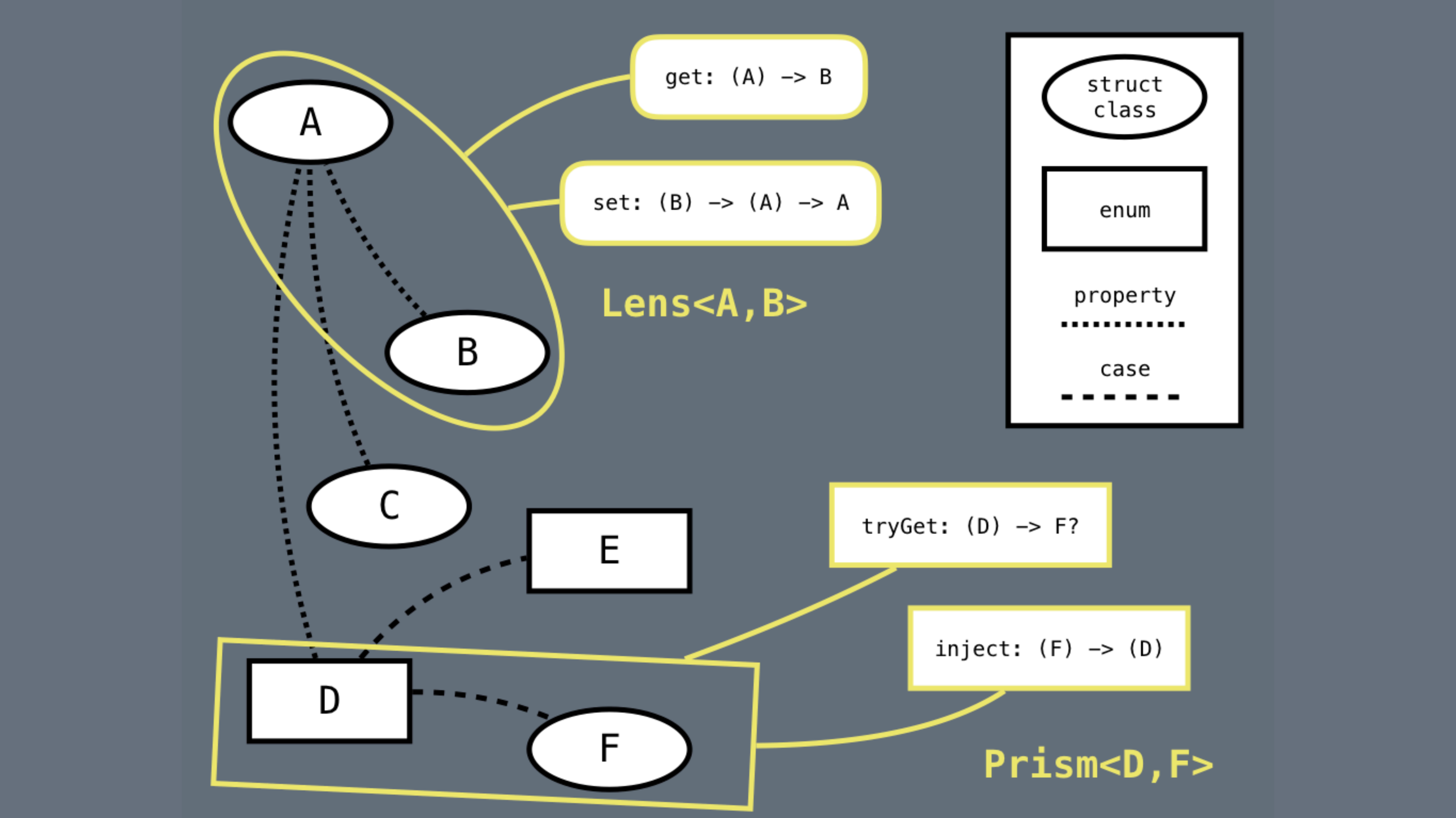
→ **composition** tools

→ build powerful **abstractions**


A
B
C
E
D
F


struct
class
enum
property
case


get: (A) -> B
set: (B) -> (A) -> A
Lens<A,B>
A
B
C
D
E
F
struct
class
enum
property
case


get: (A) -> B
set: (B) -> (A) -> A
A
B
Lens<A,B>
struct
class
enum
property
case
C
E
tryGet: (D) -> F?
inject: (F) -> (D)
D
F
Prism<D,F>

```
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
}
```

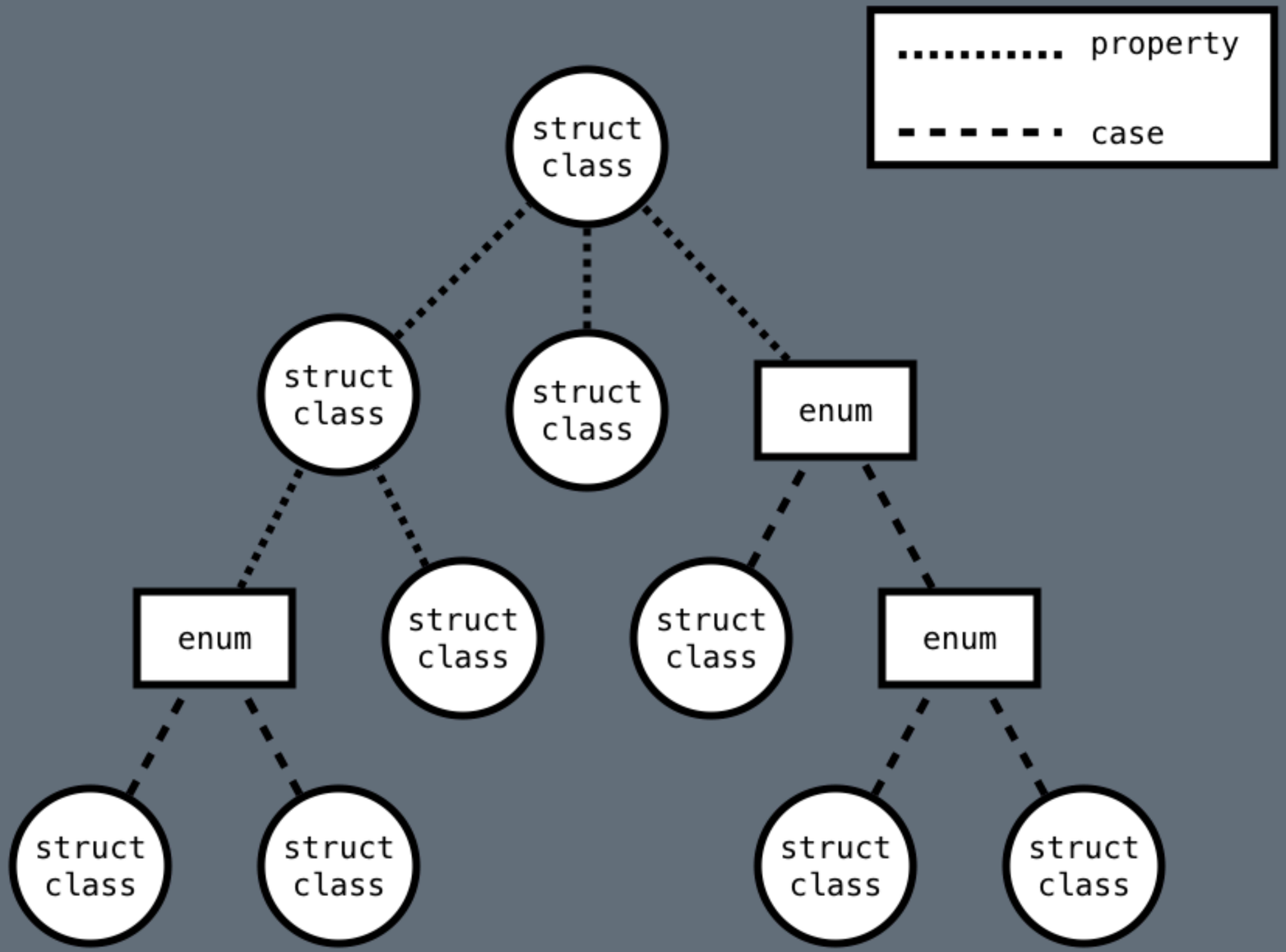
```
enum ViewState<T> {
   case empty
   case processing(String)
   case failed(Error)
   case completed(T)
}
```

```
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
}
```


property
case
struct class
struct class
struct class
enum
enum
struct class
struct class
enum
struct class
struct class
struct class
struct class

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

```swift
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>

============

Prism<A,B> + Prism<B,C> = Prism<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)>
==============
Prism<A,B1> + Prism<A,B2> = Prism<A,Either<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
  }
}

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

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

```
/// ((ViewState<Button>) -> ViewState<Button>) -> (LoginPage) -> LoginPage
let modifyLoginPage = buttonStateLens.modify

/// Prism<ViewState<Button>,String>
let processingPrism = ViewState<Button>.prism.processing

/// ((String) -> String) -> (ViewState<Button>) -> ViewState<Button>
let modifyProcessingMessage = processingPrism.tryModify
```

```
/// ((ViewState<Button>) -> ViewState<Button>) -> (LoginPage) -> LoginPage
let modifyLoginPage = buttonStateLens.modify

/// Prism<ViewState<Button>,String>
let processingPrism = ViewState<Button>.prism.processing

/// ((String) -> String) -> (ViewState<Button>) -> ViewState<Button>
let modifyProcessingMessage = processingPrism.tryModify
```

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

```
infix operator >>>

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

```
/// ((String) -> String) -> (LoginPage) -> LoginPage
let onProcessing =  modifyProcessingMessage >>> modifyLoginPage

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

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

>>>

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

=

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

```
(String -> String) -> (VS<Button> -> VS<Button>)

>>>

(VS<Button> -> VS<Button>) -> (LoginPage -> LoginPage)

=

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

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

let passwordLensAgain = °\LoginPage.credentials.passwordField.text
```

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> = ?
```

```
extension Optional {
  static var prism: Prism<Optional,Wrapped> {
    return Prism<Optional,Wrapped>.init(
      tryGet: { $0 },
      inject: Optional.some
    )
  }
}
```

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

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

/// laws are similar to lens'
/// es. Affine to an index of an Array
```

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

```
Lens<A,B> -> Affine<A,B>

Prism<A,B> -> Affine<A,B>
```

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

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

Affine<A,B> + Affine<B,C> =

Affine<A,C>
```

FunctionalKit library

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

FunctionalKit library

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

→ Lens, Prism, Affine **definitions**

FunctionalKit library

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

→ Lens, Prism, Affine **definitions**

→ some **combinators**

FunctionalKit library

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

→ Lens, Prism, Affine **definitions**

→ some **combinators**

→ **Law** functions

Thanks

Links and Contacts

→ @_logicist

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

→ https://github.com/facile-it/FunctionalKit

→ Brandon Williams's original talk on lenses: https://youtu.be/ofjehH9f-CU