Swift 5.1 Ver.2

2020/01/28

Kosuke Matsuda

- Default Arguments In Enum Cases
- Synthesize default values for the memberwise initializer
- Implicit returns from single-expression functions
- Warnings for ambiguous none cases
- Matching optional enums against non-optionals
- Expanding Swift Self to class members and value types
- Opaque Result Types
- Static and class subscripts
- Ordered Collection Diffing
- Key Path Member Lookup
- **Property Wrappers**
- ...

Property Wrapper

A property wrapper type is a type that can be used as a property wrapper. There are two basic requirements for a property wrapper type:

1. The property wrapper type must be defined with the attribute @propertyWrapper. The attribute indicates that the type is meant to be used as a property wrapper type, and provides a point at which the compiler can verify any other consistency rules.

2. The property wrapper type must have a property named wrappedValue, whose access level is the same as that of the type itself. This is the property used by the compiler to access the underlying value on the wrapper instance.

https://github.com/apple/swift-evolution/blob/master/proposals/0258-property-wrappers.md#property-wrapper-types

Property Wrapper

プロパティラッパータイプは、プロパティラッパーとして使用できるタイプです。
プロパティラッパータイプには2つの基本要件があります。

1. プロパティラッパータイプは、属性@propertyWrapperで定義する必要があります。 この属性は、そのタイプがプロパティラッパータイプとして使用されることを意味し、コンパイラが他の一貫性ルールを検証できるポイントを提供します。

2. プロパティラッパータイプには、wrappedValueという名前のプロパティが必要です。このプロパティのアクセスレベルは、タイプ自体のアクセスレベルと同じです。 これは、ラッパーインスタンスの基になる値にアクセスするためにコンパイラが使用するプロパティです。

https://github.com/apple/swift-evolution/blob/master/proposals/0258-property-wrappers.md#property-wrapper-types

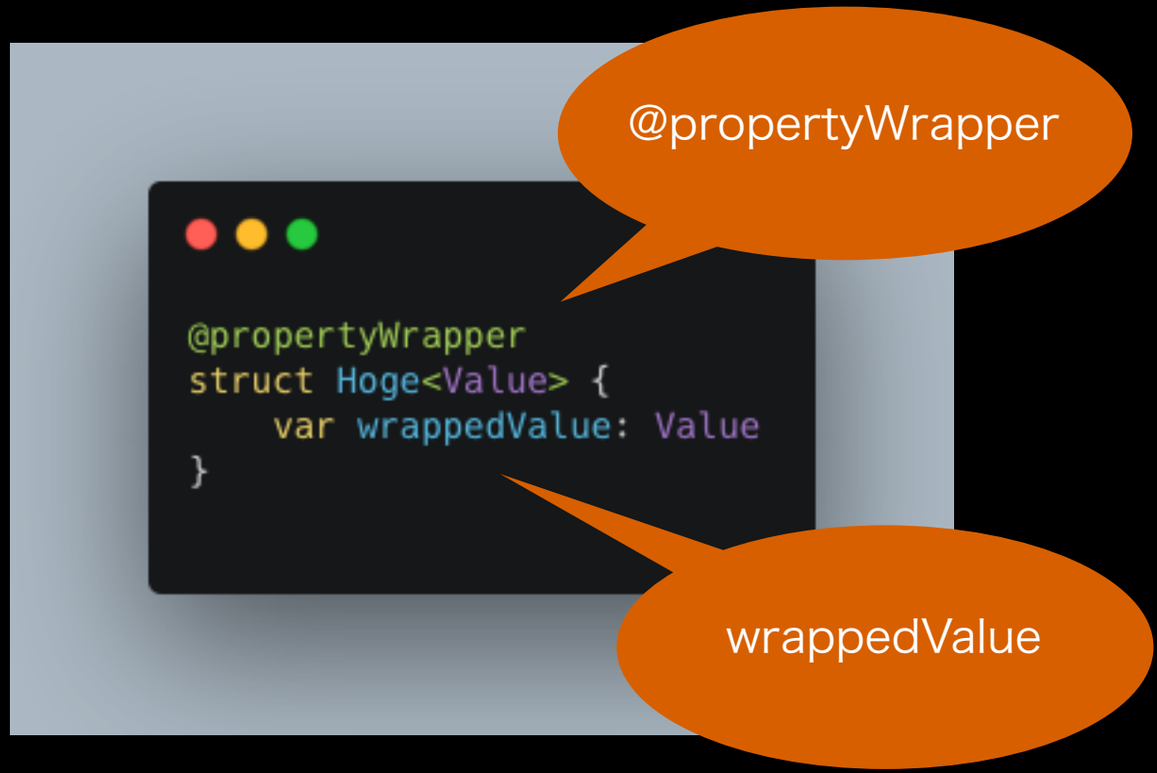
- 値が変更されるたびにそれをトリガーに何らかの関連ロジックを実行する。

- 例えば一連のルールに従って新しい値を検証したり、割り当てられた値を何らかの方法で変換したり、値が変更された時にオブザーバーに通知したり。

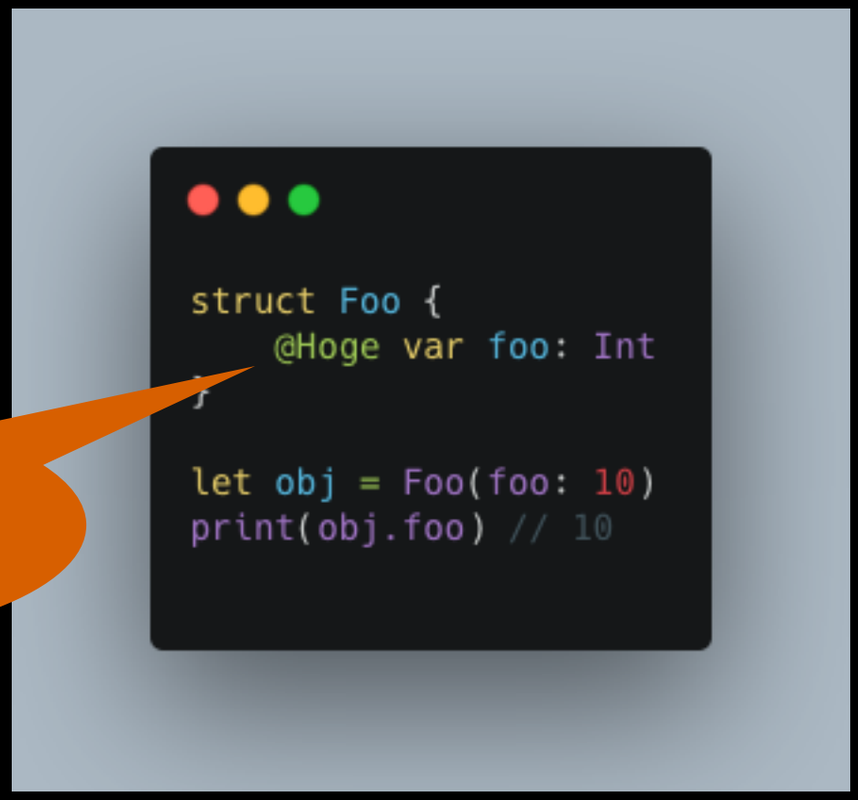
- そのような振る舞いを再利用できるようにする。

@propertyWrapper

@propertyWrapper



@propertyWrapper



@propertyWrapper

```
struct Foo {
    @Hoge var foo: Int

    // ... implemented as
    // private var _foo: Hoge<Int>
    // var foo: Int {
    //     get { _foo.wrappedValue }
    //     set { _foo.wrappedValue = newValue }
    // }
}

let obj = Foo(foo: 10)
// foo is equal to _foo.wrappedValue
print(obj.foo) // 10
```

projectedValue

- PropertyWrapper自身にアクセスする

- projectedValueプロパティを定義するとproperty wrapper typesへの投影変数（$xxx）が提供される

projectedValue

projectedValue

```
struct Foo {
    @Hoge var foo: Int
}

let obj = Foo(foo: 10)
print(obj.$foo) // Hoge<Int>(wrappedValue: 10)
```

$foo

projectedValue

```
struct Foo {
    @Hoge var foo: Int

    // ... implemented as
    // private var _foo: Hoge<Int>
    // var foo: Int {
    //     get { _foo.wrappedValue }
    //     set { _foo.wrappedValue = newValue }
    // }
    // var $foo: Hoge<Int> {
    //     get { _foo.projectedValue }
    //     set { _foo.projectedValue = newValue }
    // }
}

let obj = Foo(foo: 10)
// $foo is equal to _foo.projectedValue
print(obj.$foo) // Hoge<Int>(wrappedValue: 10)
```

ex.) Capitalized

```swift
@propertyWrapper struct Capitalized {
    var wrappedValue: String {
        didSet { wrappedValue = wrappedValue.capitalized }
    }

    init(wrappedValue: String) {
        self.wrappedValue = wrappedValue.capitalized
    }
}

struct User {
    @Capitalized var firstName: String
    @Capitalized var lastName: String

    var fullName: String { firstName + " " + lastName }
}

var user = User(firstName: "taro", lastName: "yamada")
print(user.fullName) // Taro Yamada
user.lastName = "suzuki"
print(user.fullName) // Taro Suzuki
```

ex.) UserDefault

```swift
@propertyWrapper
struct UserDefault<T> {
    let key: String
    let defaultValue: T

    var wrappedValue: T {
        get {
            UserDefaults.standard.object(forKey: key) as? T ?? defaultValue
        }
        set {
            UserDefaults.standard.set(newValue, forKey: key)
        }
    }
}
```

ex.) UserDefault

```swift
enum GlobalSettings {
    @UserDefault(key: "isLaunch", defaultValue: false)
    static var launched: Bool

    @UserDefault(key: "launchDate", defaultValue: Date())
    static var stateDate: Date
}

print(GlobalSettings.launched) // false
print(GlobalSettings.stateDate) // 2020-01-19 13:46:44 +0000

GlobalSettings.launched = true
GlobalSettings.stateDate = Date().addingTimeInterval(60*60)

print(GlobalSettings.launched) // true
print(GlobalSettings.stateDate) // 2020-01-19 14:46:44 +0000
```