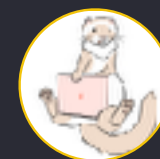


Secret Swift tour

2018/3/1
try! Swift Conference
Yuka Ezura



みなさん、おはようございます。

ezura と申します。LINE で iOS エンジニアをしています。

これからお見せするコードの多くはみなさんが日々目にしたり書いたりする、慣れ親しんだコードです。特別なものは何一つありません。しかし、いつもと少しだけ違う顔を見せるかもしれません。

それは、楽しい発見かもしれませんし、バグを生み出すような危険なものかもしれません。

今日はそんな、Swift をいつもと違う視点で探索する話をします。

Secret Swift tour、楽しんでいただけると幸いです。

```
var myVariable = 42  
myVariable = 50
```

まずは、公式のチュートリアルである、A Swift tour の入り口のコードから出発しましょう。

変数に値を代入する処理、日常的に書いていますよね

では、この代入式、

```
var myVariable = 42  
let r = (myVariable = 50)
```

Void

何が返りますか？

@@

Swift だと Void、つまり空の tuple です。

では、こちらはどうでしょうか？

```
obj?.myVariable = 50
```

変数の`obj`は`Optional`型です。そして、そのプロパティである`myVariable`に値を代入しようとしています。

```
(obj?.myVariable = 50)
```

Optional<Void>

こちらは Void?、つまり Void の Optional が返ります。

変数 obj が nil だと何もせずに nil が返り、obj が nil でないなら代入処理が行われ空のtupleが返ります。

このような動作、珍しいものではないですよ。そうです、optional chaining です。

1:55

```
(obj?.myVariable = 50)
    .map { /* do something */ }
```

ちょっとした応用例としては

Optional に対する map の性質を使って、代入の成功時のみ処理を実行するという書き方もできます。

さて、演算子は全てこのような挙動でしょうか...？

2:12

```
obj?.myVariable + 50
```

```
! error: Value of optional type 'Int?' not unwrapped;
```

例えば、+ 演算子はどうでしょう。

@@

コンパイルエラーです。内容としては、左辺の値が Optional なため、unwrap しないと加算処理ができない。ということです。

どうすれば = 演算子のように Optional chaining のシステムに乗ってくれるでしょうか。

```
infix operator ★  
func ★ (left: Int, right: Int) -> Int {  
    return left + right  
}
```

では、新たに 二項演算子を定義してみます。

右辺と左辺に Int 型をとって、足し合わせて Int 型を返す。シンプルな挙動です。

この中のどこにもオプショナルはいません。

とりあえずこの状態で先ほど + 演算子でエラーとなった式を実行してみます。

3:04


```
infix operator ★  
func ★ (left: Int, right: Int) -> Int {  
    return left + right  
}
```

```
obj?.myVariable ★ 5
```

! error: Value of optional type 'Int?' not unwrapped;

同じエラーです。では、魔法をかけましょう。

```
infix operator ★  
func ★ (left: Int, right: Int) -> Int {  
    return left + right  
}
```

```
precedencegroup FoldedIntoOptionalChaining {  
    assignment: true  
}  
  
infix operator ★ :FoldedIntoOptionalChaining  
func ★ (left: Int, right: Int) -> Int {  
    return left + right  
}
```

operator に対して設定を追加します。

`assignment` を `true` にするだけです。これでどんなことが起きたでしょうか。

3:22

```
obj?.myVariable ★ 5 // => Optional<Int>  
1 ★ 5 // => Int
```

先ほどコンパイルエラーになっていた、左辺に `Optional` の値を持つ式が `Int` の `Optional`を返すようになりました。

そして、`Optional` を持たない場合、`Optional` ではなく、ただの `Int` 型の値を返します。

冒頭の、`=` 演算子での挙動と同様になりました。

「左辺が `nil` だったら処理を止め、`nil` を返す」という `=` 演算子のような柔軟な動作、それに対して「事前に `nil` をハンドリングすることを強制する」という `+` 演算子で見た意思表示、我々はそれを使い分けられる自由があります。そして、すでにある実装でもうまくデザインされており、自然と我々を助けてくれています。

4:30

```
let myVariable: Int = /* ?? */
```

さて、代入に関連してもう一つ

この右辺に入るのはどんな式でしょうか。

Int 型の変数や、整数リテラル、あとは、Int 型の値を返す function を実行させたり…。

他に何かありますか？

例えば、

```
let myVariable: Int = {  
    while(true) {}  
}()
```

値を返さないクロージャを即時実行させる式です。

値を返さない、というのは、while(true) で無限ループさせたり、例外を投げたり、fatalError などを使って、return を使ったときと違って値を返さない状態にさせることです。

これは、戻り値が Never の function を作るときの要領です。例えば、fatalError も戻り値は Never ですよね。

では、このスライドに書かれているコードのクロージャの戻り値は Never でしょうか。

```
let naver: Never = unsafeBitCast((),  
                                to: Never.self)  
let myVariable: Int = naver  
! error: Cannot convert value of type 'Never' to specified type 'Int'
```

しかし、NeverはInt の sub type ではありません。つまり、継承関係などがありません。

実際に試してみても、Int 型に代入できず、タイプエラーになります。

5:51

```
let myVariable: Int = {  
  while(true) {}  
}()
```

では、このクロージャの型は何でしょうか。
型が解決済みのASTをみてみましょう。


```

(source_file
  (top_level_code_decl
    (brace_stmt
      (pattern_binding_decl
        (pattern_typed type='Int'
          (pattern_named type='Int' 'myVariable')
          (type_ident
            (component id='Int' bind=Swift.(file).Int)))
        (call_expr type='Int' location=sample.swift:1:23 range=[sample.swift:1:23 - line:3:3] nothrow arg_labels=
          (closure_expr type='() -> Int' location=sample.swift:1:23 range=[sample.swift:1:23 - line:3:1]
discriminator=0
          (parameter_list)
          (brace_stmt
            (while_stmt
              (call_expr implicit type='Int1' location=sample.swift:2:10 range=[sample.swift:2:10 - line:2:15]
nothrow arg_labels=
              (dot_syntax_call_expr implicit type='() -> Int1' location=sample.swift:2:10 range=[sample.swift:2:10
- line:2:15] nothrow
              (declref_expr implicit type='(Bool) -> () -> Int1' location=sample.swift:2:11 range=[sample.swift:
2:11 - line:2:11] decl=Swift.(file).Bool._getBuiltinLogicValue() function_ref=double)
              (paren_expr type='(Bool)' location=sample.swift:2:11 range=[sample.swift:2:10 - line:2:15]
              (call_expr implicit type='Bool' location=sample.swift:2:11 range=[sample.swift:2:11 - line:2:11]
nothrow arg_labels=_builtinBooleanLiteral:
              (constructor_ref_call_expr implicit type='(Int1) -> Bool' location=sample.swift:2:11
range=[sample.swift:2:11 - line:2:11] nothrow
              (declref_expr implicit type='(Bool.Type) -> (Int1) -> Bool' location=sample.swift:2:11
range=[sample.swift:2:11 - line:2:11] decl=Swift.(file).Bool.init(_builtinBooleanLiteral:) function_ref=single)
              (type_expr implicit type='Bool.Type' location=sample.swift:2:11 range=[sample.swift:2:11 -
line:2:11] typerepr='Bool'))
              (tuple_expr implicit type='(_builtinBooleanLiteral: Builtin.Int1)' location=sample.swift:2:11
range=[sample.swift:2:11 - line:2:11] names=_builtinBooleanLiteral
              (boolean_literal_expr type='Builtin.Int1' location=sample.swift:2:11 range=[sample.swift:2:11
- line:2:11] value=true))))))
              (tuple_expr implicit type='()'))
              (brace_stmt))))
              (tuple_expr type='()' location=sample.swift:3:2 range=[sample.swift:3:2 - line:3:3]))))
))
  (var_decl "myVariable" type='Int' interface type='Int' access=internal let storage_kind=stored))

```

この中から、クロージャの部分に注目してみます。

```
(closure_expr type='() -> Int'  
  location=sample.swift:1:23  
  range=[sample.swift:1:23 - line:3:1]  
  discriminator=0
```

```
let myVariable: Int = {  
  while(true) {}  
}()
```

型推論の結果、戻り値が Int と型がつけられています。

つまり、左辺の変数 myVariable の型に合わせた型となっています。

```
let _: Int = { preconditionFailure() }()  
let _: Void = { fatalError() }()  
let _: Never = { while(true) {} }()
```

そうです。このクロージャ式は、返り値をどんな型としても扱えるのです。
この性質をうまく利用している例として、

```
class SomeClass {  
    lazy var v: Int = {  
        preconditionFailure("Variable '\(#function)'  
                               used before being initialized")  
    }()  
}
```

以前、twitter でこんなコードを見かけました。

このコードがどんな問題を解決するのかというと、

6:46

```
class SomeClass {  
    var v: Int!  
    lazy var v: Int = {  
        preconditionFailure("Variable '\(#function)'  
                               used before being initialized")  
    }()  
}
```

使用する際は nil ではないけれど、init の中では決まらないプロパティに対して、implicitly unwrapped optional を使うことが定石ですよね。例えば、IBoutlet でつないでいる view や ViewModel などによく見る形かと思います。

値が決まるまでは初期値として nil を入れておくため、Optional 型の扱いになってしまうのですが

実際に使う段階では nil にならないはずの変数なのに Optional 型、つまり nil 許容の変数として存在するよりも、非Optionalな型である方が望ましいですね。

そこで、初期値として、nil ではなく、かといって、意味のないダミーの値でもなく、

@@

先ほど紹介した値を返さないクロージャ式を指定することで 非Optionalな型にすることに成功しています。

いつの間にか、代入の話から離れてきましたね。

7:30

Closure

では、仕切り直して、もっと Closure よりの話に移動しましょう。

一言に Closure といっても、function も Closure の一種として考えることができますし、あまりにも大きい話題です。

なので、今回は、使用頻度の高い、関数オブジェクトの生成に焦点を当ててみます。

```
let _: (Int) -> Int = { $0 + 1 }
```

このように Closure 式を使って、関数オブジェクトを生成する機会が多いと思いますが、この他にどんな生成方法があるでしょうか。
いくつか挙げてみましょう。

```
func f(x: Int...) -> String { ... }  
let _: (Int...) -> String = f
```

まず、一つ目、

function の引数を指定せずを書く方法です。

トップレベルにある function だけでなく、つまり、function の中で定義した function も同様です。

(「nested function = function の中で定義した function」という意図です)

次に、2つ目

8:50


```
struct SomeType {  
    func f(x: Int) -> String { ... }  
}  
  
let _: (Int) -> String = SomeType().f  
  
let _: (SomeType) -> (Int) -> String = SomeType.f
```

もちろん instance method に対しても書けますよね。この場合、実行主が未定の状態でもクロージャが作れます。

```
enum Rank: Int {  
    case ace = 1  
    case two, three, four, five, six,  
        seven, eight, nine, ten  
    case jack, queen, king  
}
```

```
let _: (Int) -> Rank? = Rank.init
```

init?(rawValue: Int)

method だけではなく、initializer でも同様です。

ちなみに、この`init`は`rawValue`を受け取って列挙型の値を作る initializer です。

```
let _: (Int, Int) -> (Int) = (+)
```

```
let _: (inout Int, Int) -> () = (+=)
```

operator method も使うことがあると思います。

あとは何があるでしょうか。

9:30

```
enum Page {  
  case settings  
  case externalSite(URL)  
}  
  
let _: (URL) -> Page = Page.externalSite
```

enum です。`enum` の associated value を求める関数オブジェクトも作り出すことができます。

一見、今まで挙げたものと比べて異質に見えるかもしれません。

では、enum の仕組みを振り返ってみましょう。

```
enum Page {  
    case settings  
    case externalSite(URL)  
    static var settings: Page {  
        return Page.settings  
    }  
  
    static func externalSite(_ url: URL) -> Page {  
        return Page.externalSite(url)  
    }  
}  
  
let _: (URL) -> Page = Page.externalSite
```

associated value を持つ enum case を作る処理は実質的には enum の static function を呼ぶような扱いです。associated value を持たない enum の場合、static property を呼ぶ処理です。これは、UIColor などでもお馴染みですね。UIColor はこれと同じように自身のインスタンスを返す static property を持っているので、UIColor.red とか、または省略して.red と書いて自身の instance を作りますよね。

このように補足した中であらためて見ると

@@

この式も自然に見えてきますよね。

ちなみに、コードをコンパイルしてみると再定義エラーになります。

10:40

```
[1.1, 3.2, 3.5].map(round)

[1.1, 3.2, 3.5].sorted(by: <)

["www.ezura.me", "🐱"].flatMap(URL.init)
```

私たちはクロージャを引数として受け付ける関数を、それこそ、息をするように使います。このインターフェースに繋げることができるプラグは様々な場所に存在していることが確認できました。

関数の引数としてクロージャを渡すといえば、`@escape`が指定されない限り、関数の中でそのクロージャを非同期で実行できなかったり、どこか関数の外にある変数に保存できなかったり...

クロージャがその関数より長く生存しないことを保証してくれますよね。

クロージャ以外にもそのような生存期間の制限のある引数がありますよね。

11:46

inout

`inout` を指定した引数です。

```
let _: (Int...) -> String = f
let _: (Int) -> String = SomeType().f
let _: (SomeType) -> (Int) -> String = SomeType.f
let _: (Int, Int) -> (Int) = (+)
let _: (inout Int, Int) -> () = (+=)
let _: (URL) -> Page = Page.externalSite
```

先ほどのコードの中にも表れていました。


```
static func +=(lhs: inout Self, rhs: Self)

var x = 1 // x: 1
x += 1 // x: 2
```

+= は左辺の引数を関数の内部で書き換えますよね。この引数は内部で書き換えられるように inout が指定されています。

そんな inout ですが、inout で指定した引数が関数の処理が終わった後も使われるような処理を書いた場合





12:18

```
func f(_ arg: inout String) -> () -> () {  
    return {  
        print(arg)  
    }  
}
```

**! error: Escaping closures can only capture
inout parameters explicitly by value**

コンパイルエラーになります。

`inout`はその他にも面白い深い動作をしますよね。

```
func f(_ arg: inout String) {  
    arg = "🐣"  
    arg = "🐓"   
}  
  
var testString = "🥚"   
    didSet {  
        print("changed: \(testString)")  
    }   
}  
  
f(&testString) 
```

さて、こちらのコード、出力結果はどうなるでしょうか。

コードの概要を説明しますと@@まず、この関数 f は inout を指定した引数を受け取り、2回書き換えます。

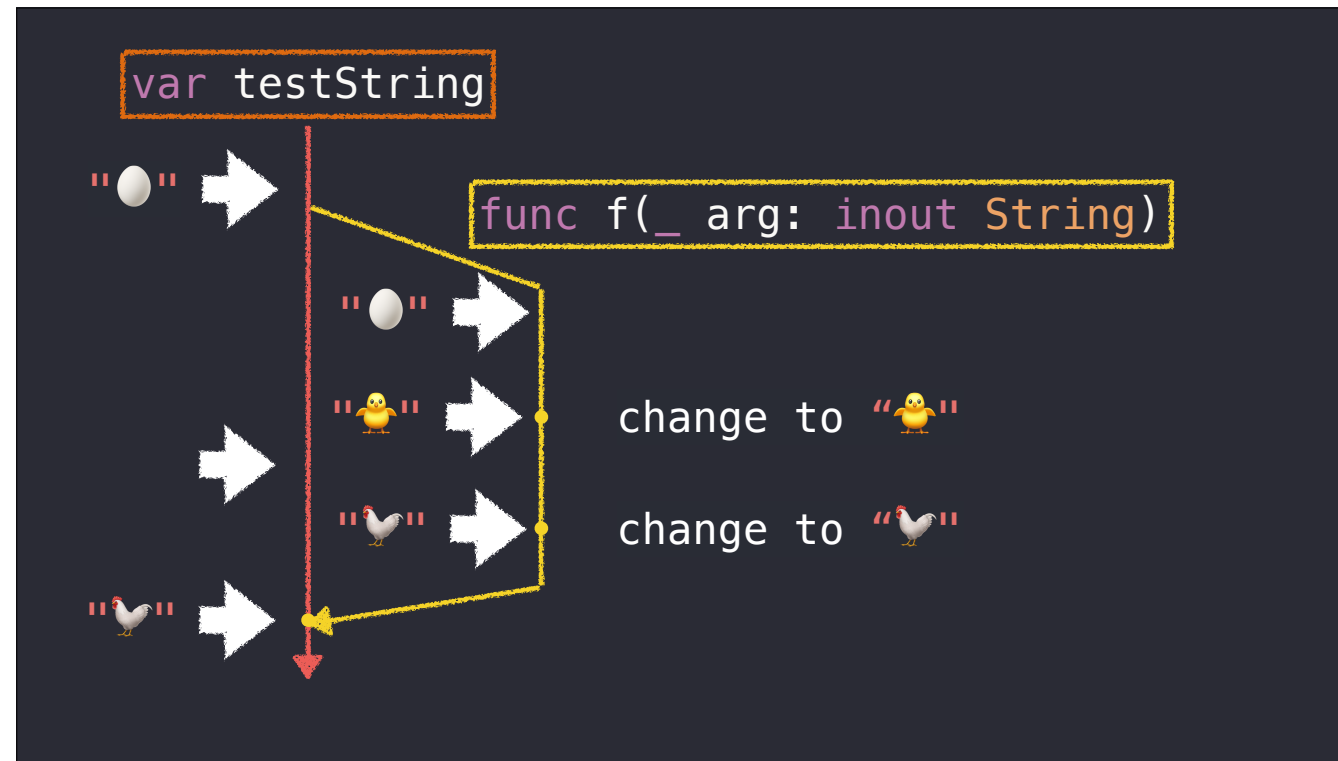
@@

この関数に、初期値が卵の変数を渡します。値の変化を知るために、この変数の値が変化したら、@@

変更後の値を print するようにします。@@

結果は、@@ こちらです。2回変数の値を変更したにも関わらず、最後に代入した値のみが出力されました。

13:21



copy-in copy-out、つまり、関数の最初に値がコピーされ、そのコピーを入れた変数が関数の中では使われます。そして、関数の最後にその変数に入っている値を`inout`指定で渡した変数に代入します。

つまり、関数`f`の中の `arg` は渡した `testString` とは異なる存在です。

先ほどの `inout` の生存期間の保証も、この仕組みの安定性に貢献していますね。

念のため、関数`f`を実行中の、この時点の `testString` はどうなっているでしょうか。

調べてみましょう。

```

func f(_ arg: inout String) {
    arg = "🐥"
    sleep(2)
    arg = "🐓"
}

var testString = "🥚" {
    didSet {
        print("changed: \(testString)")
    }
}
DispatchQueue.global().asyncAfter(deadline: .now() + .seconds(1)) {
    print("current testString: \(testString)")
}
f(&testString)

```

current testString: 🥚
changed: 🐓

関数f の途中、引数をヒヨコに変化させてから鶏になる間に sleep させて、

@@

その隙にこっそり testString の値を覗いてみましょうか。

出力は

@@

こうなります。testStringは初期値の卵のままです！

二つ出力があると紛らわしいですね。

testString の didSet の役目は終わっているので消しておきましょうか。

13:50

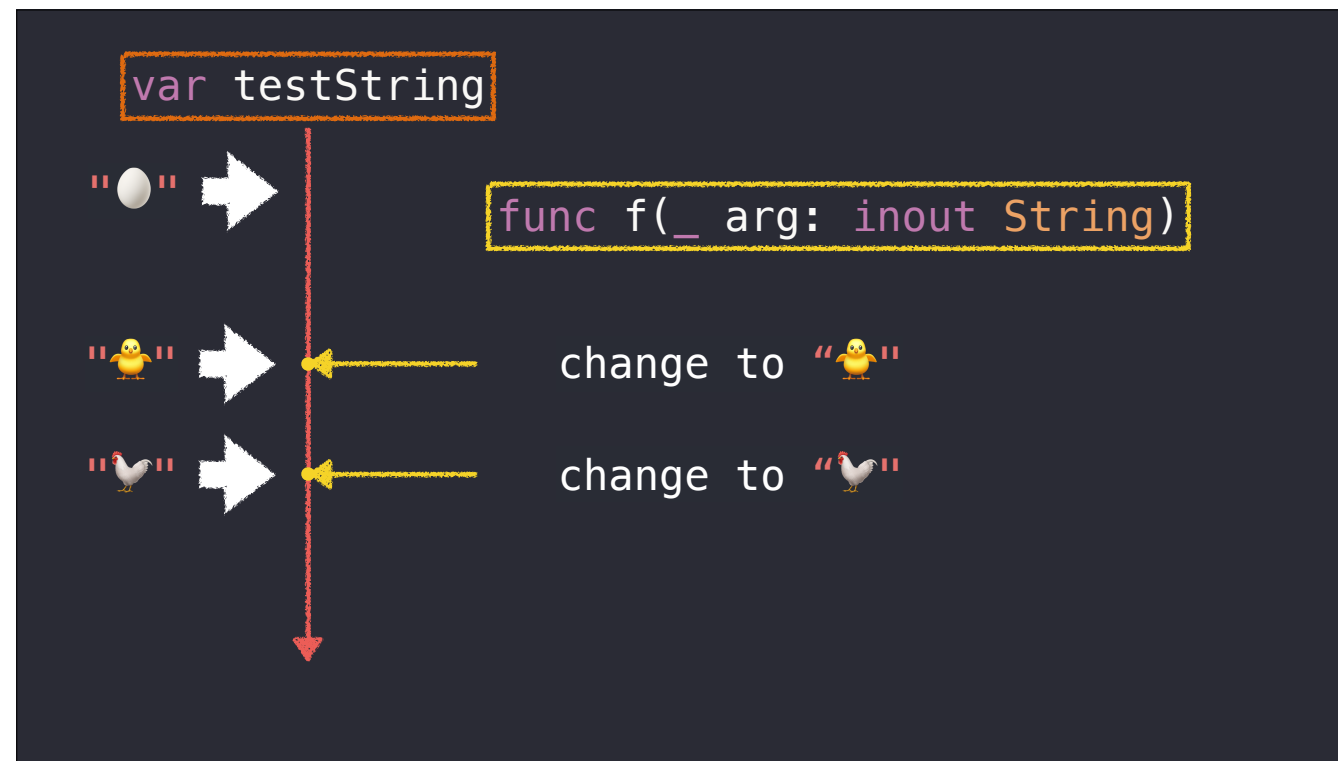
```
func f(_ arg: inout String) {
    arg = "🐥"
    sleep(2)
    arg = "🐓"
}

var testString = "🥚" /* {
    didSet {
        print("changed: \(testString)")
    }
} */
DispatchQueue.global().asyncAfter(deadline: .now() + .seconds(1)) {
    print("current testString: \(testString)")
}
f(&testString)
```

current testString: 🐥

おっと、出力結果が変わってしまいました。didSet を消す前は卵でしたがヒヨコになっています。

ひよこ、というのは、関数fの中で代入した値です。先ほどの理論では、関数が終わるまでは渡した変数は書き変わらないはずですが。



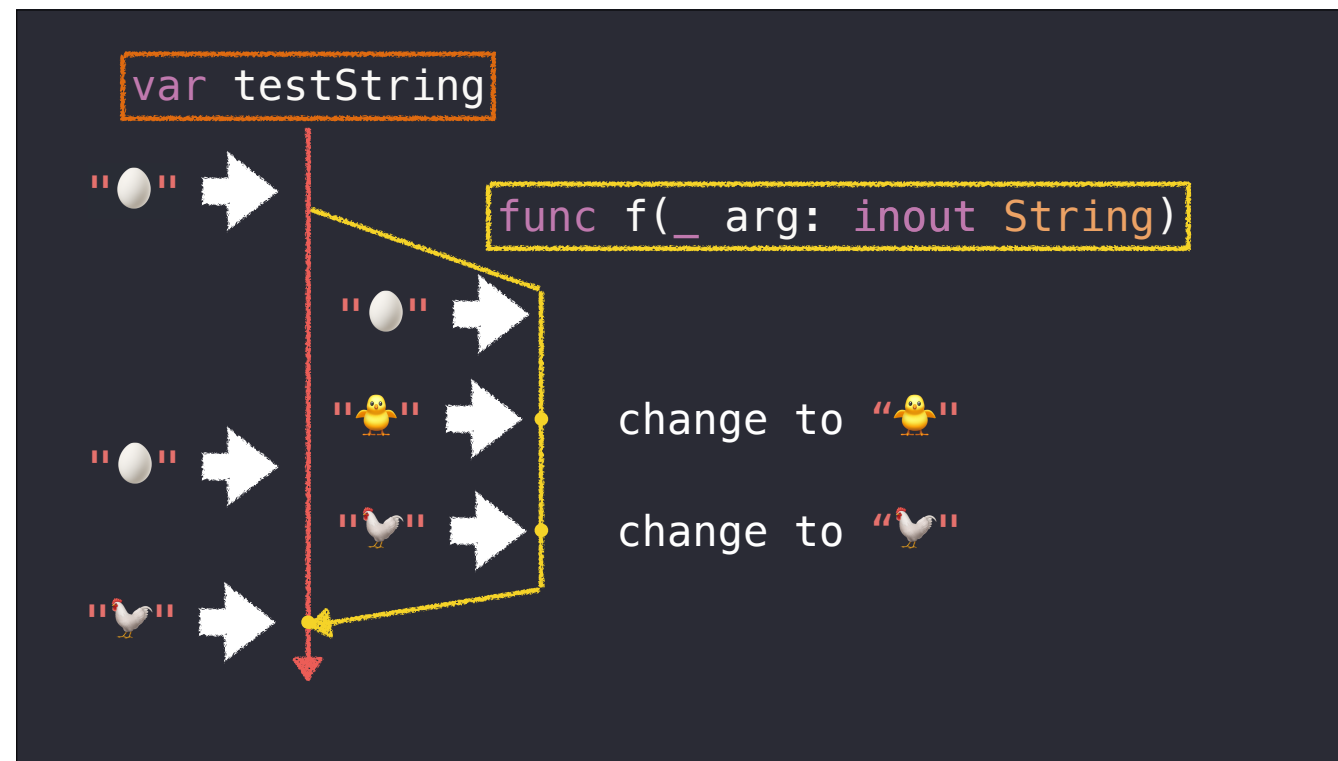
`didSet`, `willSet` を持たない `stored property` はコピーのコストをなくすために`inout`の引数に参照が渡されるのです。

なので、関数`f`の中での変更は渡された引数に即座に反映されます。

このように、関数自体は全く変えていなくても、引数として渡した変数の種類によって動作が異なるのです。もちろん、書き込み処理中の変数を読みに行くのは避けるべきことです。なので、このような動作の違いはは気にしなくて良いのかもしれません。

しかし、非同期処理の制御は難しいので、意図せずそうになってしまうこともありますし...

何より、遭遇した不思議な挙動を探るのって、なんだか楽しいですね。



`inout` を使うと、基本的には `function` の終わりに値が代入されますが、この、「`function` の終わり」というのは具体的にいつでしょうか。
「終わり」と聞いて、思い浮かぶものがあります。

inout

defer return

`return`、そして、`defer`です。

defer

execute after all other code in the scope

defer って何でしたっけ。

たしか...「defer が置かれているスコープの、全ての処理が完了した後に実行されるブロック」...で本当に良いですか？

では、最後に`defer`について。

今まで話してきたような、用意された、明確な答えに向かっていくのではなく、

私がいつも現象を探って行く、思考の過程を大事にしながら

3つのコードからその特異性を探って終わりにしましょう。

x++, ++x or neither?

```
/* ??? */ func ++(x: inout Int) -> Int {  
  defer { x += 1 }  
  return x  
}
```

まず、一つ目のコードです。

@@

inout, defer, return と、最後に何かする処理のオンパレードです。

この関数に、0が入っている変数を渡した場合、その変数と戻り値は最終的に何になるのでしょうか。

それでは、紐解いていきましょう。

defer はスコープを抜ける直前、つまりスコープの中の全ての処理が終わったら実行されます。全ての処理とはなんのでしょうか。今回の場合、return される値を決定する処理も含まれます。なので、ここは@@引数そのままの値である 0 で確定します。

これでスコープの中の全ての処理が終わりました！

ここで defer block が実行されます。@@ x は +1 されます。そして、この関数の中の処理が本当に終わりです。inout の効果で@@引数として渡した変数にxの値が反映されます。

x++

```
postfix func ++(x: inout Int) -> Int {  
    defer { x += 1 }  
    return x  
}
```

実は、この関数は、以前 Swift にも存在していた演算子、x++ の再実装です。

では、

```
func f() {  
    let v: String  
  
    defer { print(v) }  
  
    v = "init value"  
}
```

2つめのコードです。

このコードはコンパイルに成功するでしょうか。

@@

初期化前の段階の変数を使おうとすると、canonical SILに変換する時点でコンパイルエラーとなります。

しかし、このコードはコンパイルに成功します。

スコープを抜けるの全てのパスで変数の初期化をしているならばコンパイルが通るのです。

defer ブロックの「最後に実行される」という性質は、

```
func f() {  
    let v: String  
  
    v = "init value"  
  
    defer { print(v) }  
}
```

もっと言うと、スコープを抜ける場所にブロックごと移動しているイメージとしてもみることができます。

移動するブロック、なにやら怪しい雰囲気です。では、最後にその怪しさをもっと堪能して終わりにしましょう。

```
class SomeType {  
    var v: String { didSet { ... } }  
  
    init() {  
        defer { v = "🐦 in defer" }  
        v = "🐱"  
        v = "🐶"  
    }  
}
```

initializer に舞台を移します。

とある class があります。一つのストアプロパティを持っていて、値の変更を監視しています。

この initializer 内で didSet は何度呼ばれるでしょうか？

答えは、1回 です。

didSet を呼んだのは

@@

defer ブロックの中での代入です。

initializer 内でプロパティに代入しても didSet, willSet が呼ばれないはずですが、呼ばれてしまいます。

deinit でも同じ現象が起こります。defer のこの動きはバグとしても報告されています。

もしも初期化、解放処理を didSet, willSet に任せていたとしたら、または、原則に従ってそれらが呼ばれない前提で書いていたとしたら、私たちの意図していない動作となっているかもしれません。

```
deinit {  
  defer { v = "goodbye 🕊 in defer" }  
  changeV()  
  _ = { v = "goodbye 🐶 in block" }()  
}
```

その他に didSet, willSet が呼ばれる現象が起きるのは、function やクロージャ内で値を変更した場合です。

SILを見ると、defer block は 関数のような扱いを受けていますし、

ここまで見てくると「スコープを抜ける最後に実行される」では止められないほど、defer が深い存在に思えてきますね。

Swift Tour

=

```
(obj?.myVariable = 50)
    .map { /* ... */ }
{ while(true) {} }()
```

Closure

```
let _: (URL) -> Enum = Enum.case
let _: (inout Int, Int) -> () = (+=)
```

時間が来てしまいました。

最後にみなさんと見て歩いた道のりを振り返ってみます。

代入演算から始まり、演算子の Optional chaining、そして Closure について話しました。

inout

```
func f(_ arg: inout String) -> () -> () {  
    return {  
        print(arg)  
    }  
}
```

copy-in copy-out / Optimized

```
inout defer return
```

そして、その途中で何気なく登場した inout について。

defer

```
init() {  
    defer { v = "🕊 in defer" }  
    v = "🐱 initialize"  
}  
  
deinit {  
    v = "goodbye 🐱"  
    defer { v = "goodbye 🕊 in defer" }  
}
```

最後に defer の特異性を一緒に探っていきましょう。

Swift Tour

- myVariable = 50
- obj?.myVariable = 50
- Closure
- enum
- Never
- inout
- return
- didSet
- defer
- init, deinit

この発表で登場したのは、すべて日頃使うような Swift の標準的な機能でした。

しかし、慣れ親しんだからこそ見落としてしまうこと、そして、慣れ親しんだからこそ、気づき、理解できることがあると思います。

今日はそんな、一周回ってからの Swift と、その楽しさをお話ししました。

Thank you for listening!!

この3日間がみなさんにとって素晴らしい時間となりますように。
ご静聴ありがとうございました！