CSC 471 / 371 Mobile Application Development for iOS

Prof. Xiaoping Jia
School of Computing, CDM
DePaul University
xjia@cdm.depaul.edu
@DePaulSWEng

Outline

- Optional types
- Unwrapping optional values
- Optional binding
- Optional chaining
- Implicitly unwrapped optional types

DePaul University 2

A Swift Primer, Part 2 Optional Types

Representing a Missing Value

- Consider the following code

```
let inputString = input from user
let num = Int(inputString)
```

Expecting string representing an integer

Convert a string to integer

- What should the type of `num` be?
  - `Int`
    - What if the input string is *not* a well-formed integer?
    - An integer with a *possibly* invalid or missing value
  - How to represent a invalid or missing value?

    `NULL null nil 0 -1 INT_MAX`

    throw an exception

DePaul University 4

Special Value `nil`

- Swift adopts a different approach for Java, C++, even Objective-C
- Introducing a special value `nil`
  - Represents a missing value, i.e., the absence of a valid value or object
  - Not equal to any valid value or object.
    - It is not `0`, `-1`, etc.
  - Can only be used in an *optional type*
  - Does not belong to any non-optional type
- Values of non-optional types, i.e., regular types, can *never* be `nil`

A new concept. Different from regular types.

DePaul University 5

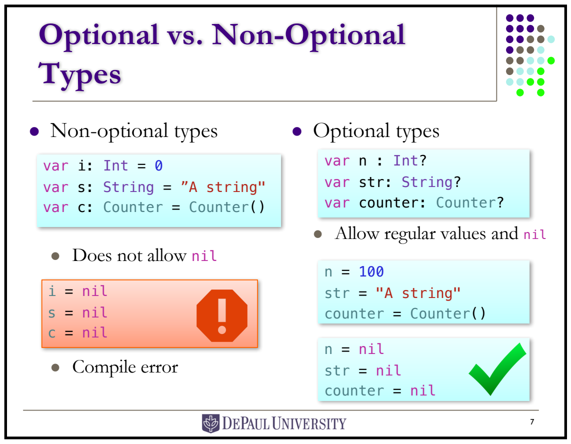
Optional Types

- An *optional type*, or *optional*, is derived from a regular type, i.e., non-optional, *Type*, by appending a ?

  Type?
  - Any non-optional type, basic type or class type, can become an optional type

    `Int?` `Counter?`
  - An optional type is a wrapper that can wrap any type
- An optional type *Type*? means, it contains either
  - A valid value or object of *Type*, or
  - No value at all, which is represented as `nil`
- Synonyms: nullable types, non-nullable types

DePaul University 6



Optional vs. Non-Optional Types

- Non-optional types

```
var i: Int = 0
var s: String = "A string"
var c: Counter = Counter()
```

  - Does not allow `nil`

```
i = nil
s = nil
c = nil
```

  - Compile error

- Optional types

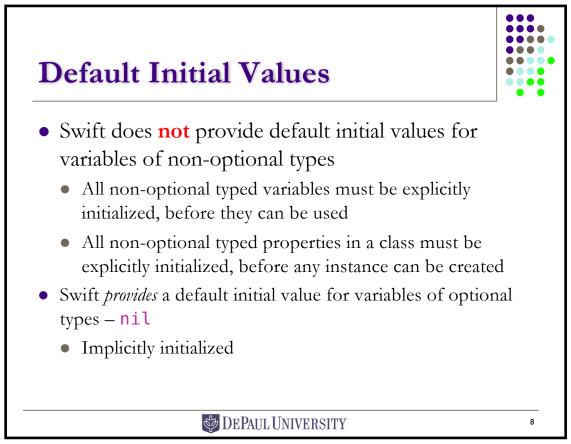
```
var n : Int?
var str: String?
var counter: Counter?
```

  - Allow regular values and `nil`

```
n = 100
str = "A string"
counter = Counter()
```

```
n = nil
str = nil
counter = nil
```

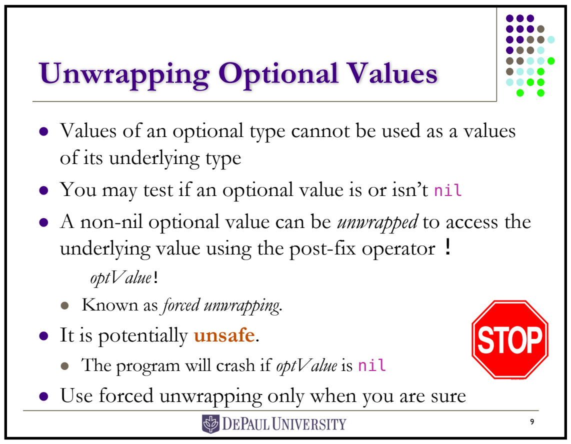
DEPAUL UNIVERSITY 7



Default Initial Values

- Swift does **not** provide default initial values for variables of non-optional types
  - All non-optional typed variables must be explicitly initialized, before they can be used
  - All non-optional typed properties in a class must be explicitly initialized, before any instance can be created
- Swift *provides* a default initial value for variables of optional types – `nil`
  - Implicitly initialized

DEPAUL UNIVERSITY 8



Unwrapping Optional Values

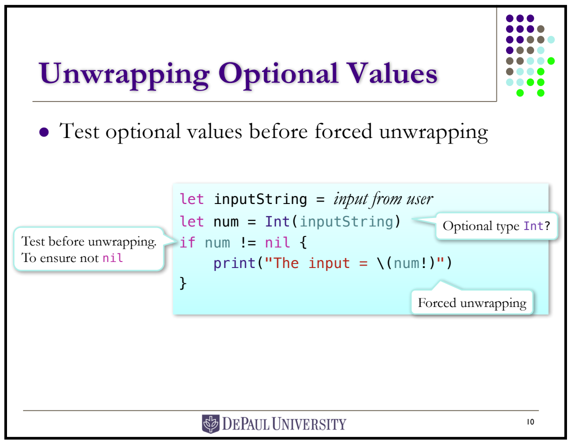
- Values of an optional type cannot be used as a values of its underlying type
- You may test if an optional value is or isn't `nil`
- A non-nil optional value can be *unwrapped* to access the underlying value using the post-fix operator **!**

  optValue!

  - Known as *forced unwrapping*.
- It is potentially **unsafe**.
  - The program will crash if *optValue* is `nil`
- Use forced unwrapping only when you are sure

STOP

DEPAUL UNIVERSITY 9



Unwrapping Optional Values

- Test optional values before forced unwrapping

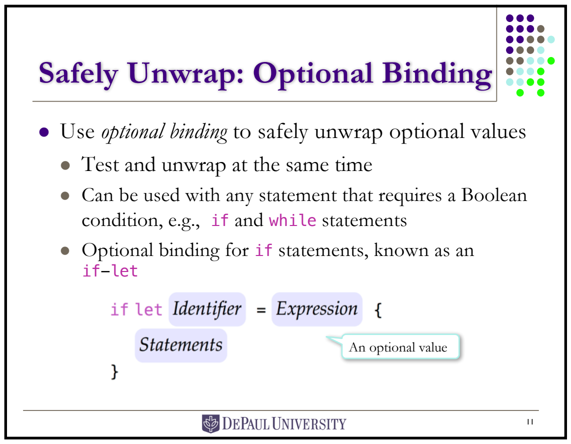
```
let inputString = input from user
let num = Int(inputString)
if num != nil {
    print("The input = \(num!)")
}
```

Optional type `Int?`

Test before unwrapping. To ensure not `nil`

Forced unwrapping

DEPAUL UNIVERSITY 10



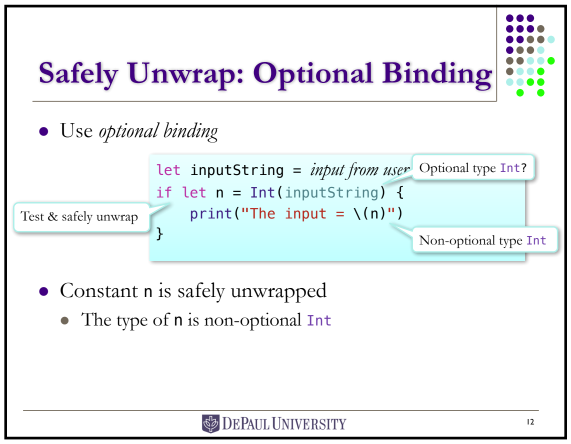
Safely Unwrap: Optional Binding

- Use *optional binding* to safely unwrap optional values
  - Test and unwrap at the same time
  - Can be used with any statement that requires a Boolean condition, e.g., `if` and `while` statements
  - Optional binding for `if` statements, known as an `if-let`

```
if let Identifier = Expression {
    Statements
}
```

An optional value

DEPAUL UNIVERSITY 11



Safely Unwrap: Optional Binding

- Use *optional binding*

```
let inputString = input from user
if let n = Int(inputString) {
    print("The input = \(n)")
}
```

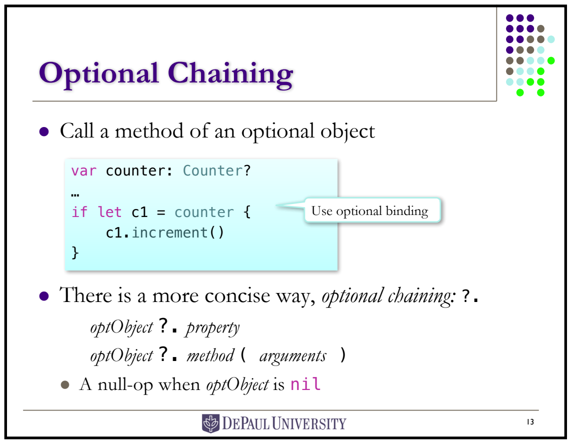
Optional type `Int?`

Test & safely unwrap

Non-optional type `Int`

- Constant n is safely unwrapped
  - The type of n is non-optional `Int`

DEPAUL UNIVERSITY 12



Optional Chaining

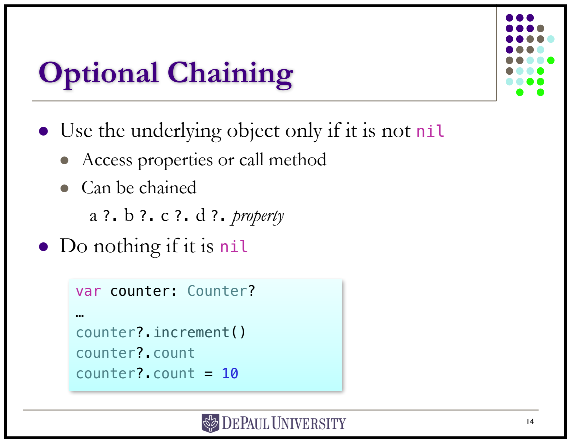
- Call a method of an optional object

```
var counter: Counter?
…
if let c1 = counter {
   c1.increment()
}
```

Use optional binding

- There is a more concise way, *optional chaining:* `?.`
  - *optObject* `?.` *property*
  - *optObject* `?.` *method* `(` *arguments* `)`
  - A null-op when *optObject* is `nil`

DEPAUL UNIVERSITY 13

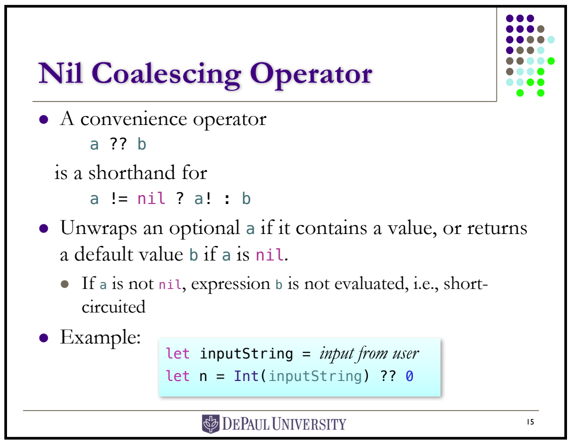


Optional Chaining

- Use the underlying object only if it is not `nil`
  - Access properties or call method
  - Can be chained
    - a `?.` b `?.` c `?.` d `?.` *property*
- Do nothing if it is `nil`

```
var counter: Counter?
…
counter?.increment()
counter?.count
counter?.count = 10
```

DEPAUL UNIVERSITY 14

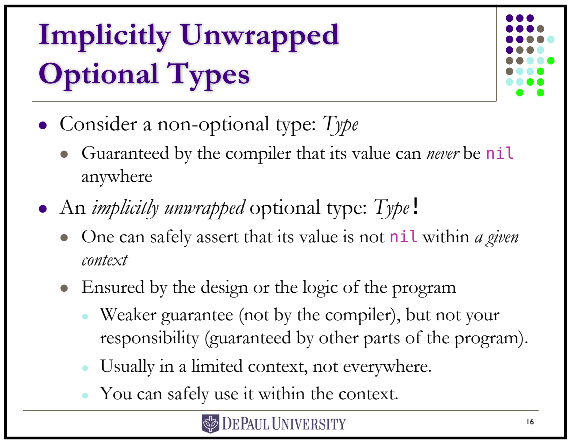


Nil Coalescing Operator

- A convenience operator
  - `a ?? b`
  - is a shorthand for
  - `a != nil ? a! : b`
- Unwraps an optional `a` if it contains a value, or returns a default value `b` if `a` is `nil`.
  - If `a` is not `nil`, expression `b` is not evaluated, i.e., short-circuited
- Example:

```
let inputString = input from user
let n = Int(inputString) ?? 0
```

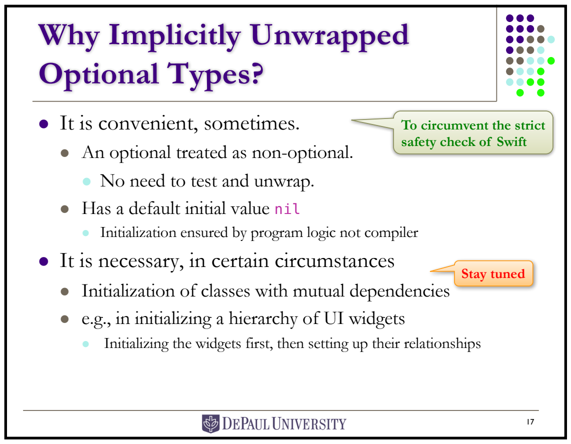
DEPAUL UNIVERSITY 15



Implicitly Unwrapped Optional Types

- Consider a non-optional type: *Type*
  - Guaranteed by the compiler that its value can *never* be `nil` anywhere
- An *implicitly unwrapped* optional type: *Type*`!`
  - One can safely assert that its value is not `nil` within *a given context*
  - Ensured by the design or the logic of the program
    - Weaker guarantee (not by the compiler), but not your responsibility (guaranteed by other parts of the program).
    - Usually in a limited context, not everywhere.
    - You can safely use it within the context.

DEPAUL UNIVERSITY 16



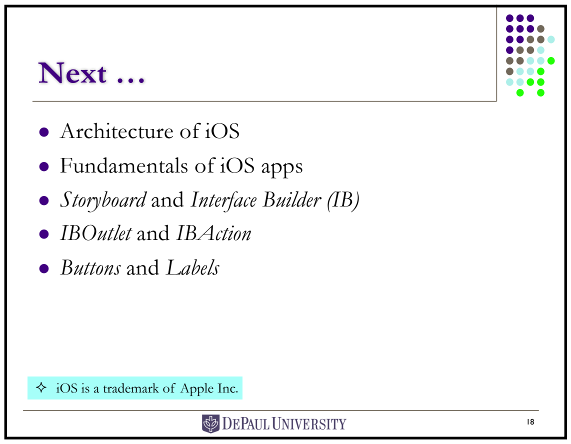
Why Implicitly Unwrapped Optional Types?

- It is convenient, sometimes.
  - An optional treated as non-optional.
    - No need to test and unwrap.
  - Has a default initial value `nil`
    - Initialization ensured by program logic not compiler
- It is necessary, in certain circumstances
  - Initialization of classes with mutual dependencies
  - e.g., in initializing a hierarchy of UI widgets
    - Initializing the widgets first, then setting up their relationships

To circumvent the strict safety check of Swift

Stay tuned

DEPAUL UNIVERSITY 17



Next …

- Architecture of iOS
- Fundamentals of iOS apps
- *Storyboard* and *Interface Builder (IB)*
- *IBOutlet* and *IBAction*
- *Buttons* and *Labels*

✧ iOS is a trademark of Apple Inc.

DEPAUL UNIVERSITY 18