

Swift Primer, Part 5
Protocols and
Extensions

1

Outline

• Protocols
• Extensions

• DePaul University

Protocols

- A protocol defines a blueprint of methods, properties, and initializer requirements
  - Does not provide implementations
  - A class with no implementation, declaration only.
  - Some requirements can be optional
- Implementation of the requirements are provided by classes, structs, or enums that *adopt* the protocol
- A protocol defines the capabilities or services provided by those that adopt the protocol
- · Protocols are analogous to interfaces in Java

DEPAUL UNIVERSITY

4

2

3

## **Adopting Protocols**

- A class, a struct, or an enum conforms to or adopts a protocol
  - If it implements all the non-optional requirements of the protocol
- Syntax

class ClassName : SuperClass , Protocols ...
struct StructName : Protocols ...
enum EnumName : RawValueType , Protocols ...

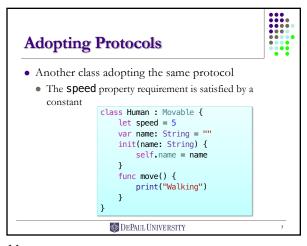
Multiple protocols can be adopted using a comma-separated list

DEPAUL UNIVERSITY

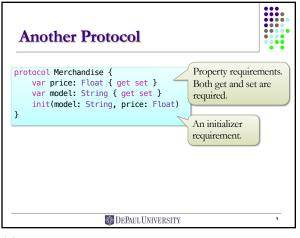
**A Simple Protocol** protocol Movable { A method requirement. func move() A property requirement. Get is required. Set is optional. var speed: Int { get } <</pre> class Car : Vehical { • Adopting the protocol override init() { class Vehical : Movable { super.init() speed = 35var speed: Int = 0 func move() { print("Moving") override func move() { print("Driving") DEPAUL UNIVERSITY

10

5



11 13



```
Adopting Multiple Protocols

class Vehical: Movable, Merchandise {
  var speed: Int = 0
  var price: Float
  var model: String
  required init(model: String, price: Float) {
    self.model = model
    self.price = price
  }
  func move() {
    print("Moving")
  }
}
```

15

14

```
Adopted Protocols in Subclasses

Protocol requirements apply to the subclasses as well.

Implementation of property and method requirements can be inherited

Initializer requirements must be implemented by each subclass.

Class Car: Vehical {
    required init(model: String, price: Float) {
        super.init(model: model, price: price)
        speed = 35
    }
    override func move() {
        print("Driving")
    }
}
```

```
Super Types vs. Super Classes

class Computer: Merchandise {
  var price: Float
  var model: String
  required init(model: String, price: Float) {
    self.model = model
    self.price = price
  }
}

Both Vehicle and Computer are root classes.

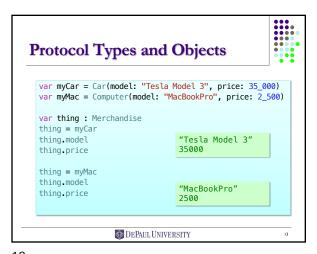
They don't share a common superclass. No code reuse.

They adopt a common protocol, i.e., share a common super type.
```

16 17

© Xiaoping Jia, 2015-2021

2



**Using Protocols for Delegation** 

- Protocols define the interface of the work to be done
- Delegates are the objects that do the work by implementing the methods declared in the protocols
- Using protocols, a class can effectively delegate the implementation and work to other classes
  - Widely used in Cocoa Touch
- A work-around of single inheritance.

DEPAUL UNIVERSITY

19

20

```
The Clock Protocol

protocol Clock {
    var hour: Int { get }
    var minute: Int { get }
}

The get property requirements can be satisfied by
Stored properties, or
Constants, or
Computed properties

DEPAUL UNIVERSITY

15
```

```
An Implementation of
the Clock Protocol

class MyClock: Clock {
  var hour : Int {
    let calendar = Calendar.current
    return calendar.component(.hour, from: Date())
  }
  var minute : Int {
    let calendar = Calendar.current
    return calendar.component(.minute, from: Date())
  }
}

DEPAUL UNIVERSITY
```

21

22

```
An Clock Implementation with

Delegation

class AppleWatch: Computer, Clock {
    var clock = MyClock()

    required init(model: String, price: Float) {
        super.init(model: model, price: price)
    }
    var hour : Int {
        return clock.hour
    }
    var minute : Int {
        return clock.minute
    }
}

DEPAUL UNIVERSITY
```

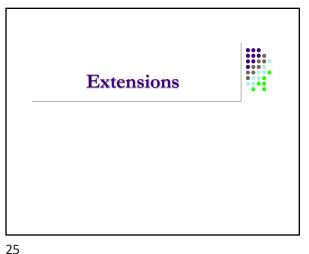
```
An Clock Implementation with Delegation

var clock = MyClock()
clock.hour
clock.minute

var myWatch = AppleWatch(model: "Gold", price: 17_000)
myWatch.hour
myWatch.minute
```

23 24

© Xiaoping Jia, 2015-2021



**Extensions** • Extensions add new functionality to an existing class, structure, or enumeration type. • Even to types in the libraries, without source code • Extensions can add Properties • Methods • Initializers • Extensions can make an existing type conform to a new protocol

DEPAUL UNIVERSITY

26

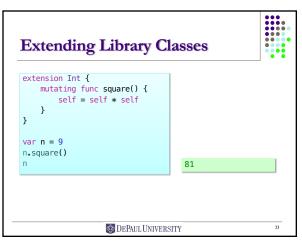
36

A Simple Extension class Human : Movable { func move() { ... } var me = Human(name: "Tim Cook") me.move() extension Human { func think() { print("Cogito ergo sum") me.think() "Cogito ergo sum"

DEPAUL UNIVERSITY

**Extending Library Classes** extension Double { var km: Double { return self \* 1\_000.0 } var m: Double { return self } Assume the var cm: Double { return self / 100.0 } unit is 1 meter var mm: Double { return self / 1\_000.0 } var ft: Double { return self / 3.28084 } let oneInch = 25.4.mm print("One inch is \((oneInch) meters") "One inch is 0.0254 meters" let threeFeet = 3.ft print("Three feet is \((threeFeet) meters") "Three feet is 0.914399970739201 meters" let aMarathon = 42.km + 195.mprint("A marathon is \((aMarathon) meters long") "A marathon is 42195.0 meters long"

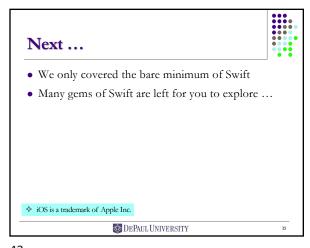
29



```
Extending Library Classes
extension Int {
    func repetitions(task: () -> ()) {
        for _ in 0...self {
                                           A closure
            task()
        }
    }
                                                6 times
5.repetitions(of: {
                                               "Hello!
    print("Hello!")
                                                "Hello!"
})
                                                "Hello!"
"Hello!"
"Hello!"
5.repetitions {
                             Trailing closure
    print("Hello!")
                                                "Hello!"
                   DEPAUL UNIVERSITY
```

38 42

© Xiaoping Jia, 2015-2021



43