CSC 471 / 371 Mobile Application Development for iOS

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

Swift Primer, Part 5 Protocols and Extensions

Outline

- Protocols
- Extensions

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

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

A Simple Protocol

```
protocol Movable {
    func move()
    var speed: Int { get }
}
```

A method requirement.

A property requirement. Get is required. Set is optional.

- Adopting the protocol

```
class Vehical : Movable {
    var speed: Int = 0
    func move() {
        print("Moving")
    }
}
```

```
class Car : Vehical {
    override init() {
        super.init()
        speed = 35
    }
    override func move() {
        print("Driving")
    }
}
```

Adopting Protocols

- Another class adopting the same protocol
  - The `speed` property requirement is satisfied by a constant

```
class Human : Movable {
    let speed = 5
    var name: String = ""
    init(name: String) {
        self.name = name
    }
    func move() {
        print("Walking")
    }
}
```

DePaul University 7

Protocols are Types

- The classes, structs, and enums that adopt protocols, are subtypes of the protocols they adopt.

```
var myCar = Car()
var me = Human(name: "Tim Cook")

var movableObject : Movable
movableObject = me
movableObject.move()
movableObject.speed

movableObject = myCar
movableObject.move()
movableObject.speed
```

```
"Walking"
5

"Driving"
35
```

DePaul University 8

Another Protocol

```
protocol Goods {
    var price: Float { get set }
    var model: String { get set }
    init(model: String, price: Float)
}
```

Property requirements. Both get and set are required.

An initializer requirement.

DePaul University 9

Adopting Multiple Protocols

```
class Vehical : Movable, Goods {
    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")
    }
}
```

DePaul University 10

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")
    }
}
```

DePaul University 11

Super Types vs. Super Classes

```
class Computer : Goods {
    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.

DePaul University 12

Protocol Types and Objects

```
var myCar = Car(model: "Tesla", price: 70_000)
var myMac = Computer(model: "MacBookPro", price: 2_500)

var thing : Goods
thing = myCar
thing.model
thing.price

thing = myMac
thing.model
thing.price
```

```
"Tesla"
70,000.0

"MacBookPro"
2,500.0
```


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.

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

An Implementation of the Clock Protocol

```
class MyClock: Clock {
    var hour : Int {
        let calendar = NSCalendar.currentCalendar()
        let components = calendar.components(
            .CalendarUnitHour,fromDate: NSDate())
        return components.hour
    }
    var minute : Int {
        let calendar = NSCalendar.currentCalendar()
        let components = calendar.components(
            .CalendarUnitMinute, fromDate: NSDate())
        return components.minute
    }
}
```


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
    }
}
```

A delegate.

The implementation is provided by the delegate.

An Clock Implementation with Delegation

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

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

Extensions

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

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"
```


Extending Library Classes

```
extension Double {
    var km: Double { return self * 1_000.0 }
    var m:  Double { return self }
    var cm: Double { return self / 100.0 }
    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")

let threeFeet = 3.ft
print("Three feet is \(threeFeet) meters")

let aMarathon = 42.km + 195.m
print("A marathon is \(aMarathon) meters long")
```

Assume the unit is 1 meter

```
"One inch is 0.0254 meters"
"Three feet is 0.914399970739201 meters"
"A marathon is 42195.0 meters long"
```

Extending Library Classes

```
extension Int {
    mutating func square() {
        self = self * self
    }
}

var n = 9
n.square()
n
```

```
81
```


Extending Library Classes

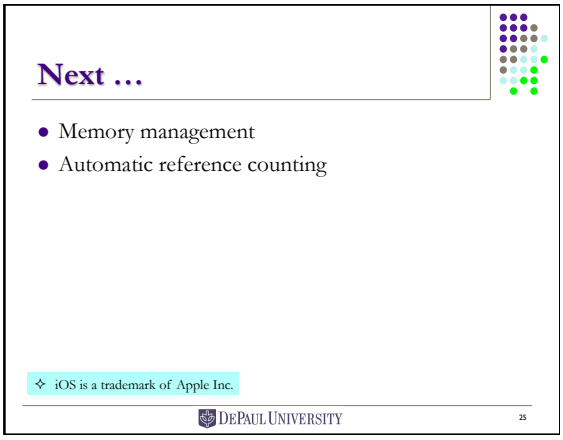
```
extension Int {
    func repetitions(task: () -> ()) {
        for _ in 0...self {
            task()
        }
    }
}

5.repetitions({
    print("Hello!")
})
5.repetitions {
    print("Hello!")
}
```

A closure

Trailing closure

6 times

```
"Hello!"
"Hello!"
"Hello!"
"Hello!"
"Hello!"
"Hello!"
```


Next …
Memory management
Automatic reference counting
iOS is a trademark of Apple Inc.
DePaul University
25