Introduction to Swift
CMSC 436
Protocols

Similar to Java Interface

protocol MyProtocol { ... } struct MyStruct: MyProtocol { ... }

protocol MyOtherProtocol {
    var prop1: Int { get }
    var prop2: String { get set }
    static var typeProp: Int { get set }

    func f() -> Double
    static func g() -> String
    mutating func h()
}
Inheritance

Inheritance looks a lot like Java:

- Type name followed by a colon, then comma-separated list of base, protocols
- Single inheritance, but as many protocols as you need
- Override properties and methods with `override`
- `super` to access superclass’s version
- Prevent overriding with `final`
- Classes can inherit classes, protocols can inherit protocols

There is no common base class, but:

- `Any` is a type that is satisfied by any data type
- `AnyObject` is a protocol that is satisfied by any class
Casting

Test an object `bar` as an instance of type `Foo`:

```swift
bar is Foo (result is a Boolean)
```

Downcast `bar` to type `Foo`:

```swift
bar as? Foo (result is an Optional)
bar as! Foo (result is a Foo, or generates runtime error)
```

In a switch, this can get complex:

```swift
switch thing {
    case 1 as Int: ...
    case let a as String: ...
    case let b as Int where b < 10: ...
    case is Foo: ...
    case let bar as Bar: ...
    default: ...
}
```
Extensions

Allows us to add functionality (including protocols) to an existing type or object:

```swift
extension Int {
    func square() -> Int { self * self }
}

print(3.square())

You can add properties, methods, protocols, nested types, ...
```
Generics (teaser)

Similar to Java:

```swift
struct Stack<Element> {
    var items = [Element]()
    mutating func push(_ item: Element) {
        items.append(item)
    }
    mutating func pop() -> Element {
        return items.removeLast()
    }
}

var tower: Stack<Ring>

func swapTwoValues<T>(_ a: inout T, _ b: inout T) {
    let tempA = a
    a = b; b = tempA
}
```
Generic Extensions

The *name* of the type parameter must match the definition!

```swift
extension Stack {
    var topItem: Element? { // Correct!
        return items.isEmpty ? nil : items[items.count - 1]
    }

    var bottomItem: T? { // Wrong!
        return items.isEmpty ? nil : items[0]
    }
}
```
Type Associations

Protocols can use associatedtype to make them generic-like:

```swift
protocol Container {
    associatedtype Item
    mutating func append(_ item: Item)
    var count: Int { get }
    subscript(i: Int) -> Item { get }
}

struct Stack<Element>: Container {
    var items = [Element]()
    mutating func append(_ item: Element) {
        self.push(item)
    }
    // ...
}
```

Item is inferred to be of type Element, which itself will be bound in a concrete instance.
Error Handling

Similar to Java’s exceptions:

```swift
func foo() throws -> Int {
    // ... some stuff
    throw MyErrorType()
}

do {
    let x = try foo()
} catch is MyErrorType { ... }

do {
    let x = try foo()
} catch let e where e is MyErrorType { ... }

do {
    let x = try foo()
} catch MyErrorType(let a) { ... }
```
Optionals, of course:

// x is of type Int?
let x = try? foo()

// y is of type Int (forced unwrapping)
let y = try! foo() // runtime error if throws

// z is of type Int in the block (optional binding)
if let z = try? foo() { ... }
Error Handling

The `defer` keyword defines a block executed at the end of scope.

Similar to `finally` in Java.

The block can be anywhere in the appropriate scope, and is useful for ensuring cleanup.
Swift uses *Automatic Reference Counting*:  
- allocate correctly, and it does everything for you  
- works for classes and closures (reference data)  
- *does not* work for structures and enumerations (value data) 

Object has no *strong references* ⇒ reclaimed 

Strong references via  
- properties  
- constants  
- variables
How Reference Counting Works

Create class instance $\Rightarrow$ memory allocated
How Reference Counting Works

Create class instance ⇒ memory allocated

Instance is one reference
How Reference Counting Works

Create class instance $\Rightarrow$ memory allocated

Instance is one reference

Instance/property passed to another class $\Rightarrow$ reference count increased
How Reference Counting Works

Create class instance ⇒ memory allocated

Instance is one reference

Instance/property passed to another class ⇒ reference count increased

Methods/objects holding references go away ⇒ reference count decreased
How Reference Counting Works

Create class instance $\Rightarrow$ memory allocated

Instance is one reference

Instance/property passed to another class $\Rightarrow$ reference count increased

Methods/objects holding references go away $\Rightarrow$ reference count decreased

Reference count reaches 0 $\Rightarrow$ memory freed
Object Lifecycle

1. Allocation (from stack or heap)

2. Initialization (init() method)

3. Usage

4. Deinitialization (deinit() method)

5. Deallocation (memory returned)
What Happens Here?

class Person { // references: 0
    let name: String
    init(name: String) { self.name = name }
    var apartment: Apartment?
    deinit { print("\(name) is being deinitialized") } }

class Apartment { // references: 0
    let unit: String
    init(unit: String) { self.unit = unit }
    var tenant: Person?
    deinit { print("Apartment \(unit) is being deinitialized") } }

var john: Person? = Person(name: "John Doe")
var unit4A: Apartment? = Apartment(unit: "4A")
john!.apartment = unit4A
unit4A!.tenant = john
john = nil
unit4A = nil
What Happens Here?

class Person { // references: 1
    let name: String
    init(name: String) { self.name = name }
    var apartment: Apartment?
    deinit { print("\(name) is being deinitialized") } 
}

class Apartment { // references: 0
    let unit: String
    init(unit: String) { self.unit = unit }
    var tenant: Person?
    deinit { print("Apartment \(unit) is being deinitialized") } 
}

var john: Person? = Person(name: "John Doe")
var unit4A: Apartment? = Apartment(unit: "4A")
john!.apartment = unit4A
unit4A!.tenant = john
john = nil
unit4A = nil

Allocate a new Person
Strong reference from john
What Happens Here?

```swift
class Person { // references: 1
    let name: String
    init(name: String) { self.name = name }
    var apartment: Apartment?
    deinit { print("@\(name) is being deinitialized") } }

class Apartment { // references: 1
    let unit: String
    init(unit: String) { self.unit = unit }
    var tenant: Person?
    deinit { print("Apartment @\(unit) is being deinitialized") } }

var john: Person? = Person(name: "John Doe")
var unit4A: Apartment? = Apartment(unit: "4A")
john!.apartment = unit4A
unit4A!.tenant = john
john = nil
unit4A = nil

Allocate a new Apartment
Strong reference from unit4A
```
What Happens Here?

class Person { // references: 1
    let name: String
    init(name: String) { self.name = name }
    var apartment: Apartment?
    deinit { print("\(name) is being deinitialized") } }

class Apartment { // references: 2
    let unit: String
    init(unit: String) { self.unit = unit }
    var tenant: Person?
    deinit { print("Apartment \(unit) is being deinitialized") } }

var john: Person? = Person(name: "John Doe")
var unit4A: Apartment? = Apartment(unit: "4A")
john!.apartment = unit4A
unit4A!.tenant = john
john = nil
unit4A = nil

Pass a unit4A reference to john
Strong reference
What Happens Here?

class Person { // references: 2
    let name: String
    init(name: String) { self.name = name }
    var apartment: Apartment?
    deinit { print("\(name) is being deinitialized") }
}

class Apartment { // references: 2
    let unit: String
    init(unit: String) { self.unit = unit }
    var tenant: Person?
    deinit { print("Apartment \(unit) is being deinitialized") }
}

var john: Person? = Person(name: "John Doe")
var unit4A: Apartment? = Apartment(unit: "4A")
john!.apartment = unit4A
unit4A!.tenant = john
john = nil
unit4A = nil

Pass a john reference to unit4a
Strong reference
What Happens Here?

class Person { // references: 1
    let name: String
    init(name: String) { self.name = name }
    var apartment: Apartment?
    deinit { print("\(name) is being deinitialized") }
}

class Apartment { // references: 2
    let unit: String
    init(unit: String) { self.unit = unit }
    var tenant: Person?
    deinit { print("Apartment \(unit) is being deinitialized") }
}

var john: Person? = Person(name: "John Doe")
var unit4A: Apartment? = Apartment(unit: "4A")
john!.apartment = unit4A
unit4A!.tenant = john
john = nil
unit4A = nil

Remove the strong reference from john
class Person { // references: 1
    let name: String
    init(name: String) { self.name = name }
    var apartment: Apartment?
    deinit { print("(name) is being deinitialized") }
}

class Apartment { // references: 1
    let unit: String
    init(unit: String) { self.unit = unit }
    var tenant: Person?
    deinit { print("Apartment \(unit) is being deinitialized") }
}

var john: Person? = Person(name: "John Doe")
var unit4A: Apartment? = Apartment(unit: "4A")
john!.apartment = unit4A
unit4A!.tenant = john
john = nil
unit4A = nil

Remove the strong reference from unit4A
What Happens Here?

class Person { // references: 1
    let name:String
    init(name: String) { self.name = name }
    var apartment: Apartment?
    deinit { print("\(name) is being deinitialized") }
}

class Apartment { // references: 1
    let unit: String
    init(unit: String) { self.unit = unit }
    var tenant: Person?
    deinit { print("Apartment \(unit) is being deinitialized") }
}

var john:Person? = Person(name: "John Doe")
var unit4A:Apartment? = Apartment(unit: "4A")
john!.apartment = unit4A
unit4A!.tenant = john
john = nil
unit4A = nil

We still have strong references!
Reference Loops

We have a *Strong Cycle*

Reference count never hits 0 ⇒ objects never deinitialized!

We can resolve this with *Weak References* or *Unowned References*

<table>
<thead>
<tr>
<th></th>
<th>Weak</th>
<th>Unowned</th>
</tr>
</thead>
<tbody>
<tr>
<td>nil-able?</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Optional?</td>
<td>required</td>
<td>no</td>
</tr>
<tr>
<td>referenced thing’s lifetime</td>
<td>shorter</td>
<td>longer</td>
</tr>
</tbody>
</table>

Only *strong* references contribute to reference count

Unowned references can lead to bad dereferences!
Declaring Weak/Unowned References

class Person {
    let name: String
    var apartment: Apartment?
    var card: CreditCard?
    init(name: String) { self.name = name }
}

class Apartment {
    let unit: String
    weak var tenant: Person?
    init(unit: String) { self.unit = unit }
}

class CreditCard {
    let number: UInt64
    unowned let customer: Person
    init(number: UInt64, customer: Person) {
        self.number = number; self.customer = customer
    }
}

var john: Person? = Person(name: "John")
var unit4A = Apartment(unit: "4A")
let card = CreditCard(number: 1234_5678_9012_3456, customer: john!)
john!.apartment = unit4A; unit4A.tenant = john
john!.card = card

If John is erased from existence by a shadowy cabal (john=nil):
unit4A.tenant will be nil
card.customer will generate a runtime error!
A Third Case

Both weak and unowned references require an Optional on one side
What if this isn't possible?

class Country {
    let name: String
    var capitalCity: City
    init(nameIn: String, capitalName: String) {
        name = nameIn
        capitalCity = City(nameIn: capitalName, countryIn: self)
    }
}

class City {
    let name: String
    let country: Country
    init(nameIn: String, countryIn: Country) {
        name = nameIn
        country = countryIn
    }
}

var country = Country(nameIn: "Bangladesh", capitalName: "Dhaka")

This is a problem; let’s see why
How Initialization Works

1. Phase 1
   1.1 Initializer ensures **all** properties have values
   1.2 Superclass initializer does the same (all the way up the chain)
   1.3 Initialization **now considered complete**

2. Phase 2
   2.1 Superclass initializer may continue customizing
   2.2 Initializer may continue customizing
      ▶ self may be accessed
      ▶ properties may be modified
      ▶ instance methods may be called
Why Our Previous Code Doesn’t Work

class Country {
    let name: String
    var capitalCity: City
    init(nameIn: String, capitalName: String) {
        name = nameIn
        capitalCity = City(nameIn: capitalName, countryIn: self)
    }
}

class City {
    let name: String
    let country: Country
    init(nameIn: String, countryIn: Country) {
        name = nameIn
        country = countryIn
    }
}

var country = Country(nameIn: "Bangladesh", capitalName: "Dhaka")

We’re trying to use self before Phase 1 completes!
Fixing With Unowned and Implicitly Unwrapped Optionals

class Country {
    let name: String
    var capitalCity: City!
    init(nameIn: String, capitalName: String) {
        name = nameIn
        capitalCity = City(nameIn: capitalName, countryIn: self)
    }
}

class City {
    let name: String
    unowned var country: Country
    init(nameIn: String, countryIn: Country) {
        name = nameIn
        country = countryIn
    }
}

var country = Country(nameIn: "Bangladesh", capitalName: "Dhaka")

City holds an unowned reference to Country
Country holds an Implicitly Unwrapped Reference to City
Optional Country.capitalCity can be initialized with nil
Assigned to non-nil instance during Phase 2 (self is available)
Closures and Reference Cycles

class Person {
    var firstName: String?
    var lastName: String?
    var fullName: () -> String = {
        return "\(self.firstName!) \(self.lastName!)"
    }
}

Closures and Reference Cycles

class Person {
    var firstName: String?
    var lastName: String?
    lazy var fullName: () -> String = {
        [unowned self] in
        return "\(self.firstName!) \(self.lastName!)"
    }
}

Lazy properties are not initialized until they’re called

Specifying unowned or weak in capture list prevents strong cycles in captures