SCIENCES SORBONNE UNIVERSITÉ

Swift, about inheritance

Fabrice.Kordon@lip6.fr

LIP6

As an introduction…

- **Swift is an OO language**
  - So it supports inheritance
- **Simple inheritance**
  - But associated with templates (à la C++)
    - **Not discussed here**
  - Also an extension mechanism
    - **Not discussed here**

Inheritance, an example


```
class Student       {
    private var number = 0
    private var name = ""

    init (nm : String, nu : Int) {
        name = nm
        number = nu
    }

    func identity () -> String {
        return "\(name) (\(number))"
    }

    func myName() -> String {
        return name
    }

    func myNumber() -> Int {
        return number
    }
}
```

Inheritance, an example


```
class PairedStudent : Student {

    private var pair : PairedStudent?

    func setMyPair (p : PairedStudent) {
        pair = p
    }

    func myPair () -> PairedStudent {
        return pair! // I want to crash if there is none
    }

    func myPairOrNone () -> PairedStudent? {
        return pair // May return "nil"
    }
}

let s1 = PairedStudent(nm : "Stan Laurel", nu :1000)
let s2 = PairedStudent(nm : "Oliver Hardy", nu :1001)
let s3 = PairedStudent(nm : "Charlot", nu :1002)

s1.setMyPair(p: s2)
s2.setMyPair(p: s1)

print ("\(s1.myName()) is paired with \(s1.myPair().myName())")
print ("\(s2.myName()) is paired with \(s2.myPairOrNone()?.myName() ?? "nobody")")
print ("\(s3.myName()) is paired with \(s3.myPairOrNone()?.myName() ?? "nobody")")
```

Inheritance, an example


```
class PairedStudent : Student {

    private var pair : PairedStudent?

    func setMyPair (p : PairedStudent) {
        pair = p
    }

    func myPair () -> PairedStudent {
        return pair! // I want to crash if there is none
    }

    func myPairOrNone () -> PairedStudent? {
        return pair // May return "nil"
    }
}
```

```
Stan Laurel is paired with Oliver Hardy
Oliver Hardy is paired with Stan Laurel
Charlot is paired with nobody
```

Initialization?

Be aware, call to super.init after

- Not like Objective-C

```
enum Color {case red, black, yellow}

class Car {
    var paint: Color
    init(c: Color) {
        paint = c
    }
}
class RaceCar: Car {
    var hasTurbo: Bool

    init(c: Color, turbo: Bool) {
        hasTurbo = turbo
        super.init(c: c) // oposite of Objective-C
    }
}
```

Initialization?

Be aware, call to super.init after

- Not like Objective-C

```
enum Color {case red, black, yellow}

class Car {
    var paint: Color
    init(c: Color) {
        paint = c
    }
}
class RaceCar: Car {
    var hasTurbo: Bool

    init(c: Color, turbo: Bool) {
        hasTurbo = turbo
        super.init(c: c) // oposite of Objective-C
    }
}
```

Advanced initialization

«Convenience» versus «designated»

- Strict hierarchical invocations

Advanced initialization

«Convenience» versus «designated»

- Strict hierarchical invocations

Designated initializer

main ones to be used
at least one per class

Advanced initialization

«Convenience» versus «designated»

- Strict hierarchical invocations

Designated initializer

main ones to be used
at least one per class

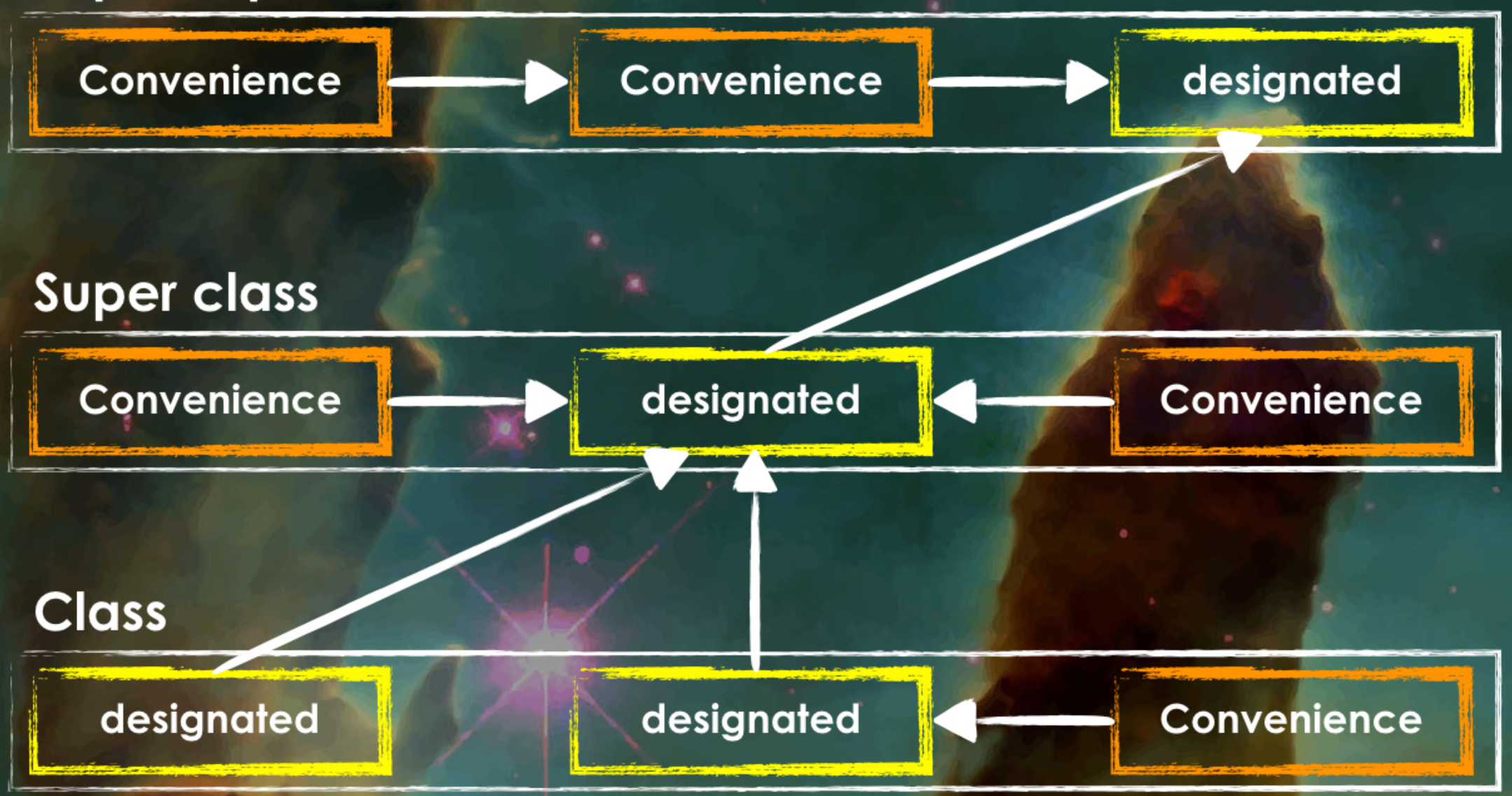
Convenience initializer

Practical for programers
no reference to initializers of the super

Advanced initialization

«Convenience» versus «designated»

- Strict hierarchical invocations



An example


```
class RaceCar: Car {
    var hasTurbo: Bool

    init(c: Color, turbo: Bool) {
        hasTurbo = turbo
        super.init(c: c)
    }

    convenience override init (c: Color) { // overrides init inherited from super
        self.init(c: c, turbo: true)
    }

    convenience init () {
        self.init(c: .red)
    }
}
```

An example


```
class RaceCar: Car {
    var hasTurbo: Bool

    init(c: Color, turbo: Bool) {
        hasTurbo = turbo
        super.init(c: c)
    }

    convenience override init (c: Color) { // overrides init inherited from super
        self.init(c: c, turbo: true)
    }

    convenience init () {
        self.init(c: .red)
    }
}
```

An example


```
class RaceCar: Car {
    var hasTurbo: Bool

    init(c: Color, turbo: Bool) {
        hasTurbo = turbo
        super.init(c: c)
    }

    convenience override init (c: Color) { // overrides init inherited from super
        self.init(c: c, turbo: true)
    }

    convenience init () {
        self.init(c: .red)
    }
}
```

As a conclusion…

Classical OO mechanisms

Embeds all you need for memory management

- Elaborate secure applications…
- Remind
  - **The ecosystem is the challenge**
  - **The ecosystem is enriched by developers**
  - **So programming must be made easy**
  - **The compiler does more work**

- By the way…
  - **Issue identified in the late 60's**
  - **Remember the HOLWG (1975, origin of Ada)**