Swift

Class e Structs

Conteúdos

- O que são classes
- Class vs estruturas
- Como criar classes e estruturas
- Metodos
- Herança

Conteúdos

- Propriedades
    - Lazy property
    - Propriedades computadas
    - Property observer
-

Class

- São o molde para criar objectos
- Tem métodos (funções) e propriedades
- Devem ser declaradas no seu próprio ficheiro
- O seu nome deve começar por maiúscula e ser CamelCase (AlunoUniv)

Class vs estruturas

- As estruturas e class têm muito em comum:
    - Definem propriedades
    - Definem metodos
    - Definem sub-scripts
    - Podem conter um constructor personalizado
    - Entre outras

Class vs estruturas

- as classes podem:
  - Herdar de outras classes
  - Podem ter mais de uma referência

class / structs

- classes são Reference Type
- structs:
  - são value type
  - tem um init predefinido

Criar class

```
class nome{

//codigo da class

}
```

Criar struct

```
struct nome{

//codigo da struct

}
```

Instanciar uma struct

```
struct resulocao{
  var w:Int
  var h:Int
}

let porjector = resulocao(w:800, h:900)
```

Instanciar uma class

```
class aluno{
  var nome:String?
  var idade:Int?


}

let joao = aluno()

joao.nome = “João”
```

Inicializar uma class

```
class Carro{

    private var marca:String
    private var ano:Int

    init(nome:String, ano:Int) {
        self.marca = nome
        self.ano = ano
    }

}

let bmw = Carro(nome:“bmw”, ano:2000)
```

Operador de identicidade

- === -> identica
- !== -> não identica

DEMO

Quando usar Structs

- Agrupar valores simples
- Quando os valores são value type
- Quando não é necessário herdar comportamentos ou propriedades

Quando usar Structs

- Definir formas geométricas
- Coordenadas
- etc

Propriedades

Propriedades

- Uma propriedade é uma variável declarada no top level de uma class
- Há dois tipos de propriedades
    - Instance properties
    - Static/class properties.

.

Instance properties

- *Instance properties*
  - *S*e nada foi dito em contrario uma propriedade e sempre *instance* property.
  - O seu valor pode variar a cada instancia da class
  - A variável “vive” enquanto a instância a qual esta associada “viver”

.

Static/class properties.

- A property is a static/class property if its declaration is preceded by the keyword static or class.

- Its lifetime is the same as the lifetime of the object type.

Local variables

- A local variable is a variable declared inside a function body.
- A local variable lives only as long as its surrounding curly-braces scope lives

Computed Initializer

Computed Initializer

- Por vezes necessitados de ter varias linhas de código para inicializar uma variável, em Swift isto pode ser feito de forma compacta

Computed Initializer

```
let timed : Bool = {
    if val == 1 {
        return true
    } else {
        return false
    }
}()
```

Computed Variables

Computed Variables

- Até agora inicializamos variáveis atribuindo-lhe um valor.
- Em Swift as variáveis pode ser computadas, ou seja em vez de terem valores têm funções
  - setter - chamada quando lhe e atribuído valor
  - getter - chamada quando acedemos ao “valor” ou seja quando esta e referida

Computed Variables

- As variáveis computadas podem ser usadas nas mais vaiadas situações
- Variáveis so de leitura
- Façade for a function
- Façade for other variables

Computed Variables

```
class Quadrado {
    var lado:Float?
    var area:Float {
        set{
            self.lado = sqrt(newValue)
        }

        get{
            return pow(self.lado!, 2)
        }
    }

    var perimetro:Float {

        set{
            self.lado = newValue/4
        }

        get{
            return self.lado! * 4
        }
    }
}
```

Variáveis so de leitura

- As variáveis computadas pode ser apenas de leitura ou seja não sendo possível serem alteradas

Variáveis so de leitura

```
var estado : String {
        get {
            if media >= 9.5 {
                return "Aprovado com media de \(self.media)"
            }else{
                return "não aprovado com media de \(self.media)"
            }
        }
    }
```

Façade for a function

- Variáveis computadas podem retornar apenas o resultado de uma função

Façade for a function

```
var mp : MPMusicPlayerController {
            return MPMusicPlayerController.systemMusicPlayer()
}
```

Façade for other variables

- Uma variável computada pode fazer de “interface” entre o utilizador e o programa, funcionando como um “gatekeeper”

- Semelhante ao getter e setter em C#

Façade for other variables

```
private var _p : String

var p : String {
    get {
        return self._p
    }
    set {
        self._p = newValue
    }
 }
```

Setter Observers

Setter Observers

- Outra forma de injectar funcionalidade em settres
- Os Observers são funções chamadas antes e depois de uma variável ser alterada
- Não são chamados quando a variável e inicializada

Setter Observers

```
var s = "whatever" {
        willSet {
            print(newValue)
        }

        didSet {
            print(oldValue)
        }
}
```

DEMO

Lazy property

Lazy property

- Se uma variável é inicializada com um determinado valor e for usada a lazy initialization este valor só é avaliado quando o programa tenta aceder a essa variável

Lazy property

- Em Swift há 3 tipos de propriedades que podem ser “lazy”
  - Variáveis Globais
  - Static properties
  - Instance properties

Lazy property

- As Lazy instance properties:
  - Não podem ser let
  - Não podem ter observers
  - Não podem ser so de leitura

Lazy property

- Para que serve?

- Em que situações faz sentido que sejam usadas ?

Lazy property

- Quando o valor inicial e “caro” de gerar / ler
- Quando o seu valor depende de factores externos

Lazy property

- Podem fazer coisas que as propriedades normais não podem:
  - Podem-se referir a propria instancia

Lazy property

```
class MyView : UIView {
        lazy var arrow : UIImage =
self.arrowImage()


        func arrowImage () -> UIImage {
            // ... big image-generating
code goes here ...
        }

}
```

Herança

Herança

- Uma class pode herdar métodos e propriedades de outra
    - A class que herda é denominada de sub class
    - A class da qual se herda é denominada de super class
- É possível aceder a métodos e propriedades
- O overriding é possível
    - O swift tem forma de garantir que este processo e feito correctamente

Herança

- Podem ser adicionados observers as propriedades herdadas

Sintax

```
class SubClass: SuperClass{

//code here

}
```

Exemplo

```
class Pessoa{
    var nome:String
    var idade:Int

    init(nome:String,idade:Int ){
        self.nome = nome
        self.idade = idade
    }

    func infos(){
        print("\(self.idade), \
(self.nome)")

    }

}
```

Exemplo

```
class Medico: Pessoa{

    var exp:String

     init(nome: String, idade: Int, exp:String) {
        self.exp = exp
       super.init(nome: nome, idade: idade)
    }

    func dados() {
        super.infos()
        print("-------------")
        print(“nome: \(super.nome),\nage: \(super.idade),\nexp: \(self.exp)")
    }
}
```

Exemplo

```
var rita = Medico(nome: "rita", idade: 30, exp: "clinica geral")
rita.dados()

rita.infos();
```

DEMO

Overriding

Overriding

- Forma de uma sub class alterar a funcionalidade de instance methods, class methods, instance propertys, class propertys e sub-scripts

- Em Swift é necessário indicar de forma explicita que um método re-escreve um método da super class

Overriding

```
class Dog {
        func bark () {
            print("woof")
        }
}

class NoisyDog : Dog {
        override func bark () {
            print("woof woof woof")
        }
}
```

Overriding

- Apesar de re-escrito ainda podemos aceder ao método original

Overriding

```
class Dog {
        func bark () {
            print("woof")
        }
    }

class NoisyDog : Dog {
        override func bark () {
            for _ in 1...3 {
             super.bark()
            }
        }
}

class VeryNoisyDog : Dog {
        override func bark () {
            for _ in 1…5 {
             super.bark()
            }
        }
}
```

Overriding

- Para evitar que algo seja re-escrito deve ser declarado como final

Overriding

```
class Dog {
        final func bark () {
             print("woof")
         }
}
```

Overriding

- Com o Overriding a propriedades podemos adicionar gettres e settres observers próprios

- A class que herda não sabe nada das propriedades herdades para alem do seu nome e tipo

Overriding

```
class SuperCar:Car{
    var gear = 1
    override var desc: String{
      return super.desc + “ in gear \
      (gear)”
    }
}
```

Protocolos

Protocolos

- Implementam um “blueprint” de métodos e propriedades
- Forma de partilhar propriedades e funcionalidades por objectos não relacionados
- Atribui um novo tipo ao objecto
- Semelhante as interfaces do Java / C#

Protocolos

```
protocol someProtocol {

//code goes here

}
```

Protocolos

- Podem ser implementados tanto em class como em struts

Protocolos

```
struct structName: protocolA, protocolB {

//code goes here

}
```

Protocolos

```
class className: superClass, protocolA, protocolB {

//code goes here

}
```

Protocolos

- Tudo o que estiver declarado no protocolo **deve** ser implementado

- Pode implementar protocolos

- Se o protocolo implementa uma propriedade

    - Com getter e settres nunca poderá ser declarada como constante ou de leitura

    - Se apenas tiver implementado o getter, pode ser declarada como qualquer propriedades

Protocolos

```
protocol SomeProtocol {

  var someInt:Int {get set}
  var someOtherInt:Int {get}

}
```

Protocolos

```
protocol SomeProtocol2 {

  func random() -> Double

}
```

Protocolos

```
class className: SomeProtocol, SomeProtocol2 {

    var someInt: Int = 0;

    var someOtherInt: Int = 0;

    func random() -> Double{

      //code goes here

    }

}
```

Qual a importancia dos Protocolos?

Protocolos

- Usado em alguns design patterns
- Polimorfismo