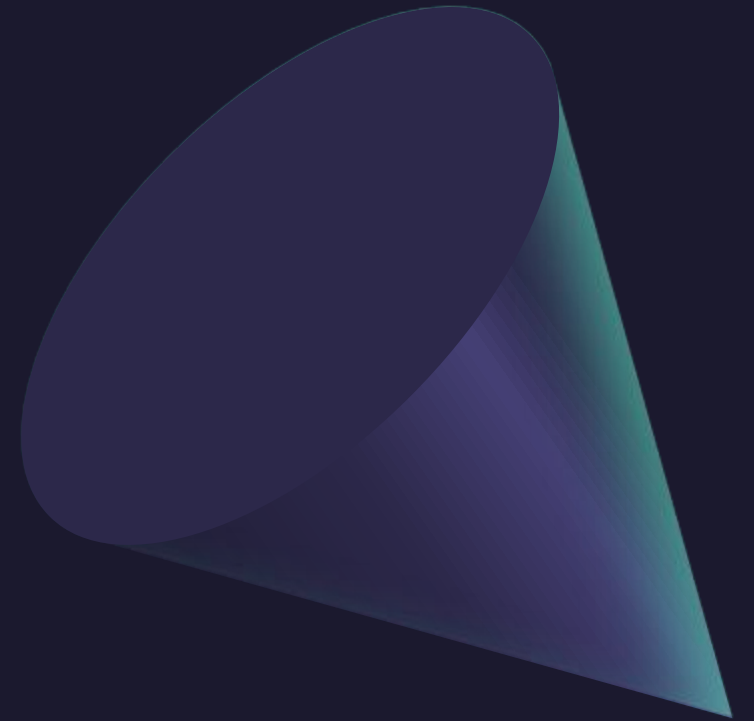


Expressividade em Linguagens de Programação

Alunos:

Kaylan Rocha Freitas Rosa

Pedro Henrique Marcos Rocancourt Cavadas

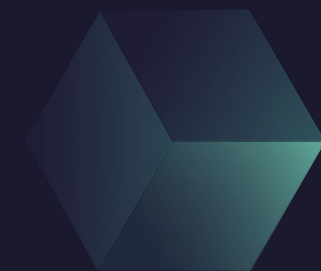


Origens e influências:

- Swift é uma linguagem de programação multiparadigma desenvolvida pela Apple em 2014, com o objetivo de substituir o Objective-C como linguagem padrão de criação de apps para iOS, Mac, Apple TV e Apple Watch
- Teve fortes influências de Rust, Haskell, Ruby, Python, C#, CLU, e, claro, Objective-C

Histórico de Versões

Data de Lançamento	Versão
09/09/2014	Swift 1.0
22/10/2014	Swift 1.1
08/04/2015	Swift 1.2
21/09/2015	Swift 2.0
13/09/2016	Swift 3.0
19/09/2017	Swift 4.0
29/03/2018	Swift 4.1
17/09/2018	Swift 4.2
25/03/2019	Swift 5.0



Classificação da linguagem:

- Multiparadigma: possui funcionalidades imperativas, funcionais, e de orientação a objetos
- Híbrida: possui características tanto de linguagens dinâmicas quanto estáticas. Por exemplo, possui um REPL modernizado, chamado Playground, assim como tipagem estática.
- Usos: criação de aplicativos para sistemas proprietários da Apple, como iOS, Mac, Apple TV e Apple Watch

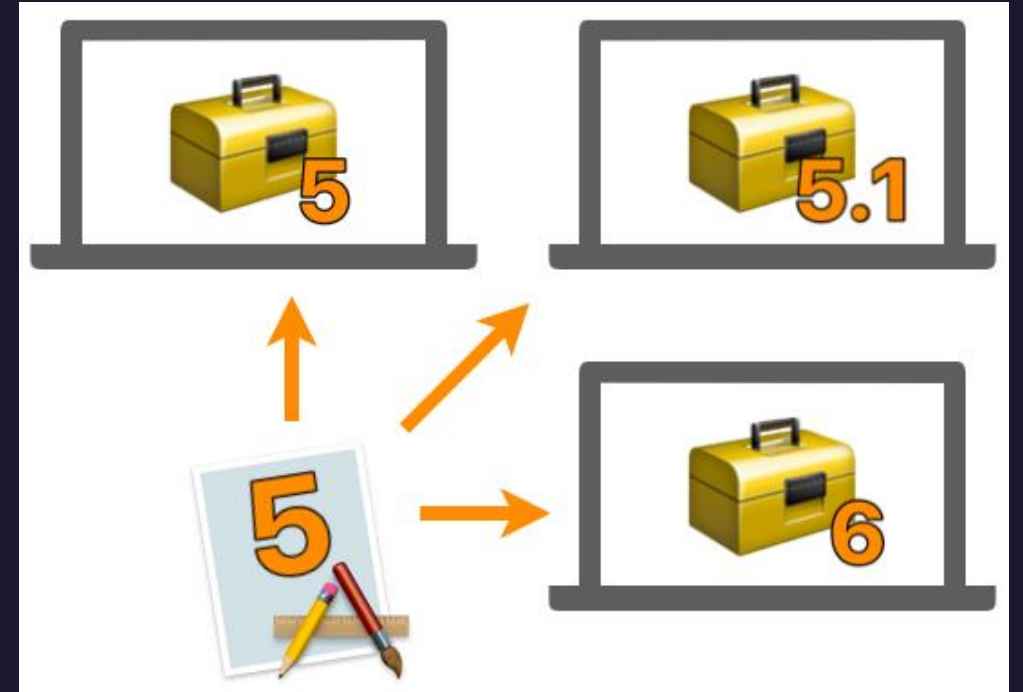


Chris Lattner

Funcionalidade 1:

Swift ABI Stability

Quando uma linguagem possui uma ABI-estável (Interface Binária do Aplicativo), isso significa que ela está empacotada e vinculada ao próprio sistema operacional, neste caso, iOS. O código Swift que você compila em seu computador tem uma interface binária própria sistema operacional, em vez de qualquer biblioteca dinâmica que você empacota com seu aplicativo. Por causa disso, a Apple deve ser capaz de garantir que seu código Swift, quando compilado para código de máquina (bitcode, LLVM-IR), será capaz de interagir adequadamente com o resto do sistema operacional, e (provavelmente mais importante) não vai quebrar entre as versões do iOS/ Swift.



Funcionalidade 2:

Operator Overloading

A sobrecarga de operador de swift permite um código mais conciso ao lidar com cálculos complexos. É muito mais natural ler expressões numéricas usando operadores do que uma sequência de chamadas de método. Na verdade, os objetos envolvidos nem precisam ser do mesmo tipo, o que facilita muito durante a escrita de códigos muito extensos além de reaproveita operadores que não teriam utilidade na interação entre objetos, como por exemplo:

Se quisermos criar um terceiro objeto a partir de 2 que já possuímos, basta fazermos a soma dos dois objetos que já temos e teremos um terceiro com basicamente uma linha `objeto3 = (objeto1 + objeto2)`, tornado bem mais simples do que ter que iterar manualmente sobre cada variável do objeto ou usar um método, como em python. Em termos de códigos simples, a vantagem é pouca, mas para projetos que exigem muitas variáveis e muitas interações complexas, a legibilidade é extremamente melhorada e também a redigibilidade, tendo em vista que vão estar em uso operadores comuns.

```
class Vector2D:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def imprimir(self):
        print("x = {:.2f} e y = {:.2f}".format(self.x, self.y))
    def soma(self, vet2):
        return Vector2D(self.x + vet2.x, self.y + vet2.y)

vet1 = Vector2D(30.0, 10.0)
vet2 = Vector2D(10.0, 20.0)
vet3 = vet1.soma(vet2)
vet3.imprimir()
```

```
struct Vetor2D {
    var x = 0.0, y = 0.0
}

extension Vetor2D {
    static func + (left: Vetor2D, right: Vetor2D) -> Vetor2D {
        return Vetor2D(x: left.x + right.x, y: left.y + right.y)
        //atribui um processo desejado a um operador
        //especifico para os objetos nesse caso os vetores
    }
}

let vetor = Vetor2D(x: 3.0, y: 1.0)
let outroVetor = Vetor2D(x: 2.0, y: 4.0)
let vetorCombinado = vetor + outroVetor
print(vetorCombinado)
```

Em python:

Em python, temos uma implementação clássica, utilizando um método, onde dependendo do programador, pode ficar confuso e dificultar a legibilidade para manutenções futuras além de consumir mais tempo para implementação

```
class Dimensoes:#os objetos tem sempre a base quadrada e caixa também
    def __init__(self, peso, formato, base, alt):#define as propriedades basicas dos objetos e das caixas
        self.peso = peso
        self.formato = formato
        self.base = base
        self.alt = alt

class caixa (Dimensoes):
    def __init__(self, peso, base, alt):
        super().__init__(peso, "quadrado", base, alt)#estende dimensões init
    def empacotar(self, objeto):#faz a verificação se é possível guardar dentro da caixa o objeto
        if (self.peso > objeto.peso):#verifica se o peso que a caixa suporta é maior que o peso do objeto
            if (objeto.formato == "liquido"):#verifica o estado, se for liquido é retornado verdadeiro
                return True
            elif (objeto.formato == "esferico" and ((self.base*self.alt)>((12.56*((objeto.alt//2)^3)/3)+5))):
                return True#retorna true se estiver dentro da margem de base e altura para que se tenha a folga para a acomodação de amortecedores
            elif (objeto.formato == "quadrado" and self.base >= objeto.base+(objeto.base*0.1) and self.alt >= objeto.alt+(objeto.alt*0.1)):
                return True#retorna true se estiver dentro da margem de base e altura para que se tenha a folga para a acomodação de amortecedores
            else:
                return False#retorna falso se foge de algum dos critérios
class objeto (Dimensoes):
    def __init__(self, peso, formato, base, alt):
        super().__init__(peso, formato, base, alt)#passa as variáveis atribuindo as dimensões definidas

caixa1 = caixa(3.0, 10, 10)
objeto1 = objeto(3.0,"esferico", 2, 2)
print(caixa1.empacotar(objeto1))
```

Em swift:

O código fica bem mais intuitivo e claro para outros darem manutenção, além de reutilizar operadores que seriam inutilizados dentro das interações entre os diferentes objetos que estão sendo tratados

```
//programa que verifica se a objeto pode ser empacotado, um exemplo real do uso:
struct Dimensoes {
    var peso: Float//peso do objeto e peso que a caixa suporta
    var formato: String// pode ser esferico, quadrado ou liquido
    var base: Float// em esferas e liquidos é desprezível
    var alt: Float// em liquidos é desprezível
}
extension caixa{
    //verifica se o peso da caixa suporta o do objeto, e se o objeto se encaixa em algum dos casos
    static func - (left: caixa, right: objeto)-> Bool{//retorna um booleano verdadeiro ou falso, nos casos sempre é verificada uma margem para os suportes e amortecedores exceto nos liquidos
        return (( left._Dimensoes.peso > right._Dimensoes.peso ) && (( right._Dimensoes.formato == "liquido" ) ||//
            (right._Dimensoes.formato=="esferico")&&((left._Dimensoes.base*left._Dimensoes.alt)>(12.56*(right._Dimensoes.alt/2*right._Dimensoes.alt/2)/3)+5) ||
            (right._Dimensoes.formato=="quadrado")&&(left._Dimensoes.base>=right._Dimensoes.base+(right._Dimensoes.base*0.1))&&(left._Dimensoes.alt >= right._Dimensoes.alt+(right._Dimensoes.alt*0.1))))
    }
}
class caixa{
    var _Dimensoes: Dimensoes
    init(peso: Float, base: Float, alt: Float){
        _Dimensoes = Dimensoes(peso: peso, formato: "quadrado", base: base, alt: alt)
    }
}
class objeto{ // cria uma classe objeto respeitando as Dimensoes
    var _Dimensoes: Dimensoes
    init(peso: Float, tipo: String, base: Float, alt: Float){
        _Dimensoes = Dimensoes(peso: peso, formato: tipo, base: base, alt: alt)//inicia as variaveis
    }
}
let caixa1 = caixa(peso: 3.0, base: 10, alt: 10)//cria um objeto da classe caixa
let objeto1 = objeto(peso: 2.0, tipo: "esferico", base: 0, alt: 2)//cria um objeto da classe objeto
let valido = caixa1 - objeto1// usa o operando menos para verificar se as propriedades dos 2 objetos estão dentro dos parametros bem intuitivamente
print(valido)
```


Funcionalidade 3:

Opcionais

- Swift introduz os tipos opcionais, que lidam com a ausência de valores. Isto é, em tempo de execução, uma variável poderá ter um valor atribuído, ou ter **nil**, e isto não afetará na execução do programa

```
Person {  
    var firstname: String, var middlename: String?, var lastname: String  
  
    func fullName() -> () {  
        if (middlename != nil) {  
            print("Your fullname is \(firstname) \(middlename!) \(lastname)")  
        } else {  
            print("Your fullname is \(firstname) \(lastname)")  
        }  
    }  
}  
  
let anotherPerson = Person(firstname: "Abel", middlename: nil, lastname: "Adeyemi")  
  
anotherPerson.fullName() //This will not throw an error
```

Perceba o caracter "?" em "var middlename: String?". Assim é feita a declaração de um tipo opcional.

Exemplo real de Opcionais

```
import UIKit

@UIApplicationMain
class AppDelegate: UIResponder, UIApplicationDelegate {

    var window: UIWindow?
```

- AppDelegate é o objeto raiz de todo aplicativo em Swift
- Variável "window" se refere à uma janela da aplicação que pode ou não estar aberta
- Por causa dessa dúvida, deve ser opcional
- Seu unwrapping é feito mais pra frente no código

Comparação com outras linguagens

Swift

```
1  if let unwrapped = window {  
2      //janela aberta, executa instruções  
3  } else {}  
4      //janela fechada, executa outras instruções  
5  }
```

Python

```
if window = None:  
    #janela está fechada, e segue as instruções  
else:  
    #janela aberta, e segue outras instruções
```

- O unwrapping seguro em Swift é feito de maneira semelhante em Python
- Isso é possível pois Python possui o objeto "None", equivalente a "nil" em Swift, permitindo comparações com variáveis



```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int procura(int array[])
5  {
6      int i = 0;
7      for (i; i < 3; i++)
8      {
9          if (array[i] == 100)
10             return i;
11      }
12  }
13
14
15  int main(void)
16  {
17      int a[3] = {1, 2, 3};
18      int index = procura(a);
19      printf ("%d", index); //imprimiria 3, um índice que está fora do array
20      return 0;
21  }

```

```

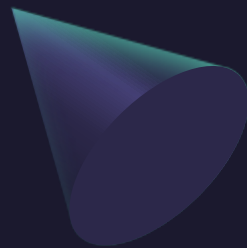
def procura(array):
    for i in range(len(array)):
        if (array[i] == 100):
            return i;

a = [1, 2, 3]
print (procura(a)) #imprime None

```

Análise crítica

- Os tipos opcionais de Swift, quando comparados a linguagens de tipagem dinâmica que possuem um objeto Nulo como Python, Perl e PHP, são desnecessários, pois essas linguagens permitem o comportamento de unwrapping seguro nativamente, assim como possuem funções que podem ou não retornar algum valor
- Funcionam em Swift pois ela é uma linguagem de tipagem estática que possui o objeto Nulo, pois além de possibilitarem o unwrapping seguro, também permitem que as funções tenham essa característica de retorno ou não de valor
- Nas imagens, a mesma função em C (tipagem estática e ausência de objeto Nulo) e Python. Qual retorno faz mais sentido? 3 em C, um índice que sequer está presente no array, ou None em Python?



Funcionalidade 4: Extensões

- Swift introduz as extensões, que permitem a adição de novos métodos à classes, estruturas, enumerações e protocolos já existentes.

```
class Airplane
{
    var altitude: Double = 0

    func setAltitude(feet: Double) {
        altitude = feet
    }
}
```

A classe "Airplane" definida

```
extension Airplane
{
    func setAltitude(meter: Double) {
        altitude = meter * 3.28084
    }
}
```

Aqui adicionamos um método em que calculamos a altitude em metros à classe "Airplane"

```
let boeing = Airplane()
boeing.setAltitude(meter: 12000)
print(boeing.altitude) // Output: 39370.08
```

O método é utilizado



Exemplo real de Extensões

- Classe "Circulo" com somente a propriedade "raio", definida no código de outro programador no mesmo projeto, o qual você não tem acesso
- Para que o cálculo da circunferência seja feito, a classe Circulo é estendida e a propriedade computada "circunferencia" é adicionada em seu código
- Após a extensão, a nova propriedade pode facilmente ser acessada

Código do outro programador

```
class Circulo {  
    var raio: Double = 0  
}
```

Seu código

```
extension Circulo  
{  
    var circunferencia: Double {  
        return raio * .pi * 2  
    }  
}
```

```
let circulo = Circulo()  
circulo.raio = 10  
print(circulo.circunferencia) // imprime: 62.83185307179586
```

Comparação com outras linguagens

Java

- O código da extensão em Java seria algo como:

```
1  import java.lang.Math.*;
2
3  public class Circunferencia extends Circulo
4  {
5      public double circunferencia(double raio)
6      {
7          return super.raio * 3.14 * 2;
8      }
9  }
```

- A main seria:

```
1  public class MyClass
2  {
3      public static void main(String[] args) {
4          Circunferencia circulo = new Circunferencia();
5          circulo.setRaio(10);
6          System.out.println(circulo.circunferencia(circulo.getRaio()));
7      }
8  }
```

Análise crítica

- Foi utilizado um exemplo trivial, mas que consegue ilustrar bem os possíveis problemas que a falta das extensões acarretaria em Java
- Em uma situação de código extenso e muitas classes, onde o acesso ao código da classe original não é possível por algum motivo, seria necessária a criação de várias novas subclasses e objetos para cada método novo a ser adicionado, mesmo que tais novas subclasses não necessitem utilizar nenhum campo das superclasses,
- Isso gera um código com legibilidade e redigibilidade muito prejudicadas, assim como a manutenção de possíveis falhas
- Ainda que o acesso ao código da classe original seja possível, adicionar novos métodos manualmente poderia prejudicar o funcionamento do restante do código, gerando bugs
- Por isso a utilização de extensões: simplificam a adição de novos métodos às classes, sem afetar o funcionamento original do programa, permitindo que os mesmos objetos já existentes trabalhem com os novos métodos