SWIFT: UMA ANÁLISE CRITICA DE SEUS PARADIGMAS E APLICAÇÕES NO MERCADO

JOÃO COMINI CÉSAR DE ANDRADE, MATEUS SOARES GATTI VASCONCELLOS, PEDRO AUGUSTO SILVA FERREIRA

HISTÓRIA

Quem criou

- Desenvolvido dentro da Apple a partir de 2010
- Lançado ao público em 2014 na WWDC

Porque foi criado

- **Objective-C era antigo e inseguro**
- **Muitos erros de ponteiros nulos**
- **Precisavam de uma linguagem:**

-Mais segura

-Mais rápida

-Mais moderna e fácil de usar

MOTIVAÇÕES PRINCIPAIS

O Swift foi criado para ser:

Mais seguro

- Introduziu Optionals para evitar “nil crashes”

Mais rápido

- Usa LLVM → performance próxima de C++

Mais moderno e expressivo

- Sintaxe simples
- Generics, POP

MOTIVAÇÕES PRINCIPAIS

Generics : Exemplo Swap de duas variaveis

```
// Versão para Int
func trocarDoisInts(_ a: inout Int, _ b: inout Int) { /* ... */ }

// Versão para String
func trocarDuasStrings(_ a: inout String, _ b: inout String) { /* ... */ }
```

```
func trocarDoisValores<T>(_ a: inout T, _ b: inout T) {
    let tempA = a
    a = b
    b = tempA
}

// Usando o código com Int
var num1 = 10
var num2 = 20
trocarDoisValores(&num1, &num2) // T é inferido como Int

// Usando o código com String
var nome1 = "Alice"
var nome2 = "Bob"
trocarDoisValores(&nome1, &nome2) // T é inferido como String
```

INFLUÊNCIAS DA LINGUAGEM

O Swift mistura ideias de várias linguagens:

Objective-C

- Interoperabilidade e parte do runtime

C# e C++

- Structs vs. classes
- Propriedades calculadas (get e set)

Python e Ruby

- Sintaxe simples
- Menos verbosidade

APLICAÇÕES NO MERCADO

1. Ecossistema Apple
   - iOS
   - macOS
   - watchOS
   - tvOS
   - SwiftUI (UI declarativa moderna)
2. Empresas globais
   - Twitter
   - Slack
   - Netflix

SEGURANÇA E OPTIONALS (O FIM DO "NULL POINTER EXCEPTION")

Objetivo: Mostrar como o Swift elimina erros de valor nulo (nil) que travam apps em outras linguagens.
Optional:É um tipo especial que "embrulha" um valor, representando duas possibilidades:

1. Existe um valor (.some(Valor))
2. O valor é ausente (.none ou nil)

O compilador força o desenvolvedor a verificar com segurança se um opcional contém um valor antes de usá-lo, um processo chamado unwrapping (desembrulhar).

SEGURANÇA E OPTIONALS (O FIM DO "NULL POINTER EXCEPTION")

Cenário: Um sensor de temperatura que pode falhar na leitura (retornar nulo).

```swift
// 1. Declaração: O sensor pode retornar um Double ou nada (nil)
var leituraSensor: Double? = nil

// 2. Verificação segura com 'if let' (Optional Binding)
// Tenta "desembrulhar" o valor de forma segura
if let temperatura = leituraSensor {
    // Este bloco só executa se o sensor leu algo válido.
    // Aqui dentro, 'temperatura' é um Double normal (seguro).
    print("A temperatura atual é \(temperatura)°C")
} else {
    // Este bloco executa se for nil (o sensor falhou)
    print("Erro: Não foi possível ler o sensor.")
}

// 3. Valor Padrão com 'Nil-Coalescing' (??)
// Se o sensor falhar, assumimos uma temperatura segura de 20.0
let temperaturaFinal = leituraSensor ?? 20.0
```

CONCISÃO E PARADIGMA FUNCIONAL

Objetivo: Demonstrar como o Swift reduz a verbosidade ("boilerplate") para manipulação de dados.

Código Imperativo (O jeito "braçal")

Cenário: Pegar uma lista de preços, filtrar só os caros (> 100) e aplicar desconto.

```
// Lista de preços
List<Double> precos = [50.0, 150.0, 300.0, 80.0];
List<Double> resultado = new List<Double>();

// Loop manual: Você diz COMO fazer passo a passo
for (Double preco : precos) {
    if (preco > 100.0) {
        double comDesconto = preco * 0.9;
        resultado.add(comDesconto);
    }
}
// Resultado: [135.0, 270.0]
```

Código Swift (Funcional e Declarativo)

Solução: Você diz O QUE você quer, encadeando operações.

```
let precos = [50.0, 150.0, 300.0, 80.0]

// filter: pega os maiores que 100
// map: aplica o desconto em cada um
let resultado = precos
    .filter { $0 > 100 }
    .map { $0 * 0.9 }

// Resultado: [135.0, 270.0]
```

PROGRAMAÇÃO ORIENTADA A PROTOCOLOS (POP) VS HERANÇA

Objetivo: Mostrar como Swift foge da herança rígida (onde tudo precisa ser "filho" de alguém) usando composição.

O Problema da Herança (OO Clássico)

Cenário: Um jogo com Robôs e Zumbis. Ambos atacam, mas só Robô precisa de bateria. Se você herdar de Inimigo, fica difícil separar.

```
class Inimigo {
    void atacar() { print("Atacando!"); }
    // Problema: Zumbi não tem bateria, mas herdaria isso se estiver na classe pai errada
    void recarregar() { ... }
}
```

A Solução Swift (POP)

Solução: Criamos "habilidades" (Protocolos) e colamos elas nos tipos, como peças de Lego. Usamos Extensão para dar o código padrão.

```
// Definimos uma "habilidade"
protocol Atacante {
    func atacar()
}

// Damos uma implementação padrão (ninguém precisa reescrever isso!)
extension Atacante {
    func atacar() { print("Causou 10 de dano!") }
}

// Criamos peças soltas que ganham poderes
struct Zumbi: Atacante { } // Ganha atacar() de graça
struct Robo: Atacante { }  // Ganha atacar() de graça

let meuRobo = Robo()
meuRobo.atacar() // "Causou 10 de dano!"
```

PONTOS FORTES

Performance (LLVM, C/C++)
– Código Swift otimizado roda quase tão rápido quanto C/C++.
async/await + Actors

– Modelo moderno que evita race conditions automaticamente.
Segurança: Optionals

– O uso de Optionals impede acessos nulos acidentais.
Menos crashes

– Apps Swift têm menos falhas por causa do sistema de tipos seguro.

PONTOS FRACOS

ABI instável (pré-Swift 5)
– Versões antigas quebravam código e dificultavam manter projetos grandes.

Compilação lenta
– Projetos grandes sofrem com inferência de tipos pesada e builds demorados.

Ecossistema fora da Apple fraco
– No backend e multiplataforma, Swift ainda perde para Node, Python e Go.

Curva de aprendizado alta
– Conceitos como Optionals, generics e value/reference podem confundir iniciantes.

LINGUAGENS QUE O SWIFT INFLUENCIOU

Kotlin: null-safety, extensions

– Kotlin adotou diretamente ideias do Swift como tipos anuláveis e extensions.

Rust: segurança, Option

– A filosofia de segurança de memória de Rust converge com a do Swift.

TypeScript/JS: optional chaining

– O sucesso do optional chaining no Swift influenciou sua adoção na Web.

CONCLUSÃO

Swift = seguro + rápido
– Combina performance alta com forte segurança de tipos.

Forte no ecossistema Apple
– É essencial para apps modernos de iOS, macOS e SwiftUI.

Adoção limitada fora dele
– No backend e multiplataforma, ainda não ganhou grande tração.

Grande impacto em linguagens modernas
– Inspirou recursos adotados em Kotlin, Rust, JavaScript e outras.

OBRIGADO!