







Introdução a Orientação a Objetos 

- Problemas da Programação Procedural
 - Descentralização (repetição) de código
 - Dificuldade de manutenção
 - Dificuldade em substituir desenvolvedores
 - Muitas pessoas responsáveis por mesmos códigos em diferentes partes do projeto
 - Pouco reaproveitamento de código



Introdução a Orientação a Objetos 

- Orientação a Objetos
 - Resolve os problemas apresentados da Programação Procedural
 - Concentra responsabilidades nos locais certos
 - Flexibiliza a aplicação
 - Encapsula a lógica de negócio
 - Melhora a comunicação entre desenvolvedores
 - Aumenta a reutilização de código
 - Aumenta o ciclo de vida dos projetos
 - Menor custo de desenvolvimento e criação



Orientação a Objetos



- Siglas
 - OO: Orientação a Objetos
 - LOO: Linguagem Orientada a Objetos
 - POO: Programação Orientada a Objetos
- Características
 - Códigos pertencem a classes
 - Classes possuem propriedades próprias
 - Classes possuem funcionalidades próprias
 - Classes interagem com outras classes

Como funciona (visão geral)



Modelo Procedural

nome
sobrenome
velocidade
endereco
anos

INSERT INTO USUARIOS...

INSERT INTO CARROS...

Cad. 1

INSERT..

Cad. 2

INSERT..

Cad. 3

INSERT..

Modelo OO

[usuario nome]
[usuario sobrenome]
[carro velocidade]
[carro endereco]
[usuario anos]

[usuario salvar]

[carro salvar]

Cad. 1

Cad. 2

Cad. 3

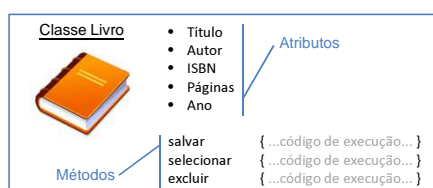
Classe Usuario

INSERT..

Classes



- Estrutura de dados
- Representa um tipo de dado
- Comportamento próprio



Atributos e Métodos



- **Atributos**
 - Características da classe
 - Tipos primitivos e/ou outras classes
 - Geralmente representados por **substantivos**
 - Exemplos: *nome, idade, endereço, cor, tamanho*
- **Métodos**
 - Ações que a classe pode realizar
 - Podem receber parâmetros e retornar valores
 - Geralmente representados por **verbos**
 - Exemplos: *salvar, ler, abrir, fechar, emprestar, devolver*

Criando classes



- Operador **class**

```
class NomeDaClasse
{
    /* Atributos */
    /* Métodos */
}
```

```
class Usuario
{
    /* Atributos */
    var nome : String
    var idade : Int

    /* Métodos */
    func apresentar() -> String
    {
        return "Meu nome é \{(nome)"
    }
}
```

Objetos



- Classe não é um objeto
- Classe é um modelo de objeto (template)
- Objetos são instâncias de classes



Construtores



- Método **init** customizado
- Customiza o código de inicialização da classe

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

    init(x: String, y: Int)
    {
        nome = x
        idade = y
    }

    func apresentar() -> String
    {
        return "Meu nome é \"(nome)\""
    }
}
```

```
// Utilizando o construtor padrão
// Só funciona se não houver construtor customizado
var u : Usuario = Usuario()
u.nome = "André"
u.idade = 30

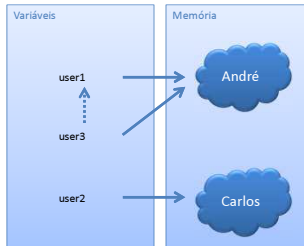
// Utilizando o construtor customizado
var u : Usuario = Usuario(x: "André", y: 30)
```

Criando objetos



- Operador **()**
- Cria uma instância a partir de uma classe

```
var user1 : Usuario = Usuario(x: "André", y: 30)
var user2 : Usuario = Usuario(x: "Carlos", y: 29)
var user3 : Usuario = user1
```



Operador **self**



- Diferencia um atributo do objeto de um atributo do método
- Fornece a referência do próprio objeto para outro método
- Funciona somente dentro do próprio objeto

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

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

Modificadores de acesso



- Definem o acesso aos atributos e métodos
 - private**: acesso apenas para a própria classe
 - public**: acesso público

```
class Usuario {
  public var pub : String
  private var pri : String
}
```

```
var user : Usuario = Usuario()

user.pub // Correto
user.pri // Erro
```

Forma de utilização



- Geralmente utilizados da seguinte forma:
 - Construtores só podem ser **public**
 - Atributos são declarados como **private**
 - Métodos são declarados como **public**, *exceto métodos de uso exclusivo interno da classe*
 - Acesso realizado via **setters**, **getters** ou outros

Getters e Setters



- Permitem interagir com os atributos
- Segurança: validações no lugar certo

```
class Usuario {
  private var nome : String

  init(nome: String) {
    self.nome = nome
  }

  func setNome(nome: String) -> Void {
    // Validações, se necessário
    self.nome = nome
  }

  func getNome() -> String {
    // Validações e/ou formatações, se necessário
    return self.nome
  }
}
```

```
var user : Usuário = Usuario("André")

user.nome = "André" // Errado
user.setNome("André") // Certo
user.getNome()
```

Getters e Setters booleanos



- **set** para definir valor
- **is** para consultar, ao invés de **get**

```
class Usuario {
    private var graduado : Bool

    func setGraduado(graduado: Bool) -> Void {
        // Validações, se necessário
        self.graduado = graduado
    }

    func isGraduado() -> Bool {
        // Validações e/ou formatações, se necessário
        return self.graduado
    }
}
```

Tratamento de retorno nulo



- Métodos que possam vir a retornar valores nulos podem precisar de tratamento específico
- Operador **!**
- Exemplo
 - `func meuMetodo() -> String`
 - `func meuMetodo() -> String!`

Aulas práticas e manuais on-line



Assista agora às aulas práticas.

[Clique aqui](#) para visualizar as aulas práticas disponíveis
