Marisangila Alves, MSc
marisangila.alves@catolicasc.org.br
marisangila.com.br

Católica de Santa Catarina

2025/2

Programação Orientada a Objetos

POO

Sumário

História

História da Programação Orientada a Objetos I

A **Programação Orientada a Objetos (POO)** surgiu como resposta à necessidade de criar sistemas mais **modulares**, **reutilizáveis** e **fáceis de manter**.

- **1960s**: Linguagem **Simula** introduz o conceito de **classes** e **objetos** (para simulações de sistemas complexos).
- **1970s**: Surge **Smalltalk**, que populariza o paradigma totalmente orientado a objetos.
- **1980s**: Linguagens como **C++** unem programação estruturada e POO.

- **1990s**: **Java** consolida a POO no desenvolvimento empresarial.
- **2000s em diante**: POO se torna padrão em linguagens como **C#**, **Python**, **csharp**, entre outras.

Nota:

A POO não substituiu outros paradigmas, mas se tornou o **mais utilizado** em aplicações de grande porte.

Definição

O que é Programação Orientada a Objetos? I

A **Programação Orientada a Objetos (POO)** é um paradigma de programação que organiza o software em **objetos**, que combinam **dados (atributos)** e **comportamentos (métodos)**.

- Facilita a **manutenção** e o **reuso** do código.
- Reflete conceitos do mundo real no software.
- Usa princípios como **abstração**, **encapsulamento**, **herança** e **polimorfismo**.

Programação Estruturada e **POO** têm abordagens diferentes para resolver problemas:

- **Estruturada**:
  - O foco está em **funções e procedimentos**.
  - Os dados são **separados** das funções.
  - Mais adequada para programas pequenos e simples.
- **Orientada a Objetos**:
  - O foco está em **objetos**, que reúnem dados e comportamentos.
  - Os dados são **encapsulados** nas classes.
  - Mais adequada para sistemas grandes e complexos.

Outros paradguimas:

- **Estruturado**: usa controle de fluxo estruturado (ex: Pascal, C).
- **Orientado a Objetos (POO)**: organiza código em classes e objetos, com encapsulamento, herança e polimorfismo (ex: Java, C++, PHP, C#, Python, Swift[1], Kotlin[2]).
- **Funcional**: enfatiza funções puras e imutabilidade, evitando efeitos colaterais (ex: Haskell, Elixir, JavaScript).

3 4

POO vs Programação Estruturada III

Atenção!

A POO não substitui a programação estruturada: ela a **complementa**, fornecendo novas ferramentas para organizar o código.

[1] Swift é muito usado em desenvolvimento mobile iOS

[2] Kotlin é usado para Android

[3] Programação orientada a eventos é muito utilizada em interfaces gráficas, onde a execução do código depende de ações do usuário como cliques e digitação.

[4] Muitas linguagens modernas são **multiparadigma**, ou seja, suportam mais de um paradigma, como Python (estruturado, OO, funcional) e JavaScript (funcional, OO, orientada a eventos).

- **Classe**: o molde que define atributos e métodos.
- **Objeto**: instância concreta de uma classe.
- **Encapsulamento**: proteção dos dados internos, controlando o acesso com modificadores (`public`, `private`, `protected`).
- **Herança**: uma classe pode **herdar** atributos e métodos de outra.
- **Polimorfismo**: diferentes classes podem redefinir métodos de formas distintas.
- **Abstração**: capacidade de representar conceitos do mundo real em modelos simplificados.

Classe

Classe:

É um **modelo ou molde** que define a estrutura e o comportamento de objetos em programação orientada a objetos.

Propriedades

As **propriedades** — também chamadas de **atributos** — representam as **características ou dados** que descrevem um objeto.

- São variáveis declaradas dentro da classe.
- Cada objeto possui sua **própria cópia** das propriedades.
- Exemplos: `nome`, `idade`, `saldo`, `cor`.

Exemplo:

Na classe **Pessoa**, atributos poderiam ser: `nome`, `idade`, `cpf`.

```
1 class Pessoa
2 {
3     // Atributos (propriedades)
4     public string nome;
5     public int idade;
6     public string cpf;
7 }
```

Código 1: Classe com atributos em C#

Métodos

Os **métodos** são **funções** definidas dentro de uma classe que descrevem o **comportamento** do objeto.

- Podem manipular ou acessar atributos.
- Representam ações que o objeto pode realizar.
- Exemplos: `apresentar()`, `depositar()`, `calcularIdade()`.

Exemplo:

Na classe **Pessoa**, um método poderia ser `apresentar()`, que imprime o nome e a idade.

```csharp
using System;

class Pessoa
{
    public string nome;
    public int idade;

    // Método (comportamento)
    public void apresentar()
    {
        Console.WriteLine($"Olá, meu nome é {this.nome} e tenho {this.idade} anos.");
    }
}
```

Código 2: Classe com método em C#

Objeto

É comum confundir **classe** com **objeto**. Mas eles são conceitos distintos:

- **Classe**: é o **molde**, o projeto ou a receita.
- **Objeto**: é a **instância** concreta criada a partir da classe.

Nota:

Uma classe define **o que um objeto pode ter e fazer**. Um objeto é um **exemplo real** daquela definição.

Exemplo:

A classe **Carro** pode ter atributos como `cor` e `modelo`, e métodos como `acelerar()` e `frear()`. O objeto seria um carro específico, por exemplo: um Fiat Uno vermelho de 2010.

```
using System;

// Classe (molde)
class Carro
{
    public string modelo;
    public string cor;

    public void acelerar()
    {
        Console.WriteLine("O carro está acelerando!");
    }
}

// Objeto (instância da classe)
class Program
{
    static void Main()
    {
        Carro meuCarro = new Carro();
        meuCarro.modelo = "Fiat Uno";
```

```
22          meuCarro.cor = "Vermelho";
23
24          Console.WriteLine($"Modelo: {meuCarro.modelo}, Cor: {meuCarro.cor}");
25          meuCarro.acelerar();
26      }
27  }
```

Código 3: Diferença entre classe e objeto

Construtor

Construtores em Programação Orientada a Objetos I

Construtor é um método especial de uma classe que é chamado automaticamente sempre que um objeto é criado. Ele é usado para **inicializar propriedades** do objeto e garantir que ele comece em um estado válido.

- No C#, o construtor é definido com um método que possui o **mesmo nome da classe**.
- Pode receber parâmetros para inicializar atributos do objeto.
- Pode conter regras de validação ou configuração inicial do objeto.

Nota:

Um construtor evita que o usuário da classe tenha que chamar métodos de inicialização manualmente, garantindo consistência e segurança no estado do objeto.

Construtores em Programação Orientada a Objetos II

O exemplo a seguir mostra uma classe **Produto** com construtor para inicializar nome e preço:

```
using System;

class Produto
{
    public string nome;
    public double preco;

    // Construtor
    public Produto(string nome, double preco)
    {
        this.nome = nome;
        this.preco = preco;
        Console.WriteLine($"Produto '{this.nome}' criado com preço R$ {this.preco}.");
    }
}

// Criando objetos
class Programa
{
```

```
20      static void Main()
21      {
22          Produto produto1 = new Produto("Caneta", 1.5);
23          Produto produto2 = new Produto("Caderno", 12.0);
24      }
25  }
```

Código 4: Exemplo de construtor em C#

Encapsulamento

Encapsulamento significa proteger os dados de um objeto, restringindo o acesso direto aos atributos e fornecendo métodos de acesso (**getters** e **setters**).

- Usa modificadores: `public`, `private`, `protected`.
- Evita alterações indevidas nos atributos.

```
using System;

class ContaBancaria
{
    private double saldo;

    public ContaBancaria(double saldoInicial)
    {
        this.saldo = saldoInicial;
    }

    // Getter
    public double getSaldo()
    {
```

```
15          return this.saldo;
16      }
17
18      // Setter controlado
19      public void depositar(double valor)
20      {
21          if (valor > 0)
22          {
23              this.saldo += valor;
24          }
25      }
26  }
27
28  class Program
29  {
```

```
30      static void Main()
31      {
32          ContaBancaria conta = new ContaBancaria(100);
33          conta.depositar(50);
34          Console.WriteLine("Saldo atual: " + conta.getSaldo);
35      }
36  }
```

Código 5: Encapsulamento em C#

Herança

Herança permite que uma classe (filha) reutilize atributos e métodos de outra (pai).

- Promove reuso de código.
- Facilita extensibilidade.

```
using System;

class Pessoa
{
    public string nome;

    public Pessoa(string nome)
    {
        this.nome = nome;
    }

    public virtual void apresentar()
    {
        Console.WriteLine($"Olá, eu sou {this.nome}");
    }
}

class Aluno : Pessoa
{
    public string curso;

```

```
22      public Aluno(string nome, string curso) : base(nome)
23      {
24          this.curso = curso;
25      }
26
27      public override void apresentar()
28      {
29          Console.WriteLine($"Sou {this.nome}, aluno de {this.curso}");
30      }
31  }
32
33  class Program
34  {
35      static void Main()
36      {
37          Aluno aluno = new Aluno("João", "Computação");
38          aluno.apresentar();
39      }
40  }
```

Código 6: Herança em C#

Polimorfismo

Polimorfismo significa "muitas formas". Permite que diferentes classes implementem o mesmo método de maneiras distintas.

- Métodos com o mesmo nome, mas comportamentos diferentes.
- Facilita o uso de código genérico.

```
1  using System;
2  using System.Collections.Generic;
3
4  class Animal
5  {
6      public virtual void falar()
7      {
8          Console.WriteLine("O animal faz um som.");
9      }
10 }
11
12 class Cachorro : Animal
13 {
14     public override void falar()
```

```
15      {
16          Console.WriteLine("O cachorro late: Au Au!");
17      }
18  }
19
20  class Gato : Animal
21  {
22      public override void falar()
23      {
24          Console.WriteLine("O gato mia: Miau!");
25      }
26  }
27
28  class Program
29  {
```

```
30      static void Main()
31      {
32          List<Animal> animais = new List<Animal>()
33          {
34              new Cachorro(),
35              new Gato()
36          };
37
38          foreach (var a in animais)
39          {
40              a.falar(); // comportamento diferente com o mesmo
                ↪ método
41          }
42      }
43  }
```

Código 7: Polimorfismo em C#

Abstração

Abstração é a capacidade de modelar conceitos do mundo real em classes. Em C#, é implementada com **classes abstratas** ou **interfaces**.

- Define apenas o que deve ser feito, não como.
- Obriga classes filhas a implementarem os métodos.

```
using System;
using System.Collections.Generic;

abstract class Forma
{
    public abstract double CalcularArea();
}

class Quadrado : Forma
{
    private double lado;

    public Quadrado(double lado)
    {
        this.lado = lado;
    }

    public override double CalcularArea()
    {
        return this.lado * this.lado;
    }
```

```
22  }
23
24  class Circulo : Forma
25  {
26      private double raio;
27
28      public Circulo(double raio)
29      {
30          this.raio = raio;
31      }
32
33      public override double CalcularArea()
34      {
35          return Math.PI * (this.raio * this.raio);
36      }
37  }
38
39  class Program
40  {
41      static void Main()
42      {
43          List<Forma> formas = new List<Forma>() {
```

```
44              new Quadrado(4),
45              new Circulo(3)
46          };
47
48          foreach (var f in formas)
49          {
50              Console.WriteLine("Área: " + f.CalcularArea());
51          }
52      }
53  }
```

Código 8: Abstração em C#

Tópicos Avançados

Padrões de arquitetura são soluções comprovadas para problemas recorrentes no design de software. Eles ajudam a organizar o sistema de forma **modular**, **manutenível** e **escalável**.

- Fornecem estruturas e diretrizes para a construção do software.
- Permitem comunicação mais clara entre desenvolvedores.
- Exemplos de padrões de arquitetura:

- **MVC (Model-View-Controller)**: separa dados, lógica de negócios e interface.
- **MVP (Model-View-Presenter)**: similar ao MVC, mas o Presenter manipula a lógica da view.
- **MVVM (Model-View-ViewModel)**: usado em aplicações com bindings, popular em .NET/WPF.
- **Arquitetura em Camadas**: divide o sistema em camadas como View, BLL (Business Logic Layer) e DAL (Data Access Layer).

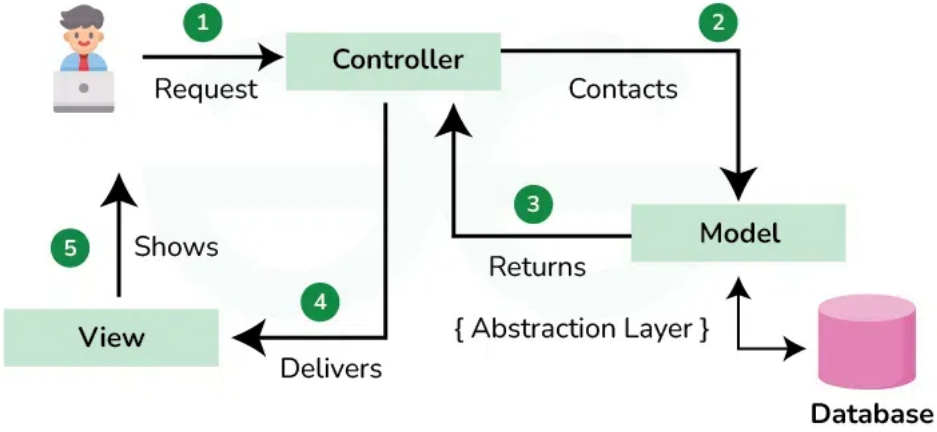
- **Hexagonal**: promove independência de *frameworks* e infraestrutura.
- **Microservices**: cada serviço é independente, focado em um domínio específico.
- **Event-Driven**: comunicação baseada em eventos, desacoplamento entre componentes.
- **Domain-Driven Design (DDD)**: foco no modelo de domínio e regras de negócio, organiza o sistema em bounded contexts e facilita manutenção de sistemas complexos.

Atenção!

Padrões de arquitetura são diferentes de **design patterns**; eles estruturam todo o sistema, enquanto design patterns resolvem problemas de componentes específicos.

O **MVC** é um padrão de arquitetura que organiza a aplicação em três componentes principais:

MVC: Model-View-Controller II



- **Model (Modelo)**: representa os dados e a lógica de negócio.
- **View (Visão)**: responsável pela interface e apresentação dos dados.
- **Controller (Controlador)**: recebe as entradas do usuário, interage com o modelo e atualiza a visão.

Vantagens do MVC

- Separação de responsabilidades.
- Facilita manutenção e testes.
- Permite múltiplas visões para os mesmos dados.

```
using System.Collections.Generic;
using System.Data.SqlClient;

public class ProdutoModel
{
    private readonly ConnectionFactory factory;

    public ProdutoModel(ConnectionFactory factory)
    {
        this.factory = factory;
    }

    public List<Produto> ListarProdutos()
    {
        List<Produto> lista = new List<Produto>();

        using (SqlConnection con = factory.GetConnection())
        {
            con.Open();

            string sql = "SELECT nome, preco FROM produtos";
            using (SqlCommand cmd = new SqlCommand(sql, con))
```

```
23              using (SqlDataReader reader = cmd.ExecuteReader())
24              {
25                  while (reader.Read())
26                  {
27                      string nome = reader.GetString(0);
28                      double preco = reader.GetDouble(1);
29
30                      lista.Add(new Produto(nome, preco));
31                  }
32              }
33          }
34
35          return lista;
36      }
37  }
```

Código 9: Exemplo de Model.

```
using System;
using System.Collections.Generic;

public class ProdutoView
{
    public void ExibirProdutos(List<Produto> produtos)
    {
        Console.WriteLine("=== LISTA DE PRODUTOS ===\n");

        foreach (var p in produtos)
        {
            Console.WriteLine($"Nome: {p.Nome} | Preço: R$ {p.Preco}");
        }
    }
}
```

Código 10: Exemplo de Constroller.

```cs
using System;
using System.Collections.Generic;

public class ProdutoView
{
    public void ExibirProdutos(List<Produto> produtos)
    {
        Console.WriteLine("=== LISTA DE PRODUTOS ===\n");

        foreach (var p in produtos)
        {
            Console.WriteLine($"Nome: {p.Nome} | Preço: R$ {p.Preco}");
        }

        Console.WriteLine("\n==========================");
    }
}
```

Código 11: Exemplo de View para WebSites.

Além dos conceitos básicos, existem vários tópicos avançados que são importantes para estudo futuro em **POO**.

- **Interfaces e Classes Abstratas**: definição de contratos e implementação parcial de classes.
- **Namespaces**: organização de classes.
- **Design Patterns**: padrões de projeto como Singleton, Factory, Observer, Strategy, Decorator.
- **Polimorfismo avançado**: sobrecarga, sobrescrita, polimorfismo paramétrico (generics).

- **Encapsulamento avançado**: uso correto de atributos e métodos privados, protegidos e públicos, getters/setters com validação.
- **Princípios SOLID**: SRP, OCP, LSP, ISP e DIP para código limpo e modular.
- **Exceptions e Tratamento de Erros**: lançar e capturar exceções, exceções customizadas para regras de negócio.
- **Testabilidade**: Testes unitários.
- **Arquitetura avançada**: MVC, MVVM, MVP, Hexagonal, Domain-Driven Design (DDD), camadas, serviços e repositórios.

(Deitel *et al.*, 2003)

DEITEL, H. M. *et al.* **C# : como programar**. São Paulo: Pearson Education, 2003. ISBN 8534614598.

GAMMA, Erich *et al.* **Padrões de Projetos: Soluções Reutilizáveis de Software Orientados a Objetos**. [*S. l.*]: Bookman, 2000. ISBN 9788573076103.

MARTIN, Robert C. **Arquitetura Limpa: O Guia do Artesão para Estrutura e Design de Software**. [*S. l.*]: Alta Books, 2019. p. 432. ISBN 978-85-508-0460-6.

MICROSOFT. **C# Documentation**. Accessed: 2025-12-02. Microsoft Learn. 2025. Disponível em: `https://learn.microsoft.com/en-us/dotnet/csharp/`.

Estes slides estão protegidos por uma licença Creative Commons

Este modelo foi adaptado de Maxime Chupin.

Marisangila Alves, MSc
marisangila.alves@catolicasc.org.br
marisangila.com.br

Católica de Santa Catarina

2025/2

Programação Orientada a Objetos

POO