Marisangila Alves, MSc
marisangila.alves@edu.sc.senai.br
marisangila.com.br

2025/2

Desenvolvimento de Sistemas

Programação Orientada a Objetos

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**, **PHP**, 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`.

```php
<?php
class Pessoa {
    // Atributos (propriedades)
    public $nome;
    public $idade;
    public $cpf;
}
?>
```

Código 1: Classe com atributos em PHP

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.

```php
<?php
class Pessoa {
    public $nome;
    public $idade;

    // Método (comportamento)
    public function apresentar() {
        echo "Olá, meu nome é {$this->nome} e tenho {$this->idade} anos.\n";
    }
}
?>
```

Código 2: Classe com método em PHP

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.

```php
<?php
// Classe (molde)
class Carro {
    public $modelo;
    public $cor;

    public function acelerar() {
        echo "O carro está acelerando!\n";
    }
}

// Objeto (instância da classe)
$meuCarro = new Carro();
$meuCarro->modelo = "Fiat Uno";
$meuCarro->cor = "Vermelho";

echo "Modelo: {$meuCarro->modelo}, Cor: {$meuCarro->cor}\n";
$meuCarro->acelerar();
?>
```

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 PHP, o construtor é definido com o método **__construct()**.
- 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:

```php
<?php
class Produto {
    public $nome;
    public $preco;
    // Construtor
    public function __construct($nome, $preco) {
        $this->nome = $nome;
        $this->preco = $preco;
        echo "Produto '{$this->nome}' criado com preço R$ {$this->preco}.\n";
    }
}
// Criando objetos
$produto1 = new Produto("Caneta", 1.5);
$produto2 = new Produto("Caderno", 12.0);
?>
```

Código 4: Exemplo de construtor em PHP

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.

```php
<?php
class ContaBancaria {
    private $saldo;

    public function __construct($saldoInicial) {
        $this->saldo = $saldoInicial;
    }

    // Getter
    public function getSaldo() {
        return $this->saldo;
    }

    // Setter controlado
```

```php
    public function depositar($valor) {
        if ($valor > 0) {
            $this->saldo += $valor;
        }
    }
}

$conta = new ContaBancaria(100);
$conta->depositar(50);
echo "Saldo atual: " . $conta->getSaldo(); // 150
?>
```

Código 5: Encapsulamento em PHP

Herança

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

- Promove reuso de código.
- Facilita extensibilidade.

```php
<?php
class Pessoa {
    public $nome;

    public function __construct($nome) {
        $this->nome = $nome;
    }

    public function apresentar() {
        echo "Olá, eu sou {$this->nome}\n";
    }
}

class Aluno extends Pessoa {
    public $curso;

    public function __construct($nome, $curso) {
        parent::__construct($nome);
        $this->curso = $curso;
    }

```

```php
    public function apresentar() {
        echo "Sou {$this->nome}, aluno de {$this->curso}\n";
    }
}

$aluno = new Aluno("João", "Computação");
$aluno->apresentar();
?>
```

Código 6: Herança em PHP

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.

```php
<?php
class Animal {
    public function falar() {
        echo "O animal faz um som.\n";
    }
}

class Cachorro extends Animal {
    public function falar() {
        echo "O cachorro late: Au Au!\n";
    }
}

class Gato extends Animal {
```

```php
    public function falar() {
        echo "O gato mia: Miau!\n";
    }
}

$animais = [new Cachorro(), new Gato()];
foreach ($animais as $a) {
    $a->falar(); // comportamento diferente com o mesmo método
}
?>
```

Código 7: Polimorfismo em PHP

Abstração

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

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

```php
<?php
abstract class Forma {
    abstract public function calcularArea();
}

class Quadrado extends Forma {
    private $lado;

    public function __construct($lado) {
        $this->lado = $lado;
    }

    public function calcularArea() {
        return $this->lado * $this->lado;
    }
}

class Circulo extends Forma {
    private $raio;

    public function __construct($raio) {
```

```php
        $this->raio = $raio;
    }

    public function calcularArea() {
        return pi() * ($this->raio * $this->raio);
    }
}

$formas = [new Quadrado(4), new Circulo(3)];
foreach ($formas as $f) {
    echo "Área: " . $f->calcularArea() . "\n";
}
?>
```

Código 8: Abstração em PHP

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 **escálavel**.

- 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).

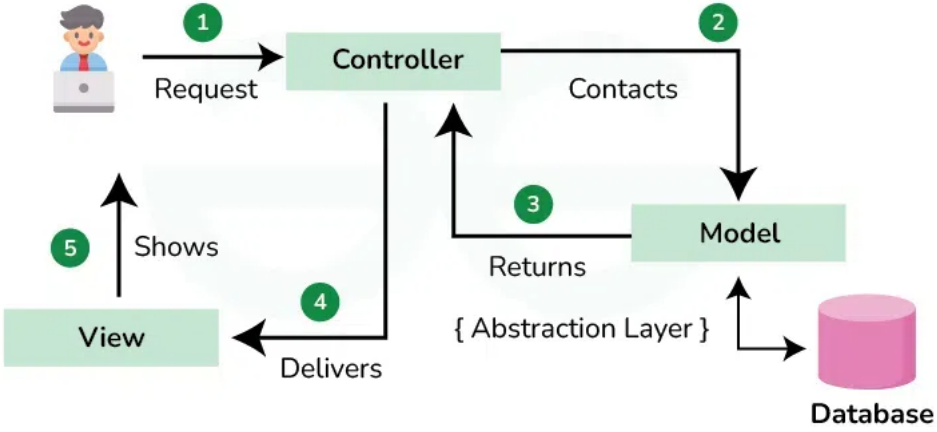
- **Hexagonal**: promove independência de *frameworks* e infraestrutura.
- **Microservices**: cada serviço é independente, focado em um domínio específico.
- **Event-Driven**: comunicação baseada em eventos, desacoplamento entre componentes.
- **Domain-Driven Design (DDD)**: foco no modelo de domínio 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.

```php
<?php
include 'conexao.php';
class Produto {
    public $nome;
    public $preco;
    public function __construct($nome, $preco) {
        $this->nome = $nome;
        $this->preco = $preco;
    }
    public function inserirNoBanco() {
        global $conexao;
        $sql = "INSERT INTO produtos (nome, preco) VALUES (?, ?)";
        $consulta = $conexao->prepare($sql);
        $consulta->execute([$this->nome, $this->preco]);
    }
}
?>
```

Código 9: Exemplo de Model.

```php
<?php
include 'Produto.php';

class ProdutoController {
    private $precoMinimo = 1.0;

    public function adicionarProduto($nome, $preco) {
        if ($preco < $this->precoMinimo) {
            echo "Erro: preço mínimo permitido é R$ {$this->precoMinimo}.\n";
            return;
        }

        $produto = new Produto($nome, $preco);
        $produto->inserirNoBanco();
    }
}
?>
```

Código 10: Exemplo de Controller.

```php
<?php
include 'ProdutoController.php';

$controller = new ProdutoController();

// Verifica se o formulário foi enviado
if ($_SERVER['REQUEST_METHOD'] === 'POST') {
    $nome = $_POST['nome'] ?? '';
    $preco = (float) ($_POST['preco'] ?? 0);

    // Chama o controller para adicionar o produto
    $controller->adicionarProduto($nome, $preco);

    echo "<p>Produto <strong>{$nome}</strong> adicionado com sucesso!</p>";
}
?>

<!DOCTYPE html>
<html lang="pt-BR">
<head>
    <meta charset="UTF-8">
```

```
    <title>Adicionar Produto</title>
</head>
<body>
    <h1>Adicionar Produto</h1>
    <form method="POST" action="">
        <label for="nome">Nome do Produto:</label><br>
        <input type="text" id="nome" name="nome" required><br><br>

        <label for="preco">Preço do Produto:</label><br>
        <input type="number" step="0.01" id="preco" name="preco" required><br><br>

        <input type="submit" value="Adicionar">
    </form>
</body>
</html>
```

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 e Autoloading**: 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.

DALL'OGLIO, Pablo. **PHP: Programando com Orientação a Objetos**. 2. ed. São Paulo: Novatec, 2009.

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

GROUP, The PHP. **PHP Manual**. [*S. l.: s. n.*], 2025. Acesso em: 25 set. 2025. Disponível em: `https://www.php.net/manual/pt_BR/`.

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.

Estes slides estão protegidos por uma licença Creative Commons

Este modelo foi adaptado de Maxime Chupin.

Marisangila Alves, MSc
marisangila.alves@edu.sc.senai.br
marisangila.com.br

2025/2

Desenvolvimento de Sistemas

Programação Orientada a Objetos