

# Paradigma Orientado a Objetos

Grupo 1

29 de maio de 2024

## Resumo

Esse material tem como ênfase focar na explicação conceitual do Paradigma Orientado a Objetos (POO) de forma didática, com auxílio de exercícios e exemplos de código, ambos em Java.

## 1 Introdução

Para tratar de forma didática a explicação do Paradigma em questão, esse artigo se divide nos seguintes tópicos:

2. O Paradigma Orientado a Objetos
3. Linha do tempo da Programação Orientada a Objetos
4. Vantagens e desvantagens
5. Comparação com outros paradigmas
6. Exemplos de código
7. Exercícios

## 2 O Paradigma Orientado a Objetos

Segundo Peter Van Roy [Van Roy, 2009][tradução nossa]: "paradigma da programação é uma aproximação da programação de computadores baseada em uma teoria matemática ou em um conjunto coerente de princípios. Cada paradigma suporta um conjunto de conceitos que o faz melhor para resolver certos problemas. Por exemplo, o paradigma orientado a objetos é o melhor para problemas com um largo numero de dados abstratos organizados em uma hierarquia". Em resumo, a programação orientada a objetos organiza o código em torno de objetos, que são entidades autônomas que contêm dados (atributos) e comportamentos (métodos). O paradigma orientado a objetos (POO) pode ser decomposto em 5 conceitos principais que formam os 4 pilares.

### 2.1 Principais conceitos

- <sup>1º</sup> **Objetos** são a base do paradigma orientado a objetos [Gabbrielli and Martini, 2023]. Como citado anteriormente, um objeto representa um agrupamento de dados (atributos de um objeto), também chamado de estado do objeto, e um conjunto de métodos, também chamado de comportamento do objeto, responsáveis por realizar a interação do objeto com o exterior (restante do código). Um objeto é uma instância de uma classe.
- <sup>2º</sup> **Classes** Uma classe define um modelo para um conjunto de objetos. Ela estabelece quais serão seus atributos (dados que o objeto irá armazenar), visibilidade deles e seus métodos (funções e operações). Todo objeto deve pertencer a pelo menos uma classe [Gabbrielli and Martini, 2023].
- <sup>3º</sup> **Herança** é uma relação entre classes que permite que a definição e a implementação de uma classe, seja baseada em outra classe já existente [Korson and McGregor, 1990][tradução nossa]. Dos conceitos, somente a herança é exclusiva do paradigma orientado a objeto.

4º **Polimorfismo**, em termos gerais, é a habilidade de algo possuir múltiplas formas. Em termos de linguagens de programação, polimorfismo é a habilidade de uma mesma estrutura possuir formas ou comportamentos diferentes quando se encontra em situações diferentes. Esse é um conceito importante tanto para entender o paradigma, quanto para aprender sobre conceitos de linguagens de programação no geral.

5º **Encapsulamento** na programação orientada a objetos, pode ser definida como "uma técnica para minimizar interdependência de módulos escritos separadamente, definindo uma interface externa estrita" [Snyder, 1986][tradução nossa]. Vale ressaltar que encapsulamento, dependendo da definição utilizada, não é um conceito exclusivo do paradigma orientado a objetos. Para esclarecer os principais conceitos e trazer definições mais aprofundadas, recomendamos o estudo os artigos e livros contidos nas referências.

## 2.2 Pilares do Paradigma-Orientado-a-Objetos

1º **Abstração**: A abstração é o processo de identificar as características essenciais de um objeto do mundo real e representá-las no código. Em POO, isso significa modelar objetos como entidades que têm atributos (dados) e métodos (comportamentos). Por exemplo, um carro pode ser abstraído como um objeto com atributos como cor, modelo e velocidade, e métodos como acelerar, frear e virar.

2º **Encapsulamento**: O encapsulamento é o princípio de esconder os detalhes internos de um objeto e expor apenas uma interface clara para interagir com ele. Isso é alcançado através do uso de modificadores de acesso, como public, private e protected, que controlam a visibilidade dos atributos e métodos de uma classe. O encapsulamento ajuda a proteger os dados de um objeto contra acesso não autorizado e a garantir que o objeto mantenha um estado consistente.

3º **Herança**: A herança é um mecanismo que permite que uma classe (chamada de classe derivada ou subclasse) herde atributos e métodos de outra classe (chamada de classe base ou superclasse). Isso promove a reutilização de código, pois as subclasses podem estender ou modificar o comportamento das superclasses sem precisar reescrever todo o código. Por exemplo, uma classe "Carro Esportivo" pode herdar atributos e métodos de uma classe "Carro" mais genérica, adicionando funcionalidades específicas de carros esportivos, como turbo e spoiler.

4º **Polimorfismo**: O polimorfismo é a capacidade de objetos de diferentes classes responderem à mesma mensagem de maneiras diferentes. Isso é alcançado através do conceito de sobrescrita de método (override) e sobrecarga de método (overload). O polimorfismo permite que o código seja mais flexível e genérico, pois os objetos podem ser tratados de forma uniforme, independentemente de sua classe específica. Por exemplo, uma mesma mensagem "mover()" pode ser enviada para diferentes objetos, como um carro, um avião ou um navio, e cada objeto pode responder de maneira apropriada de acordo com sua própria implementação do método "mover()".

Esses quatro pilares fornecem os princípios fundamentais para a construção de sistemas robustos, flexíveis e de fácil manutenção em programação orientada a objetos.

## 3 Linha do tempo da Programação Orientada a Objetos:

### 3.1 Anos 1960: Início Conceitual

A Programação Orientada a Objetos tem início nos anos 60, com a criação da Simula. A Simula é uma família de linguagens de programação, projetadas com o objetivo de simular eventos discretos, criada em 1962 por Kristen Nygaard e Ole-Johan Dahl no Centro Norueguês de Computação em Oslo. Neste centro trabalhava-se em simulações de naves, que foram confundidas pela explosão combinatória de como as diversas qualidades de diferentes naves poderiam afetar umas às outras. O primeiro projeto da Simula foi a Simula I, baseada em ALGOL 60, linguagem que foi criada em 1962, mas só mais tarde foi criada a primeira linguagem de programação orientada a objetos: a Simula-67. Simula-67 é outra das linguagens da família Simula, e foi ela que introduziu os conceitos de objetos e troca de mensagens para construção de programas. A ideia surgiu ao agrupar os diversos tipos de naves em diversas classes de objetos, sendo responsável cada classe de objetos por definir os seus "próprios" dados e comportamentos, introduzindo conceitos fundamentais como classes, objetos, herança e instâncias.

### 3.2 Anos 1970: Consolidação dos Conceitos

Os conceitos da POO foram posteriormente amadurecidos e aprimorados durante a década de 1970 pela linguagem de programação Smalltalk, criada no laboratório de pesquisa da Xerox, nos Estados Unidos. Desenvolvida por Alan Kay, Dan Ingalls, e Adele Goldberg no Xerox PARC. Smalltalk é uma linguagem de programação dinamicamente tipada que implementou a POO de forma mais pura que a Simula. Em Smalltalk tudo é objeto: os números, as classes, os métodos, blocos de código, etc. O Smalltalk passou por algumas versões até chegar no produto final, a linguagem evoluiu da versão 71 para a versão 72, e mais tarde para o Smalltalk-76, considerado o primeiro Smalltalk moderno. Mais tarde, foi lançado o Smalltalk-80, a primeira versão pública do Smalltalk, disponibilizado para empresas como a Apple, DEC, e IBM como teste de portabilidade de ambiente. O Smalltalk-80 virou um padrão para as diversas versões de Smalltalk. Introduziu a ideia de que "tudo é um objeto" e popularizou os conceitos de mensagens entre objetos e a herança.

### 3.3 Anos 1980: Popularização e Desenvolvimento

Em 1983, trabalhando na AT&T Bell Labs, Bjarne Stroustrup desenvolveu o C++ como uma extensão da linguagem C. O objetivo era adicionar funcionalidades de programação orientada a objeto ao C, combinando alguns pontos positivos de cada linguagem, o C++ logo se tornou uma linguagem poderosa e versátil, e que logo foi adotada na indústria de software. O surgimento do C++ foi muito significativo na indústria de tecnologia, empresas de software começaram a adotar essa nova linguagem para o desenvolvimento de produtos e serviços, incluindo sistemas operacionais, ferramentas de produtividade, jogos de computador e aplicativos empresariais. No ano seguinte, em 1984, Brad Cox e Tom Love desenvolveram o Objective-C. Eles combinaram a linguagem C com os conceitos do Smalltalk, criando uma linguagem que incorporava os paradigmas de POO de maneira robusta. O Objective-C se tornou a linguagem principal usada pela Apple no desenvolvimento de softwares, utilizada durante anos para criar aplicativos para macOS e iOS.

### 3.4 Anos 1990: Explosão da POO

Os anos 90 foram cruciais para o crescimento da POO. O Java, por exemplo, é uma das linguagens mais utilizadas nos dias atuais. Ela foi oficialmente lançada em 1995, desenvolvida por James Gosling e sua equipe na Sun Microsystems, Java foi pensado com a ideia de ser independente de plataforma, o que significa que os programas escritos em Java podem ser executados em qualquer dispositivo que possua uma máquina virtual Java. A linguagem Java também foi projetada com uma forte segurança e robustez, o que o torna uma escolha popular no desenvolvimento de uma ampla variedade de aplicações, desde aplicações web até sistemas corporativos complexos.

Nesta época também surgiram algumas outras linguagens que não se popularizaram tanto quanto o Java, mas tiveram sua importância no desenvolvimento da computação e na evolução da programação orientada a objetos e paradigmas de programação em geral, exemplos são: Ada, uma linguagem originalmente desenvolvida para aplicações militares; Lisp, que é uma das linguagens de programação mais antigas ainda em uso conhecida por sua flexibilidade e poder em processamento simbólico; e Pascal, uma linguagem estruturada com foco em ensino e clareza de código.

Além disso, devemos lembrar de outras duas linguagens que são muito clássicas, utilizadas até nos dias atuais: Ruby e PHP. Ruby é uma linguagem de programação valorizada por sua sintaxe simples e expressiva, sendo popular para desenvolvimento web. Já o PHP é uma linguagem de script de servidor amplamente utilizada para desenvolvimento web dinâmico e suportada por uma grande comunidade de desenvolvedores.

### 3.5 Anos 2000 em diante: Evolução Contínua

Criada pela Microsoft como parte da plataforma .NET, C# foi inspirada em C++ e Java, oferecendo uma sintaxe limpa e uma robusta biblioteca de classes, facilitando o desenvolvimento de software para o Windows. Muitas linguagens tradicionais, como Python e JavaScript, incorporaram ou aprimoraram recursos de POO. Python, por exemplo, é amplamente utilizado devido à sua sintaxe simples e poderosa, enquanto JavaScript evoluiu com frameworks como Angular e React, que fazem uso intenso de conceitos de POO. Python, embora criado em 1991, viu um aumento significativo em popularidade nos anos 2000 e 2010. Sua simplicidade e clareza de sintaxe fizeram dela uma escolha popular tanto para iniciantes quanto para desenvolvedores experientes.

## 3.6 Quem era Alan Curtis Kay?

### 3.6.1 Formação acadêmica

Alan Kay formou-se em Matemática e Biologia Molecular. Sua formação multidisciplinar ajudou a moldar sua abordagem inovadora em tecnologia e educação.

### 3.6.2 Interesse em educação

Kay sempre teve um grande interesse em educação, especialmente no ensino de crianças. Ele acreditava no poder da tecnologia para transformar a maneira como as crianças aprendem e interagem com o mundo.

### 3.6.3 Trabalho na Xerox

Um dos períodos mais importantes da carreira de Kay foi seu trabalho no Xerox Palo Alto Research Center (PARC). Lá, ele esteve envolvido em vários projetos revolucionários que moldaram o futuro da computação. Durante sua estadia na Xerox PARC, ele liderou o desenvolvimento da linguagem de programação Smalltalk, uma das primeiras linguagens orientadas a objetos.

### 3.6.4 Influência e Legado

Para Alan Kay, a POO não era apenas uma técnica de programação, mas uma maneira de pensar sobre sistemas de software e seu design. Ele viu a POO como uma forma de tornar o software mais modular, reutilizável e mais fácil de entender e manter, o que reflete sua crença na importância da educação e da acessibilidade na computação.

## 4 Vantagens e desvantagens

### 4.1 Vantagens:

#### 4.1.1 Reutilização

A reutilização de código é um princípio chave na POO, possibilitando a criação de classes genéricas que podem ser usadas em diferentes contextos. Na Figura 1, a classe Veiculo é uma classe base que pode ser reutilizada para criar outras classes: Já na Figura 2, houve a reutilização do código para criar outras duas classes genéricas (Carro e Moto) de mesmo contexto.

#### 4.1.2 Facilidade na manutenção

A POO contribui para um código mais organizado e mais fácil de manter com a Divisão modular e Localização de erros simplificada.

#### 4.1.3 Modelagem Realista

A POO permite modelar sistemas de forma mais próxima à realidade, usando a abstração e objetos para representar entidades do mundo real. Na Figura 4 podemos ver a classe Carro com seus respectivos comportamentos.

```
// Subclasse Carro que estende a classe Veiculo
class Carro extends Veiculo { 2 usages
    // Construtor da classe Carro que chama o construtor da classe base (Veiculo)
    > public Carro(String marca, String modelo) { super(marca, modelo); }

    // Método público específico para a classe Carro para abrir a porta
    > public void abrirPorta() { System.out.println("Porta do " + marca + " " + modelo + " está aberta."); }
}
```

Figura 4: Código em Java

```

1  class Veiculo { 2 usages 2 inheritors
2      // Atributos protegidos da classe Veiculo
3      protected String marca; 5 usages
4      protected String modelo; 5 usages
5
6      // Construtor da classe Veiculo para inicializar os atributos
7      public Veiculo(String marca, String modelo) { 2 usages
8          this.marca = marca;
9          this.modelo = modelo;
10     }
11
12     // Método público para simular a aceleração do veículo
13     public void acelerar() { System.out.println(marca + " " + modelo + " está acelerando."); }
14
15     // Método público para simular a frenagem do veículo
16     public void frear() { System.out.println(marca + " " + modelo + " está freando."); }
17
18     // Subclasse Carro que estende a classe Veiculo
19     class Carro extends Veiculo { 2 usages
20         // Construtor da classe Carro que chama o construtor da classe base (Veiculo)
21         public Carro(String marca, String modelo) { super(marca, modelo); }
22
23         // Método público específico para a classe Carro para abrir a porta
24         public void abrirPorta() { System.out.println("Porta do " + marca + " " + modelo + " está aberta."); }
25     }
26
27     // Subclasse Moto que estende a classe Veiculo
28     class Moto extends Veiculo { 2 usages
29         // Construtor da classe Moto que chama o construtor da classe base (Veiculo)
30         public Moto(String marca, String modelo) { super(marca, modelo); }
31
32         // Método público específico para a classe Moto para empinar
33         public void empinar() { System.out.println(marca + " " + modelo + " está empinando."); }
34     }
35 }

```

Figura 1: Código em Java

```

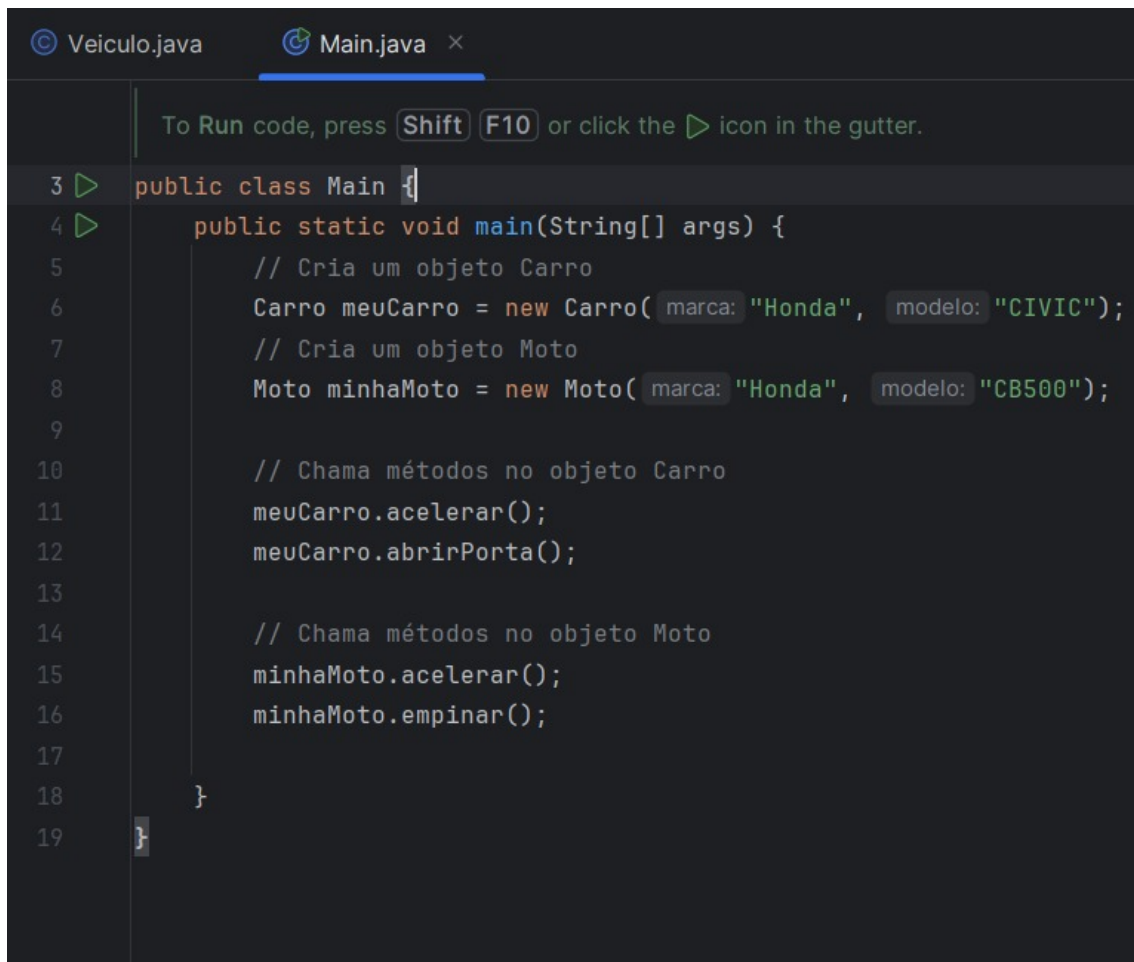
23 // Subclasse Carro que estende a classe Veiculo
24 class Carro extends Veiculo { no usages
25     // Construtor da classe Carro que chama o construtor da classe base (Veiculo)
26     public Carro(String marca, String modelo) { super(marca, modelo); }
27
28     // Método público específico para a classe Carro para abrir a porta
29     public void abrirPorta() { System.out.println("Porta do " + marca + " " + modelo + " está aberta."); }
30 }
31
32 // Subclasse Moto que estende a classe Veiculo
33 class Moto extends Veiculo { no usages
34     // Construtor da classe Moto que chama o construtor da classe base (Veiculo)
35     public Moto(String marca, String modelo) { super(marca, modelo); }
36
37     // Método público específico para a classe Moto para empinar
38     public void empinar() { System.out.println(marca + " " + modelo + " está empinando."); }
39 }

```

Figura 2: Código em Java

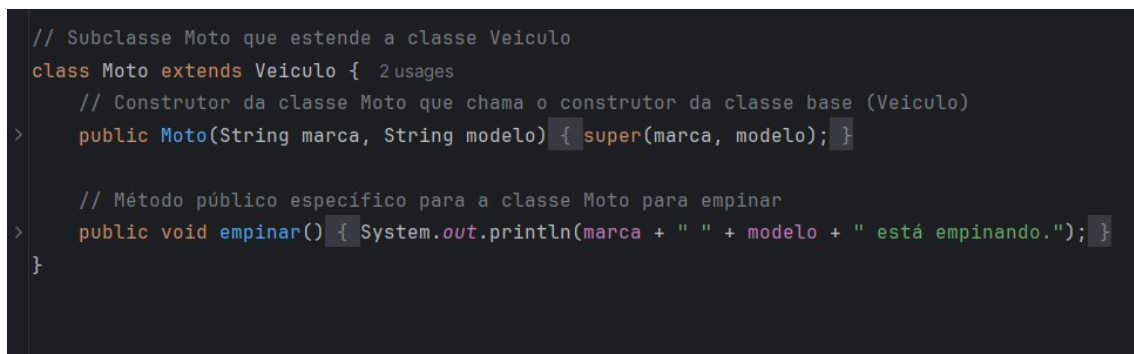
#### 4.1.4 Extensibilidade

No exemplo abaixo, adicionar novas funcionalidades específicas para Moto é fácil, sem necessidade de alterar a classe base Veiculo.



```
Veiculo.java Main.java x
To Run code, press Shift F10 or click the play icon in the gutter.
3 public class Main {
4     public static void main(String[] args) {
5         // Cria um objeto Carro
6         Carro meuCarro = new Carro( marca: "Honda", modelo: "CIVIC");
7         // Cria um objeto Moto
8         Moto minhaMoto = new Moto( marca: "Honda", modelo: "CB500");
9
10        // Chama métodos no objeto Carro
11        meuCarro.acelerar();
12        meuCarro.abrirPorta();
13
14        // Chama métodos no objeto Moto
15        minhaMoto.acelerar();
16        minhaMoto.empinar();
17    }
18 }
19 }
```

Figura 3: Código em Java



```
// Subclasse Moto que estende a classe Veiculo
class Moto extends Veiculo { 2usages
    // Construtor da classe Moto que chama o construtor da classe base (Veiculo)
> public Moto(String marca, String modelo) { super(marca, modelo); }

    // Método público específico para a classe Moto para empinar
> public void empinar() { System.out.println(marca + " " + modelo + " está empinando."); }
}
```

Figura 5: Código em Java

## 4.2 Desvantagens:

### 4.2.1 *Complexidade Inicial*

Exemplo: Comparado a uma abordagem procedural simples, a POO requer mais esforço inicial para definir classes e objetos.

```
// Abordagem POO
class Pessoa { no usages
    private final String nome; 2 usages
    private final int idade; 2 usages

    public Pessoa(String nome, int idade) { no usages
        this.nome = nome;
        this.idade = idade;
    }

    public void detalhes() { no usages
        System.out.println("Nome: " + nome + ", Idade: " + idade);
    }
}
```

Figura 6: Código em Java

### 4.2.2 *Overhead de Performance*

Exemplo: A criação de múltiplos objetos pode introduzir overhead de memória e processamento

```
// Criação de muitos objetos pode introduzir overhead
Sensor[] sensores = new Sensor[1000]; 2 usages
for (int i = 0; i < sensores.length; i++) {
    sensores[i] = new Sensor(i);
}
```

Figura 7: Código em Java

### 4.2.3 *Maior Uso de Memória*

Exemplo: Objetos geralmente consomem mais memória comparado a variáveis simples em abordagens procedurais.



```
// Criando um milhão de objetos
Dispositivo[] dispositivos = new Dispositivo[1000000]; 2 usages
for (int i = 0; i < dispositivos.length; i++) {
    dispositivos[i] = new Dispositivo(i);
}
```

Figura 8: Código em Java

#### 4.2.4 Possibilidade de Over-Engineering

Exemplo: A criação de estruturas de classes complexas pode levar a um design excessivamente complexo para problemas simples

```
// Classe abstrata e subclasses podem ser over-engineering para problemas simples
abstract class FormaGeometrica { 1usage 1inheritor
    abstract double area(); no usages 1implementation
}

class Retangulo extends FormaGeometrica { no usages
    private double largura; 2 usages
    private double altura; 2 usages

    public Retangulo(double largura, double altura) { no usages
        this.largura = largura;
        this.altura = altura;
    }

    @Override no usages
    double area() {
        return largura * altura;
    }
}
```

Figura 9: Classe abstrata FormaGeometrica

## 5 Comparação com outros Paradigmas

### 5.1 POO VS FUNCIONAL

Quando falamos de POO, é importante discernirmos os diferentes tipos e características de paradigmas. É importante ressaltar que a escolha de determinado paradigma depende do problema em questão e de que não há uma solução pronta para tudo. Usando POO e PF (programação funcional) como exemplo, vemos que a programação orientada a objetos é feita para adaptar problemas reais e de grande complexidade. Um exemplo comum de aplicação da POO é o desenvolvimento de sistemas de gerenciamento de usuários e gestão de estoque, uma vez que a abordagem modular e reutilizável facilita a manutenção e expansão do sistema. Já a programação funcional é um paradigma de programação que se concentra na avaliação de expressões e na aplicação de funções matemáticas. A PF se baseia em funções, utilizando-se de funções puras e da imutabilidade dessas funções, ela foi feita para ser usada na análise e no processamento de dados.

### 5.2 POO VS IMPERATIVO

Como já visto, os paradigmas são formas de abordar e resolver uma situação com a programação. Dito isso, para compararmos o nosso paradigma (OO) com os outros apresentados, devemos no



mínimo ter uma base de cada um. No paradigma OO, a programação é feita utilizando objetos para encapsulamento de dados e classes, objetos dos quais são instanciados por classes. Os objetos interagem entre si através de métodos. Agora no paradigma imperativo, a programação é baseada em instruções que alteram o estado do programa. O foco está em como realizar tarefas, ou seja, a sequência de comandos que o computador deve seguir para atingir um objetivo. Assim, enquanto o paradigma imperativo divide seu código em uma sequência de instruções e funções, o paradigma POO divide o problema em objetos que combinam dados e métodos. O paradigma imperativo dá ênfase à descrição assertiva das operações para realizar a modelagem do problema. Já o paradigma POO, dá ênfase na modelagem do problema com um grupo de objetos que, encapsulam tanto o estado (dados) quanto o comportamento (método), promovendo maior reutilização de código e deixando o código mais organizado.

### 5.3 POO VS Lógico

Continuando nossa comparação, no paradigma Lógico, a programação é baseada em declarações de fatos e regras, utilizando a lógica computacional para resolver problemas. Em vez de comandos sequenciais, é definido o que é verdadeiro e o que você quer descobrir. Após isso, o sistema de inferência deduz as respostas automaticamente a partir dessas declarações. Visto isso enquanto o paradigma lógico divide o problema em fatos e regras lógicas que descrevem as relações e restrições no problema, além disso não existe um estado explícito mantido pelo programa, em vez disso a solução é derivada a partir de fatos e regras. Já no OO é dividido o problema com objetos que combinam dados e comportamentos, ou seja, enquanto no OO é priorizado a estrutura e a facilidade de modelagem, no paradigma lógico é priorizado a definição do que é verdadeiro e na inferência automática de respostas a partir de fatos e regras, dando mais valor sobre a lógica declarativa do que a estrutura e comportamento de dados. Salientando as diferenças, o paradigma lógico foca na especificação de regras e fatos que representam o conhecimento do problema. A lógica declarativa permite que o programador defina o que é necessário sem detalhar como isso deve ser alcançado. Já POO foca na modelagem do mundo real ou do problema através de objetos que encapsulam dados e métodos.

## 6 Exemplos de código

Para exemplificar melhor a orientação a objetos e fazer com que fique entendível e compreensível o paradigma e suas características, podemos começar com exemplos mais simples onde se poderia aplicar a orientação a objeto e depois seguiremos para um exemplo mais complicado.

### 6.1 Modelo de exemplo: Carros

No caso da Figura 7.2.1 temos uma classe Carro, que possui 2 atributos que seriam o **modelo** e a cor do carro, temos o **construtor** que pode ser posteriormente utilizado para criar um novo objeto da classe carro passando cor e modelo com **parâmetros** e por último temos o método acelerar que retorna um “print”. Ao criarmos uma main podemos criar um novo objeto:

```
New Corolla = New Carro('Branco' , 'Esportivo')
```

E podemos chamar o seu método acelerar():

```
Corolla.acelerar()
```

Que retornará um print no console:

```
”O carro esta acelerando...”
```

### 6.2 Modelo de exemplo: Biblioteca

Na Figura 11 parte de um código de sistema de cadastro de uma biblioteca ,algo que se aproxima mais de uma aplicação para o mundo real.

A classe Livro em Java representa um livro em uma biblioteca, com funcionalidades básicas para gerenciar seu estado de empréstimo. A classe possui dois atributos: título, que armazena o título do livro, e disponível, que indica se o livro está disponível para empréstimo. O construtor inicializa o livro com um título e define sua disponibilidade como verdadeira. A classe inclui métodos para emprestar (emprestar()) e devolver (devolver()) o livro. O método emprestar() marca o livro como indisponível se ele estiver disponível, e devolver() marca o livro como disponível novamente.

Este código ilustra como gerenciar a disponibilidade de livros em um sistema de biblioteca simples.

1. **Criação de um Livro:**

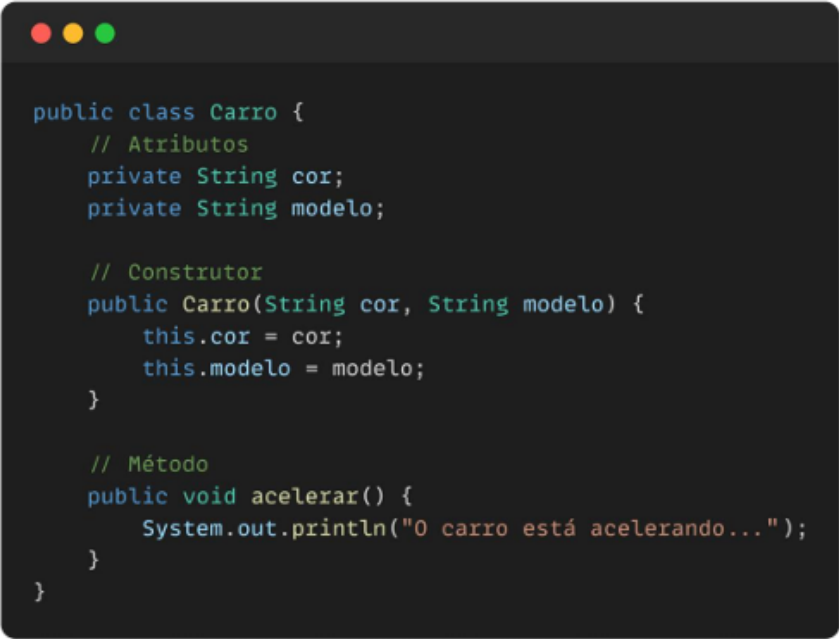
Quando um livro é criado (new Livro(”Java para Iniciantes”)), ele está disponível para empréstimo.

2. **Empréstimo de um Livro:**

O método emprestar() verifica se o livro está disponível. Se estiver, ele marca o livro como emprestado e retorna verdadeiro.

3. **Devolução de um Livro:**

O método devolver() redefine o estado do livro para disponível. A Figura 12 exemplifica uma implementação simples de um sistema de gestão de livros que pode ser usado como base para construir uma aplicação de biblioteca mais complexa.



```

public class Carro {
    // Atributos
    private String cor;
    private String modelo;

    // Construtor
    public Carro(String cor, String modelo) {
        this.cor = cor;
        this.modelo = modelo;
    }

    // Método
    public void acelerar() {
        System.out.println("O carro está acelerando...");
    }
}

```

Figura 10: Código em Java



```

class Livro {
    String titulo;
    boolean disponivel;

    Livro(String titulo) {
        this.titulo = titulo;
        this.disponivel = true;
    }

    boolean emprestar() {
        if (disponivel) {
            disponivel = false;
            return true;
        }
        return false;
    }

    void devolver() {
        disponivel = true;
    }
}

```

Figura 11: Código em Java

```

public class Main {
    public static void main(String[] args) {
        Livro livro = new Livro("Java para Iniciantes");

        // Tenta emprestar o livro
        if (livro.emprestar()) {
            System.out.println(livro.titulo + " foi emprestado com sucesso.");
        } else {
            System.out.println(livro.titulo + " não está disponível.");
        }

        // Devolve o livro
        livro.devolver();
        System.out.println(livro.titulo + " foi devolvido.");
    }
}

```

Figura 12: Código em Java

## 7 Exercícios

### 7.1 Exercícios Teóricos

1 – (SUSEP –2002 -ESAF) Analise as seguintes afirmações relativas à Programação Orientada a Objetos:

- I. Em um programa orientado a objetos, as instâncias de uma classe armazenam os mesmos tipos de informações e apresentam o mesmo comportamento.
- II. Em uma aplicação orientada a objetos, podem existir múltiplas instâncias de uma mesma classe.
- III. Em um programa orientado a objetos, as instâncias definem os serviços que podem ser solicitados aos métodos.
- IV. Em um programa orientado a objetos, o método construtor não pode ser executado quando a classe à qual pertence é executada.

Indique a opção que contenha todas as afirmações verdadeiras.

- a) I e II
- b) II e III
- c) III e IV
- d) I e III
- e) II e IV

2 - (ELETROBRAS NCE-UFRJ 2001) Em relação à tecnologia de orientação a objetos, a afirmativa de que o estado de um objeto não deve ser acessado diretamente, mas sim por intermédio de métodos de acesso (ou propriedades) está diretamente relacionada ao conceito de:

- a) Herança;
- b) Interface;
- c) Classe;
- d) Polimorfismo;
- e) Encapsulamento.

3 - (IF-PR 2010) Com relação ao paradigma de orientação a objetos, considere as seguintes afirmativas:

- I. “Herança” (ou generalização) é o mecanismo pelo qual uma classe (ou subclasse) pode estender outra classe (ou superclasse).
- II. “Polimorfismo” é o princípio pelo qual duas ou mais classes derivadas de uma mesma superclasse podem invocar métodos que têm identificação (assinatura) diferentes, mas comportamentos iguais.

III. “Interface” é um contrato entre a classe e o mundo externo.

Assinale a alternativa correta.

- a) Somente a afirmativa 1 é verdadeira.
- b) Somente a afirmativa 2 é verdadeira.
- c) Somente a afirmativa 3 é verdadeira.
- d) Somente as afirmativas 1 e 3 são verdadeiras.
- e) As afirmativas 1, 2 e 3 são verdadeiras.

4 - (ENADE 2005) A orientação a objetos é uma forma abstrata de pensar um problema utilizando-se conceitos do mundo real e não, apenas, conceitos computacionais. Nessa perspectiva, a adoção do paradigma orientado a objetos implica necessariamente que:

- a) Os usuários utilizem as aplicações de forma mais simples.
- b) Os sistemas sejam encapsulados por outros sistemas.
- c) Os programadores de aplicações sejam mais especializados.
- d) Os objetos sejam implementados de maneira eficiente e simples.
- e) A computação seja acionada por troca de mensagens entre objetos.

5 - Analise as seguintes afirmativas.

I. Ocultar dados dentro das classes e torná-los disponíveis apenas por meio de métodos é uma técnica muito usada em programas orientados a objetos e é chamada de sobrescrita de atributos.

II. Uma subclasse pode implementar novamente métodos que foram herdados de uma superclasse. Chamamos isso de sobrecarga de métodos.

III. Em Java não existe Herança múltipla como em C++. A única maneira de se obter algo parecido é via interfaces.

Estão incorretas:

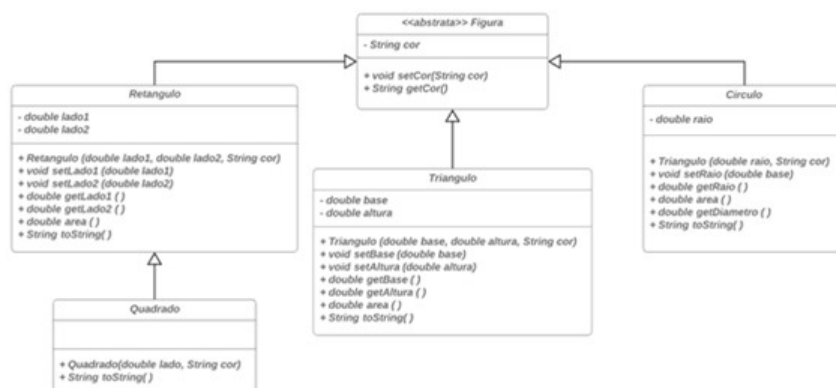
- a) I
- b) II
- c) III
- d) I e III
- e) I e II

### 7.1.1 Gabarito das questões teóricas

- 1 - a) I e II
- 2 - e) Encapsulamento.
- 3 - e) As afirmativas 1, 2 e 3 são verdadeiras.
- 4 - e) A computação seja acionada por troca de mensagens entre objetos.
- 5 - a) I

## 7.2 Exercícios Práticos

1 - Implemente a seguinte modelagem em Java:



2 - Identifique e explique os erros na classe abaixo:

```
1 public class DemoConstrutor
2 {
3     private int a,b;
4     public DemoConstrutor()
5     {
6         System.out.println("No construtor sem argumentos...");
7         DemoConstrutor(0,0);
8     }
9     public DemoConstrutor(int xa,int xb)
10    {
11        System.out.println("No construtor com argumentos...");
12        a = xa; b = xb;
13    }
14 }
```

3 - Identifique e explique os erros na classe abaixo:

```
1 public class Data
2 {
3     private byte dia,mês;
4     private short ano;
5     private Data(byte dd,byte mm,short aa)
6     {
7         dia = dd; mês = mm; ano = aa;
8     }
9 }
```

4 - Analise o código abaixo:

```
2 package media;
3
4 public class calcularMedia {
5     private double nota1;
6     private double nota2;
7     private double media;
8     private int matricula;
9     private String nome;
10
11     public void calcularMedia(double nota1, double nota2{
12         this.nota1 = nota1;
13         this.nota2 = nota2;
14         media = (nota1 - nota2)/2;
15     }
16
17     public void cadastrarAluno(int cod, String matricula){
18         this.cod = cod;
19         this.matricula = matricula;
20     }
21 }
```

É correto afirmar que:

- A) A classe "calcularMedia" segue a regra geral para nomes de classes.
- B) Esta classe não apresenta coesão.
- C) Esta classe não está dentro de nenhum pacote.
- D) A classe está escrita totalmente correta.
- E) O método "calcularMedia não irá executar a expressão aritmética.

5 - Identifique e explique os erros na classe abaixo:

```
1 public class MudaString
2 {
3     public static void main(String[] argumentos)
4     {
5         String nome = "Dan Gusfield";
6         nome.charAt(3) = '+';
7         System.out.println(nome);
8     }
9 }
```

### 7.2.1 Gabarito das questões práticas

Questão 1 -

```
1 package exAula08;
2
3 public abstract class Figura {
4     private String cor;
5
6     public Figura() {
7     }
8
9     public Figura(String cor) {
10         super();
11         this.cor = cor;
12     }
13
14     public String getCor() {
15         return cor;
16     }
17
18     public void setCor(String cor) {
19         this.cor = cor;
20     }
21
22     @Override
23     public String toString() {
24         return "Figura [cor=" + cor + "]";
25     }
26
27     public abstract double area();
28 }
```



```

1  package exAula08;
2
3  public class Retangulo extends Figura {
4      private double lado1;
5      private double lado2;
6
7      @Override
8      public double area() {
9          return this.lado1 * this.lado2;
10     }
11
12     public Retangulo(String cor, double lado1, double lado2) {
13         super(cor);
14         this.lado1 = lado1;
15         this.lado2 = lado2;
16     }
17
18     public Retangulo() {
19     }
20
21     public double getLado1() {
22         return lado1;
23     }
24
25     public void setLado1(double lado1) {
26         this.lado1 = lado1;
27     }
28
29     public double getLado2() {
30         return lado2;
31     }
32
33     public void setLado2(double lado2) {
34         this.lado2 = lado2;
35     }
36
37     @Override
38     public String toString() {
39         return "Retangulo [lado1= " + lado1 + ", lado2= " + lado2 + ", cor= " + super.getCor() + "];";
40     }
41 }

```

```

1  package exAula08;
2
3  public class Triangulo extends Figura {
4      private double base;
5      private double altura;
6
7      public Triangulo() {
8      }
9
10     public Triangulo(String cor, double base, double altura) {
11         super(cor);
12         this.base = base;
13         this.altura = altura;
14     }
15
16     public double getBase() {
17         return base;
18     }
19
20     public void setBase(double base) {
21         this.base = base;
22     }
23
24     public double getAltura() {
25         return altura;
26     }
27
28     public void setAltura(double altura) {
29         this.altura = altura;
30     }
31
32     @Override
33     public double area() {
34         return base * altura;
35     }
36
37     @Override
38     public String toString() {
39         return "Triangulo [base=" + base + ", altura=" + altura + ", cor= " + super.getCor() + "];";
40     }
41 }

```

```

1 package exAula08;
2
3 public class Circulo extends Figura {
4     private double raio;
5
6     public Circulo(String cor, double raio) {
7         super(cor);
8         this.raio = raio;
9     }
10
11     public Circulo() {
12     }
13
14     public double getRaio() {
15         return raio;
16     }
17
18     public void setRaio(double raio) {
19         this.raio = raio;
20     }
21
22     @Override
23     public String toString() {
24         return "Circulo [raio=" + raio + ", cor= " + super.getCor() + "]\n";
25     }
26
27     @Override
28     public double area() {
29         return raio * 3.14 * 2;
30     }
31 }

```

```

1 public class Quadrado extends Retangulo {
2     public Quadrado() {
3         super();
4     }
5
6     public Quadrado(String cor, double lado1, double lado2) {
7         super(cor, lado1, lado2);
8     }
9
10    public Quadrado(String cor, double lado) {
11        lado = super.getLado1();
12    }
13
14    @Override
15    public double area() {
16        return this.getLado1() * this.getLado1();
17    }
18
19    @Override
20    public String toString() {
21        return "Quadrado [area()=" + area() + ", cor= " + super.getCor() + "]\n";
22    }
23 }

```

```

1 package exAula08;
2
3 public class Teste {
4     public static void main(String[] args) {
5         Quadrado quadrado = new Quadrado();
6         Triangulo triangulo = new Triangulo("Azul", 5, 12);
7         Circulo circulo = new Circulo("Verde", 10.2);
8         Retangulo retangulo = new Retangulo("Rosa", 32, 12);
9
10        quadrado.setCor("Roxo");
11        quadrado.setLado1(4);
12
13        System.out.println(quadrado);
14        System.out.println(triangulo);
15        System.out.println(circulo);
16        System.out.println(retangulo);
17    }
18 }

```

Questão 2 -

- 1º No construtor DemoConstrutor(), há uma tentativa de chamar o construtor DemoConstrutor(int xa, int xb) usando DemoConstrutor(0,0);. No entanto, para chamar outro construtor da mesma classe, deve-se usar a palavra-chave this em vez do nome da classe.
- 2º Correção: Alterar DemoConstrutor(0,0); para this.DemoConstrutor(0, 0);.

```

1 public class Data
2 {
3     private byte dia,mês;
4     private short ano;
5     private Data(byte dd,byte mm,short aa)
6     {
7         dia = dd; mês = mm; ano = aa;
8     }
9 }

```

Questão 3 -

- 1º Alterar os nomes das variáveis para evitar caracteres especiais, por exemplo, mes e ano.
- 2º Usar this para referenciar os campos da classe. Assim, ficaria this.dia = dd; this.mes = mm; this.ano = aa;.

Questão 4 - B) Esta classe não apresenta coesão.

Questão 5 - Correção: Para modificar uma String, é necessário criar uma nova String com as alterações desejadas. Uma forma de fazer isso é usando StringBuilder ou manipulando substrings.

## Referências

- [Gabbrielli and Martini, 2023] Gabbrielli, M. and Martini, S. (2023). *Programming languages: principles and paradigms*. Springer Nature.
- [Korson and McGregor, 1990] Korson, T. and McGregor, J. D. (1990). Understanding object-oriented: A unifying paradigm. *Communications of the ACM*, 33(9):40–60.
- [Snyder, 1986] Snyder, A. (1986). Encapsulation and inheritance in object-oriented programming languages. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 38–45.
- [Van Roy, 2009] Van Roy, P. (2009). Programming paradigms for dummies: What every programmer should know. *New computational paradigms for computer music*, 104:616–621.

[Curso POO Teoria 01a - O que é Programação Orientada a Objetos](#)  
[Dio. - O que é a POO](#)  
[Wikipedia - Programação orientada a objetos](#)  
[Wikipedia - Simula](#)  
[Wikipedia - Smalltalk](#)  
[LocalWeb - Programacao funcional e POO](#)  
[Feministech - Programação Orientada a Objetos e Funcional](#)  
[Bemol Digital - Paradigmas no desenvolvimento de software POO](#)  
[Lecture 14: Programming Languages and Programming on the Web](#)  
[Lecture 1: Programming Paradigms](#)  
[Wikipedia - Programação Lógica](#)  
[Wikipedia - Programação Imperativa](#)