

Programação Orientada a Objetos

Henrique dos Santos de Oliveira
Professor-José Eduardo Soares de Lima
Escola Ulbra São Lucas
29/08/2024

RESUMO

A programação orientada a objetos (POO) é um paradigma de programação que organiza o código em torno de objetos, que representam entidades do mundo real e possuem características (atributos) e comportamentos (métodos).

Na programação orientada a objetos, um objeto é uma instância de uma classe. Sendo que uma classe é uma estrutura que define as propriedades e comportamentos que um objeto pode ter. Por exemplo, imagine uma classe chamada “Carro”. Essa classe pode ter atributos como cor, modelo e velocidade, e métodos como “acelerar” e “frear”. Assim, os objetos criados a partir dessa classe são instâncias individuais de carros, com valores específicos para seus atributos.

Palavras-chave: Programação Orientada a Objetos, POO, Classes, Objetos, Encapsulamento, Polimorfismo, Herança, Sobrecarregamento de Métodos, Sobrescrição de Métodos

• INTRODUÇÃO

A programação orientada a objetos é importante por diversos motivos. Primeiramente, ela promove a reutilização de código por meio do conceito de herança, possibilitando que classes derivadas herdam características e comportamentos de classes base. Isso resulta em um desenvolvimento mais eficiente, pois evita a necessidade de reescrever o mesmo código várias vezes.

Outrossim, a programação orientada a objetos facilita a organização do código em módulos independentes, chamados de classes. Essa modularidade torna o código mais fácil de entender, modificar e depurar, além de facilitar a colaboração em equipe. Assim, alterações em

uma classe específica não afetam diretamente outras partes do sistema, o que simplifica a manutenção do software.

A programação orientada a objetos é uma abordagem valiosa no desenvolvimento de software, fornecendo benefícios como reutilização de código, modularidade, facilidade de manutenção, abstração do mundo real e flexibilidade. Esses aspectos contribuem para um desenvolvimento mais eficiente, sistemas mais robustos e uma base sólida para a construção de software complexo.

- **OBJETO**

Objeto é o elemento individual criado dentro de uma classe. Também conhecido como instância, possui atributos e métodos. Ou seja, é uma entidade concreta baseada no molde da classe. O processo de criação de um objeto chama-se instanciação.

- **CLASSES**

Uma classe é uma forma de definir um tipo de dado em uma linguagem orientada a objeto. Ela é formada por dados e comportamentos.

Para definir os dados são utilizados os atributos, e para definir o comportamento são utilizados métodos. Depois que uma classe é definida podem ser criados diferentes objetos que utilizam a classe. A imagem abaixo mostra a definição da classe Empresa, que tem os atributos nome, endereço, CNPJ, data de fundação, faturamento, e também o método imprimir, que apenas mostra os dados da empresa.

```

public class Empresa {

    private String nome;
    private String cnpj;
    private String endereco;
    private Date dataFundacao;
    private float faturamento;

    public void imprimir() {
        System.out.println("Nome: " + nome);
        System.out.println("CNPJ: " + cnpj);
        System.out.println("Endereço: " + endereco);
        System.out.println("Data de Fundação: " + dataFundacao);
    }

    public String getNome() {
        return nome;
    }
    public void setNome(String nome) {
        this.nome = nome;
    }
    public String getCnpj() {
        return cnpj;
    }
    public void setCnpj(String cnpj) {
        this.cnpj = cnpj;
    }
    public String getEndereco() {
        return endereco;
    }
    public void setEndereco(String endereco) {
        this.endereco = endereco;
    }
    public Date getDataFundacao() {
        return dataFundacao;
    }
    public void setDataFundacao(Date dataFundacao) {
        this.dataFundacao = dataFundacao;
    }
}

```

- **VARIÁVEIS**

Uma variável representa um contêiner ou espaço na memória física ou virtual de um computador, onde diferentes tipos de dados (valores) são armazenados durante a execução de um programa. A cada variável é atribuído um nome descritivo ou identificador que se refere ao valor salvo. Os dados armazenados podem mudar de valor ou serem constantes.

- **CONSTANTES**

Ao contrário das variáveis, que podemos alterar o valor conforme a necessidade do algoritmo a ser desenvolvido, as constantes precisam ser inicializadas e não podem ter o seu valor alterado.

Por exemplo, caso seja necessário calcular a área de uma circunferência, podemos adotar o seguinte algoritmo:

algoritmo

declare real área;

declare real raio = 5;

declare constante real pi = 3.14;

area = pi * (raio * raio);

exibir area;

Este algoritmo ficará da seguinte forma:

```
void main() {  
    float area;  
    float raio = 5;  
    const float pi = 3.14;  
  
    area = pi * (raio * raio);  
  
    printf("\nA área é de %0.2f", area);  
}
```

Teremos o seguinte retorno:

```
A área é de 78.50  
  
...Program finished with exit code 20  
Press ENTER to exit console.□
```

No caso a constante pi recebeu o valor de 3.14, afinal, de acordo com as leis matemáticas PI é um valor fixo, se por algum motivo o algoritmo tentar mudar o valor de pi, como se pode ver utilizando C, retornará um erro como abaixo:

```
9  #include <stdio.h>
10
11 void main()
12 {
13     float area;
14     float raio = 5;
15     const float pi = 3.14;
16
17     area = pi * (raio * raio);
18
19     printf("\nA área é de %.2f", area);
20
21     pi = 10;
22 }
23
```

input

Compilation failed due to following error(s).

```
main.c: In function 'main':
main.c:21:8: error: assignment of read-only variable 'pi'
    pi = 10;
    ^
```

- **ENCAPSULAMENTO**

O encapsulamento é um conceito fundamental na POO, que visa controlar o acesso aos atributos e métodos de um objeto. Isso é alcançado através da definição de níveis de visibilidade, como público, protegido e privado. A ideia por trás do encapsulamento é que os detalhes internos de um objeto devem ser ocultados do mundo exterior, permitindo que apenas as operações essenciais sejam realizadas por meio de uma interface pública. Ao ocultar os detalhes internos, é possível proteger os dados de alterações indesejadas e manter a integridade do objeto. Esse pilar promove a modularidade, facilita a manutenção e evolução do código e ajuda a prevenir erros decorrentes de acessos incorretos aos dados.

Exemplo:

```
public class Veiculo {  
    private String placa;  
  
    public void setPlaca(String placa) {  
        this.placa = placa;  
    }  
  
    public void exibirPlaca() {  
        System.out.println("Placa do veículo: " + placa);  
    }  
}
```

Neste exemplo, temos a classe Veiculo que possui um atributo privado chamado placa. O encapsulamento é aplicado ao controlar o acesso direto a esse atributo e fornecer métodos públicos para interagir com ele. O método setPlaca é usado para definir a placa do veículo, enquanto o método exibirPlaca é usado para mostrar a placa na saída. Essa abordagem protege o atributo placa de modificações indesejadas e centraliza o acesso a ele por meio de métodos controlados.

- **POLIMORFISMO**

O polimorfismo se refere à capacidade de um objeto executar diferentes comportamentos com base no contexto em que é utilizado. Isso significa que objetos de classes diferentes podem responder a uma mesma mensagem (chamada de método) de maneiras distintas. O polimorfismo contribui para a flexibilidade do código, permitindo que diferentes classes compartilhem uma mesma interface e, ao mesmo tempo, implementem comportamentos específicos. Isso simplifica a extensão e a manutenção do sistema, pois novas classes podem ser adicionadas sem afetar o código existente que depende da interface comum.

Exemplo:

```
public class ExemploPolimorfismo {  
    public static void main(String[] args) {  
        Veiculo carro = new Carro("ABC123");  
        Veiculo motocicleta = new Motocicleta("XYZ456");  
  
        carro.ligar(); // Saída: Carro ligado.  
        motocicleta.ligar(); // Saída: Motocicleta ligada.  
    }  
}
```

O polimorfismo é demonstrado quando objetos das classes Carro e Motocicleta (subclasses de Veiculo) são tratados de forma genérica, chamando o método ligar() que é sobrescrito em cada subclasse.

- **HERANÇA**

A herança é um conceito que permite a criação de hierarquias de classes, onde uma classe mais específica pode herdar os atributos e métodos de uma classe mais genérica. Isso permite a reutilização de código, pois características comuns podem ser definidas em uma classe base, e classes derivadas podem estender ou personalizar esse comportamento. A herança reflete a relação "é-um" entre classes, em que uma classe derivada é um tipo mais específico da classe base. Esse pilar contribui para a economia de tempo na codificação, pois evita a duplicação de código e promove uma estrutura organizada e hierárquica.

Exemplo:

```

public class Veiculo {
    protected String placa;

    public Veiculo(String placa) {
        this.placa = placa;
    }

    public void ligar() {
        System.out.println("Veiculo ligado.");
    }
}

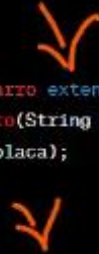
public class Carro extends Veiculo {
    public Carro(String placa) {
        super(placa);
    }

    @Override
    public void ligar() {
        System.out.println("Carro ligado.");
    }
}

public class Motocicleta extends Veiculo {
    public Motocicleta(String placa) {
        super(placa);
    }

    @Override
    public void ligar() {
        System.out.println("Motocicleta ligada.");
    }
}

```



A herança é exemplificada pela relação entre uma classe base Veiculo e suas subclasses Carro e Motocicleta. A classe base possui atributos e métodos comuns, que são herdados pelas subclasses.

• SOBRECARGA METODOS

Em orientação a objetos, a sobrecarga de método permite que existam métodos com o mesmo nome dentro de uma classe, na condição de que possuam tipo ou quantidade de argumentos diferentes, ou seja, tenham diferente assinatura. Métodos sobrecarregados podem ter, por exemplo, diferentes retornos, mas apenas se, antes, eles tiverem diferentes parâmetros, ou seja, essa não pode ser a única característica de distinção entre eles.

Exemplo:


```
public class Soma {
    public int soma(int x, int y) { return x+y; }
    public String soma(String x, String y) { return x+y; }
    public double soma(double x, double y) { return x+y; }
}
```

Quando o método soma é chamado no código, é papel do compilador identificar o método correspondente com base nos parâmetros da função.

• SOBREPOSIÇÃO METODOS

A sobreposição de métodos ocorre quando uma subclasse fornece uma implementação específica para um método que já está definido na sua superclasse. Isso permite que a subclasse modifique o comportamento do método da superclasse para atender às suas necessidades específicas. A sobrescrita é um exemplo de polimorfismo em tempo de execução e é essencial para criar comportamentos mais especializados em subclasses.

```
class Animal {
    public void fazerSom() {
        System.out.println("Som do animal");
    }
}

class Cachorro extends Animal {
    // Sobrescreve o método fazerSom da classe Animal
    @Override
    public void fazerSom() {
        System.out.println("Latido");
    }
}

class Gato extends Animal {
    // Sobrescreve o método fazerSom da classe Animal
    @Override
    public void fazerSom() {
        System.out.println("Miau");
    }
}
```

No exemplo acima, as classes Cachorro e Gato fornecem suas próprias implementações do método fazerSom, que substituem a implementação fornecida pela classe Animal.

• CONSIDERAÇÕES FINAIS

A programação orientada a objetos (POO) é um paradigma de desenvolvimento que facilita a criação de sistemas robustos e escaláveis por meio da utilização de conceitos como classes, objetos, encapsulamento, herança, polimorfismo e sobrecarga de métodos. Esses conceitos promovem a modularidade e a reutilização de código, tornando o desenvolvimento mais eficiente e a manutenção mais fácil.

Através da POO, é possível criar software mais organizado e flexível, onde as alterações em uma parte do sistema têm um impacto mínimo sobre outras partes. A capacidade de modelar o mundo real em termos de objetos e suas interações também contribui para a criação de sistemas mais intuitivos e fáceis de entender.

Portanto, a programação orientada a objetos não apenas melhora a estrutura e a qualidade do código, mas também contribui para o desenvolvimento de soluções de software que podem evoluir e se adaptar às mudanças ao longo do tempo.

REFERÊNCIAS

https://www.alura.com.br/artigos/poo-programacao-orientada-a-objetos?srsId=AfmBOorNFf-5yacDGOujfRIXgzhFEFYAaqfIGM1s_BiKcEbSURkah6KU

<https://hub.asimov.academy/blog/programacao-orientada-a-objetos-conceito-e-pilares/>

<https://www.devmedia.com.br/principais-conceitos-da-programacao-orientada-a-objetos/32285#:~:text=Uma%20classe%20%C3%A9%20uma%20forma,o%20comportamento%20s%C3%A3o%20utilizados%20m%C3%A9todos.>

<https://ebaonline.com.br/blog/variaveis-na-programacao-seo>

<https://www.treinaweb.com.br/blog/variaveis-e-constantas-na-programacao>

<https://www.dio.me/articles/os-4-pilares-da-programacao-orientada-a-objetos-SSU4Q9>

https://pt.wikipedia.org/wiki/Sobrecarga_de_fun%C3%A7%C3%A3o#:~:text=Em%20orienta%C3%A7%C3%A3o%20a%20objetos%2C%20a,ou%20seja%2C%20tenham%20diferente%20assinatura.