Orientação a Objetos

A programação orientada a objetos (POO) é um dos principais paradigmas usados no desenvolvimento de software moderno, sendo fundamental para linguagens como Java, Python e C++. Seu objetivo é estruturar o código de maneira que ele reflita o mundo real, utilizando objetos que possuem atributos e métodos. Nesta aula, vamos explorar os conceitos essenciais da POO, como classes, objetos, herança, encapsulamento, abstração e polimorfismo. Entender esses fundamentos é crucial para desenvolver software modular, reutilizável e flexível, alinhando teoria e prática no desenvolvimento dinâmico de sistemas.

Conceitos de Orientação a Objetos

A orientação a objetos é um paradigma de programação que organiza o desenvolvimento de software de maneira a simular o mundo real. Diferentemente de outros paradigmas, como o imperativo ou o funcional, a orientação a objetos foca em agrupar dados e comportamentos em entidades chamadas "objetos". Esses objetos são projetados para representar elementos do mundo real, como uma "pessoa", um "carro" ou até mesmo conceitos abstratos, como "transações financeiras".

O uso da orientação a objetos facilita a modularização do código, a reutilização de componentes e a manutenção do software. Isso ocorre porque cada objeto contém tanto os dados quanto os métodos (funções) que manipulam esses dados, criando uma separação clara entre as partes de um programa. O código se torna mais organizado, flexível e reutilizável, permitindo que os desenvolvedores criem estruturas complexas a partir de componentes menores e mais simples.

Por exemplo, em um sistema bancário, podemos ter objetos que representam "Clientes", "Contas Bancárias" e "Transações". Cada um desses objetos pode interagir com os outros de maneira controlada. O "Cliente" pode realizar uma "Transação" em sua "Conta Bancária", e todas essas interações são implementadas de forma que reflitam o comportamento no mundo real. Essa proximidade com a realidade é uma das grandes vantagens da orientação a objetos, pois permite que desenvolvedores pensem nos problemas de programação de forma mais natural.

Além disso, ao utilizar esse paradigma, podemos resolver questões complexas de forma modular. Se algo der errado em um componente, como uma transação bancária específica, podemos investigar e corrigir apenas aquele objeto ou método envolvido, sem afetar o restante do sistema. Assim, a programação orientada a objetos ajuda a organizar e simplificar projetos grandes, tornando-os mais fáceis de escalar e manter.

Definição de Classes e Objetos

A base da orientação a objetos está nas classes e nos objetos. Uma classe pode ser vista como um "molde" ou "modelo" que define as características e comportamentos de um tipo de objeto. Já um objeto é uma instância dessa classe, ou seja, ele é um exemplo concreto de um determinado tipo.

Imaginemos uma classe chamada "Carro". Essa classe pode ter atributos como "cor", "marca" e "ano de fabricação", além de métodos como "acelerar" e "frear". A partir dessa classe "Carro", podemos criar vários objetos, ou seja, carros específicos com diferentes combinações de atributos. Por exemplo, um carro pode ser vermelho, da marca Toyota e fabricado em 2020, enquanto outro carro pode ser azul, da marca Ford e fabricado em 2018. Ambos são objetos diferentes, mas compartilham a mesma estrutura básica fornecida pela classe "Carro".

Código exemplo:

```
python

class Carro:

    def __init__(self, cor, marca, ano):

        self.cor = cor

        self.marca = marca

        self.ano = ano



    def acelerar(self):

        print(f"O {self.marca} está acelerando!")

# Criando dois objetos a partir da classe Carro

carro1 = Carro("Vermelho", "Toyota", 2020)

carro2 = Carro("Azul", "Ford", 2018)

# Usando os métodos

carro1.acelerar()  # Saída: O Toyota está acelerando!

carro2.acelerar()  # Saída: O Ford está acelerando!
```

O exemplo acima mostra como a classe "Carro" é utilizada para criar dois objetos distintos: "carro1" e "carro2". Ambos possuem características próprias, mas compartilham o comportamento definido pelos métodos da classe.

As classes não têm limite de criação. Podemos criar quantas classes forem necessárias para organizar e estruturar nosso código. Cada classe representa um conceito, e os objetos dessa classe são os exemplos concretos. Ao desenvolvermos software, utilizamos classes para descrever diferentes partes do sistema, tornando o código mais modular e fácil de entender.

Essa ideia de modularidade é muito importante. Ao agrupar dados e comportamentos dentro de uma classe, criamos blocos de construção reutilizáveis. Por exemplo, se precisarmos implementar outro sistema que também envolva veículos, poderemos reutilizar a classe "Carro" e adaptá-la com pequenas mudanças, sem precisar reescrever tudo do zero.

Herança e Encapsulamento

Dois dos principais pilares da orientação a objetos são a herança e o encapsulamento, conceitos fundamentais para criar hierarquias e controlar o acesso aos dados dentro do sistema.

Herança

A herança permite que uma classe "filha" herde atributos e métodos de uma classe "pai". Isso significa que podemos criar novas classes que reutilizam o comportamento de classes já existentes, adicionando ou alterando funcionalidades conforme necessário. Esse conceito é especialmente útil para evitar a duplicação de código.

Por exemplo, imagine que temos uma classe "Veículo", que tem atributos como "velocidade" e métodos como "mover". A partir dessa classe "Veículo", podemos criar outras classes como "Carro" e "Bicicleta", que herdam de "Veículo", mas adicionam comportamentos específicos.

Código exemplo:

```
python

class Veiculo:
    def __init__(self, velocidade):
        self.velocidade = velocidade

    def mover(self):
        print(f"O veículo está se movendo a {self.velocidade} km/h")
class Carro(Veiculo):
    def __init__(self, velocidade, marca):
        super().__init__(velocidade)  # Herda o atributo velocidade de Veiculo
        self.marca = marca

    def acelerar(self):
        print(f"O {self.marca} está acelerando!")
carro = Carro(120, "Toyota")
carro.mover()      # Saída: O veículo está se movendo a 120 km/h
carro.acelerar()   # Saída: O Toyota está acelerando!
```

Aqui, a classe “Carro” herda da classe “Veículo”, aproveitando o método “mover”. Isso permite que o comportamento seja compartilhado entre diferentes classes sem necessidade de reescrever o código. Assim, a herança facilita a reutilização e a organização do código.

Encapsulamento

Já o encapsulamento diz respeito à proteção dos dados de um objeto. Para garantir que os atributos de um objeto não sejam modificados de maneira inapropriada, utilizamos o encapsulamento para esconder esses dados do acesso direto, permitindo que sejam alterados ou consultados apenas por métodos específicos, como os “getters” e “setters”.

No exemplo de uma classe “Conta Bancária”, podemos querer que o saldo de uma conta só possa ser modificado por métodos como “depositar” ou “sacar”, impedindo que qualquer parte do programa altere o saldo diretamente.

Código exemplo:

```
python

class ContaBancaria:

    def __init__(self, saldo_inicial):

        self.__saldo = saldo_inicial  # Atributo privado



    def depositar(self, valor):

        self.__saldo += valor
```

```
    def sacar(self, valor):
        if valor <= self.__saldo:
            self.__saldo -= valor
        else:
            print("Saldo insuficiente")

    def consultar_saldo(self):
        return self.__saldo
# Criando uma conta e utilizando os métodos
conta = ContaBancaria(1000)
conta.depositar(500)
print(conta.consultar_saldo())  # Saída: 1500
conta.sacar(2000)  # Saída: Saldo insuficiente
```

No exemplo acima, o saldo é encapsulado (definido como privado com "__saldo"), e só pode ser modificado ou consultado através dos métodos fornecidos pela classe. Isso garante que as regras de negócio sejam respeitadas, evitando erros no sistema.

Abstração e Polimorfismo

Abstração

A abstração na orientação a objetos é o processo de simplificar a representação de um objeto do mundo real no código, focando apenas nos detalhes essenciais e ignorando aspectos irrelevantes. Esse processo permite que os desenvolvedores lidem apenas com o que é necessário para resolver um problema específico, sem sobrecarregar o código com detalhes desnecessários.

Por exemplo, imagine que você esteja criando um sistema de gerenciamento imobiliário. Você pode abstrair um “Imóvel” como uma classe que contém apenas os atributos essenciais, como “endereço” e “tamanho”, ignorando detalhes como o tipo de material usado na construção ou a cor das paredes.

Código exemplo:

python

```
class Imovel:

    def init(self, endereco, tamanho):

        self.endereco = endereco

        self.tamanho = tamanho



    def descrever(self):

        print(f"Imóvel localizado em {self.endereco} com {self.tamanho} metros quadrados")

# Criando um objeto do tipo Imovel

imovel = Imovel("Rua Principal, 123", 100)
```

imovel.descrever() # Saída: Imóvel localizado em Rua Principal, 123 com 100 metros quadrados

A abstração nos permite focar no que é mais importante para resolver o problema, sem precisar representar todos os detalhes de um imóvel no mundo real.

Polimorfismo

O polimorfismo permite que um objeto de uma classe possa ser tratado como se fosse de uma classe pai, ou que um método possa se comportar de diferentes maneiras dependendo do objeto com o qual está lidando. Isso é particularmente útil quando temos várias classes que herdam de uma classe base e queremos que elas compartilhem um comportamento, mas com suas próprias implementações.

Por exemplo, imagine que temos as classes “Casa” e “Apartamento”, que herdam de “Imóvel”. Ambas podem ter o método “descrever”, mas a implementação de cada uma será diferente.

Código exemplo:

```
python

class Casa(Imovel):

    def __init__(self, endereco, tamanho, cor):

        super().__init__(endereco, tamanho)

        self.cor = cor
```

```
    def descrever(self):
        print(f"Casa de cor {self.cor}, localizada em {self.endereco}, com {self.tamanho} metros quadrados")

class Apartamento(Imovel):
    def __init__(self, endereco, tamanho, andar):
        super().__init__(endereco, tamanho)
        self.andar = andar

    def descrever(self):
        print(f"Apartamento no {self.andar}º andar, localizado em {self.endereco}, com {self.tamanho} metros quadrados")

# Criando objetos de diferentes classes
casa = Casa("Rua dos Lírios, 45", 150, "Azul")
apartamento = Apartamento("Av. Paulista, 1000", 80, 15)

# Ambos objetos podem chamar o método "descrever", mas o comportamento é específico
casa.descrever()  # Saída: Casa de cor Azul, localizada em Rua dos Lírios, 45, com 150 metros quadrados
apartamento.descrever()  # Saída: Apartamento no 15º andar, localizado em Av. Paulista, 1000, com 80 metros quadrados
```

Aqui vemos o polimorfismo em ação. O método “descrever” é chamado em objetos diferentes, e o comportamento varia de acordo com a classe à qual o objeto pertence, mesmo que ambos herdem da classe “Imóvel”. Isso torna o código mais flexível e adaptável.

Conteúdo Bônus

FreeCodeCamp oferece um excelente guia sobre Programação Orientada a Objetos, abordando conceitos como classes, objetos, encapsulamento, herança e polimorfismo, com exemplos práticos e explicações detalhadas sobre como esses princípios são aplicados no desenvolvimento de software.

Título: Significado de OOP – o que é Programação Orientada a Objetos?

Tradutor: Kristoffer da Silva Lagerström

Autora: Hillary Nyakundi (em inglês)

Plataforma: FreeCodeCamp

Referência Bibliográfica

DEITEL, P. J.; DEITEL, H. M. Ajax, rich internet applications e desenvolvimento web para programadores. Pearson: 2008

FELIX, R. (Org.). Programação orientada a objetos. Pearson: 2016

FERREIRA, R. D. Linguagem de programação. Contentus: 2020.

FLATSCHART, F.; BACHINI, C.; CUSIN, C. Open Web Platform. Brasport: 2013.

NEVES, M. C. B. de A. Sites de Alta Performance. Contentus: 2020

PAGE-JONES, M. Fundamentos do desenho orientado a objeto com UML. Pearson: 2001.

PUGA, S.; RISSETTI, G. Lógica de programação e estruturas de dados, com aplicações em Java. Pearson: 2016.

SEGURADO, V. S. (Org.). Projeto de interface com o usuário. Pearson: 2016.

Atividade Prática 8 – Orientação a Objetos

Título da Prática: Entendendo a definição e uso de Orientação a Objetos
Objetivos: Praticar conceitos para melhor entendimento de Orientação ao Objetos
Materiais, Métodos e Ferramentas: Computador com uma IDE instalada (recomendado: VS Code) e Javascript instalado via node/npm e o entendimento do contexto abaixo

Atividade Prática

A orientação a objetos (OO) é um paradigma de programação que se baseia no conceito de "objetos", que podem conter dados na forma de campos, também conhecidos como atributos. Esse paradigma ajuda a estruturar e projetar o mundo real no código, baseando-se em objetos que interagem entre si.

Uma classe é uma estrutura de dados que define um tipo de objeto. Ela descreve as propriedades e comportamentos que os objetos desse tipo terão.

Um objeto é uma instância de uma classe. Ele é uma entidade real que existe em tempo de execução e pode ser manipulada pelo programa.

O encapsulamento, a herança, a abstração e o polimorfismo são conceitos importantes e que te ajudarão a entender e identificar melhor o uso da Orientação a Objetos. Seguem definições para fixação e melhor entendimento.

- Encapsulamento é o conceito de agrupar dados e os métodos que operam nesses dados em uma única unidade, chamada de objeto. Isso ajuda a proteger os dados de acesso não autorizado e permite que os objetos controlem como seus dados são manipulados.

- Herança é um mecanismo pelo qual uma classe pode herdar atributos e métodos de outra classe. Isso promove a reutilização de código e ajuda a organizar e estruturar programas de forma mais eficiente.

- Abstração é o processo de simplificar a complexidade do mundo real, identificando as características essenciais de um objeto e ignorando os detalhes irrelevantes. Por exemplo, ao modelar um carro em um programa, podemos nos concentrar nas características essenciais, como cor, modelo e velocidade, e ignorar detalhes irrelevantes, como a composição química dos materiais usados.

- Polimorfismo é a capacidade de um objeto de uma classe ser tratado como um objeto de sua classe pai. Isso permite que objetos de diferentes classes sejam tratados de maneira uniforme, simplificando a estrutura e o comportamento do código.

1. O que é uma classe em orientação a objetos?

a. Uma instância de um objeto

b. Um bloco de código que executa uma tarefa específica

c. Uma estrutura de dados que define um tipo de objeto, descrevendo suas propriedades e comportamentos

d. Um método que protege os dados de acesso não autorizado

2. Qual é o conceito que envolve agrupar dados e métodos em uma única unidade?

a. Polimorfismo

b. Encapsulamento

c. Abstração

d. Herança

3. O que permite que uma classe herde atributos e métodos de outra classe

a. Polimorfismo

b. Herança

c. Encapsulamento

d. Abstração

4. Qual é o benefício do polimorfismo em orientação a objetos?

a. Ele agrupa dados em uma única unidade

b. Ele permite que objetos de diferentes classes sejam tratados de forma uniforme

c. Ele define a estrutura de uma classe

d. Ele protege os dados de acesso externo

Gabarito Atividade Prática

1. c. Uma estrutura de dados que define um tipo de objeto, descrevendo suas propriedades e comportamentos.

Comentário: Uma classe é o plano que define as propriedades e comportamentos dos objetos que serão criados a partir dela.

2. b. Encapsulamento.

Comentário: O encapsulamento é o conceito de agrupar dados e métodos que operam nesses dados, ajudando a proteger as informações de acesso indevido.

3. b. Herança.

Comentário: A herança é o mecanismo que permite a uma classe reaproveitar atributos e métodos de outra, promovendo a reutilização de código.

4. b. Ele permite que objetos de diferentes classes sejam tratados de forma uniforme.

Comentário: O polimorfismo facilita o tratamento de diferentes tipos de objetos de forma similar, tornando o código mais flexível e generalizado.

Ir para exercício