Programação em Java

OXETECH LAB

Módulo 3

Programação Orientação a Objetos (POO)

- Introdução à POO
- Classes e Objetos
- Construtores
- Encapsulamento
- Herança
- Polimorfismo
- Classe Abstrata
- Interfaces

Introdução à POO

A Programação Orientada a Objetos (POO) é um paradigma de programação que se concentra em organizar o código em torno de objetos, que são instâncias de classes. A POO foi projetada para lidar com sistemas de software complexos, promovendo uma estrutura modular que facilita a manutenção, extensão e reutilização do código.

Conceito de Orientação a Objetos

Na POO, tudo gira em torno de objetos e classes:

- **Objetos** representam entidades do mundo real, com propriedades e comportamentos. Em um sistema bancário, por exemplo, uma conta bancária pode ser um objeto, com atributos como saldo e número da conta, e métodos para depositar e sacar dinheiro.
- **Classes** são "moldes" ou "planos" a partir dos quais objetos são criados. A classe define as características (atributos) e ações (métodos) comuns aos objetos que dela derivam.

Princípios Fundamentais da Orientação a Objetos

Encapsulamento: O encapsulamento é o princípio de restringir o acesso direto aos dados de um objeto. Através de métodos (getters e setters), os dados são protegidos e acessados de forma

controlada. Em Java, usamos modificadores de acesso (public, private, protected) para determinar a visibilidade dos atributos e métodos.

Herança: A herança permite que uma classe (subclasse) herde os atributos e métodos de outra classe (superclasse). Esse princípio promove o reuso de código e permite que novas classes sejam construídas a partir de classes existentes, reduzindo a redundância.

Polimorfismo: O polimorfismo permite que uma mesma operação se comporte de formas diferentes em objetos distintos. Ele se manifesta de duas formas principais:

- Sobrecarga de métodos (compile-time): permite definir métodos com o mesmo nome, mas com diferentes parâmetros.
- Sobrescrita de métodos (runtime): permite que uma subclasse forneça uma implementação específica para um método já definido na superclasse.

Abstração: A abstração é o processo de simplificar um sistema, expondo apenas os detalhes essenciais, enquanto esconde os detalhes de implementação. Em Java, usamos classes abstratas e interfaces para definir contratos de métodos sem implementar o comportamento completo, permitindo uma organização mais clara e um design focado em funcionalidades.

Principais Vantagens e Aplicações da POO

A POO traz várias vantagens, especialmente para o desenvolvimento de sistemas complexos:

- **Modularidade:** O código é dividido em pequenas partes (objetos e classes) que podem ser desenvolvidas, testadas e mantidas separadamente.
- **Reutilização de Código:** Com herança e composição, é possível reaproveitar funcionalidades existentes em novas classes, diminuindo a duplicação de código.
- **Facilidade de Manutenção:** Alterações em uma parte do código geralmente não afetam outras partes, facilitando a atualização e a correção de erros.
- **Escalabilidade:** A POO facilita a criação de sistemas grandes e complexos, que podem crescer em tamanho e funcionalidade sem se tornarem difíceis de manter.

A Orientação a Objetos é uma abordagem poderosa para o desenvolvimento de software, pois facilita o design de sistemas complexos, promove a reutilização de código e melhora a modularidade e manutenção do código. Entender seus princípios e aplicá-los de forma eficaz ajuda os programadores a criar aplicações robustas e escaláveis.

Classes e Objetos

No contexto da Programação Orientada a Objetos, classes e objetos são elementos centrais. As classes definem a estrutura e o comportamento de objetos, enquanto os objetos são instâncias dessas classes, representando entidades específicas com características e ações definidas pela classe.

Classe

Uma classe é um "molde" ou "plano" que define atributos e métodos comuns para objetos do mesmo tipo. Ela especifica os dados (também conhecidos como atributos ou propriedades) e as ações (ou métodos) que os objetos derivados dessa classe podem ter.

- **Atributos:** São variáveis que armazenam o estado do objeto. Por exemplo, em uma classe Carro, atributos podem ser cor, marca, modelo.
- **Métodos:** São funções que definem o comportamento do objeto. Em uma classe Carro, métodos podem incluir acelerar, frear, ligar.

Objeto

Um objeto é uma instância de uma classe. Ao instanciar uma classe, criamos um objeto que contém valores específicos para seus atributos e pode realizar as ações definidas pelos métodos da classe.

Exemplo:

```java
public class Carro {
    String marca;
    String modelo;
    int ano;

    // Método para exibir informações do carro
    public void exibirInfo() {
        System.out.println("Marca: " + marca);
        System.out.println("Modelo: " + modelo);
        System.out.println("Ano: " + ano);
    }

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

A classe Carro tem atributos marca, modelo e ano, além de métodos como exibirInfo e acelerar. A classe define o que todos os carros devem ter em comum.

```java
public class Main {
    public static void main(String[] args) {
        Carro carro1 = new Carro("Toyota", "Corolla", 2020);
        Carro carro2 = new Carro("Honda", "Civic", 2021);

        carro1.exibirInfo();
        carro2.acelerar();
    }
}
```

Acima, carro1 e carro2 são objetos da classe Carro, com atributos e comportamentos definidos de acordo com a classe. A instrução carro1.exibirInfo() ira exibir as informações do carro1.

Construtores

Os construtores são métodos especiais das classes que têm a função de inicializar objetos. Eles são chamados automaticamente quando uma nova instância de uma classe é criada, permitindo definir valores iniciais para os atributos e garantir que o objeto esteja em um estado válido desde o início.

Conceito de Construtor

Um construtor:

- Sempre possui o mesmo nome da classe.
- Não possui tipo de retorno, nem mesmo void.
- É chamado automaticamente ao criar um objeto com new.

Tipos de Construtores

Existem dois tipos principais de construtores em Java:

- **Construtor Padrão:** Se nenhuma classe declara um construtor, o compilador Java cria automaticamente um construtor padrão sem parâmetros. Este construtor inicializa os atributos com seus valores padrão (por exemplo, 0 para int, null para objetos).
- **Construtor Definido pelo Usuário:** É um construtor implementado pelo programador, que permite inicializar os atributos da classe com valores específicos no momento da criação do objeto.

Criando um Construtor

Construtores são criados dentro da classe com o mesmo nome da classe e são usados para definir valores iniciais para os atributos. Um construtor pode ter parâmetros para receber valores durante a criação do objeto.

```
public class Pessoa {
    private String nome;
    private int idade;

    // Construtor com parâmetros
    public Pessoa(String nome, int idade) {
        this.nome = nome;
        this.idade = idade;
    }
}
```

Aqui, o construtor Pessoa(String nome, int idade) permite que o nome e a idade sejam passados ao criar um novo objeto Pessoa.

Sobrecarga de Construtores

Java permite sobrecarga de construtores, o que significa que uma classe pode ter múltiplos construtores com diferentes listas de parâmetros. Isso permite flexibilidade na criação de objetos, possibilitando diferentes formas de inicialização.

No contexto da classe Pessoa, poderíamos ter dois construtores:

```
public Pessoa() {
    this.nome = "Indefinido";
    this.idade = 0;
}

public Pessoa(String nome, int idade) {
    this.nome = nome;
    this.idade = idade;
}
```

Temos um construtor sem parâmetros, caso não seja fornecidos valores na hora de instanciar o objeto e temos outro construtor com parâmetros para o cenário oposto.

Uso do *this* em Construtores

O this é uma referência ao objeto atual e pode ser usado em construtores para diferenciar os parâmetros dos atributos da classe quando têm o mesmo nome. O this também pode ser usado para chamar outro construtor dentro da mesma classe, facilitando a reutilização de código.

```
public Pessoa(String nome, int idade) {
    this.nome = nome;
    this.idade = idade;
}
```

Na instrução “this.idade= idade;” o this é usado para fazer referência ao atributo da classe, já que o temos um parâmetro com o mesmo nome da atributo da classe.

Porque usar Construtores ?

Os motivos são:

- **Facilidade de Inicialização:** Construtores garantem que o objeto seja inicializado em um estado válido.
- **Flexibilidade na Criação de Objetos:** Com sobrecarga, é possível fornecer diferentes maneiras de criar e inicializar objetos.
- **Código Mais Limpo e Confiável:** Reduz a necessidade de métodos adicionais para configurar os atributos depois da criação do objeto, diminuindo o risco de inconsistências.

Encapsulamento

O encapsulamento é um dos pilares fundamentais da Programação Orientada a Objetos. Ele promove a segurança e a organização do código, protegendo os dados de um objeto contra acesso externo não autorizado e mantendo o controle sobre como esses dados são modificados.

Conceito de Encapsulamento

Encapsulamento consiste em restringir o acesso direto aos dados de uma classe e disponibilizar métodos específicos para manipular esses dados de maneira controlada. Essa técnica permite que as variáveis internas de um objeto sejam acessadas e alteradas apenas por métodos específicos, conhecidos como getters e setters.

- **Getters:** São métodos que permitem acessar o valor de um atributo privado.
- **Setters:** São métodos que permitem modificar o valor de um atributo privado, aplicando verificações, se necessário.

Visibilidade dos Atributos e Métodos

Para implementar o encapsulamento, é comum usar modificadores de acesso para controlar a visibilidade dos atributos e métodos de uma classe.

- **private:** Atributos ou métodos private só podem ser acessados dentro da própria classe.
- **public:** Métodos public podem ser acessados de fora da classe.
- **protected** e **package-private** (sem modificador) também são utilizados em OOP, mas são abordados mais adiante.

Ao definir atributos como private, garantimos que apenas a própria classe pode acessá-los diretamente, evitando acessos indesejados.

Implementação de Getters e Setters

Getters e setters são métodos públicos que permitem acessar e modificar os valores de atributos privados de uma forma controlada. Eles são especialmente úteis para aplicar verificações antes de alterar valores, garantindo consistência nos dados do objeto.

Veremos um exemplo a seguir onde, nome e idade são atributos privados. A classe Pessoa fornece métodos getNome, setNome, getIdade e setIdade para acessar e modificar esses atributos. O setter setIdade inclui uma validação para garantir que a idade seja um valor positivo.

```java
public class Pessoa {
    private String nome;
    private int idade;

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public int getIdade() {
        return idade;
    }

    public void setIdade(int idade) {
        if (idade ⩾ 0) {
            this.idade = idade;
        } else {
            System.out.println("Idade inválida.");
        }
    }
}
```

Vantagens do Encapsulamento

- **Segurança de Dados:** Atributos privados evitam modificações não autorizadas.
- **Facilidade de Manutenção:** Alterações nos métodos de acesso (getters/setters) não afetam o restante do código.
- **Controle sobre os Dados:** Com setters, é possível aplicar verificações antes de alterar o valor dos atributos.
- **Modularidade:** A classe controla como seus dados internos são manipulados, tornando o código mais modular e fácil de entender.

Herança

Herança é um dos pilares da Programação Orientada a Objetos e permite que uma classe herde características (atributos e métodos) de outra classe. Com a herança, podemos criar novas classes baseadas em classes já existentes, facilitando o reaproveitamento de código e promovendo uma estrutura de hierarquia entre as classes.

Conceito de Herança

Na herança:

- A **classe pai** (ou superclasse) é a classe que compartilha suas características.
- A **classe filha** (ou subclasse) é a classe que herda características da classe pai.

A herança permite que a classe filha use métodos e atributos da classe pai, além de poder estender suas funcionalidades.

Implementando Herança em Java

Em Java, a herança é implementada com a palavra-chave **extends**. Ao declarar uma classe como uma extensão de outra, a subclasse herda automaticamente os membros (atributos e métodos) não privados da superclasse.

```java
public class Animal {
  protected String nome;

  public void mostrarNome() {
    System.out.println(this.nome);
  }
}

class Cachorro extends Animal {

  public Cachorro(String nome) {
    this.nome = nome;
  }

}
```

No exemplo acima:

- Cachorro é uma subclasse de Animal.
- Cachorro herda o atributo nome e o método mostrarNome de Animal.

Modificadores de Acesso e Herança

Nem todos os membros da superclasse são acessíveis na subclasse:

- **public:** acessível em qualquer lugar.
- **protected:** acessível em subclasses e no mesmo pacote.
- **default** (sem modificador): acessível apenas dentro do mesmo pacote.
- **private:** não é acessível em subclasses.

Palavra-chave *super*

A palavra-chave super é usada para acessar membros da superclasse a partir da subclasse:

- **super.método():** chama um método da superclasse.
- **super.atributo:** acessa um atributo da superclasse.

Também pode ser usada no construtor da subclasse para chamar o construtor da superclasse.

```
public class Animal {
  protected String nome;

  public Animal(String nome) {
    this.nome = nome;
  }
}

class Cachorro extends Animal {
  public Cachorro(String nome) {
    super(nome);
  }
}
```

Aqui, super(nome) no construtor de Cachorro chama o construtor da classe Animal.

Polimorfismo

Polimorfismo é um conceito fundamental da Programação Orientada a Objetos que permite que objetos de diferentes classes sejam tratados como objetos de uma mesma superclasse. Ele possibilita que um método tenha comportamentos diferentes dependendo do objeto que o invoca, promovendo flexibilidade e extensibilidade no código.

O Conceito de Polimorfismo

Polimorfismo significa “muitas formas” e, isso se refere à capacidade de um objeto assumir diferentes formas. Em Java, o polimorfismo permite que métodos sejam implementados de várias maneiras, dependendo da classe que os utiliza.

Existem dois tipos principais de polimorfismo:

1. **Polimorfismo de Sobrecarga** (ou polimorfismo em tempo de compilação)
2. **Polimorfismo de Sobrescrita** (ou polimorfismo em tempo de execução)

Polimorfismo de Sobrecarga (Overloading)

A sobrecarga ocorre quando vários métodos na mesma classe têm o mesmo nome, mas parâmetros diferentes (quantidade ou tipo). Esse tipo de polimorfismo é resolvido durante a compilação.

```
public class Calculadora {
    public int soma(int a, int b) {
        return a + b;
}

    public double soma(double a, double b) {
        return a + b;
    }
}
```

No exemplo acima, o método soma é sobrecarregado para lidar tanto com inteiros quanto com números de ponto flutuante.

Polimorfismo de Sobrescrita (Overriding)

A sobrescrita ocorre quando uma subclasse implementa um método da superclasse com o mesmo nome, tipo de retorno e parâmetros. Esse tipo de polimorfismo é resolvido em tempo de execução, dependendo do objeto que invoca o método.

```
public class Animal {
    public void emitirSom() {
        System.out.println("O animal emite um som.");
   }
}

public class Cachorro extends Animal {
    @Override
    public void emitirSom() {
        System.out.println("O cachorro late.");
    }
}
```

```
public class Gato extends Animal {
    @Overridepublic void emitirSom() {
    System.out.println("O gato mia.");
    }
}
```

Ao usar o método emitirSom() em uma instância de Cachorro ou Gato, cada um terá uma resposta diferente, mesmo que o método seja chamado da mesma forma.

Uso de Classes e Métodos Polimórficos

Uma das vantagens do polimorfismo é a possibilidade de usar referências de superclasse para referenciar objetos de subclasses. Isso é comum em estruturas de código onde métodos genéricos precisam lidar com diferentes tipos de objetos.

```
public class Main {
    public static void main(String[] args) {
        Animal meuAnimal = new Cachorro();
        meuAnimal.emitirSom(); // Saída: "O cachorro late."
    }
}
```

Aqui, a referência meuAnimal é do tipo Animal, mas o objeto instanciado é do tipo Cachorro. O método emitirSom() invoca a versão específica de Cachorro, graças ao polimorfismo.

Classe Abstrata

Uma classe abstrata é uma classe em Java que não pode ser instanciada diretamente e é projetada para ser uma base para outras classes. Ela fornece uma estrutura comum para classes derivadas (subclasses), permitindo que métodos sejam implementados parcialmente ou deixados sem implementação, para que sejam definidos mais especificamente nas subclasses.

As classes abstratas são usadas quando queremos definir um modelo ou uma estrutura para uma família de classes, garantindo que todas compartilhem comportamentos essenciais, mas permitindo que cada uma implemente esses comportamentos de maneiras diferentes.

Características das Classes Abstratas

- **Não podem ser instanciadas:** Você não pode criar um objeto de uma classe abstrata diretamente.
- **Podem conter métodos abstratos e concretos:**
  - **Métodos abstratos** são declarados sem corpo e devem ser implementados nas subclasses.
  - **Métodos concretos** possuem implementação e podem ser utilizados diretamente pelas subclasses.
- **Podem ter variáveis de instância e construtores:** Embora não sejam instanciadas, classes abstratas podem definir atributos e construtores, que serão chamados ao instanciar uma subclasse.

Sintaxe de uma Classe Abstrata

Para declarar uma classe abstrata em Java, utilizamos a palavra-chave abstract.

```
abstract class Forma {

    protected String cor;
    public Forma(String cor) {
        this.cor = cor;
    }

    // Método abstrato (sem implementação)
    public abstract double calcularArea();

    // Método concreto
    public String getCor() {
        return cor;
    }
}
```

Neste exemplo:

- Forma é uma classe abstrata que serve como base para diferentes formas geométricas.
- calcularArea() é um método abstrato que precisa ser implementado nas subclasses.
- getCor() é um método concreto que retorna a cor da forma.

Implementação de Subclasses

Subclasses de uma classe abstrata devem fornecer implementação para todos os métodos abstratos da superclasse. Caso contrário, elas também deverão ser declaradas como abstratas.

```
class Circulo extends Forma {

    private double raio;
    public Circulo(String cor, double raio) {
        super(cor);
        this.raio = raio;
    }

    @Override
    public double calcularArea() {
        return Math.PI * raio * raio;
    }
}
```

Aqui, Circulo estende Forma e implementa o método calcularArea(). Agora, uma instância de Circulo pode ser criada, pois ele não é abstrato.

Interfaces

Uma interface em Java é uma estrutura que define um conjunto de métodos que uma classe deve implementar, sem fornecer a implementação desses métodos. As interfaces estabelecem um contrato que as classes devem seguir, permitindo a criação de código mais flexível e modular.

Ao utilizar interfaces, conseguimos projetar sistemas mais escaláveis, pois classes diferentes podem implementar a mesma interface de maneiras distintas, atendendo a requisitos específicos.

Características das Interfaces

- **Definem métodos abstratos:** Uma interface é composta por métodos que são implicitamente abstratos, ou seja, eles não possuem implementação na própria interface.
- **Constantes:** Todas as variáveis declaradas em uma interface são públicas, estáticas e finais, atuando como constantes.
- **Implementação múltipla:** Uma classe pode implementar várias interfaces, o que não é possível com classes abstratas, pois Java não permite herança múltipla de classes.
- **Métodos padrão e estáticos:** A partir do Java 8, as interfaces podem ter métodos default (com implementação) e métodos static.

Sintaxe de uma Interface

Para definir uma interface em Java, usamos a palavra-chave interface.

```
public interface Animal {
    void emitirSom(); // Método abstrato
    void dormir(); // Outro método abstrato
}
```

Aqui, Animal é uma interface que define dois métodos: emitirSom() e dormir(). Qualquer classe que implementar essa interface será obrigada a fornecer uma implementação para esses métodos.

Implementação de uma Interface

Uma classe que implementa uma interface deve definir todos os métodos abstratos declarados na interface.

```
public class Cachorro implements Animal {
    @Override
    public void emitirSom() {
        System.out.println("Au au!");
    }

    @Override
    public void dormir() {
        System.out.println("O cachorro está dormindo.");
    }
}
```

No exemplo acima, a classe Cachorro implementa a interface Animal e fornece uma implementação para os métodos emitirSom() e dormir().

Métodos default em Interfaces

A partir do Java 8, as interfaces podem incluir métodos com implementação, usando a palavra-chave default. Estes métodos não precisam ser implementados nas classes que implementam a interface.

```
public interface Animal {
    void emitirSom();

    default void respirar() {
        System.out.println("O animal está respirando.");
    }
}
```

Com isso, qualquer classe que implemente Animal terá acesso ao método respirar() sem a necessidade de reimplementá-lo.

Interfaces permitem que diferentes classes compartilhem um conjunto de métodos, promovendo a reutilização de código e um design mais modular e flexível.

Exercícios

1. Explique o que é Programação Orientada a Objetos (POO) e mencione seus principais pilares. Dê exemplos práticos de como ela pode ser aplicada em sistemas reais.

2. Escreva uma definição para "classe" e "objeto". Em seguida, explique a diferença entre os dois com um exemplo, usando um contexto de "Carro".

3. Crie uma classe chamada Carro com atributos marca, modelo e ano. Em seguida, instancie dois objetos dessa classe, representando carros diferentes.

4. Crie uma classe chamada Pessoa com os atributos nome, idade e endereco. Defina métodos para definir e recuperar cada um desses valores. Instancie um objeto da classe Pessoa e defina seus atributos.

5. Adicione um método fazerAniversario na classe Pessoa, que aumenta a idade em 1. Crie um objeto da classe Pessoa, defina a idade e chame o método para verificar se o valor foi atualizado.

6. Defina um construtor para a classe Pessoa que recebe nome e idade como parâmetros, e inicializa esses valores nos atributos correspondentes. Em seguida, instancie um objeto usando esse construtor e exiba as informações da pessoa..

7. Crie uma sobrecarga do construtor da classe Pessoa que aceita um parâmetro adicional endereco. Instancie um objeto usando o novo construtor e exiba as informações da pessoa.

8. Altere a classe Pessoa para que todos os atributos sejam privados. Adicione métodos get e set para cada atributo. Em seguida, crie um objeto da classe Pessoa e utilize os métodos para definir e recuperar os valores.

9. Modifique o método setIdade da classe Pessoa para garantir que a idade seja sempre um valor positivo. Teste o comportamento do método quando valores negativos forem passados.

10. Crie uma classe Animal com atributos nome e idade. Em seguida, crie uma classe Cachorro que herda de Animal e adicione um método latir. Instancie um objeto da classe Cachorro, defina seus atributos e chame o método latir.

11. Modifique a classe Cachorro para que utilize o construtor da classe Animal para inicializar os atributos herdados. Instancie um Cachorro e exiba suas informações.

12. Adicione um método fazerSom na classe Animal, que exibe "Animal fazendo som". Na classe Cachorro, sobrescreva esse método para exibir "Cachorro latindo". Instancie um Cachorro e chame o método fazerSom para verificar qual mensagem é exibida.

13. Crie outra classe chamada Gato que também herda de Animal e sobrescreva o método fazerSom para exibir "Gato miando". Instancie Cachorro e Gato e armazene-os em um array de Animal. Percorra o array chamando fazerSom para cada elemento, observando o polimorfismo em ação.

14. Crie uma classe abstrata chamada Forma com um método abstrato calcularArea. Crie classes Retangulo e Circulo que herdam de Forma e implementam o método calcularArea. Teste as classes instanciando-as e chamando o método calcularArea.

15. Adicione um atributo cor na classe Forma e um construtor que inicializa cor. Crie um método getCor que retorna a cor da forma. Instancie Retangulo e Circulo, definindo a cor e chamando getCor para verificar o valor.