Java Developer

Java Core

Sumário

Objetivos deste Módulo

Ao final deste módulo, objetiva-se que o aluno adquira:

Conhecimentos:

- Orientação a Objetos: interfaces, classes abstratas, herança, polimorfismo, override

1. Interfaces

Uma interface é uma coleção de métodos abstratos (métodos sem implementação) que uma classe pode implementar. As interfaces permitem definir um conjunto de métodos que uma classe deve fornecer, sem especificar como esses métodos devem ser implementados. Isso é útil para definir contratos ou APIs que outras classes devem seguir.

Aqui estão algumas características e conceitos importantes relacionados a interfaces em Java:

Métodos abstratos: As interfaces contêm apenas métodos abstratos, ou seja, métodos que não têm implementação. Eles são declarados sem um corpo e não contêm código real.

Exemplo de declaração de interface:

```
public interface MinhaInterface {

    void metodo1();

    int metodo2();

}
```

Implementação de interface: Uma classe pode implementar uma ou mais interfaces usando a palavra-chave **implements**. Quando uma classe implementa uma interface, ela deve fornecer implementações para todos os métodos abstratos definidos na interface.

Exemplo de implementação de interface:

```
public class MinhaClasse implements MinhaInterface {

    @Override
    public void metodo1() {
        // Implementação do método 1
    }


    @Override
    public int metodo2() {
        // Implementação do método 2
        return 0;
    }

}
```

Múltiplas interfaces: Uma classe pode implementar várias interfaces, o que permite que ela “herde” o contrato de várias fontes diferentes.

Exemplo de implementação de múltiplas interfaces:

```
public class MinhaClasse implements Interface1, Interface2 {
    // Implementações dos métodos das interfaces
}
```

Polimorfismo com uso de interface: Quando você usa interfaces, pode aproveitar o polimorfismo. Isso significa que você pode tratar objetos de classes que implementam a mesma interface de forma uniforme, independentemente de qual classe concreta eles sejam.

Exemplo de polimorfismo com o uso de interface:

```
MinhaInterface objeto = new MinhaClasse();
objeto.metodo1();
int resultado = objeto.metodo2();
```

Outro exemplo de polimorfismo com o uso de interface:

```
public interface Animal {
    void fazerSom();
}
```

Agora, criaremos duas **classes** que implementam essa interface: **Cachorro** e **Gato**. Cada uma delas fornecerá sua própria implementação do método fazerSom().

```
public class Cachorro implements Animal {

    @Override
    public void fazerSom() {
        System.out.println("O cachorro faz: Au Au!");
    }
}
```

======

```
public class Gato implements Animal {
    @Override
    public void fazerSom() {
        System.out.println("O gato faz: Miau!");
    }
}
```

Agora, podemos criar uma classe principal para demonstrar o polimorfismo:

```
public class Main {

    public static void main(String[] args) {

        Animal animal1 = new Cachorro();
        Animal animal2 = new Gato();

        animal1.fazerSom();
        animal2.fazerSom();

    }
}
```

Neste exemplo, criamos duas variáveis (objetos) do tipo Animal, animal1 e animal2. No entanto, essas variáveis podem apontar para objetos de tipos diferentes que implementam a interface Animal, ou seja, um cachorro e um gato.

Ao chamar o método fazerSom() em cada uma dessas variáveis, o Java usará a implementação apropriada do método, dependendo do tipo real do objeto ao qual a variável se refere. Isso é o que chamamos de polimorfismo.

A saída do programa será:

O cachorro faz: Au Au!
O gato faz: Miau!

Perceba que, mesmo sendo variáveis do tipo Animal, a implementação específica do método fazerSom() é chamada com base no tipo real do objeto que está sendo referenciado. Isso é uma característica fundamental do polimorfismo em Java, que permite tratar objetos de classes diferentes de forma uniforme quando elas compartilham uma interface comum.

Bora praticar?

EXERCÍCIO: Crie uma interface chamada FormaGeometrica que declare dois métodos abstratos:

double calcularArea(): Este método deve calcular e retornar a área da forma geométrica.

double calcularPerimetro(): Este método deve calcular e retornar o perímetro da forma geométrica.

Em seguida, crie duas classes, Circulo e Retangulo, que implementam a interface FormaGeometrica. As classes devem fornecer implementações para os métodos calcularArea e calcularPerimetro de acordo com suas fórmulas específicas de cálculo.

Finalmente, crie uma classe principal chamada Main que demonstre como usar as classes Circulo e Retangulo para calcular áreas e perímetros de formas geométricas diferentes.

Mais algumas características das interfaces em Java:

Constantes em interfaces: Além de métodos abstratos, uma interface pode conter constantes públicas e estáticas. Essas constantes são implicitamente públicas, estáticas (static) e finais (constantes).

Exemplo de constantes em uma interface:

```
public interface MinhaInterface {

    int CONSTANTE1 = 10;
    String CONSTANTE2 = "Algum valor";

}
```

As constantes podem ser acessadas diretamente através do nome da interface, por exemplo, MinhaInterface.CONSTANTE1.

Default Methods: A partir do Java 8, as interfaces podem conter métodos com implementações padrão (default methods). Isso permite adicionar métodos a interfaces existentes sem quebrar as classes que as implementam. As classes que implementam a interface podem optar por substituir o método padrão se desejarem.

Exemplo de um método com implementação padrão:

```
public interface MinhaInterface {

    void metodo1();

    default void metodoPadrao() {
        // Implementação padrão do método
    }
}
```

Leia mais em Métodos Default no Java

Classes que implementam MinhaInterface não são obrigadas a fornecer uma implementação para metodoPadrao, a menos que desejem substituí-lo.

Em resumo, as interfaces em Java são uma parte fundamental da linguagem e são amplamente usadas para definir contratos, estabelecer contratos entre classes e permitir o polimorfismo.

2. Classes abstratas

Uma classe abstrata é uma classe que **não pode ser instanciada** diretamente, ou seja, você não pode criar objetos diretamente a partir dela. Em vez disso, ela serve como uma **base** para outras classes que estendem (herdam) dela. As classes abstratas são declaradas com a palavra-chave **abstract**.

A principal finalidade das classes abstratas é fornecer uma estrutura comum para classes filhas (**subclasses**) que compartilham características em comum, enquanto permitindo que cada classe filha implemente seu próprio comportamento específico.

Aqui estão algumas características e regras importantes relacionadas a classes abstratas em Java:

Não é possível criar objetos a partir de uma classe abstrata. Por exemplo:

```
abstract class Animal {
    // ...
}

Animal animal = new Animal(); // Isso resultaria em erro de compilação.
```

Classes abstratas podem conter **métodos abstratos** (ou seja, métodos que não têm uma implementação) e **métodos concretos** (com implementação).

```
abstract class Animal {

    public abstract void fazerSom(); // Método abstrato

    public void mover() {
        System.out.println("O animal se move."); // Método concreto
    }
}
```

As classes que estendem uma classe abstrata (subclasses) devem implementar todos os métodos abstratos definidos na classe abstrata. Se uma subclasse não implementar todos os métodos abstratos, ela também deve ser declarada como abstrata.

```
class Cachorro extends Animal {

    public void fazerSom() {
        System.out.println("O cachorro faz latidos.");
    }

}
```

Você pode ter construtores em classes abstratas e eles podem ser chamados por construtores das subclasses usando a palavra-chave super.

Vejamos um exemplo completo:

```
// Classe abstrata que representa uma forma geométrica
abstract class Forma {

    // Atributo para armazenar a cor da forma
    private String cor;

    // Construtor da classe abstrata
    public Forma(String cor) {
        this.cor = cor;
    }

    // Método abstrato para calcular a área da forma
    public abstract double calcularArea();

    // Método concreto para exibir informações sobre a forma
    public void mostrarInformacoes() {
        System.out.println("Esta é uma forma de cor " + cor);
    }
}

=====

// Classe concreta que representa um círculo
class Circulo extends Forma {

    private double raio;

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

    // Implementação do método abstrato para calcular a área do círculo
    @Override
    public double calcularArea() {
        return Math.PI * raio * raio;
    }
}

=====

// Classe concreta que representa um retÇangulo
public class Retangulo extends Forma {

    private double base;
    private double altura;

    public Retangulo(String cor, double base, double altura) {
        super(cor);
        this.base = base;
        this.altura = altura;
    }
```

```
    @Override
    public double calcularArea() {
        return base * altura;
    }
}

=====

public class Main {

    public static void main(String[] args) {

        Forma circulo = new Circulo("Vermelho", 5.0);

        Forma retangulo = new Retangulo("Azul", 4.0, 6.0);

        // Chama o método calcularArea() para calcular a área das formas
        double areaCirculo = circulo.calcularArea();
        double areaRetangulo = retangulo.calcularArea();

        // Chama o método mostrarInformacoes() para exibir informações sobre as formas
        circulo.mostrarInformacoes();
        System.out.println("Área do círculo: " + areaCirculo);

        retangulo.mostrarInformacoes();
        System.out.println("Área do retângulo: " + areaRetangulo);
    }
}
```

As classes abstratas são úteis para criar hierarquias de classes onde você deseja fornecer um conjunto de métodos e atributos comuns, mas deixar a implementação específica para as subclasses. Isso promove a reutilização de código e ajuda a organizar o código de maneira mais eficaz.

Bora praticar?

EXERCÍCIO 1: Crie uma hierarquia de classes de animais usando classes abstratas em Java.
Criar uma classe abstrata chamada **Animal** com os seguintes atributos e métodos:

Atributos:

nome (uma String que representa o nome do animal)
idade (um inteiro que representa a idade do animal)

Métodos abstratos:

emitirSom() - Esse método deve ser implementado nas subclasses para que cada tipo de animal emita seu som característico.

Em seguida, crie pelo menos três classes concretas que estendem a classe abstrata Animal para representar diferentes tipos de animais (por exemplo, **Cachorro**, **Gato** e **Passaro**).

Implemente o método emitirSom() em cada uma dessas classes para que cada animal emita um

som diferente.

Finalmente, crie uma classe de teste chamada TesteAnimais que cria instâncias dos diferentes tipos de animais, atribui valores aos seus atributos e chama o método emitirSom() para cada um deles. Se preferir, crie testes unitários.

EXERCÍCIO 2: Modelando Produtos em um E-commerce

Vamos criar uma hierarquia de classes para modelar produtos em um sistema de e-commerce. Crie uma classe abstrata chamada **Produto** com os seguintes atributos e métodos:

Atributos:

nome (uma String que representa o nome do produto)
preco (um valor em ponto flutuante que representa o preço do produto)

Métodos abstratos:

calcularDesconto() - Esse método abstrato deve calcular o preço com o desconto aplicado com base em um percentual fornecido.

Agora, crie duas subclasses concretas, **Livro** e **Eletronico**, que estendem a classe Produto. Cada uma dessas subclasses deve implementar o método abstrato calcularDesconto() de acordo com suas regras específicas.

Livro deve ter um atributo adicional chamado autor (uma String que representa o autor do livro). O desconto em livros é de 10%.

Eletronico deve ter um atributo adicional chamado marca (uma String que representa a marca do dispositivo eletrônico). O desconto em dispositivos eletrônicos é de 5%.

Por fim, crie uma classe de teste chamada TesteEcommerce que cria instâncias de Livro e Eletronico, atribui valores aos seus atributos e chama o método calcularDesconto() para calcular o preço com desconto para cada um deles. Se preferir, crie testes unitários.

Então, qual a diferença entre classe abstrata e interface?

As interfaces e as classes abstratas são duas ferramentas poderosas na programação orientada a objetos em Java (e em muitas outras linguagens de programação), mas têm finalidades diferentes e características distintas:

Classes Abstratas:

- Propósito Principal: Uma classe abstrata é usada quando você deseja fornecer uma base comum para várias classes relacionadas. Ela pode conter métodos abstratos (métodos sem implementação) e métodos concretos (métodos com implementação) que podem ser compartilhados pelas subclasses.

- Construtores: Pode ter construtores, que são chamados quando uma instância de suas subclasses é criada.

- Herança: Pode ser usada para estabelecer uma hierarquia de classes. Uma classe abstrata pode ser estendida por outras classes.

- Múltipla Herança: Uma classe abstrata pode estender apenas uma classe, evitando assim o problema da ambiguidade na múltipla herança.

- Finalidade da Classe Abstrata: Geralmente, classes abstratas são usadas para modelar conceitos gerais e fornecer uma estrutura comum para classes relacionadas. Elas permitem compartilhar código comum e definir contratos para subclasses.

Interfaces:

- Propósito Principal: Uma interface é usada quando você deseja definir um contrato que várias classes independentes devem cumprir. Ela consiste em declarações de métodos, mas não contém implementações de métodos. A partir do Java 8, interfaces podem conter métodos com implementação padrão (default methods) que têm uma implementação padrão, mas ainda podem ser sobrescritos pelas classes que implementam a interface.

- Construtores: Não pode ter construtores, pois não é possível criar instâncias de interfaces.

- Herança: Uma classe pode implementar várias interfaces, permitindo a herança múltipla de comportamentos.

- Finalidade da Interface: As interfaces são usadas para definir contratos que as classes devem seguir. Elas permitem que classes independentes implementem funcionalidades específicas.

Em resumo, enquanto as classes abstratas são usadas para compartilhar comportamentos comuns e criar hierarquias de classes, as interfaces são usadas para definir contratos que as classes independentes devem seguir. Em Java, você pode usar ambas as classes abstratas e interfaces de acordo com as necessidades do seu projeto, muitas vezes combinando-as para criar uma estrutura de código flexível e modular.

3. Herança

Já utilizamos herança quando estudamos classes abstratas, mas vamos ver agora alguns conceitos importantes para melhor compreensão…

A herança é um conceito fundamental na programação orientada a objetos (POO) e em muitas linguagens de programação, como Java, Python, C++, entre outras. Ela permite que você crie novas classes com base em classes existentes, aproveitando os atributos e comportamentos da classe existente (conhecida como classe base ou superclasse) para criar novas classes (conhecidas como subclasses ou classes derivadas).

Aqui estão alguns conceitos-chave relacionados à herança:

Superclasse (Classe Base): É a classe original que contém atributos e métodos que podem ser compartilhados por uma ou mais subclasses. A superclasse é a classe de onde outras classes herdam atributos e métodos.

Subclasse (Classe Derivada): É a classe que herda atributos e métodos da superclasse. A subclasse pode adicionar novos atributos e métodos ou modificar os existentes, mas ela herda a estrutura básica da superclasse.

Polimorfismo: A herança permite a aplicação do polimorfismo, que é a capacidade de objetos de subclasses serem tratados como objetos da superclasse. Isso permite que você escreva código mais genérico que pode lidar com diferentes tipos de objetos de maneira eficaz.

Sobrescrita (Override): As subclasses podem substituir (sobrescrever) os métodos herdados da superclasse, fornecendo sua própria implementação do método. Isso permite que as subclasses personalizem o comportamento dos métodos herdados.

Exemplo de herança em Java:

```
// Classe base (superclasse) - Veiculo
class Veiculo {

    private String marca;
    private String modelo;
    private int ano;

    public Veiculo(String marca, String modelo, int ano) {
        this.marca = marca;
        this.modelo = modelo;
        this.ano = ano;
    }

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

    public void frear() {
        System.out.println("O veículo está freando.");
    }

    public void exibirInformacoes() {
        System.out.println("Marca: " + marca);
        System.out.println("Modelo: " + modelo);
        System.out.println("Ano: " + ano);
    }
}

====

// Subclasse - Carro
class Carro extends Veiculo {

    private int numeroPortas;

    public Carro(String marca, String modelo, int ano, int numeroPortas) {
        super(marca, modelo, ano); // Chamando o construtor da superclasse
        this.numeroPortas = numeroPortas;
    }

    public void abrirPortas() {
        System.out.println("Abrindo as " + numeroPortas + " portas do carro.");
    }
}

====

// Subclasse - Moto
class Moto extends Veiculo {
```

```
    private boolean temPartidaEletrica;

    public Moto(String marca, String modelo, int ano, boolean temPartidaEletrica) {
        super(marca, modelo, ano);
        this.temPartidaEletrica = temPartidaEletrica;
    }

    public void ligarPartidaEletrica() {
        if (temPartidaEletrica) {
            System.out.println("A moto está ligando a partida elétrica.");
        } else {
            System.out.println("A moto não possui partida elétrica.");
        }
    }
}
```

====

```
public class ExemploHeranca {
    public static void main(String[] args) {
        Carro meuCarro = new Carro("Toyota", "Corolla", 2022, 4);
        Moto minhaMoto = new Moto("Honda", "CBR 600", 2021, true);

        meuCarro.exibirInformacoes();
        meuCarro.abrirPortas();
        meuCarro.acelerar();
        meuCarro.frear();

        System.out.println();

        minhaMoto.exibirInformacoes();
        minhaMoto.ligarPartidaEletrica();
        minhaMoto.acelerar();
        minhaMoto.frear();
    }
}
```

A herança é um conceito poderoso na programação orientada a objetos que ajuda a promover a reutilização de código, a hierarquia de classes e a criação de relacionamentos entre objetos de diferentes tipos. No entanto, é importante usá-la com cuidado, uma vez que um design de classe inadequado pode levar a problemas de complexidade e acoplamento excessivo.