COM230 - Programação Orientada a Objetos
Prof. Dr. Marcelo G. Manzato

Material de Revisão

1. Introdução

A Programação Orientada a Objetos (POO) é um paradigma de programação que utiliza objetos para organizar e estruturar o código. Java é uma linguagem de programação amplamente utilizada que segue os princípios da POO de maneira rigorosa. Nesta revisão, exploraremos os principais conceitos da POO em Java, destacando sua sintaxe e fornecendo exemplos práticos.

2. Classe e Objeto

Em Java, uma classe é um modelo para criar objetos. Objetos, por sua vez, são instâncias de classes. Por exemplo, imagine uma classe Carro:

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

    public Carro(String modelo, int ano) {
        this.modelo = modelo;
        this.ano = ano;
    }
}
```

Aqui, Carro é a classe, e você pode criar objetos dessa classe, como:

```
Carro meuCarro = new Carro("Sedan", 2023);
```

Dessa forma, classe é uma estrutura abstrata que define atributos e métodos comuns a um conjunto de objetos relacionados. Já objeto é uma instância específica de uma classe, com valores particulares para seus atributos, mas seguindo a estrutura definida pela classe.

A distinção entre classe e objeto é fundamental na POO, pois permite a criação de código modular e reutilizável. Classes fornecem a estrutura, e objetos representam instâncias concretas dessa estrutura no mundo real.

3. Encapsulamento

Encapsulamento é o conceito de esconder detalhes internos de um objeto e expor apenas o necessário. Em Java, isso é frequentemente alcançado usando modificadores de acesso, como public, private e protected. Por exemplo:

```
public class ContaBancaria {
    private double saldo;

    public void depositar(double valor) {
        if (valor > 0) {
            saldo += valor;
        }
    }

    public double getSaldo() {
        return saldo;
    }
}
```

Aqui, o saldo é encapsulado e só pode ser acessado através dos métodos públicos depositar e getSaldo.

Em Java, há três modificadores de acesso:

- Public (public): Membros marcados como public são acessíveis de qualquer lugar. Eles formam a interface pública da classe.
- Private (private): Membros marcados como private são acessíveis apenas dentro da própria classe. Eles são ocultos do mundo externo.
- Protected (protected): Membros marcados como protected são acessíveis dentro da própria classe e por subclasses.

O encapsulamento não é apenas uma questão de ocultar membros, mas também de fornecer uma interface clara e controlada para interagir com os objetos. Ao aplicar o encapsulamento, promovemos a modularidade, facilitamos a manutenção e protegemos a integridade dos dados em nossos programas. É uma prática essencial na construção de sistemas robustos e escaláveis em Java e outros ambientes de POO.

4. Herança

Herança permite que uma classe herde características de outra. Por exemplo:

```
public class Animal {
    public void comer() {
        System.out.println("Animal comendo...");
    }
}
```

```
public class Cachorro extends Animal {
    public void latir() {
        System.out.println("Au au!");
    }
}
```

A classe Cachorro herda de Animal, obtendo o método comer, e possui seu próprio método latir.

Herança pode formar uma hierarquia de classes. Por exemplo, além de Animal, poderíamos ter outras subclasses como Gato, Pato etc., todas herdando de Animal. Isso cria uma hierarquia onde as classes mais específicas herdam comportamentos mais genéricos.

Como benefícios de se utilizar herança, podemos citar:

- Reutilização de Código: Evita a duplicação de código ao permitir que classes herdem comportamentos de outras classes.
- Extensibilidade: Permite estender ou modificar comportamentos existentes sem alterar a classe original.
- Organização Hierárquica: Facilita a organização de classes em uma hierarquia que reflete a relação entre os diferentes tipos de objetos.

Utilizada com moderação e compreensão, a herança pode melhorar a organização do código e promover a reutilização de código, contribuindo para sistemas mais flexíveis e expansíveis em Java.

5. Polimorfismo

De modo geral, polimorfismo refere-se à capacidade de um objeto assumir diversas formas. Isso permite que um mesmo método seja usado para realizar diferentes ações, dependendo do contexto em que é chamado. Existem dois tipos principais de polimorfismo em Java: polimorfismo de sobrecarga e polimorfismo de sobreposição.

5.1 Polimorfismo de Sobrecarga

Polimorfismo de sobrecarga ocorre quando uma classe tem dois ou mais métodos com o mesmo nome, mas com diferentes parâmetros. O compilador e o interpretador Java decidem qual método chamar com base nos tipos e números de argumentos passados. Por exemplo:

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

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

```
    }
}
```

Aqui, a classe Calculadora tem dois métodos chamados somar, um para inteiros e outro para números de ponto flutuante. O Java decidirá qual chamar com base nos argumentos fornecidos.

5.2 Polimorfismo de Sobreposição (Override)

Polimorfismo de sobreposição ocorre quando uma subclasse fornece uma implementação específica para um método que já está definido em sua superclasse. Isso permite que objetos da subclasse sejam usados no lugar de objetos da superclasse sem saber a diferença. Por exemplo:

```
public class Animal {
    public void fazerSom() {
        System.out.println("Animal fazendo som...");
    }
}

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

Aqui, Cachorro está sobrepondo o método fazerSom definido na classe Animal. Agora, um objeto Cachorro pode ser tratado como um Animal e, ainda assim, chamará seu próprio método fazerSom.

Na prática, podemos usar polimorfismo assim:

```
Animal meuAnimal = new Cachorro();
meuAnimal.fazerSom();  // Chama o método fazerSom da classe Cachorro
```

Mesmo que meuAnimal seja declarado como tipo Animal, ele pode executar o método específico da classe Cachorro devido ao polimorfismo de sobreposição.

6. Interfaces

Interfaces em Java são estruturas que definem um conjunto de métodos que uma classe deve implementar. Elas oferecem uma forma de criar contratos, especificando quais comportamentos uma classe deve fornecer, sem se preocupar com os detalhes de como esses comportamentos são implementados. Interfaces são totalmente abstratas, o que significa que não contêm implementações concretas de métodos.

Para declarar uma interface, fazemos:

```
public interface Forma {
    double calcularArea();
    void desenhar();
}
```

Aqui, Forma é uma interface que declara dois métodos, calcularArea e desenhar. Qualquer classe que implementa essa interface deve fornecer implementações para ambos os métodos. Por exemplo:

```
public class Circulo implements Forma {
    private double raio;

    public Circulo(double raio) {
        this.raio = raio;
    }

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

    public void desenhar() {
        System.out.println("Desenhando um círculo...");
    }
}
```

A classe Circulo implementa a interface Forma e fornece implementações para os métodos calcularArea e desenhar. Isso estabelece um contrato entre a interface e a classe.

Apesar de em Java não haver herança múltipla, uma classe pode implementar várias interfaces, permitindo que ela forneça comportamentos de várias fontes. Por exemplo:

```
public class Retangulo implements Forma, Colorivel {
    // Implementações para os métodos de Forma e Colorivel
}
```

Interfaces podem ser usadas como tipos, o que significa que você pode criar referências do tipo de uma interface e atribuir a elas objetos de classes que a implementam. Isso facilita a criação de código mais flexível e genérico.

```
Forma minhaForma = new Circulo(5.0);
minhaForma.desenhar(); // Chama o método desenhar da classe Circulo
```

A utilização de interfaces pode trazer alguns benefícios, como:

- Contratos Claros: Definem contratos claros entre as classes, promovendo uma separação clara entre a especificação e a implementação.
- Múltiplas Implementações: Permite que uma classe forneça implementações para vários comportamentos sem herdar de várias classes.
- Flexibilidade: Facilita a criação de código flexível e extensível, pois uma classe pode implementar várias interfaces.

7. Conclusão

A Programação Orientada a Objetos em Java oferece uma abordagem poderosa e flexível para desenvolver *software*. Este resumo abordou alguns dos principais conceitos, mas há muito mais para explorar. Ao compreender esses fundamentos, os programadores podem criar sistemas mais modularizados, reutilizáveis e fáceis de manter. A prática contínua e a exploração de projetos mais complexos são fundamentais para aprofundar a compreensão desses conceitos e aplicá-los de forma eficaz.