Orientação a Objeto - Material de Apoio.

Os paradigmas:

- Procedural:
  - Neste paradigma, o programa é estruturado em torno de procedimentos (ou funções) que contêm um conjunto de instruções para serem executadas sequencialmente. Exemplos de linguagens que seguem esse paradigma são C e Pascal.
- **Orientado a Objetos**:
  - *Neste paradigma, o programa é estruturado em torno de objetos, que são instâncias de classes. Os objetos têm propriedades (atributos) e comportamentos (métodos), e a interação entre objetos é fundamental. Exemplos de linguagens que seguem esse paradigma são Java, C++, Python e Ruby.*
- Funcional:
  - Neste paradigma, o foco está na avaliação de funções matemáticas e na imutabilidade dos dados. As funções são tratadas como valores e podem ser passadas como argumentos e retornadas como resultados. Exemplos de linguagens que seguem esse paradigma são Haskell, Lisp e Erlang.
- Lógico:
  - Neste paradigma, a lógica formal é usada para resolver problemas. Os programas são construídos em torno de fatos e regras lógicas, e a inferência lógica é usada para deduzir conclusões a partir dessas informações. Exemplos de linguagens que seguem esse paradigma são Prolog e Datalog.
- Estruturado:
  - Neste paradigma, o programa é estruturado em torno de estruturas de controle como sequência, seleção (if-else) e repetição (loops). A ideia é dividir o programa em blocos lógicos e organizados. A maioria das linguagens de programação suporta o paradigma estruturado, incluindo C, Pascal e Java.

Estruturado x Orientado a Objetos:

```c
#include <stdio.h>

// Função para calcular a área do quadrado
int calcularAreaQuadrado(int lado) {
    return lado * lado;
}

// Função para calcular a área do retângulo
int calcularAreaRetangulo(int base, int altura) {
    return base * altura;
}
```

```c
int main() {
    int lado = 4;
    int base = 6;
    int altura = 8;

    int areaQuadrado = calcularAreaQuadrado(lado);
    int areaRetangulo = calcularAreaRetangulo(base, altura);

    printf("Área do quadrado: %d\n", areaQuadrado);
    printf("Área do retângulo: %d\n", areaRetangulo);

    return 0;
}
```

```java
class Quadrado {
    private int lado;

    public Quadrado(int lado) {
        this.lado = lado;
    }

    public int calcularArea() {
        return lado * lado;
    }
}

class Retangulo {
    private int base;
    private int altura;

    public Retangulo(int base, int altura) {
        this.base = base;
        this.altura = altura;
    }

    public int calcularArea() {
        return base * altura;
    }
}

public class AreaFormas {
    public static void main(String[] args) {
        int lado = 4;
        Quadrado quadrado = new Quadrado(lado);

        int base = 6;
        int altura = 8;
        Retangulo retangulo = new Retangulo(base, altura);

        int areaQuadrado = quadrado.calcularArea();
        int areaRetangulo = retangulo.calcularArea();

        System.out.println("Área do quadrado: " + areaQuadrado);
        System.out.println("Área do retângulo: " + areaRetangulo);
```

```
    }
}
```

Struct x Classes

```
struct Cliente { char nome[50]; int idade; };
```

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

Diferenças:

- Encapsulamento e visibilidade:
  - Em uma **classe**, os membros (atributos e métodos) podem ser definidos com diferentes níveis de acesso, como `public`, `private`, `protected`, que controlam a visibilidade desses membros.
  - Em uma **struct**, todos os membros são públicos por padrão, o que significa que eles são acessíveis diretamente por outros códigos.

- Métodos e Comportamentos:
  - Em uma **classe**, você pode definir métodos para manipular os atributos e fornecer comportamentos específicos.
  - Em uma **struct**, geralmente não são definidos métodos, mas apenas atributos para armazenar dados.
- Herança:
  - As **classes** suportam `herança`, permitindo que você crie hierarquias de classes com base em relações de especialização.
  - **struct** em C não suporta `herança`.
- Construtores e Destrutores:
  - Em uma **classe**, você pode definir construtores e destrutores para inicializar e liberar recursos, respectivamente.
  - Em uma **struct**, em C, você não pode definir explicitamente construtores e destrutores.
- Orientação a Objetos:
  - As classes são o principal bloco de construção da programação orientada a objetos (POO), permitindo encapsulamento, herança e polimorfismo.
  - **struct**, em C é usado principalmente para agrupar dados relacionados, mas não

possui os recursos de POO.

No geral, as `classes` são mais flexíveis e poderosas em termos de recursos de programação orientada a objetos, como encapsulamento, herança e polimorfismo. As `structs` em C, por outro lado, são mais simples, fornecendo uma maneira de agrupar dados relacionados sem os recursos avançados da orientação a objetos.

Os 4 pilares da Orientação a Objetos:

1. Encapsulamento:
2. Herança:
3. Polimorfismo:
4. Abstração:

Encapsulamento:

Encapsulamento é o princípio de esconder os detalhes internos de um objeto e fornecer uma interface controlada para interagir com ele.
Por conta dessa técnica, o conhecimento a respeito da implementação interna da classe é desnecessário do ponto de vista do objeto, uma vez que isso passa a ser responsabilidade dos métodos internos da classe.

Antes de continuarmos...

Visibilidades:

No Java, existem quatro tipos de visibilidade que podem ser aplicados a classes, atributos, métodos e construtores. São eles:

1. `public`: A visibilidade `public` torna o elemento acessível a partir de qualquer lugar, ou seja, ele pode ser acessado por qualquer classe ou código dentro ou fora do pacote onde está definido.
2. `private`: A visibilidade `private` restringe o acesso ao elemento somente dentro da própria classe onde está definido. Ou seja, ele não pode ser acessado por outras classes ou códigos externos.
3. `protected`: A visibilidade `protected` permite o acesso ao elemento dentro da própria classe, subclasses (herdeiras) e classes do mesmo pacote. No entanto, ele não pode ser acessado por classes fora do pacote, a menos que sejam subclasses da classe onde o elemento é definido.
4. Sem modificador (padrão/package-private): Quando nenhum modificador é especificado, o elemento possui visibilidade de nível de pacote. Isso significa que ele pode ser acessado apenas por classes dentro do mesmo pacote. Fora do pacote, o elemento não é visível.

Continuando encapsulamento...

Vamos considerar um exemplo em Java, onde não usar encapsulamento pode levar a riscos.

```
public class BankAccount {
    public double balance; // Atributo público para o saldo

    public void withdraw(double amount) {
        // Lógica de saque
        balance -= amount;
    }

    public void deposit(double amount) {
        // Lógica de depósito
        balance += amount;
    }
}
```

Neste exemplo, o atributo `balance` é público, o que significa que outros códigos podem acessá-lo diretamente e modificá-lo. Isso pode levar a riscos, pois qualquer código externo pode manipular diretamente o saldo da conta, sem passar por validações ou lógicas específicas.

```
public class ExternalCode {
    public static void main(String[] args) {
        BankAccount account = new BankAccount();
        account.balance = 1000000; // Acesso direto ao atributo balance

        // ... Outras operações com o saldo ...
    }
}
```

Com o encapsulamento, podemos corrigir esse problema:

```
public class BankAccount {
    private double balance; // Atributo privado para o saldo

    public void withdraw(double amount) {
        // Lógica de saque
        balance -= amount;
    }

    public void deposit(double amount) {
        // Lógica de depósito
        balance += amount;
    }

    public double getBalance() {
        return balance;
    }
}
```

Herança:

Imagine se você tivesse herdado uma fortuna de 100 milhões de reais dos seus pais, talvez você não estivesse lendo este artigo agora. No entanto, como essa situação não ocorreu, vamos continuar nossa leitura.

"*Herança é um princípio de orientação a objetos, que permite que classes compartilhem atributos e métodos, através de "heranças". Ela é usada na intenção de reaproveitar código ou comportamento generalizado ou especializar operações ou atributos.*"
Wikipédia

Exemplo em que não se utiliza herança:

```
// Classe "Pato"
class Pato {
    private String cor;
    private int tamanho;

    public Pato(String cor, int tamanho) {
        this.cor = cor;
        this.tamanho = tamanho;
    }

    public void fazerBarulho() {
        System.out.println("Quack!");
    }

    public void exibirCor() {
        System.out.println("Cor do pato: " + cor);
    }

    public void exibirTamanho() {
        System.out.println("Tamanho do pato: " + tamanho);
    }
}

// Classe "PatoDeBorracha"
class PatoDeBorracha {
    private String cor; // linha de código existente em Pato
    private int tamanho;// linha de código existente em Pato
    private boolean boia;

    public PatoDeBorracha(String cor, int tamanho, boolean boia) {
        this.cor = cor;
        this.tamanho = tamanho;
        this.boia = boia;
    }

    public void fazerBarulho() {
        System.out.println("Squeak!");
    }

    public void exibirCor() {
        System.out.println("Cor do pato de borracha: " + cor);
```

```
    }

    public void exibirTamanho() {
        System.out.println("Tamanho do pato de borracha: " + tamanho);
    }

    public void flutuar() {
        if (boia) {
            System.out.println("O pato de borracha está flutuando.");
        } else {
            System.out.println("O pato de borracha não flutua.");
        }
    }
}

// Código principal
public class Main {
    public static void main(String[] args) {
        Pato pato = new Pato("Amarelo", 10);
        pato.fazerBarulho();
        pato.exibirCor();
        pato.exibirTamanho();

        PatoDeBorracha patoBorracha = new PatoDeBorracha("Vermelho", 5,
true);
        patoBorracha.fazerBarulho();
        patoBorracha.exibirCor();
        patoBorracha.exibirTamanho();
        patoBorracha.flutuar();
    }
}
```

Ao não utilizar herança em um contexto adequado, você pode enfrentar alguns problemas, como:

1. `Duplicação de código`: Sem a **herança**, você pode acabar replicando código em várias classes que possuem funcionalidades semelhantes. Isso leva a uma manutenção mais difícil, pois qualquer modificação precisa ser feita em várias partes do código.
2. `Falta de reutilização`: A **herança** permite que você compartilhe comportamentos e atributos comuns entre classes relacionadas. Sem ela, você perde a oportunidade de reutilizar e compartilhar código, o que pode resultar em uma base de código maior e menos eficiente.
3. `Dificuldade na extensibilidade`: Sem a **herança**, é mais difícil estender e adicionar novos comportamentos às classes existentes. Isso pode levar a um código mais rígido e menos flexível para futuras alterações ou adições de funcionalidades.
4. `Complexidade desnecessária`: Sem **herança**, você pode precisar criar estruturas e lógicas mais complexas para lidar com classes relacionadas. A herança permite uma modelagem mais simples e intuitiva das relações entre objetos.

Utilizando herança:

```
// Classe "Pato"
class Pato {
    private String cor;
    private int tamanho;

    public Pato(String cor, int tamanho) {
        this.cor = cor;
        this.tamanho = tamanho;
    }

    public void fazerBarulho() {
        System.out.println("Quack!");
    }

    public void exibirCor() {
        System.out.println("Cor do pato: " + cor);
    }

    public void exibirTamanho() {
        System.out.println("Tamanho do pato: " + tamanho);
    }
}

// Classe "PatoDeBorracha" herda da classe "Pato"
class PatoDeBorracha extends Pato {
    private boolean boia;

    public PatoDeBorracha(String cor, int tamanho, boolean boia) {
        super(cor, tamanho);
        this.boia = boia;
    }

    @Override
    public void fazerBarulho() {
        System.out.println("Squeak!");
    }

    public void flutuar() {
        if (boia) {
            System.out.println("O pato de borracha está flutuando.");
        } else {
            System.out.println("O pato de borracha não flutua.");
        }
    }
}

// Código principal
public class Main {
    public static void main(String[] args) {
        Pato pato = new Pato("Amarelo", 10);
        pato.fazerBarulho();
        pato.exibirCor();
        pato.exibirTamanho();

        PatoDeBorracha patoBorracha = new PatoDeBorracha("Vermelho", 5,
true);
        patoBorracha.fazerBarulho();
```

```
        patoBorracha.exibirCor();
        patoBorracha.exibirTamanho();
        patoBorracha.flutuar();
    }
}
```

Neste exemplo, a classe "PatoDeBorracha" herda da classe "Pato" utilizando a palavra-chave `extends`. A classe filha herda os atributos (cor e tamanho) e métodos (fazerBarulho, exibirCor, exibirTamanho) da classe pai. Além disso, a classe "PatoDeBorracha" adiciona um novo método `flutuar()` específico para patos de borracha.

Polimorfismo:

"**Definimos Polimorfismo como um princípio a partir do qual as classes derivadas de uma única classe base são capazes de invocar os métodos que, embora apresentem a mesma assinatura, comportam-se de maneira diferente para cada uma das classes derivadas**." Fonte. https://www.devmedia.com.br/

Exemplo em java:

```
// Polimorfismo
// Classe base Animal
class Animal {
    public void fazerBarulho() {
        System.out.println("O animal faz algum barulho.");
    }
}

// Classe derivada Cachorro
class Cachorro extends Animal {
    @Override
    public void fazerBarulho() {
        System.out.println("O cachorro late.");
    }
}

// Classe derivada Gato
class Gato extends Animal {
    @Override
    public void fazerBarulho() {
        System.out.println("O gato mia.");
    }
}

// Classe principal
public class PolimorfismoExemplo {
    public static void main(String[] args) {
        Animal animal1 = new Cachorro(); // Criando um objeto Cachorro e
atribuindo a uma variável Animal
        Animal animal2 = new Gato();     // Criando um objeto Gato e
atribuindo a uma variável Animal

        animal1.fazerBarulho();  // Chama o método fazerBarulho() do
```

```
objeto Cachorro
        animal2.fazerBarulho();  // Chama o método fazerBarulho() do
objeto Gato
    }
}
```

Sobre assinatura de método:

A assinatura de um método em Java é composta pelo nome do método e pelos tipos, ordem e quantidade de seus parâmetros. A assinatura é usada para identificar exclusivamente um método em uma classe, permitindo diferenciar entre métodos com o mesmo nome, mas com parâmetros diferentes.

A assinatura de um método não inclui o tipo de retorno nem os modificadores de acesso, apenas o nome do método e os parâmetros. Isso significa que dois métodos com o mesmo nome, a mesma lista de parâmetros, mas com tipos de retorno diferentes não podem coexistir em uma mesma classe.

Por exemplo, considere os seguintes métodos em uma classe chamada `Calculadora`:

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

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

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

Com base na assinatura, o compilador Java é capaz de identificar qual método deve ser chamado com base nos argumentos passados. Por exemplo:

```
Calculadora calculadora = new Calculadora();

int resultado1 = calculadora.somar(2, 3);              // Chama o primeiro
método somar(int, int)
double resultado2 = calculadora.somar(4.5, 2.7);       // Chama o segundo
método somar(double, double)
String resultado3 = calculadora.somar("Olá, ", "mundo!");  // Chama o
terceiro método somar(String, String)
```

Exemplo de **mal uso** do polimorfismo em POO pode ocorrer quando uma classe derivada altera o comportamento de um método da classe base de maneira que quebra a funcionalidade esperada.

Considere a seguinte hierarquia de classes relacionadas a formas geométricas:

```
class Forma {
    public void calcularArea() {
        System.out.println("Área da forma genérica.");
    }
}

class Retangulo extends Forma {
    private int altura;
    private int largura;

    public Retangulo(int altura, int largura) {
        this.altura = altura;
        this.largura = largura;
    }

    public void calcularArea() {
        int area = altura * largura;
        System.out.println("Área do retângulo: " + area);
    }
}

class Circulo extends Forma {
    private double raio;

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

    public void calcularArea() {
        double area = Math.PI * raio * raio;
        System.out.println("Área do círculo: " + area);
    }
}

class FormaHandler {
    public void exibirArea(Forma forma) {
        forma.calcularArea();
    }
}
```

```
class Triangulo extends Forma {
    private double base;
    private double altura;

    public Triangulo(double base, double altura) {
        this.base = base;
        this.altura = altura;
    }

    // Esqueceu de sobrescrever o método calcularArea()
}

// Uso incorreto do polimorfismo
FormaHandler handler = new FormaHandler();
```

```
Triangulo triangulo = new Triangulo(5, 8);
handler.exibirArea(triangulo);
```

Um outro exemplo de mal uso do polimorfismo em POO pode ser a violação do princípio de substituição de **Liskov**. Esse princípio estabelece que os objetos de uma classe derivada devem poder ser usados no lugar dos objetos da classe base, sem causar efeitos colaterais indesejados ou comportamentos inconsistentes.

Não confunda sobrecarga e polimorfismo:

Não, sobrecarga e polimorfismo são conceitos distintos na Programação Orientada a Objetos.
Existe Polimorfismo de sobreposição e Polimorfismo de sobrecarga.

A **sobrecarga** (*overloading*) ocorre quando uma classe possui vários métodos com o mesmo nome, mas com diferentes listas de parâmetros. A ideia é que esses métodos realizem operações semelhantes, mas com diferentes tipos de entrada ou quantidades de parâmetros. Durante a compilação ou tempo de execução, o compilador ou a máquina virtual Java determinam qual método deve ser chamado com base nos argumentos fornecidos. A **sobrecarga** permite que um método seja flexível e aceite diferentes combinações de parâmetros.

Vários construtores, é considerado sobrecarga?

Não, quando você tem vários construtores dentro da mesma classe, isso não é considerado sobrecarga de métodos. A sobrecarga ocorre quando uma classe tem vários métodos com o mesmo nome, mas com parâmetros diferentes.

Por exemplo, considere a classe `Calculadora` com dois métodos `soma` sobrecarregados:

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

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

Por outro lado, o **polimorfismo** (*polymorphism*) refere-se à capacidade de objetos de diferentes classes serem tratados de maneira uniforme, por meio do uso de uma classe base ou interface comum. O polimorfismo permite que um objeto possa ser referenciado por meio de um tipo genérico e, mesmo assim, executar o comportamento específico da classe concreta à qual pertence. Isso ocorre quando as classes derivadas substituem métodos da classe base para fornecer uma implementação especializada.

Em resumo, sobrecarga refere-se a ter vários métodos com o mesmo nome, mas com diferentes listas de parâmetros, enquanto o polimorfismo é a capacidade de objetos de diferentes classes serem tratados de forma uniforme e executarem comportamentos específicos de suas classes concretas.

Abstração:

O conceito de abstração consiste em esconder os detalhes de algo, no caso, os detalhes desnecessários.

No mundo real, utilizamos abstrações o tempo todo. Tudo que não sabemos como funciona por baixo dos panos pode ser considerado uma abstração.
Referência

A abstração nos permite focar nos aspectos essenciais de um objeto, ignorando os detalhes desnecessários. Podemos pensar nisso como uma representação simplificada de um objeto, que contém apenas as informações e comportamentos mais relevantes para o nosso programa. *chatGPT*

Por exemplo, vamos considerar um programa que simula um zoológico. Podemos ter diferentes tipos de animais, como leões, tigres e ursos. Cada animal tem características específicas, como nome, idade, tamanho e habilidades. Em vez de nos preocuparmos com todos os detalhes de cada animal, podemos abstrair essas informações para criar uma classe Animal genérica. *chatGPT*

Segue exemplos em código:

```
public abstract class Animal {
    private String nome;
    private int idade;

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

    public abstract void comer();

    public abstract void dormir();

    public String getNome() {
        return nome;
    }

    public int getIdade() {
        return idade;
    }
}
```

```
public class Leao extends Animal {

    public Leao(String nome, int idade) {
        super(nome, idade);
    }

    @Override
    public void comer() {
        System.out.println("O leão está comendo.");
    }

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

Posso considerar interface uma abstração?

Sim, a interface pode ser considerada uma forma de abstração na programação orientada a objetos.

Em orientação a objetos, uma interface define um conjunto de métodos que uma classe deve implementar. Ela descreve os comportamentos que uma classe concreta deve ter, mas não especifica como esses comportamentos devem ser implementados. Isso permite separar a definição dos comportamentos da sua implementação concreta.

As interfaces funcionam como contratos que definem o que as implementações (Classes) devem conter.
Em suma as interfaces dizem "O que" e não "Como", sendo assim, podemos tomar o "Como" como os detalhes e o "O que" como uma abstração.
Referência

```
public interface ICustomerRepository {
    void save(Customer customer);
}
```

Tomando como base a interface acima, tomamos ela como um contrato que diz que um `Customer` pode ser salvo, mas ela não diz **como** isto deve ser feito.

Este conceito de abstração por interfaces nos leva ao princípio DIP (Princípio da Inversão de Dependência) que prega o seguinte:

Sempre que puder, dependa de abstrações ao invés de implementações.

balta.io

Thiago André Cardoso Silva
twitter: @programador_who
instagram: @thi.dev.who