📊 Infográfico: Os 4 Pilares da POO em um Exemplo

Vamos explorar como **Abstração**, **Herança**, **Encapsulamento** e **Polimorfismo** trabalham juntos usando um exemplo prático: a modelagem de veículos.

🚗 O Cenário: Sistema de Veículos

Imagine que precisamos criar um sistema para gerenciar diferentes tipos de veículos. Cada veículo tem características e comportamentos em comum, mas também possui suas particularidades.

🧠 1. Abstração

O que é essencial em um veículo?

Começamos definindo um "contrato" ou uma ideia geral do que é um Veiculo. Não criamos um "veículo genérico", mas sim o conceito dele. Ele deve ser capaz de ligar, desligar e acelerar.

A Ação: Criamos uma classe abstract Veiculo que define os comportamentos essenciais, mas deixa a implementação específica para quem for herdar dela.

```
// A classe abstrata define o "contrato"
public abstract class Veiculo {
    protected String marca;
    private int velocidadeAtual; // Será encapsulado!

    // Comportamento obrigatório para todas as subclasses
    public abstract void acelerar();

    public void ligar() {
        System.out.println("Veículo ligado.");
    }
}
```

👨‍👩‍👧 2. Herança

Reutilizando a base para criar tipos específicos.

Agora, criamos classes concretas que herdam as características e comportamentos do nosso Veiculo abstrato. Um Carro **é um** Veiculo. Uma Moto **é um** Veiculo.

A Ação: Usamos a palavra-chave extends para que Carro e Moto reutilizem o código de Veiculo.

```
// Carro herda de Veiculo
public class Carro extends Veiculo {
    // Implementação específica de acelerar para Carro
    @Override
    public void acelerar() {
        // Lógica de encapsulamento aqui!
        int novaVelocidade = getVelocidadeAtual() + 10;
        setVelocidadeAtual(novaVelocidade);
        System.out.println("Carro acelerando para " + getVelocidadeAtual() + " km/h.");
    }
}
```

🔒 3. Encapsulamento

Protegendo o estado interno do veículo.

A velocidade de um veículo não deve poder ser alterada para qualquer valor de forma arbitrária. Ela deve ser controlada por métodos, como acelerar() e frear().

A Ação: Declaramos o atributo velocidadeAtual como private na classe Veiculo e criamos métodos públicos (getters e setters) para controlar seu acesso.

```
// Dentro da classe abstrata Veiculo
public abstract class Veiculo {
    // ...
    private int velocidadeAtual = 0; // Atributo privado

    // Método público para ler o valor (Getter)
    public int getVelocidadeAtual() {
        return this.velocidadeAtual;
    }

    // Método protegido/público para alterar o valor (Setter)
    protected void setVelocidadeAtual(int velocidade) {
        if (velocidade >= 0) {
            this.velocidadeAtual = velocidade;
        }
    }
    // ...
}
```

🎭 4. Polimorfismo

"Muitas formas" de acelerar.

Apesar de Carro e Moto serem Veiculos, eles aceleram de maneiras diferentes. O polimorfismo permite que, ao chamarmos o mesmo método acelerar(), o comportamento correto seja executado dependendo do objeto.

A Ação: Cada classe filha (Carro, Moto) sobrescreve (@Override) o método acelerar() com sua própria lógica.

```
// Moto também herda de Veiculo
public class Moto extends Veiculo {
    @Override
    public void acelerar() {
        int novaVelocidade = getVelocidadeAtual() + 15; // Moto acelera mais rápido
        setVelocidadeAtual(novaVelocidade);
        System.out.println("Moto acelerando para " + getVelocidadeAtual() + " km/h.");
    }
}

// Classe principal para ver a mágica acontecer
public class Garagem {
    public static void main(String[] args) {
        Veiculo meuCarro = new Carro();
        Veiculo minhaMoto = new Moto();

        // A mesma chamada de método...
        meuCarro.acelerar();  // Saída: Carro acelerando para 10 km/h.
        minhaMoto.acelerar(); // Saída: Moto acelerando para 15 km/h.
        // ...gera comportamentos diferentes!
    }
}
```

✨ Conclusão

Neste único exemplo, vimos como:

- A **Abstração** definiu o que é um veículo.
- A **Herança** criou tipos específicos de veículos reutilizando código.
- O **Encapsulamento** protegeu o estado interno (velocidade).
- O **Polimorfismo** permitiu que cada veículo acelerasse à sua maneira.

Juntos, esses pilares criam um código mais organizado, seguro, flexível e fácil de manter.

Atividade: Pilares da Programação Orientada a Objetos

Esta atividade foi projetada para testar seu conhecimento sobre os quatro conceitos fundamentais da POO: **Encapsulamento**, **Herança**, **Polimorfismo** e **Abstração**.

Questões

1. Qual pilar da POO é responsável por agrupar atributos (dados) e métodos (comportamentos) em uma única unidade chamada "classe", escondendo os detalhes complexos de implementação do mundo exterior?
a) Herança
b) Polimorfismo
c) Encapsulamento
d) Abstração
2. Observe o código Java abaixo:

```
class Funcionario {
    String nome;
    double salario;

    public Funcionario(String nome, double salario) {
        this.nome = nome;
        this.salario = salario;
    }
}

class Gerente extends Funcionario {
    double bonus;

    public Gerente(String nome, double salario, double bonus) {
        super(nome, salario);
        this.bonus = bonus;
    }
}
```

Qual conceito da POO é o mais evidente neste exemplo?
a) Polimorfismo
b) Herança

c) Abstração
d) Encapsulamento

3. A capacidade de um método se comportar de maneiras diferentes dependendo do objeto que o invoca é conhecida como:
a) Herança
b) Polimorfismo
c) Abstração
d) Encapsulamento

4. Imagine que você está modelando diferentes formas geométricas (Circulo, Quadrado, Triangulo). Todas elas precisam de um método para calcular a área, mas a fórmula para cada uma é diferente. Qual pilar da POO permite que você chame um mesmo método calcularArea() em objetos de tipos diferentes e obtenha o resultado correto para cada um?
a) Herança
b) Encapsulamento
c) Polimorfismo
d) Abstração

5. O principal objetivo da Abstração é:
a) Permitir que uma classe herde características de outra.
b) Agrupar dados e métodos em uma única unidade.
c) Esconder a complexidade e expor apenas a funcionalidade essencial de um objeto.
d) Permitir que um objeto tenha múltiplas formas.

6. Em uma classe ContaBancaria, o atributo saldo é definido como privado para que ele só possa ser alterado através de métodos como depositar() e sacar(). Esta prática é um exemplo clássico de:
a) Polimorfismo
b) Herança
c) Abstração
d) Encapsulamento

7. O que a palavra-chave super() geralmente faz em um método construtor de uma classe filha (subclasse)?
a) Invoca um método da própria classe.
b) Invoca o método construtor da classe mãe (superclasse).
c) Cria uma nova instância da classe mãe.
d) Deleta a instância da classe filha.

8. Qual pilar da POO promove o maior reuso de código, evitando que você precise reescrever a mesma lógica em múltiplas classes?
a) Abstração
b) Polimorfismo
c) Herança
d) Encapsulamento

9. Um controle remoto de TV é uma ótima analogia para qual conceito da POO? Você sabe que o botão "aumentar volume" funciona, mas não precisa conhecer os detalhes

do circuito eletrônico por trás dele.
a) Polimorfismo
b) Abstração
c) Herança
d) Encapsulamento

10. Considere o código Java abaixo:

```
abstract class Animal {
    public abstract void fazerSom();
}

class Cachorro extends Animal {
    public void fazerSom() {
        System.out.println("Au Au");
    }
}

class Gato extends Animal {
    public void fazerSom() {
        System.out.println("Miau");
    }
}
```

A definição de Animal como uma classe abstract com um método abstract fazerSom() é um exemplo de qual pilar?
a) Encapsulamento
b) Herança
c) Polimorfismo
d) Abstração

Gabarito

1. **c) Encapsulamento**
2. **b) Herança**
3. **b) Polimorfismo**
4. **c) Polimorfismo**
5. **c) Esconder a complexidade e expor apenas a funcionalidade essencial de um objeto.**
6. **d) Encapsulamento**
7. **b) Invoca o método construtor da classe mãe (superclasse).**
8. **c) Herança**
9. **b) Abstração**
10. **d) Abstração**

Exemplos e Explicações dos Pilares (em Java)

1. Encapsulamento

Agrupa dados (atributos) e os métodos que os manipulam dentro de uma classe, controlando o acesso a esses dados com modificadores como private.

- **Exemplo Prático (Java):**

```
public class ContaBancaria {
    private String titular;
    private double saldo; // Atributo privado

    public ContaBancaria(String titular) {
        this.titular = titular;
        this.saldo = 0.0;
    }

    public void depositar(double valor) {
        if (valor > 0) {
            this.saldo += valor;
            System.out.println("Depósito de R$" + valor + " realizado. Novo saldo: R$" + this.saldo);
        } else {
            System.out.println("Valor de depósito inválido.");
        }
    }

    public void sacar(double valor) {
        if (valor > 0 && valor <= this.saldo) {
            this.saldo -= valor;
```

```
            System.out.println("Saque de R$" + valor + " realizado. Novo saldo: R$" + this.saldo);
        } else {
            System.out.println("Saldo insuficiente ou valor de saque inválido.");
        }
    }

    // Método "getter" para acesso controlado
    public double getSaldo() {
        return this.saldo;
    }
}

// Para usar:
// ContaBancaria conta = new ContaBancaria("Ana");
// conta.depositar(100);
// conta.sacar(30);
// System.out.println("Saldo final consultado: R$" + conta.getSaldo());
```

Explicação: O saldo (saldo) é private, então não pode ser acessado ou modificado diretamente de fora da classe. A única forma de interagir com ele é através dos métodos públicos depositar(), sacar() e getSaldo(), garantindo a integridade dos dados.

2. Herança

Permite que uma classe (filha ou subclasse) herde atributos e métodos de outra classe (mãe ou superclasse) usando a palavra-chave extends.

- **Exemplo Prático (Java):**

```
class Veiculo {
    protected String marca;
    protected String modelo;

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

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

class Carro extends Veiculo { // Carro herda de Veiculo
```

```
    private int portas;

    public Carro(String marca, String modelo, int portas) {
        super(marca, modelo); // Chama o construtor da classe mãe
        this.portas = portas;
    }

    public void abrirPortaMalas() {
        System.out.println("Porta-malas aberto.");
    }
}

// Para usar:
// Carro meuCarro = new Carro("Ford", "Ka", 4);
// System.out.println("Meu carro é um " + meuCarro.marca + " " + meuCarro.modelo);
// meuCarro.acelerar(); // Método herdado de Veiculo
// meuCarro.abrirPortaMalas(); // Método específico de Carro
```

Explicação: A classe Carro não precisou reimplementar a lógica de marca, modelo ou o método acelerar(), pois ela "herdou" tudo isso da classe Veiculo.

3. Polimorfismo

Permite que objetos de classes diferentes respondam à mesma chamada de método de maneiras específicas, geralmente através da sobrescrita de métodos (@Override).

- **Exemplo Prático (Java):**

```
class Ave {
    public void voar() {
        System.out.println("Voando de forma genérica.");
    }
}

class Pardal extends Ave {
    @Override // Sobrescreve o método da classe mãe
    public void voar() {
        System.out.println("O pardal voa batendo as asas rapidamente.");
    }
}

class Pinguim extends Ave {
    @Override // Sobrescreve o método da classe mãe
    public void voar() {
```

```
        System.out.println("O pinguim não voa, ele nada.");
    }
}

// Classe para demonstrar o polimorfismo
class TesteAves {
    public static void fazerVoar(Ave ave) {
        ave.voar();
    }

    public static void main(String[] args) {
        Ave pardal = new Pardal();
        Ave pinguim = new Pinguim();

        fazerVoar(pardal);  // Saída: O pardal voa batendo as asas rapidamente.
        fazerVoar(pinguim); // Saída: O pinguim não voa, ele nada.
    }
}
```

Explicação: O método fazerVoar chama o mesmo método voar(), mas o comportamento executado depende do tipo de objeto (Pardal ou Pinguim) que é passado para ele.

4. Abstração

Oculta os detalhes complexos de implementação, mostrando apenas as funcionalidades essenciais, através de classes e métodos abstratos (abstract).

- **Exemplo Prático (Java):**

```
// Classe abstrata que serve como um "contrato"
abstract class ControleRemoto {
    public abstract void ligar();
    public abstract void desligar();
}

// Classe concreta que implementa o contrato
class ControleTV extends ControleRemoto {
    @Override
    public void ligar() {
        System.out.println("TV ligada: enviando sinal infravermelho...");
        // (lógica complexa de sinal aqui)
    }

    @Override
```

```
    public void desligar() {
        System.out.println("TV desligada: enviando sinal infravermelho...");
        // (lógica complexa de sinal aqui)
    }
}

// Para usar:
// ControleRemoto meuControle = new ControleTV();
// meuControle.ligar();
// meuControle.desligar();
```

Explicação: A classe ControleRemoto define o que um controle **deve fazer** (ligar, desligar), mas não como. Ela é um contrato (abstração). A classe ControleTV fornece a implementação concreta, escondendo os detalhes de "como" o sinal é enviado.