JAVA - POO

1. **Classes e Objetos**
    - **Classe**: Em Java, uma classe é uma blueprint (plano de construção) para a criação de objetos. Ela encapsula dados e comportamentos, representados por atributos (variáveis de instância) e métodos, respectivamente.
    - **Objeto**: Uma instância de uma classe. Ele é uma entidade que possui estado (valores dos atributos) e comportamento (os métodos que ele pode executar).

```
// Sintaxe
class NomeDaClasse {
    // Atributos
    tipo nomeDoAtributo;

    // Métodos
    tipoRetorno nomeDoMetodo() {
        // Corpo do método
    }
}

// Exemplo

class Carro {
    String marca;
    int ano;

    void ligar() {
        System.out.println("Carro ligado!");
    }
}
```

No exemplo acima,
`Carro` é uma **classe** com dois atributos (`marca`, `ano`) e um método (`ligar`). Um **objeto** dessa classe poderia ser criado assim:

```
Carro meuCarro = new Carro();
```

---

2. **Atributos de Classe**

   Os **atributos de classe** (também chamados de **variáveis de instância**) são usados para armazenar os dados ou estado dos objetos de uma classe. Cada objeto tem sua própria cópia desses atributos. Já os **atributos estáticos** são compartilhados por todos os objetos da classe.
    - **Atributos de Instância**: Variáveis cujos valores são específicos para cada instância (objeto) de uma classe.
    - **Atributos Estáticos (de Classe)**: Variáveis associadas à classe em si, em vez de suas instâncias. Todos os objetos da classe compartilham o mesmo valor.

```
// Sintaxe
class NomeDaClasse {
    // Atributo de instância
    tipo nomeAtributo;

    // Atributo de classe (estático)
    static tipo nomeAtributoEstatico;
}

// Exemplo
class Carro {
```

```
    // Atributo de instância
    String marca;

    // Atributo de classe (estático)
    static int totalDeCarros;
}
```

3. **Métodos de classe**

Os **métodos** são funções definidas dentro de uma classe que descrevem os comportamentos que um objeto pode ter. Métodos podem ser de instância ou de classe (estáticos).

- **Métodos de Instância**: Operam sobre instâncias (objetos) da classe.
- **Métodos Estáticos**: Operam a partir da própria classe e não precisam de um objeto para serem invocados.

```
// Sintaxe
class NomeDaClasse {
    // Método de instância
    tipoRetorno nomeDoMetodo() {
        // Corpo do método
    }

    // Método de classe (estático)
    static tipoRetorno nomeDoMetodoEstatico() {
        // Corpo do método
    }
}

// Exemplo
class Carro {
    static int totalDeCarros;

    // Método de instância
    void ligar() {
        System.out.println("Carro ligado!");
    }

    // Método de classe (estático)
    static void exibirTotalDeCarros() {
        System.out.println("Total de carros: " + totalDeCarros);
    }
}
```

4. **Construtores**

Os **construtores** são métodos especiais usados para inicializar objetos. Eles têm o mesmo nome da classe e são chamados automaticamente quando um objeto da classe é criado. Construtores podem ser sobrecarregados (múltiplos construtores com diferentes assinaturas).

```
// Sintaxe
class NomeDaClasse {
    // Construtor
    NomeDaClasse() {
        // Inicializa os atributos
    }
}

// Exemplo
```

```
class Carro {
    String marca;
    int ano;

    // Construtor com dois parâmetros
    Carro(String marca, int ano) {
        this.marca = marca;
        this.ano = ano;
    }
}
```

Aqui, o construtor `Carro(String marca, int ano)` inicializa os atributos `marca` e `ano` quando um novo objeto `Carro` é criado.

---

5. **Modificadores de acesso**

   Os **modificadores de acesso** em Java controlam a visibilidade de atributos, métodos e classes. Os principais modificadores de acesso são:

   - `public`: O membro é acessível por qualquer outra classe.
   - `private`: O membro é acessível apenas dentro da própria classe.
   - `protected`: O membro é acessível dentro do mesmo pacote ou por subclasses.
   - **Default** (sem especificar): O membro é acessível apenas dentro do mesmo pacote.

```
// Sintaxe
class NomeDaClasse {
    private tipo atributoPrivado;
    public tipo atributoPublico;

    protected tipo metodoProtegido() {
        // Corpo do método
    }
}

// Exemplo
class Pessoa {
    // Atributo privado, acessível apenas dentro da classe
    private String nome;

    // Método público para acessar o nome (getter)
    public String getNome() {
        return nome;
    }

    // Método público para modificar o nome (setter)
    public void setNome(String nome) {
        this.nome = nome;
    }
}
```

---

6. **Encapsulamento**

   O **encapsulamento** é o conceito de proteger os dados internos de uma classe, controlando o acesso a esses dados por meio de métodos. Isso ajuda a manter o controle sobre como os atributos são acessados e modificados, promovendo a integridade dos dados.

   Principais Características:

   - Atributos são normalmente privados.
   - Métodos públicos (getters e setters) são fornecidos para acessar e modificar os atributos.

```
// Exemplo
class ContaBancaria {
    private double saldo;

    // Método getter
    public double getSaldo() {
        return saldo;
    }

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

---

7. **Herança**

   A **herança** é o mecanismo que permite a criação de uma nova classe (subclasse) a partir de uma classe existente (superclasse). A subclasse herda os atributos e métodos da superclasse, podendo também adicionar novos membros ou sobrescrever os existentes.

```
// Sintaxe
class Superclasse {
    // Atributos e métodos da superclasse
}

class Subclasse extends Superclasse {
    // Atributos e métodos adicionais da subclasse
}

// Exemplo
class Animal {
    void fazerSom() {
        System.out.println("O animal faz um som.");
    }
}

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

Aqui,
`Cachorro` herda o método `fazerSom` de `Animal`, mas o sobrescreve com uma implementação específica.

---

8. **Polimorfismo**

   O **polimorfismo** permite que um objeto de uma classe seja tratado como se fosse de sua superclasse. Existem dois tipos principais de polimorfismo:

   - **Polimorfismo de sobrecarga**: Métodos com o mesmo nome, mas com parâmetros diferentes.
   - **Polimorfismo de sobrescrita**: A subclasse fornece uma implementação específica de um método que já está presente na superclasse.

```
// Sintaxe
class Superclasse {
    void metodo() {
```

```
        // Corpo do método
    }
}

class Subclasse extends Superclasse {
    @Override
    void metodo() {
        // Sobrescreve o método
    }
}

// Exemplo
class Animal {
    void fazerSom() {
        System.out.println("O animal faz um som.");
    }
}

class Gato extends Animal {
    @Override
    void fazerSom() {
        System.out.println("O gato mia.");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal meuAnimal = new Gato();  // Polimorfismo
        meuAnimal.fazerSom();  // Saída: O gato mia.
    }
}
```

9. **Classes internas**

Uma **classe interna** é uma classe definida dentro de outra classe. Ela é usada para agrupar logicamente classes que estão fortemente relacionadas e para acessar os membros da classe externa.

```
// Sintaxe
class ClasseExterna {
    // Classe interna
    class ClasseInterna {
        // Atributos e métodos da classe interna
    }
}

// Exemplo
class Externa {
    private String mensagem = "Mensagem da classe externa.";

    class Interna {
        void exibirMensagem() {
            System.out.println(mensagem);  // Acesso direto ao atributo da classe externa
        }
    }
}
```

10. **Abstração**

A **abstração** é o processo de esconder os detalhes de implementação de uma classe e expor apenas sua funcionalidade essencial. Em Java, ela pode ser implementada usando **classes abstratas** ou **interfaces**.

```
// Sintaxe
abstract class ClasseAbstrata {
    abstract void metodoAbstrato();
}

// Exemplo
abstract class Forma {
    abstract void desenhar();
}

class Circulo extends Forma {
    @Override
    void desenhar() {
        System.out.println("Desenhando um círculo.");
    }
}
```

1. **Interface**
   Uma **interface** define um contrato que as classes que a implementam devem seguir. Ela só pode conter declarações de métodos (sem implementação) e constantes.

```
// Sintaxe
interface NomeDaInterface {
    void metodo();
}

// Exemplo
interface Animal {
    void fazerSom();
}

class Passaro implements Animal {
    public void fazerSom() {
        System.out.println("O pássaro canta.");
    }
}
```

2. **Enumerações (enums)**
   Uma **enumeração** (enum) é um tipo especial em Java que define um conjunto fixo de constantes.

```
// Sintaxe
enum NomeDoEnum {
    CONSTANTE1, CONSTANTE2, CONSTANTE3;
}

// Exemplo
enum Dia {
    SEGUNDA, TERCA, QUARTA, QUINTA, SEXTA;
}
```

3. **Entrada de usuário**
   A entrada do usuário em Java geralmente é feita utilizando a classe `Scanner`, que permite capturar dados inseridos pelo teclado.

```
// Sintaxe
import java.util.Scanner;
```

```
Scanner scanner = new Scanner(System.in);
String input = scanner.nextLine();
```

4. **Data e Hora**

   A manipulação de data e hora em Java pode ser feita usando classes como `LocalDate`, `LocalTime`, e `LocalDateTime`, que fazem parte da biblioteca `java.time`.

```
// Sintaxe
import java.time.LocalDate;
import java.time.LocalDateTime;

LocalDate dataAtual = LocalDate.now();
LocalDateTime dataHoraAtual = LocalDateTime.now();
```

5. **Exceções**

   As **exceções** são eventos que ocorrem durante a execução de um programa que interrompem o fluxo normal de execução. Em Java, exceções são tratadas usando blocos `try-catch`.

```
// Sintaxe
try {
    // Código que pode lançar exceção
} catch (TipoDeExcecao e) {
    // Código para tratar a exceção
}
```

6. **Arrays**

   Os **arrays** em Java são estruturas de dados que armazenam múltiplos valores do mesmo tipo em uma única variável.

```
// Sintaxe
tipo[] nomeDoArray = new tipo[tamanho];

// Exemplo
int[] numeros = {1, 2, 3, 4, 5};
```

7. **ArrayList**

   A classe `ArrayList` é uma implementação da interface `List` que fornece uma lista dinâmica, ou seja, o tamanho do `ArrayList` pode aumentar ou diminuir conforme os elementos são adicionados ou removidos.

```
// Sintaxe
import java.util.ArrayList;

ArrayList<tipo> lista = new ArrayList<>();

// Exemplo
ArrayList<String> frutas = new ArrayList<>();
frutas.add("Maçã");
frutas.add("Banana");
```