# Programação Avançada - AULA 03

Matheus Moresco

Análise e Desenvolvimento de Sistemas- 5º Período

2025/01

# Última aula

- Revisão dos conceitos de POO
  - Encapsulamento
  - Herança
  - Polimorfismo
  - Abstração
- Sobrecarga de Métodos
- Métodos construtores

# Introdução

- Revisão das ultimas aulas
- Exercício prático

# Princípios da POO

- A Programação Orientada a Objetos (POO) se baseia em quatro princípios fundamentais:

  - **Encapsulamento**: Restringe o acesso direto aos dados de um objeto, permitindo manipulação apenas por métodos específicos, garantindo segurança e integridade.
  - **Herança**: Permite que uma classe herde atributos e métodos de outra, promovendo reutilização e organização do código.
  - **Polimorfismo**: Permite que diferentes classes sejam tratadas de forma uniforme, desde que sigam uma interface comum, aumentando a flexibilidade.
  - **Abstração**: Oculta detalhes internos e expõe apenas o necessário, reduzindo a complexidade.

# Princípios da POO - Encapsulamento

- O **encapsulamento** é um dos princípios da Programação Orientada a Objetos (POO) que restringe o acesso direto aos dados internos de um objeto e expõe apenas métodos seguros para manipular esses dados.

```
 2
 3 public class Pessoa {
 4     private String nome;
 5
 6     public void setNome(String nome) {
 7         this.nome = nome;
 8     }
 9
10     public String getNome() {
11         return nome;
12     }
13 }
```

# Princípios da POO - Herança

- A herança é um dos pilares da Programação Orientada a Objetos (POO) e permite que uma classe (filha) reutilize atributos e métodos de outra classe (pai).

```
2
3  public class Pessoa {
4      protected String nome;
5      protected int idade;
6
7      public Pessoa(String nome, int idade) {
8          this.nome = nome;
9          this.idade = idade;
10     }
11
12     public void exibirInformacoes() {
13         System.out.println("Nome: " + nome);
14         System.out.println("Idade: " + idade);
15     }
16 }
17
```

# Princípios da POO - Herança

```
 2
 3 public class Aluno extends Pessoa {
 4  private String curso;
 5
 6  public Aluno(String nome, int idade, String curso) {
 7      this.nome = nome;
 8      this.idade = idade;
 9      this.curso = curso;
10  }
11 }
12
```

```
 2
 3 public class Professor extends Pessoa {
 4  private String disciplina;
 5
 6  public Professor(String nome, int idade, String disciplina) {
 7      this.nome = nome;
 8      this.idade = idade;
 9      this.disciplina = disciplina;
10  }
11 }
12
```

# Princípios da POO - Polimorfismo

- O polimorfismo permite que um mesmo método tenha comportamentos diferentes dependendo do objeto que o utiliza. Isso é possível por meio de **métodos sobrescritos** em subclasses.

```
2
3  public class Aluno extends Pessoa {
4      private String curso;
5
6      public Aluno(String nome, int idade, String curso) {
7          this.nome = nome;
8          this.idade = idade;
9          this.curso = curso;
10     }
11
12     @Override
13     public void exibirInformacoes() {
14         System.out.println("Nome: " + nome);
15         System.out.println("Idade: " + idade);
16         System.out.println("Curso: " + curso);
17     }
18 }
19
```

```
2
3  public class Professor extends Pessoa {
4      private String disciplina;
5
6      public Professor(String nome, int idade, String disciplina) {
7          this.nome = nome;
8          this.idade = idade;
9          this.disciplina = disciplina;
10     }
11
12     @Override
13     public void exibirInformacoes() {
14         System.out.println("Nome: " + nome);
15         System.out.println("Idade: " + idade);
16         System.out.println("Disciplina: " + disciplina);
17     }
18 }
19
```

# Princípios da POO - Abstração

- A **abstração** em POO consiste em ocultar detalhes internos da implementação e expor apenas os comportamentos essenciais.
  - Utilizamos **classes abstratas** para criar um modelo base que define métodos genéricos, sem implementação completa.
  - Subclasses concretas devem implementar esses métodos.

```
 2
 3 public abstract class Pessoa {
 4     protected String nome;
 5     protected int idade;
 6
 7     public Pessoa(String nome, int idade) {
 8         this.nome = nome;
 9         this.idade = idade;
10     }
11
12     public abstract void exibirInformacoes();
13
14 }
15
```

# O que é Sobrecarga?

- A sobrecarga de métodos é um conceito do polimorfismo que consiste basicamente em criar variações de um mesmo método, ou seja, a criação de dois ou mais métodos com nomes totalmente iguais em uma classe.
- Permite que utilizemos o mesmo nome em mais de um método contanto que suas listas de argumentos sejam diferentes para que seja feita a separação dos mesmos.
- Melhora a legibilidade e a organização do código, permitindo que uma mesma operação seja realizada de formas diferentes, dependendo dos argumentos fornecidos.

# Regras da Sobrecarga de Métodos

- Os métodos devem ter o mesmo nome.
- Devem possuir assinaturas diferentes (quantidade ou tipo de parâmetros).
- Não podem diferir apenas pelo tipo de retorno.

```
2
3   public class Calculadora {
4       public int soma(int a, int b, int c) {
5           return a + b + c;
6       }
7       public int soma(int a, int b) {
8           return a + b;
9       }
10      public double soma(int a, int b) {
11          return a + b;
12      }
13  }
```

✖ Errado

```
2
3   public class Calculadora {
4       public int soma(int a, int b) {
5           return a + b;
6       }
7       public int soma(int a, int b, int c) {
8           return a + b + c;
9       }
10      public double soma(double a, double b) {
11          return a + b;
12      }
13  }
```

✓ Correto

# Métodos Construtores

- Os métodos construtores são métodos especiais em Java utilizados para inicializar objetos de uma classe. Eles têm o mesmo nome da classe e são chamados automaticamente quando um objeto é instanciado usando new.

**Objeto objeto1 = new Objeto(attr1, attr2)**

# Métodos Construtores

```java

public class Pessoa {
    String nome;
    int idade;

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


    public void exibirInformacoes() {
        System.out.println("Nome: " + nome);
        System.out.println("Idade: " + idade);
    }
}

```

```java

public class App {
    Run | Debug | Run main | Debug main
    public static void main(String[] args) throws Exception {
        Pessoa pessoa = new Pessoa(nome:"João", idade:20);

        pessoa.exibirInformacoes();
    }
}


```

# Tipos de Construtores em Java

1. Construtor Padrão (Sem Parâmetros)
2. Construtor Parametrizado
3. Sobrecarga de Construtores
4. Construtor Copiador

# 1. Construtor Padrão (Sem Parâmetros)

- Caso nenhum construtor seja definido, Java cria um construtor padrão automaticamente.

```
1
2   public class Pessoa {
3       String nome;
4       int idade;
5
6       public Pessoa() {
7           this.nome = "Sem Nome";
8       }
9
10      public void exibirInformacoes() {
11          System.out.println("Nome: " + nome);
12          System.out.println("Idade: " + idade);
13      }
14  }
15
```

# 2. Construtor Parametrizado

- Construtores podem aceitar parâmetros para inicializar atributos.

```java

public class Pessoa {
    String nome;
    int idade;

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


    public void exibirInformacoes() {
        System.out.println("Nome: " + nome);
        System.out.println("Idade: " + idade);
    }
}

```

# 3. Sobrecarga de Construtores

- Podemos criar vários construtores com diferentes parâmetros.

```java
public class Pessoa {
    String nome;
    int idade;

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

    public Pessoa(String nome) {
        this.nome = nome;
    }

    public Pessoa() {
        this.nome = "Sem Nome";
    }

    public void exibirInformacoes() {
        System.out.println("Nome: " + nome);
        System.out.println("Idade: " + idade);
    }
}
```

# 4. Construtor Copiador

- Permite criar um novo objeto a partir de outro já existente.

```java

public class Pessoa {
    String nome;
    int idade;

    public Pessoa(Pessoa outraPessoa) {
        this.nome = outraPessoa.nome;
        this.idade = outraPessoa.idade;
    }

    public void exibirInformacoes() {
        System.out.println("Nome: " + nome);
        System.out.println("Idade: " + idade);
    }
}

```

# Vantagens do Uso de Construtores

- ✓Facilitam a inicialização dos objetos
- ✓Garantem que um objeto tenha valores válidos
- ✓Melhoram a legibilidade e manutenção do código
- ✓Permitem flexibilidade na criação de instâncias (com sobrecarga)

# Exemplo Prático

- Criar uma classe **Carro**.
- Adicionar os atributos **marca**, **modelo** e **ano**.
- Adicionar método **exibirDetalhes**().
- Implementar o método construtor do **Carro** e fazer a sobrecarga dele.
- Fazer o encapsulamento da classe **Carro**.
- Criar uma classe Filha **CarroEsportivo** que herda a de **Carro** e adiciona o atributo **velocidadeMaxima**.
- Adicione o método acelerar, com implementações diferentes nas classes **Carro** e **CarroEsportivo**.
- Faça a abstração do método **acelerar** na classe **Carro**

# Trabalhos da disciplina

- Serão vários trabalhos durante o bimestre.
- Alguns trabalhos serão interligados.
- Entregues todos juntos e um único projeto.
- Entrega no final do Bimestre.
- Grupos de até 4 pessoas.

# Exercício Prático

1. Crie uma classe **Pessoa** com sobrecarga de construtores, utilize diferentes de informações (nome, idade, endereço, etc), crie um método para exibir as informações da pessoa, crie um atributo cpf e restrinja as permissões de acesso deste atributo.
2. Criar uma classe **Funcionario** que herda as informações da classe Pessoa, use sobrecarga de construtores para inicializar com diferentes conjuntos de informações (cargo, salário, etc.).
3. Faça um método **getSalario**() que indica o salário do funcionário, e depois crie uma sobrecarga do método **getSalário**() que calcula o salário a partir de um valor de bônus, que será informado como parâmetro do método.
4. Faça a sobreposição do método de exibir informações, para exibir as informações do funcionário.