



## A Orientação a Objetos

---

Nivelamento de Lógica de Programação e  
Programação Orientada a Objeto

# A Orientação a Objetos

O Básico sobre Orientação a Objetos

Java é uma linguagem fortemente tipada e orientada a objetos.

Portanto, temos trabalhado com classes, instâncias e métodos mesmo sem, no entanto, nos aprofundar.

Agora chegamos ao momento de abordar como Java implementa a POO.

# A Orientação a Objetos

O Básico sobre Orientação a Objetos

A Programação Orientada a Objetos é um dos paradigmas mais populares - *talvez junto à programação funcional* - na engenharia de software moderna.

A forma de trabalhar abstrações de problemas em código representando seus atributos e comportamentos pode, a princípio, causar algum impacto, mas também parece muito natural.

Java adota totalmente os pilares da **Abstração**, **Encapsulamento**, **Herança** e **Polimorfismo**, combinando segurança e flexibilidade às aplicações.

# A Orientação a Objetos

O Básico sobre Orientação a Objetos

Como Java é totalmente aderente a POO, então desde o início temos trabalhado e declarado classes:

```
public class MinhaClasse {  
    //conteudo da classe  
}
```

# A Orientação a Objetos

O Básico sobre Orientação a Objetos

Uma classe pode definir métodos e atributos (ou propriedades), sendo que esses podem ser de instância ou estáticos.

Vamos decifrar o "Hello World"(clássico) e entender melhor.

```
public class Main {  
    public static void main(String[] args) {  
        String nome = "Ada Tech";  
        System.out.println("Hello, " + nome);  
    }  
}
```

# A Orientação a Objetos

## O Básico sobre Orientação a Objetos

Para definir novos métodos que serão referenciados diretamente dentro de main é necessário que sejam estáticos (**static**), ou seja, são métodos da classe.

Se, porém, quiser definir comportamentos específicos para cada instância criada, será preciso omitir palavra-chave **static** e executá-los a partir de objetos da classe.

```
public class Main {  
    public static void main(String[] args) {  
        String nome = "Ada Tech";  
        System.out.println("Hello, " + nome);  
    }  
}
```

# A Orientação a Objetos

## O Básico sobre Orientação a Objetos

```
public class Main {
    static int contGlobal = 0;
    int contLocal = 0;
    public static void main(String[] args) {
        Main m1 = new Main();
        m1.incrementar();
        m1.incrementar();
        System.out.println(m1.contLocal);
        Main m2 = new Main();
        m2.incrementar();
        System.out.println(m2.contLocal);
        System.out.println(contGlobal);
    }
    int incrementar() {
        contLocal += 1;
        incrementarGlobal();
        return contLocal;
    }
    static int incrementarGlobal() {
        contGlobal += 1;
        return contGlobal;
    }
}
```

Abstração

# A Orientação a Objetos

## Abstração

**Abstração** é um dos pilares ou fundamentos da orientação a objetos e trata sobre como queremos representar as coisas.

Pessoas são seres complexos com muitas características físicas, de personalidade e de conhecimento.

Não há uma forma única de representar uma classe **Pessoa**, portanto representamos o que é relevante em um dado contexto.

O que chamamos de abstração é como algo é representado naquele conjunto de atributos e métodos conforme o contexto.

# A Orientação a Objetos

## Abstração

Nas redes sociais podemos escolher o nome de usuário, os interesses e uma foto.

Para uma aplicação bancária podemos considerar data de nascimento, profissão e renda.

Veja que estamos falando da mesma pessoa, mas considerando abstrações diferentes. Até dentro da mesma aplicação é possível criar abstrações numa funcionalidade.

Portanto, não há motivos razoáveis para ter uma classe com muitos atributos nem métodos. Sempre considere a abstração necessária para aquele contexto.

Construtores

# A Orientação a Objetos

## Construtores

Java define um construtor padrão vazio para todas as classes. Por isso conseguimos instanciar `Main` mesmo não vendo seu construtor.

Ainda é possível definir outros construtores com diferentes assinaturas.

Porém, havendo qualquer construtor explícito implica em não gerar o construtor vazio, cabendo a você fazer essa declaração.

# Construtores

## Assinatura de métodos

A assinatura do método é composta do **nome** e os **tipos** dos parâmetros. Ou seja, o compilador não entende o nome do parâmetro definido, mas apenas os tipos.

**Cada assinatura deve ser única numa classe.**

Quando criamos variações de assinatura de métodos com o mesmo nome, chamamos de sobrecarga.

# Construtores

Assinatura de métodos

Tomando como base a discussão sobre construtores. Vamos ao código a seguir:

```
public class Pessoa {  
    String nome;  
    String sobrenome;  
    public Pessoa(String nome, String sobrenome) {  
        this.nome = nome;  
        this.sobrenome = sobrenome;  
    }  
    public Pessoa(String nome) {  
        this.nome = nome;  
    }  
    public Pessoa() {  
    }  
}
```

# Encapsulamento

# Encapsulamento

Encapsulamento é mais um pilar fundamental da orientação a objetos.

Cada classe deve ser responsável pelo controle ao acesso e manipulação dos seus atributos.

Nos exemplos de classes que trabalhamos até aqui, tanto em `Main` como em `Pessoa`, os atributos foram declarados como uma variável comum tipo `nomeVariavel`.

Isso permite que outras classes consigam acessar esses atributos e manipular diretamente.

# Encapsulamento

Em Java, trabalhamos o encapsulamento restringindo a visibilidade direta ao atributo e expondo métodos **get** e **set**.

```
public class Pessoa {  
    private String nome;  
    public String getNome() {  
        return nome;  
    }  
    public void setNome(String nome) {  
        this.nome = nome;  
    }  
}  
class Main {  
    public static void main(String[] args) {  
        Pessoa pessoa = new Pessoa();  
        pessoa.setNome("Aluno");  
        System.out.println(pessoa.getNome());  
    }  
}
```

# Encapsulamento

## Imutabilidade

Se desejar definir um atributo como imutável, declare como **final** e omita o **setter**.

Nesse caso, o valor para **nome** deve ser informado no momento de criação do objeto, ou seja, passado pelo construtor.

```
public class Pessoa {  
    private final String nome;  
    public Pessoa(String nome) {  
        this.nome = nome;  
    }  
    public String getNome() {  
        return nome;  
    }  
}  
  
class Main {  
    public static void main(String[] args) {  
        Pessoa pessoa = new Pessoa("Aluno");  
  
        System.out.println(pessoa.getNome());  
    }  
}
```

# Modificadores de Acesso

## Modificadores de Acesso

Até agora usamos os modificadores `public` e `private`, porém, sem discutir ou explicar.

Em Java, são quatro as opções de visibilidade ou acesso

- **`private`**: restrito ao escopo da classe
- não informado: quando não informado, a visibilidade padrão é restrita ao mesmo pacote; também chamada "package", "default" e "default-package".
- **`protected`**: restrito à classe e aos seus subtipos
- **`public`**: sem restrições.

Herança

## Herança

**Herança** é o terceiro pilar fundamental da orientação a objetos que abordamos nessa aula.

Herança define uma relação entre "tipo" e "subtipo", onde as definições no "tipo" também estarão presentes no "subtipo".

É uma forma prática de evitar repetições de código e representar hierarquia ou generalização nas abstrações.

## Herança

Considere uma abstração para um sistema de escola.

Podemos considerar uma abstração geral para **Pessoa** e os subtipos **Aluno** e **Instrutor**.

Assim, as características e comportamentos em comum podem ser definidos em **Pessoa** e as particularidades em cada subtipo específico.

## Herança

Java só permite herança simples, ou seja, cada subtipo só pode estar ligado a um tipo.

Porém, essa hierarquia permite múltiplos níveis.

```
class Aluno extends Pessoa {
    private final String matricula;
    public Aluno(String nome, String matricula) {
        super(nome);
        this.matricula = matricula;
    }
}

class Instrutor extends Pessoa {
    private final String area;
    public Instrutor(String nome, String area) {
        super(nome);
        this.area = area;
    }
}

class InstrutorTemporario extends Instrutor {
    private final LocalDate dataInicio;
    private final LocalDate dataFim;
    public InstrutorTemporario (String nome, String area,
        LocalDate dataInicio, LocalDate dataFim) {
        super(nome, area);
        this.dataInicio = dataInicio;
        this.dataFim = dataFim;
    }
}
```

# Herança

Composição, agregação e herança

**Composição** e **agregação** são conceitos importantes em orientação a objetos, complementares à herança.

Enquanto a herança define uma relação "é-um" (onde uma classe é uma especialização de outra), a composição e a agregação definem relações "tem-um" (onde uma classe é composta por ou agregada a outra).

# Herança

Composição, agregação e herança

**Composição** e **agregação** são conceitos importantes em orientação a objetos, complementares à herança.

Enquanto a herança define uma relação "é-um" (onde uma classe é uma especialização de outra), a composição e a agregação definem relações "tem-um" (onde uma classe é composta por ou agregada a outra).

- **Composição:** Indica que uma classe é composta por outras classes, e essas partes não podem existir sem a classe principal.

Por exemplo, uma classe **Carro** pode ser composta por **Motor**, **Rodas**, e **Portas**. Se o Carro deixar de existir, suas partes também deixam de existir.

# Herança

Composição, agregação e herança

**Composição** e **agregação** são conceitos importantes em orientação a objetos, complementares à herança.

Enquanto a herança define uma relação "é-um" (onde uma classe é uma especialização de outra), a composição e a agregação definem relações "tem-um" (onde uma classe é composta por ou agregada a outra).

- **Agregação:** Semelhante à composição, mas com uma diferença fundamental: as partes podem existir independentemente da classe principal.

Um exemplo clássico é uma **Turma** que agrega **Aluno(s)**. Se a **Turma** for desfeita, os **Alunos** continuam a existir.

# Herança

Composição, agregação e herança

A escolha entre composição e agregação depende do relacionamento que você deseja modelar entre os objetos.

Na prática, a composição tende a ser preferida quando a vida do objeto composto está intimamente ligada ao ciclo de vida da classe principal, enquanto a agregação é utilizada quando os objetos podem existir de forma independente.

Esses conceitos, juntamente com a herança, permitem criar modelos de software que são mais representativos do mundo real e mais flexíveis na manutenção e evolução do código.

Em toda relação "tipo" e "subtipo" é possível que o subtipo redefina o funcionamento de um método visível do tipo.

Basta repetir a assinatura do método no subtipo e codificar a nova implementação.

Também é possível usar a anotação `@Override` para marcar que aquela é uma redefinição ou sobrescrita de um comportamento do tipo.

java.lang.Object

## java.lang.Object

Em Java, todas as classes herdam de `java.lang.Object`, a qual define os métodos `hashCode`, `equals` e `toString`.

O método `toString` define a representação do estado de um objeto como String.

```
public class Pessoa {  
    private final String nome;  
    public Pessoa(String nome) {  
        this.nome = nome;  
    }  
    public String getNome() {  
        return nome;  
    }  
    @Override  
    public String toString() {  
        return "Pessoa{" +  
            "nome='" + nome + '\'' +  
            '}';  
    }  
}
```

# Exercicios

## Exercício de código

### **Exercício 1: Sistema de Gerenciamento de Biblioteca**

Definição das primeiras classes (Livro, Autor, Categoria) aplicando composição e encapsulamento.

Introdução à herança com subclasses como `LivroDigital` e `LivroFisico`.

### **Desafio**

Expandir o modelo inicial para incluir uma classe `Biblioteca`, com métodos para adicionar, remover e listar livros, aplicando herança e encapsulamento.

## Exercício 2: Classe ContaBancaria

Crie uma classe chamada ContaBancaria com os seguintes requisitos:

- **Atributos:** Deve ter um atributo para o saldo e outro para o número da conta. Pense nos modificadores de acesso corretos para garantir que o saldo só possa ser alterado por meio de métodos específicos.
- **Construtor:** O construtor deve inicializar a conta com um número e o saldo em zero.
- **Métodos:**
  - Crie um método para depositar um valor. O método deve ser capaz de lidar com valores de depósito inválidos (como valores negativos).
  - Crie um método para sacar um valor. Este método precisa garantir que o saldo não se torne negativo.
  - Crie um método para verificar o saldo.

## Exercício de código

### Exercício 3: Classe Calculadora

Crie uma classe chamada Calculadora com os seguintes requisitos:

- **Atributo:** Crie um atributo para armazenar o histórico de operações. Pense se ele deve ser acessível de fora da classe.
- **Métodos de Instância:**
  - Crie métodos de soma e subtração. Eles devem atualizar o histórico de operações.
  - Adicione um método para exibir o histórico.
- **Métodos Estáticos:**
  - Crie métodos de multiplicação e divisão. Eles não devem usar o histórico, pois a operação não pertence a uma instância específica da calculadora.
  - O método de divisão deve ter uma proteção contra divisão por zero.

Obrigada