

BOB JAVAPANTS



**INTRODUÇÃO À ORIENTAÇÃO A
OBJETOS EM JAVA**

Olá, marujo!

Bem-vindo ao mundo subaquático da programação orientada a objetos em Java, guiado por Bob Esponja e sua turma! Neste ebook, exploraremos cada conceito de forma clara e acessível, desde classes e objetos até os complexos relacionamentos entre eles. Prepare-se para se tornar um mestre da programação orientada a objetos em Java enquanto mergulhamos juntos nesse oceano de conhecimento!



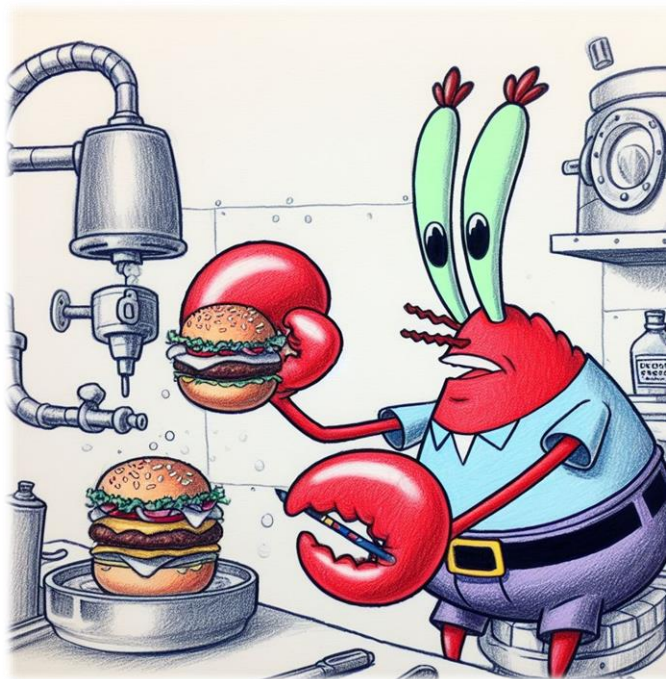
Vocês estão prontas, crianças ?



PROGRAMAÇÃO ORIENTADA A OBJETOS

Introdução à Orientação a Objetos

Antes de tudo, vamos lembrar que a programação procedural é como seguir uma receita de hambúrguer do Sr. Sirigueijo, passo a passo.



Por outro lado, a Orientação a Objetos é como construir uma cidade submarina em Fenda do Biquíni: você tem objetos, como casas e carros, que interagem entre si de maneira inteligente.

Introdução à Orientação a Objetos

Agora, por que escolher a Orientação a Objetos? Bom, imagine que você é o Patrick tentando organizar uma festa na rocha do Lula Molusco. Com a POO, você pode pensar em cada coisa como um objeto: o refrigerante, o karaokê, até mesmo os convites. Isso torna tudo mais organizado e fácil de entender, além de facilitar a reutilização de código e deixar as coisas mais flexíveis para mudanças. É como ter um Plankton que se adapta a qualquer situação!



Introdução à Orientação a Objetos

A seguir estão algumas vantagens do uso da Programação Orientada a Objetos (POO):

Modularidade: divisão do código em pedacinhos independentes (objetos), facilitando a organização e manutenção do programa.

Reutilização de Código: Com o uso de classes, podemos reaproveitar partes do nosso código em diferentes partes do programa.

Encapsulamento: Podemos esconder partes do nosso código que não queremos que ninguém veja, mantendo as coisas organizadas e seguras.

Escalabilidade: É possível adicionar novas partes ao nosso programa sem bagunçar o que já fizemos.

Abstração: nos permite criar modelos simplificados de objetos do mundo real por meio da descrição das suas características (atributos) e métodos (ações).



CLASSES E OBJETOS

Classes e Objetos

As Classes

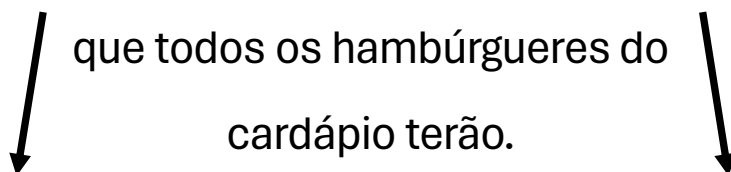
Uma classe é como o **plano** de uma casa na Fenda do Biquíni - é onde definimos as características (atributos) e comportamentos (métodos) dos objetos que vamos criar.

Imagine que estamos criando um programa para o restaurante do Sr. Siriguejo, onde vamos montar um cardápio que possui diferentes hambúrgueres de siri:

Classe "HambúrguerDeSiri":

Aqui definimos todas as

características e comportamentos



Atributos da Classe

Métodos da Classe

Classes e Objetos

As Classes

Atributos da Classe "HambúrguerDeSiri"

Vamos definir as características compartilhadas por todos os hambúrgueres de siri do cardápio:

- Nome – **Exemplo:** “X-Burguer de Siri”
- Tamanho – pequeno (**P**), médio (**M**) ou grande (**G**)
- Preço – **Exemplo:** R\$ 15,00
- Ingredientes – **Exemplo:** pão, carne, cheddar e bacon

Métodos da Classe "HambúrguerDeSiri"

Agora, definiremos as ações/comportamentos que todos os hambúrgueres podem realizar/sofrer:

- Imprimir informações gerais (nome, tamanho, preço e ingredientes)
- Adicionar Ingrediente
- Remover Ingrediente
- Calcular Preço

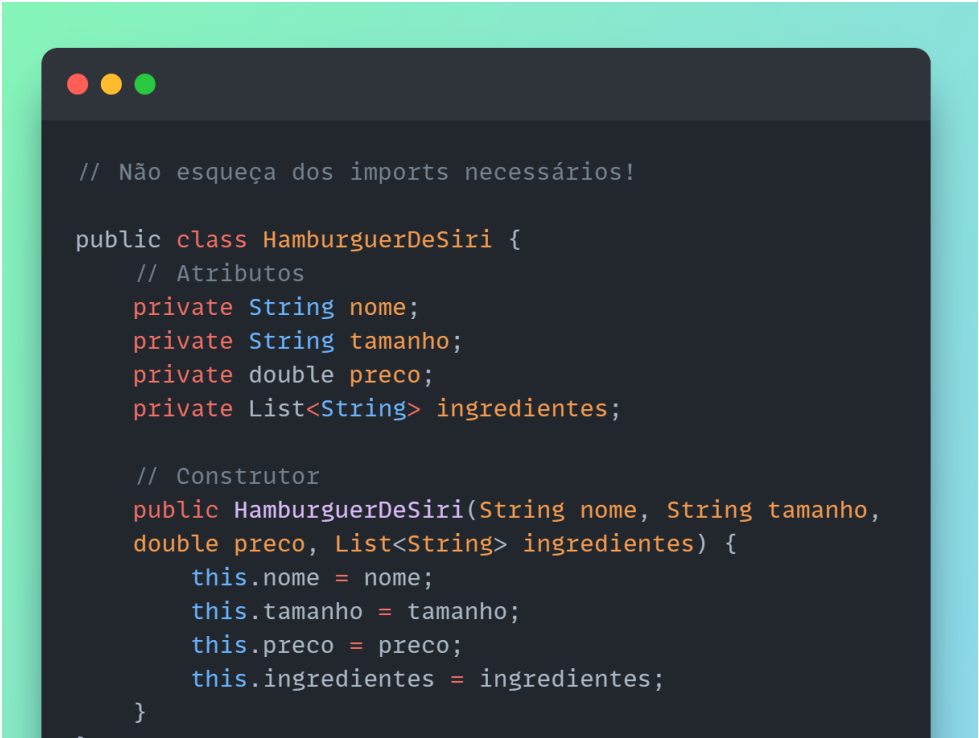
Classes e Objetos

Construtor

Definição: é um método especial dentro de uma classe que é chamado automaticamente quando um objeto é criado. Ele é utilizado para inicializar os atributos do objeto com os valores desejados.

No código...

1. Definição dos atributos e construtor



```
// Não esqueça dos imports necessários!

public class HamburgerDeSiri {
    // Atributos
    private String nome;
    private String tamanho;
    private double preco;
    private List<String> ingredientes;

    // Construtor
    public HamburgerDeSiri(String nome, String tamanho,
        double preco, List<String> ingredientes) {
        this.nome = nome;
        this.tamanho = tamanho;
        this.preco = preco;
        this.ingredientes = ingredientes;
    }
}
```

Classes e Objetos

No código...

2. Definição dos métodos

```
// Método para imprimir informações gerais do hambúrguer
public void imprimirInformacoes() {
    System.out.println("Hambúrguer: " + nome);
    System.out.println("Tamanho: " + tamanho);
    System.out.println("Preço: R$" + preco);
    System.out.println("Ingredientes:");
    for (String ingrediente : ingredientes) {
        System.out.println("- " + ingrediente);
    }
}

// Método para adicionar um ingrediente extra
public void adicionarIngrediente(String novoIngrediente) {
    ingredientes.add(novoIngrediente);
}

// Método para remover um ingrediente
public void removerIngrediente(String ingredienteRemovido) {
    ingredientes.remove(ingredienteRemovido);
}

// Método para calcular o preço
public double calcularPreco() {
    return preco;
}
}
```

(continuação do trecho de código anterior)

Classes e Objetos

Objetos

Agora, quando queremos adicionar um novo hambúrguer de siri ao cardápio, como o "Hambúrguer Siri Supremo" ou o "Hambúrguer Vegano de Siri", criamos objetos a partir da classe "HambúrguerDeSiri". Na linguagem técnica, isso significa que queremos **instanciar** um novo objeto.

Cada objeto terá seus próprios valores para os atributos, como tipo de pão, tamanho do hambúrguer, ingredientes especiais de siri, que serão únicos para aquele hambúrguer específico no cardápio do restaurante do Sr. Siriguejo.



Classes e Objetos

No código...

1. Instanciando um novo objeto:

```
// Não esqueça dos imports necessários!

public class Main {
    public static void main(String[] args) {
        // Criando uma lista de ingredientes para um hambúrguer específico
        List<String> ingredientes = new ArrayList<>();
        ingredientes.add("Carne de Siri");
        ingredientes.add("Alface");
        ingredientes.add("Queijo");
        ingredientes.add("Molho Especial");

        // Instanciando um objeto da classe HamburguerDeSiri
        HamburguerDeSiri burger = new HamburguerDeSiri("X-Burger", "G", 15.90, ingredientes);
    }
}
```

2. Utilizando os métodos da classe

```
// Imprimindo informações do hambúrguer
burger.imprimirInformacoes();

// Adicionando um ingrediente extra
burger.adicionarIngrediente("Bacon de Alga Marinha");

// Removendo um ingrediente
burger.removerIngrediente("Alface");

// Calculando o preço atualizado
double precoAtualizado = burger.calcularPreco();
System.out.println("Preço Atualizado: R$" + precoAtualizado);
}
```

Saída (Output):

```
Hambúrguer: X-Burger
Tamanho: G
Preço: R$15.9
Ingredientes:
- Carne de Siri
- Alface
- Queijo
- Molho Especial
Preço Atualizado: R$15.9
```



HERANÇA

Herança

Conceito de Herança

Na programação orientada a objetos, herança é um conceito importante. Ela permite que uma classe **herde** atributos e métodos de outra classe, chamada superclasse. Isso ajuda a reutilizar código e organizar hierarquicamente as classes.

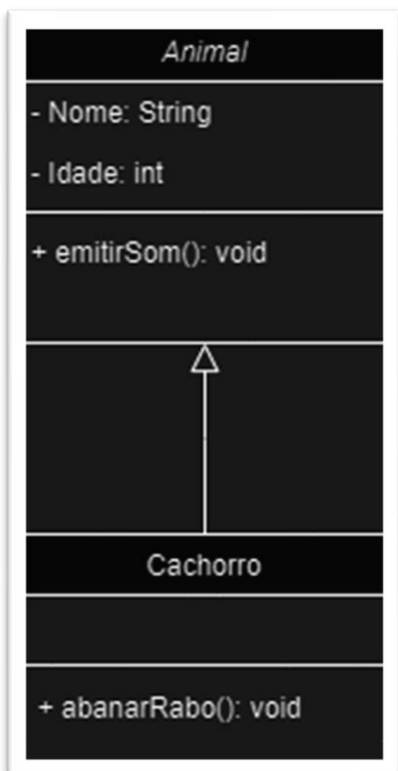
É como o Bob Esponja aprendendo habilidades do Lula Molusco: ele pode aprender a fazer hambúrgueres como o Lula e adicionar suas próprias receitas. Assim, a classe filha (Bob Esponja) herda os atributos e métodos de uma classe mãe (Lula Molusco).



Herança

Superclasses e Subclasses

Na herança, a classe que dá origem a outras é chamada de superclasse. A que herda as suas características, a classe filha, é também chamada de subclasse. Por exemplo, a classe "Animal" pode ser uma superclasse e "Cachorro" uma subclasse. O cachorro herda características comuns a todos os animais, como comer e dormir.



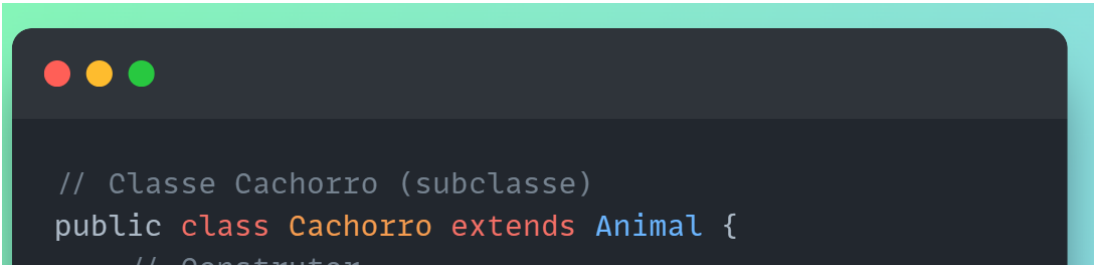
No exemplo ao lado, a subclasse “Cachorro” herda os atributos e métodos da superclasse “Animal”. Além disso, ela também adiciona um método próprio: `abanarRabo()`.

Herança

A palavra “extends” na linguagem Java

Em Java, usamos a palavra "extends" para mostrar que uma classe herda de outra. No exemplo anterior, para criar a subclasse "Cachorro" que estende a superclasse "Animal", usamos "public class Cachorro extends Animal". Isso indica que a classe Cachorro herda da classe Animal.

Em seguida, podemos adicionar novos atributos e métodos à classe filha, além de reutilizar os existentes da superclasse. É como construir um castelo de areia com moldes: a superclasse é o molde principal e a subclasse adiciona detalhes extras.



```
// Classe Cachorro (subclasse)
public class Cachorro extends Animal {
    // Construtor
```

Herança

No código...

Superclasse “Animal”:

```
// Classe Animal (superclasse)
public class Animal {
    // Atributos
    private String nome;
    private int idade;

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

    // Método para emitir som genérico
    public void emitirSom() {
        System.out.println("Emitindo som...");
    }
}
```

Subclasse “Cachorro”:

- Uso do “super()”
- Uso do “@Override”
- Adição de métodos próprios da classe filha: abanarRabo()

```
// Classe Cachorro (subclasse)
public class Cachorro extends Animal {
    // Construtor
    public Cachorro(String nome, int idade) {
        super(nome, idade);
        // Chamada ao construtor da superclasse Animal
    }

    // Método para emitir som específico do cachorro
    @Override
    public void emitirSom() {
        System.out.println("Latindo: Au au!");
    }

    // Método para abanar o rabo
    public void abanarRabo() {
        System.out.println("Abanando o rabo...");
    }
}
```

Herança

Uso do *Super()*

Em Java, o `super()` é usado principalmente para acessar membros da superclasse (classe mãe) dentro da subclasse (classe filha). No exemplo anterior, a subclasse “Cachorro” utilizou `super()` para reutilizar o construtor da superclasse “Animal” e, por isso, não foi necessário redeclarar os atributos repetidos dentro da classe filha.

Na Superclasse “Animal”:

```
public class Animal {  
    // Atributos  
    private String nome;  
    private int idade;  
  
    // Construtor  
    public Animal(String nome, int idade) {  
        this.nome = nome;  
        this.idade = idade;  
    }  
}
```

Na Subclasse “Cachorro”:

```
public class Cachorro extends Animal {  
    // Construtor  
    public Cachorro(String nome, int idade) {  
        super(nome, idade);  
        // Chamada ao construtor da superclasse Animal  
    }  
}
```



POLIMORFISMO

Polimorfismo

Conceito de Polimorfismo

Polimorfismo é como os ingredientes de um hambúrguer podem ser combinados de diferentes maneiras para criar receitas únicas. Na programação, isso significa que um objeto pode se comportar de várias formas, dependendo do contexto em que é utilizado.

Tipos de Polimorfismo

Existem dois tipos principais de polimorfismo: dinâmico e estático.



Polimorfismo

Polimorfismo Dinâmico (Sobrescrita)

O polimorfismo dinâmico, também conhecido como sobrescrita, ocorre quando uma subclasse implementa um método que já existe na sua superclasse.

```
// Superclasse Animal
public class Animal {
    // Método genérico para fazer um som
    public void fazerSom() {
        System.out.println("Emitindo som...");
    }
}

// Subclasse Cachorro
public class Cachorro extends Animal {
    // Sobrescrita do método fazerSom()
    // na subclasse Cachorro

    @Override
    public void fazerSom() {
        System.out.println("Au au!");
    }
}

// Subclasse Gato
public class Gato extends Animal {
    // Sobrescrita do método fazerSom()
    // na subclasse Gato

    @Override
    public void fazerSom() {
        System.out.println("Miau!");
    }
}
```

Polimorfismo

Uso do `@Override`

Já a palavra `@Override`, é utilizada para indicar que um método na subclasse está substituindo (ou sobrescrevendo) um método da superclasse com o mesmo nome e assinatura. No exemplo anterior, a subclasse “Cachorro” sobrescreveu o método “emitirSom()” da superclasse “Animal”, trocando a emissão de um som genérico para um som específico.

Na superclasse “Animal”:

```
// Método para emitir som genérico
public void emitirSom() {
    System.out.println("Emitindo som...");
}
```

Na subclasse “Cachorro”:

```
// Método para emitir som específico do cachorro
@Override
public void emitirSom() {
    System.out.println("Latindo: Au au!");
}
```

Polimorfismo

Polimorfismo Estático (Sobrecarga)

O polimorfismo estático, também conhecido como sobrecarga, ocorre quando temos vários métodos com o mesmo nome, mas diferentes parâmetros. Em outras palavras, é ter diferentes versões de um método onde cada um deles recebe e retorna algo diferente.

```
// Classe Animal (superclasse)
public class Animal {
    // Método genérico para fazer um som
    public void fazerSom() {
        System.out.println("Emitindo som ...");
    }
}

// Subclasse Cachorro
public class Cachorro extends Animal {
    // Sobrescrita do método fazerSom()
    // na subclasse Cachorro

    @Override
    public void fazerSom() {
        System.out.println("Au au!");
    }

    // Sobrecarga do método fazerSom()
    // com parâmetros
    public void fazerSom(String nome) {
        System.out.println(nome + ": Au au!");
    }
}
```




ENCAPSULAMENTO E MODIFICADORES DE ACESSO

Encapsulamento e modificadores de acesso

Importância do Encapsulamento

Encapsulamento é como quando guardamos os segredos de uma receita de hambúrguer em um cofre. Ele protege os detalhes internos de uma classe, como seus atributos e métodos, tornando-os acessíveis apenas de forma controlada. Dessa forma, essas informações só podem ser acessadas e modificadas por métodos específicos da classe, mantendo a integridade dos dados.



Encapsulamento e modificadores de acesso

Modificadores de Acesso

Os modificadores de acesso são como as chaves que controlam o acesso ao cofre dos ingredientes de um hambúrguer:

- **public:** Todos podem acessar e modificar livremente.
- **private:** Acesso restrito somente à própria classe.
- **protected:** Acesso permitido para a própria classe e suas subclasses.
- **default:** Acesso permitido apenas dentro do mesmo pacote.

Estes modificadores ajudam a garantir que apenas as partes necessárias do código possam interagir com os dados de uma classe, promovendo a segurança e a organização do código.

Encapsulamento e modificadores de acesso

Modificadores de Acesso

Neste e-book de Introdução a Orientação a Objetos em Java, exploraremos apenas dois modificadores de acesso, o *public* e o *private*.

O uso do *public*

No universo do Bob Esponja, o *public* é como compartilhar uma receita de hambúrguer com todos os moradores da Fenda do Biquíni. Quando um método ou atributo é declarado como público em Java, ele pode ser acessado por qualquer classe, em qualquer lugar do código, facilitando a interação entre diferentes partes do programa e promovendo a reutilização de código.

Encapsulamento e modificadores de acesso

O uso do private

Por outro lado, o *private* é como guardar a fórmula secreta do hambúrguer do Siri Cascudo. Quando um método ou atributo é declarado como privado em Java, ele só pode ser acessado pela própria classe em que está definido. Isso garante a segurança dos dados e evita que outras partes do programa interfiram indevidamente, mantendo a integridade e a coesão do código.



Encapsulamento e modificadores de acesso

No código...

```
public class ReceitaHamburguer {  
    // Atributos privados  
    private String ingredientePrincipal;  
    private int quantidade;  
  
    // Método público para definir ingrediente principal  
    public void definirIngredientePrincipal(String ingrediente, int quantidade) {  
        ingredientePrincipal = ingrediente;  
        quantidade = quantidade;  
        System.out.println("Ingrediente principal: " + ingrediente);  
        System.out.println("Quantidade: " + quantidade);  
    }  
  
    // Método público para obter o ingrediente principal  
    public String getIngredientePrincipal() {  
        return nomeIngredientePrincipal;  
    }  
}
```

Neste exemplo, a classe *ReceitaHamburguer* possui os atributos privados *ingredientePrincipal* e *quantidade*, que só podem ser acessados dentro da própria classe. Os métodos públicos *definirIngredientePrincipal()* e *getIngredientePrincipal()* permitem manipular esses atributos fora da classe.



AGRADECIMENTOS

Obrigada por ler até aqui!

Esse Ebook foi gerado por IA e diagramado por humano, visando propósitos didáticos e não lucrativos. O passo a passo se encontra no meu GitHub.

Esse conteúdo foi proposto pelo Desafio de Projeto: Criando um Ebook com IA, fornecido no Bootcamp Standander 2024 – Fundamentos de IA para Devs, disponível na plataforma da Dio.



GitHub

<https://github.com/danielle-soaress>



LinkedIn

<https://www.linkedin.com/in/danielle-soares-712910206/>

OBRIGADA, MARUJO!

