

# Abstração

## Objetivo da Aula

Entender o conceito de **abstração na Programação Orientada a Objetos (POO)** em Java, mostrando **como criar classes e métodos genéricos** que representam **ideias** do mundo real sem se prender aos detalhes de completos de implementação.

## Vantagens da Abstração

1.  **Simplifica o código** – Mostra só o que importa.
2.  **Facilita manutenção** – Mudanças afetam menos partes do sistema.
3.  **Reaproveita código** – Classes abstratas podem ser usadas em vários lugares.
4.  **Organiza melhor o projeto** – Cada classe tem sua função clara no sistema.
5.  **Permite polimorfismo** – Um mesmo método funciona para objetos diferentes.
6.  **Deixa o código mais claro** – Fica mais fácil entender o fluxo do sistema.
7.  **Ajuda nos testes** – Podemos simular partes do sistema de forma mais fácil.
8.  **Protege detalhes internos** – O usuário usa, mas não altera o que não deve, permite proteção.

## Problemas que a Abstração Resolve

1.  **Código repetido** → deixa tudo mais reaproveitável.
2.  **Muitos detalhes** → esconde o que não é importante.
3.  **Mudanças difíceis** → facilita trocar partes do sistema.
4.  **Código confuso** → deixa mais fácil de entender.
5.  **Classes muito presas umas às outras** → reduz dependências.
6.  **Sistema travado** → deixa o projeto mais flexível.
7.  **Difícil de testar** → permite usar simulações mais simples do sistema.

## Exemplo para o mundo real

### Classe → OncePerRequestFilter

Uma das classes abstratas **mais usadas no Spring Framework**.

Serve para criar **filtros customizados** que interceptam requisições HTTP **apenas uma vez por request** (ótimo para logs, autenticação, CORS, etc, centralizar requisições de todo o sistema).

```

public class LogRequestFilter extends OncePerRequestFilter {

    @Override
    protected void doFilterInternal(HttpServletRequest request,
                                    HttpServletResponse response,
                                    FilterChain filterChain)
        throws ServletException, IOException {
        System.out.println("Requisição recebida: " + request.getRequestURI());

        // Continua o fluxo normal da requisição
        filterChain.doFilter(request, response);
    }
}

```



## Abstração resolvida:

Você não precisa implementar toda a lógica, apenas sobrescrever o método `doFilterInternal()` — o Spring já abstrai a parte complexa do ciclo de requisição.

## AbstractController

Permite criar **controladores personalizados** sem precisar anotar com `@RestController`. Ideal para casos onde você quer **reutilizar comportamento comum** entre vários controllers.

```

public class MeuControllerBase extends AbstractController {

    @Override
    protected ModelAndView handleRequestInternal(HttpServletRequest request,
                                                HttpServletResponse response) {
        System.out.println("Request recebido em: " + request.getRequestURI());
        return new ModelAndView("home");
    }
}

```

## Abstração resolvida:

O Spring já implementa toda a lógica de controle HTTP — você só define o que quer fazer na requisição.

O Spring usa **muitas classes abstratas** que nós podemos **estender** para simplificar a criação de componentes reutilizáveis.

Aqui estão os **exemplos mais comuns e úteis**

🧠 Resumo — Classes Abstratas Mais Úteis no Spring Boot		
Classe Abstrata	Onde é usada	Para quê serve
OncePerRequestFilter	Web / Security	Interceptar requisições HTTP
AbstractController	MVC	Criar controladores personalizados
AbstractRoutingDataSource	DataSource	Multi-tenancy / banco dinâmico
AbstractCacheManager	Cache	Implementar caches customizados
AbstractAuthenticationProcessingFilter	Security	Criar autenticação customizada



# Introdução e Conceito de Abstração

**Objetivo:** contextualizar a abstração dentro dos pilares da POO.

- Pilares da POO: **Encapsulamento, Herança, Polimorfismo e Abstração**
- **Abstração:** é o ato de esconder detalhes complexos e mostrar apenas o essencial.
- Exemplo conceitual:

Um carro tem muitas peças, mas quando dirigimos, só pensamos em **acelerar, frear, ligar** — não em como o motor funciona.

## ✳ Definição simples:

“Abstrair é representar algo do mundo real em código, mostrando apenas o que é importante.”

## Classes e Métodos Abstratos

**Objetivo:** apresentar a sintaxe e como aplicar o conceito.

### 🧠 Quando usar Abstração

- Quando você quer **definir um comportamento genérico** que será **implementado de formas diferentes** por subclasses.
- Quando quer **proteger o programador** de detalhes desnecessários.

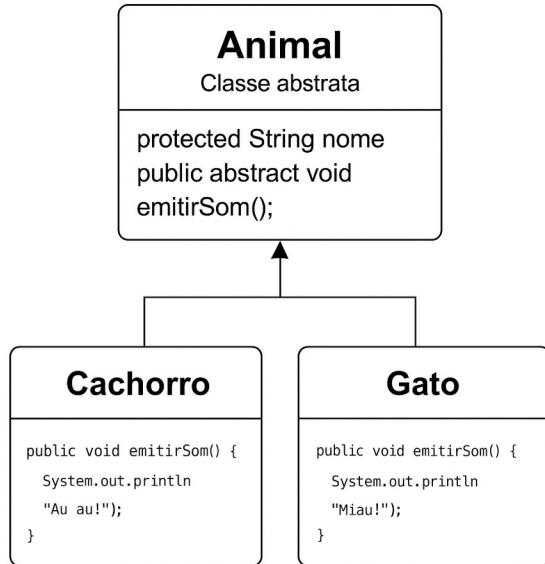
**VEJA EXEMPLO 1 NA IDE**

- **Abstract class** → A classe abstrata não deve ser **instanciada diretamente** por um motivo bem importante no design orientado a objetos:

### Porque ela é incompleta

Uma classe abstrata serve como **modelo ou esqueleto** para outras classes.

- **Abstract Method** → deve ser implementado pelas subclasses.
- **Protected atributos** → servem para estabelecer visibilidade para subclasses



## Exemplo de Código

**Objetivo:** mostrar a abstração aplicada em um caso real simples.

[VEJA EXEMPLO 2 NA IDE](#)

👤 **Explicação passo a passo:**

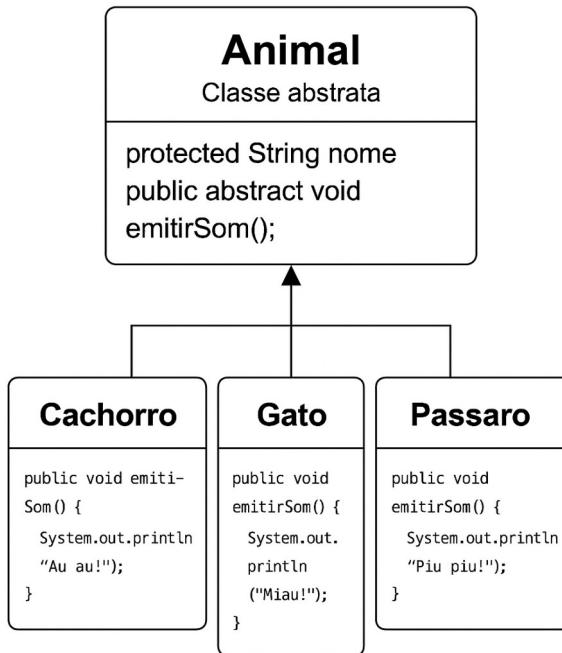
- Animal define **o que todo animal deve saber fazer** (`emitirSom`, `dormir`).
- Cachorro e Gato **implementam o comportamento de forma diferente**.
- Isso mostra **como abstração e polimorfismo** trabalham juntos.

## Demonstração Interativa

- O que mais poderia ser uma subclasse de Animal?"
- Adicione juntos uma nova classe Passaro que implementa `emitirSom()` com "Piu piu!".

[VEJA EXEMPLO 3 NA IDE](#)

## Demonstração Interativa



## Exercício

### Desafio rápido:

Crie uma classe abstrata Veiculo com:

- método **abstrato** mover()
- método **concreto** parar()

Depois, crie duas subclasses:

- Carro
- Bicicleta

E chame os métodos no main teste.

[Ver exemplo 4 na IDE](#)

**Fluxo de um Zoológico:** vamos a um exemplo de uma classe de negócio que recebe um animal qualquer e tenha um método de interação com o animal onde cada animal pode chamar seus métodos e comportamentos.

[Ver exemplo 5 na IDE](#)

## Conceitos aplicados

- **Abstração:** classe Animal define o modelo.
- **Herança:** Cachorro e Gato herdam de Animal.

- **Polimorfismo:** emitirSom() muda conforme o tipo real do objeto.
- **Encapsulamento protegido:** atributo nome é protected, acessível pelas subclasses.

## Conclusão

- Reforce: **Abstração = esconder detalhes e mostrar apenas o essencial.**
- **Melhora a manutenção e organização** do código.
- **Dica:** “Quanto mais o código reflete o mundo real, mais fácil ele será de entender e evoluir.”



## Vantagens de Usar Abstração em Java

### Reduz a complexidade do sistema

A abstração esconde os detalhes técnicos e expõe apenas o que é necessário.

→ Exemplo: você pode usar o método enviarMensagem() sem precisar saber como a mensagem é criptografada ou enviada por rede.

---

### Facilita a manutenção e evolução do código

Quando os detalhes estão isolados em classes concretas, é possível **alterar a implementação sem afetar o restante do sistema.**

→ Exemplo: mudar a forma como um BancoDeDados conecta sem alterar o código que usa essa abstração.

---

### Promove o reuso de código

Classes abstratas definem **contratos genéricos** que podem ser reutilizados em diferentes contextos.

---

### Melhora a organização do projeto

A abstração ajuda a estruturar o código de forma lógica, **separando responsabilidades.**

→ Exemplo: a classe Pagamento cuida da lógica geral, e subclasses como Pix, CartaoCredito, Boleto tratam dos detalhes específicos como: transferência, estorno, pagamento, emissão

---

### Facilita o polimorfismo

Permite criar código genérico que trabalha com diferentes implementações sem saber seus tipos concretos.

### Torna o código mais legível e intuitivo

Ao usar abstrações bem nomeadas, o código se torna mais próximo da linguagem do domínio do problema.

→ Exemplo: Transportadora.entregarPedido() é mais claro do que manipular detalhes de rota, caminhão e motorista.

---

## Facilita testes unitários

Ao abstrair dependências você pode criar **mocks** e testar partes do sistema isoladamente.

→ Exemplo: testar ServicoDeEmail simulando o envio sem precisar de um servidor real.

---

## Aumenta a segurança do código

A abstração **controla o acesso aos detalhes internos**, evitando que outras partes do sistema modifiquem diretamente o funcionamento interno de um objeto.

→ Exemplo: um usuário pode chamar carro.ligar(), mas não alterar diretamente o estado do motor.

---

## Suporte à extensão de funcionalidades

Novas classes concretas podem ser adicionadas sem precisar modificar o código existente que depende da abstração.

→ Exemplo: adicionar uma nova forma de pagamento Pix sem alterar a classe ProcessadorDePagamento.

---

## Alinha o código com o pensamento humano

Abstrações aproximam o código da forma como pensamos o mundo — **trabalhar com conceitos e não com detalhes técnicos**.

→ Isso facilita o entendimento entre desenvolvedores, analistas e até clientes.

## Tarefa de casa

### Sistema de Pagamentos Bancários (Abstração e Polimorfismo)

#### Objetivo da atividade

O objetivo desta atividade é aplicar os conceitos de **abstração, herança e polimorfismo** em um contexto real de um **sistema bancário**, onde diferentes formas de pagamento precisam ser processadas de maneira flexível e organizada.

#### Descrição do problema

Imagine que você foi contratado por um banco digital para desenvolver o **módulo de pagamentos** do novo sistema.

Esse banco precisa permitir que os clientes realizem **pagamentos de diferentes tipos**, como:

-  **PIX** — transferência instantânea.
-  **Boleto** — pagamento com geração de código e compensação.
-  **Cartão de crédito** — pagamento via operadora financeira.

O sistema deve ser capaz de **processar qualquer tipo de pagamento** de forma **genérica**, sem precisar saber qual é o tipo específico.

Ou seja, o sistema deve **receber um pagamento genérico**, e o Java deve ser capaz de **executar o comportamento correto automaticamente**, de acordo com o tipo de pagamento criado.

## Requisitos técnicos

1. Crie uma **classe abstrata Pagamento** que contenha:

- Um atributo `protected double valor`.
- Um **método abstrato** `efetuarPagamento()` (sem corpo).
- Um método **comum a todas as formas**, chamado `confirmarPagamento()`, que imprime uma mensagem genérica de confirmação.

2. Crie **três classes concretas (Subclasses)**:

- `PagamentoPIX`
- `PagamentoBoleto`
- `PagamentoCartao`

Cada uma deve **herdar** de `Pagamento` e **implementar o método abstrato** `efetuarPagamento()` com uma mensagem personalizada.

3. Crie uma classe **SistemaBancario** com um método `processarPagamento(Pagamento pagamento)` que:

- Exiba uma mensagem inicial de processamento.
- Chame o método `efetuarPagamento()` (o comportamento varia conforme o tipo).
- Chame `confirmarPagamento()` ao final.

4. No método `main()`:

- Crie um objeto `SistemaBancario`.
- Crie três pagamentos diferentes (PIX, Boleto e Cartão).
- Chame o método `processarPagamento()` para cada um.

## Conceitos que o aluno deve aplicar

- **Abstração:** criar uma classe base genérica (`Pagamento`).
- **Herança:** especializar o comportamento em subclasses (PIX, Boleto, Cartão).
- **Polimorfismo:** fazer o Java decidir dinamicamente qual método chamar.
- **Encapsulamento:** proteger o valor com `protected` e acessar via construtor.

## Entrega

O aluno deve entregar:

- O código completo do projeto Java.
- Um pequeno **comentário no topo do código** explicando, com suas palavras, onde foi aplicado **abstração, herança e polimorfismo**.