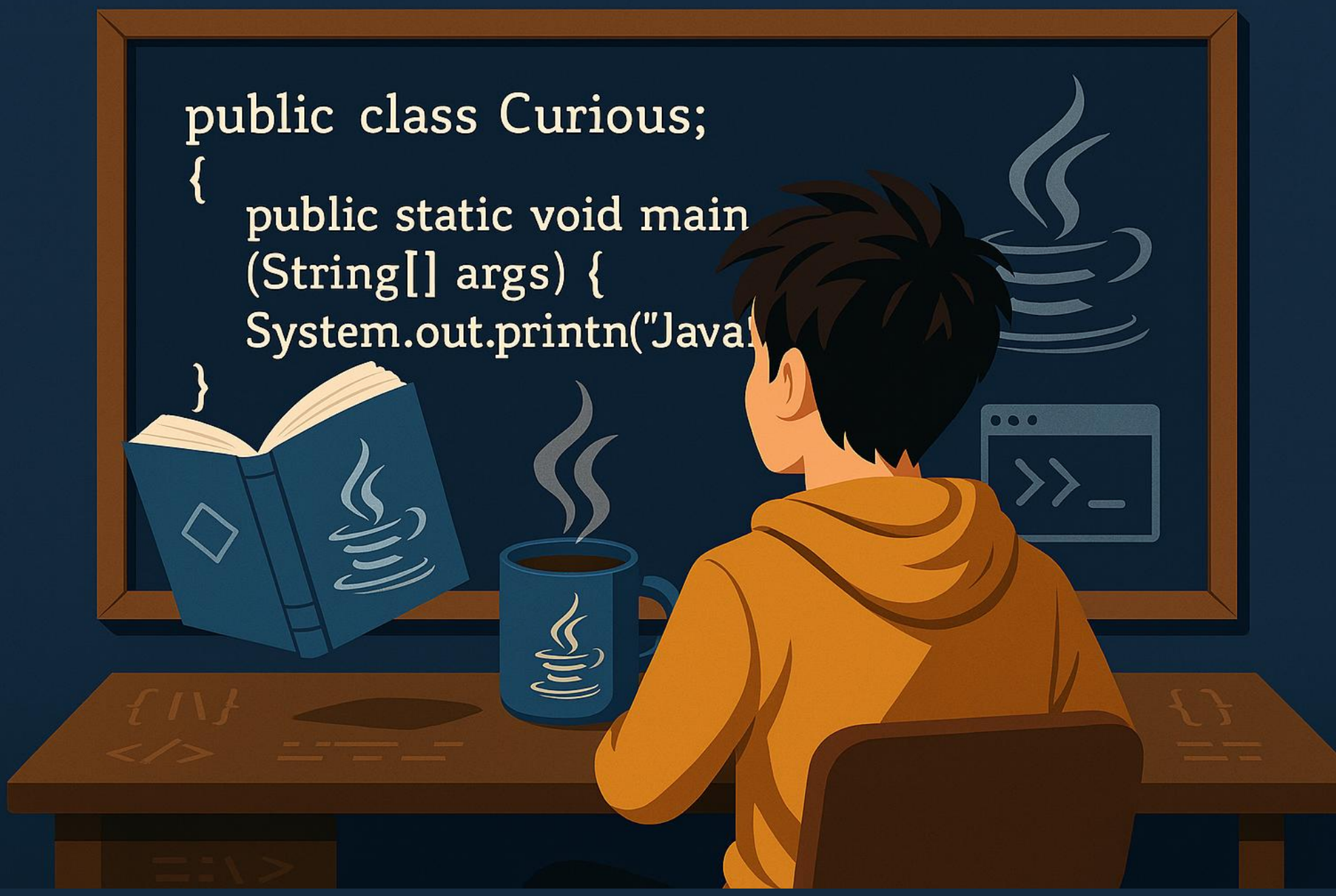
Java para Mentes Curiosas

O Guia Prático de Boas Práticas para o Java



Programe em Java seguindo as boas práticas

Fabrício de Araújo Santana

Introdução

POR QUE APLICAR BOAS PRÁTICAS EM JAVA?

Quando começamos a programar, o foco costuma ser “fazer funcionar”. Porém, com o tempo, percebemos que só isso não basta. O como o código é escrito importa — e muito. Java é uma linguagem orientada a objetos, amplamente usada em grandes sistemas, bancos, apps Android e servidores. Por isso, aplicar boas práticas é essencial para garantir que o código seja:

- **Claro:** fácil de entender por você e outros desenvolvedores;
- **Manutenível:** simples de modificar, corrigir ou evoluir;
- **Reutilizável:** com partes de código que podem ser reaproveitadas;
- **Seguro e confiável:** com menos chances de erros ou falhas ocultas.

CÓDIGO LIMPO É O CÓDIGO QUE DURA

A importância do Clean Code (Código Limpo)

Imagine um sistema feito sem organização, nomes confusos e lógica repetida. Difícil de entender, difícil de mexer. Agora imagine um código bem estruturado, com nomes claros, métodos curtos e responsabilidades separadas. Mais fácil, certo?

Seguir boas práticas não é “frescura”, é o que torna seu código profissional. Em projetos reais, isso faz toda a diferença para trabalhar em equipe, evitar bugs e manter o sistema funcionando por anos.

Java facilita, mas você precisa usar bem

O Java ajuda, mas temos que saber usar

A linguagem Java já oferece muitos recursos que incentivam boas práticas — como classes, interfaces, tratamento de exceções, collections, enums e muito mais. Mas é você quem decide como vai usar tudo isso.

Este eBook vai te ajudar a aplicar essas boas práticas no dia a dia com Java, com exemplos diretos e reais.
Vamos nessa?

01

Convenções de Código Java

Um bom código começa por nomes claros e padronizados. Em Java, existem convenções que ajudam a deixar o código mais legível, organizado e profissional. A seguir, veja os principais tipos de nomeação e como usá-los com exemplos reais.

NOMEAÇÃO EM JAVA

Porque dar bons nomes é tão importante?

Dar nomes não é fácil, mas é essencial na programação. Em Java, seguir boas convenções de nomeação torna o código mais claro, compreensível e fácil de manter. Nomes bem escolhidos ajudam tanto você quanto outros desenvolvedores a entenderem rapidamente o propósito de uma classe, método ou atributo. E isso Não é frescura, é prática profissional. Evite usar muitas abreviações, use um nome que seja de fácil compreensão. Um código com boa nomenclatura é mais limpo, colaborativo e duradouro.

NOMEAÇÃO DE CLASSES

Classes, Interfaces, Records, Enum

Comece com letra maiúscula e use **camelCase**. O nome deve ser um substantivo que represente bem o que a classe faz ou representa. O nome desse padrão é **PascalCase**

ExemplosNomenclatura.java

```
public class Usuario {}
public class CalculadoraFinanceira {}
```

NOMEAÇÃO DE VARIÁVEIS/ATRIBUTOS

Variáveis/Atributos, Variáveis final

Comece com letras em minúsculas e a cada nova palavra subsequente, a primeira letra deve ser escrita em maiúscula. Deve descrever claramente o que a variável representa. O nome desse padrão é **camelCase**

ExemplosNomenclatura.java

```
Usuario usuarioLogado = new Usuario();
int quantidadeProdutos = 10;
final double taxaJuros = 0.05;
```

NOMEAÇÃO DE MÉTODOS

Métodos, Procedimentos, Funções

Comece com verbo no infinitivo, as letras em minúsculas e a cada nova palavra subsequente, a primeira letra deve ser escrita em maiúscula. O nome desse padrão é **camelCase**

ExemplosNomenclatura.java

```
public void calcularTotal() {}
public String obterNomeCompleto() {}
```

NOMEAÇÃO DE CONSTANTES

Atributos static final

Esse tipo de nomenclatura pode gerar um pouco de confusão, quando declaramos uma variável ou atributo como final ela não representa uma constante global. Ela só é constante global quando tem o **static final.** Nesse tipo de atributo as letras são escritas em maiúsculas, com palavras separadas por _ (**underscore**). O nome desse padrão é UPPER_SNACK_CASE

ExemplosNomenclatura.java

```java
public static final int MAX_TENTATIVAS = 5;
public static final double PI = 3.1415;
```

02

Prioridades de Bons Projetos

O problema mais fundamental em Ciência de computação é a tarefa de decompor os problemas complexos em partes que possam ser resolvidas de forma independente - John Ousterhout

PRIORIDADES ESSENCIAIS EM PROJETOS

Por que elas são prioridades?

As prioridades de um bom projeto orientado a objetos são: **Integridade Conceitual**, **Ocultamento de Informações**, **Alta Coesão** e **Baixo Acoplamento**. Elas formam a base para qualquer sistema bem estruturado. Um projeto pode até funcionar sem seguir esses princípios, mas as chances de ser sustentável e de fácil manutenção são muito menores.

Essas prioridades não são regras obrigatórias, mas são diretrizes que nos ajudam a modelar sistemas de forma mais organizada, clara e eficiente. Quando bem aplicadas, facilitam a evolução do código, reduzem erros e promovem maior qualidade técnica.

INTEGRIDADE CONCEITUAL

Princípios que nos ajudam a melhorar os sistemas

O sistema deve ter coerência e padronização de funcionalidades, projeto e implementação. Ela é a consideração mais importante no projeto de sistemas, pois facilita o uso e entendimento de um sistema. Esse tópico trata da padronização nos projetos, onde ela vale para. Uma dica, Evite misturar estilos diferentes no mesmo sistema.

ExemploIntegridadeConceitual.java

```java
/*
 * Todos os métodos que fazem algo parecido
 * tem um nome parecido. Isso é a padronização
 */
public class Pedido {
    public double calcularPrecoFinal() { }
    public double calcularDesconto() { }
}
```

OCULTAMENTO DE INFORMAÇÃO

Princípios que nos ajudam a melhorar os sistemas

As classes precisam de um pouco de privacidade. As classes devem ocultar detalhes internos de sua implementação (usando modificador private), principalmente aqueles sujeitos à mudança. Adicionalmente, interfaces da classe deve ser estável

ExemploOcultamentoInformacao.java

```java
/*
 * O atributo senha não pode ser acessado diretamente
 */
public class Usuario {
    private String senha;

    public void alterarSenha(String novaSenha) {
        // validação antes de alterar
        this.senha = novaSenha;
    }
}
```

ALTA COESÃO

Princípios que nos ajudam a melhorar os sistemas

Cada classe deve ter uma única responsabilidade bem definida. Ou seja, seus métodos e atributos devem estar fortemente relacionados entre si. Isso também vale para outras unidades, como funções, métodos, pacotes entre outros.

ExemploAltaCoesao.java

```java
/* Um relatório pode gerar um PDF ou um
 * Excel, tendo alta coesão
 */
public class RelatorioFinanceiro {
    public void gerarPDF() { }
    public void exportarExcel() { }
}
```

INTEGRIDADE CONCEITUAL

Princípios que nos ajudam a melhorar os sistemas

As classes devem ser o mais independentes umas das outras. Isso porque nenhuma classe é uma ilha, pois classes dependem uma das outras, porém, a questão principal é a qualidade desse acoplamento. Um acoplamento “bom” seria uma classe que usa um serviço de outra classe. O estado da outra classe não interfere na classe que está usando seus serviços

ExemploAltaCoesao.java

```java
public interface Notificador {
    void enviar(String mensagem);
}

public class EmailService implements Notificador {
    public void enviar(String mensagem) { }
}

public class UsuarioService {
    private Notificador notificador;

    public void registrarUsuario(String nome) {
        // lógica de cadastro
        notificador.enviar("mensagem");
    }
}
```

03

Entendendo SOLID no Java

Quando o código começa a crescer, a bagunça também cresce — a menos que você use boas práticas. É aí que entra o SOLID, um conjunto de cinco princípios da programação orientada a objetos que ajudam a manter o código organizado, flexível e fácil de manter.

POR QUE APLICAR SOLID NO JAVA?

Princípios que nos ajudam a melhorar os sistemas

Java é amplamente usado em sistemas grandes e de longa vida útil. Aplicar os princípios do SOLID evita códigos engessados, difíceis de testar e complicados de manter. Com SOLID, seu sistema cresce de forma saudável e você se destaca como um desenvolvedor que escreve código limpo, modular e escalável.

Os princípios que serão mostrados são recomendações. Não devemos ser radicais e achar que eles são regras, eles nos guiam no que deveria ser feito, mas nem sempre isso é possível

OS PRINCIPIOS DO SOLID

O SOLID conta com 5 princípios

S – Single Responsibility Principle (SRP)

- Princípio de Responsabilidade Única

O – Open/Close Principle (OCP)

- Princípio Aberto-Fechado

L – Liskov Substitution Principle (LSP)

- Princípio de Substituição de Liskov

I – Interface Segregation Principle (ISP)

- Princípio e Segregação da Interface

D – Dependency Inversion Principle (DIP).

- Princípio da Inversão da Dependência

SINGLE RESPONSABILITY PRINCIPLE (SRP)

Princípio de Responsabilidade Única (SRP)

Declara que as entidades do software devem ser projetadas de maneira a permitir que novos componentes sejam adicionados sem a necessidade de modificar o código existente. Isso significa que, ao estender as funcionalidades do sistema, devemos ser capazes de fazer isso através de adição de novas classes ou módulos, em vez de alterar o código existente

ExemplosSolid.java

```java
// Correto: separação de responsabilidades
public class GeradorDeRelatorio {
    public void gerarPDF() { }
}

public class EnviadorDeEmail {
    public void enviar(String email) { }
}
```

OPEN/CLOSED PRINCIPLE (OCP)

Princípio Aberto-Fechado (OCP)

Declara que uma classe, deve ter apenas uma responsabilidade e, portanto, deve ter apenas um motivo para mudar. Cada classe deve ser responsável por fazer uma única tarefa e fazer bem. Isso ajuda a evitar acoplamento excessivo entre classes e torna o código mais modular e testável

ExemplosSolid.java

```java
// O código esta aberto para extensão, mas fechado para modificação.
public interface Desconto {
    double aplicar(double valor);
}

public class DescontoNatal implements Desconto {
    public double aplicar(double valor) {}
}

public class DescontoClienteVip implements Desconto {
    public double aplicar(double valor) {}
}
```

LISKOV SUBSTITUION PRINCIPLE (LSP)

Princípio de Substituição de Liskov (LSP)

Estabelece que uma classe derivada deve poder substituir sua classe base, mantendo a consistência do sistema. Se uma classe A é um subtipo de uma classe B, então os objetos do tipo B podem ser substituídos pelos objetos do tipo A sem que isso afete o funcionamento correto do sistema. Ou seja, a classe derivada deve ser capaz de atender a todas as pré-condições, pós-condições e invariantes definidos pela classe base

ExemplosSolid.java

```java
// O ideal seria que Pinguim não herdasse de Ave se não pode voar
public class Ave {
    public void voar() {}
}

public class Andorinha extends Ave {}

public class Pinguim extends Ave {
    // quebra o princípio
    public void voar() { }
}
```

INTERFACE SEGREGATION PRINCIPLE (ISP)

Princípio e Segregação da Interface (ISP)

Estabelece que uma classe não deve ser forçada a depender de interfaces que não sejam utilizadas por completo. Uma interface deve ser coesa e ter apenas o mínimo necessário para seus clientes. As interfaces devem ser segregadas de forma a cada cliente depender dos métodos que precisa utilizar, evitando assim a dependência de funcionalidades desnecessárias

ExemplosSolid.java

```java
public interface Trabalhador {
    void trabalhar();
    void comer(); // ruim se for um robô
}

public interface Trabalhador {
    void trabalhar();
}

public interface SerHumano {
    void comer();
}
```

DEPENDENCY INVERSION PRINCIPLE (DIP)

Princípio da Inversão da Dependência (DIP)

Propõe que as entidades de nível superior não devam depender diretamente das entidades de nível inferior. Em vez disso, ambas devem depender de abstrações. Isso permite que as dependências sejam invertidas, facilitando a extensibilidade, testabilidade e manutenção do código. Poderíamos chamar esse princípio de “Prefira Interfaces a Classe”, pois transmite melhor a sua ideia

ExemplosSolid.java

```java
public interface EnviadorMensagem {
    void enviar(String msg);
}

public class EmailService implements EnviadorMensagem {
    public void enviar(String msg) {}
}

public class Notificacao {
    private EnviadorMensagem enviador;

    public void notificar(String mensagem) {
        enviador.enviar(mensagem);
    }
}
```

Referências

Referências

- **Material do Professor José Roberto Chile Silva**
- **Material do Professor Marcelo Gomes de Paoli**

Esses materiais são de aula dos professores, disponibilizados para os alunos do Centro Universitário Senac – Santo Amaro, na graduação em Analista e Desenvolvedor de Sistemas.

Agradecimentos

OBRIGADO POR LER ATÉ AQUI

Esse ebook foi gerado com ajuda de IA, e diagramado totalmente por humano. O passo a passo está no meu Github

Esse conteúdo foi gerado para fins didáticos de construção, foi realizado uma validação, mas ainda pode conter erros sutis. Espero que gostei do ebook

https://github.com/Fabriciobr5975