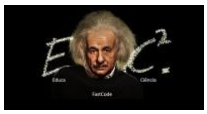


Boas Práticas no Desenvolvimento de Software com Java 17:

Um Guia Técnico Avançado para a Comunidade EducaCiência FastCode

No cenário atual de desenvolvimento de software, a eficiência, escalabilidade e manutenibilidade são fatores essenciais para o sucesso de qualquer projeto. Com o lançamento do Java 17, a linguagem oferece novos recursos e melhorias que tornam o código mais conciso, seguro e expressivo. No entanto, o uso adequado desses recursos é o que diferencia o código comum do código de qualidade superior.

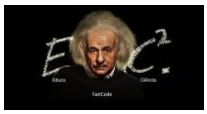
Este documento explora as melhores práticas recomendadas pela comunidade EducaCiência FastCode para o desenvolvimento de software robusto em Java 17, aplicando conceitos avançados de design de software, modularidade e automação de testes, sempre com foco na legibilidade, reusabilidade e confiabilidade do código.

1. Convenções de Nomeação e Padrões de Código

A clareza e a legibilidade do código são primordiais para equipes que trabalham em grandes sistemas. Seguir as Java Naming Conventions é essencial para que o código seja compreensível e de fácil manutenção.

Exemplo de padrão adequado:

```
Public class OrderService {

   Public double calculateOrderTotal(Order order) {

      Return order.getItems().stream()

            .mapToDouble(Item::getPrice)

            .sum();

   }

}
```



Aqui, OrderService e calculateOrderTotal seguem um padrão claro de nomeação, evidenciando a função de cada elemento

2. Aproveitamento dos Recursos Modernos do Java 17

O Java 17 traz melhorias significativas que devem ser utilizadas de maneira criteriosa, como o novo switch com expressões e sealed classes. Esses recursos reduzem a complexidade, eliminam código boilerplate e previnem erros em tempo de compilação.

```
Public String processPaymentStatus(PaymentStatus status) {
   Return switch (status) {
      Case PENDING -> “Payment is pending”;
      Case COMPLETED -> “Payment completed successfully”;
      Case FAILED -> “Payment failed”;
      Default -> throw new IllegalStateException(“Unexpected status: “ + status);
   };
}
```

3. Design de APIs com sealed classes

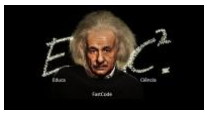
As sealed classes permitem um controle rigoroso sobre hierarquias de classe, garantindo que todas as subclasses possíveis sejam conhecidas em tempo de compilação. Isso melhora a segurança do código e facilita a manutenção.

```
Public sealed interface PaymentMethod
   Permits CreditCard, Paypal, BankTransfer {}
```

4. Princípios SOLID e Modularidade com JPMS

Os princípios do SOLID, aplicados em conjunto com o Java Platform Module System (JPMS), promovem um design orientado a responsabilidades claras, facilitando a escalabilidade e a manutenção dos sistemas.

```
Public class OrderProcessor {
   Private final PaymentService paymentService;
```



```
    Public OrderProcessor(PaymentService paymentService) {
        This.paymentService = paymentService;
    }

    Public void process(Order order) {
        paymentService.processPayment(order.getPayment());
    }
}
```

5. Imutabilidade e Modelagem de Dados com record

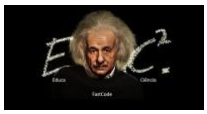
Em um ambiente multi-threaded, garantir que os dados não sejam modificáveis após a criação é uma prática importante. O record é uma estrutura concisa que permite a criação de objetos imutáveis com menos esforço.

```
Public record Customer(String name, String email) {}
```

6. Tratamento Adequado de Exceções

O tratamento eficiente de exceções é crucial para manter a estabilidade do sistema. Use checked exceptions para condições que o cliente da API pode tratar e unchecked exceptions para erros que são falhas de programação.

```
Public void processFile(String filePath) throws IOException {
    Try (BufferedReader reader = Files.newBufferedReader(Paths.get(filePath)))
{
        Reader.lines().forEach(System.out::println);
    } catch (NoSuchFileException e) {
        Throw new FileNotFoundException(“Arquivo não encontrado: “ + filePath);
    }
}
```



7. Automação de Testes e TDD

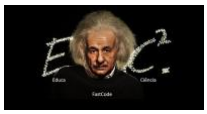
Testes automatizados, especialmente com JUnit 5 e Mockito, são uma prática essencial para garantir que o sistema funcione corretamente após modificações e expansões. O Test-Driven Development (TDD) ajuda a assegurar que o código seja sempre escrito com base nos requisitos reais.

```
Import org.junit.jupiter.api.Test;

Import static org.mockito.Mockito.*;

Public class OrderProcessorTest {

   @Test

   Void shouldProcessOrderCorrectly() {

      PaymentService paymentService = mock(PaymentService.class);

      OrderProcessor processor = new OrderProcessor(paymentService);

      Order order = new Order(...);

      Processor.process(order);


      Verify(paymentService).processPayment(order.getPayment());

   }

}
```

8. Streams API e Programação Funcional

A Streams API oferece uma maneira eficiente e declarativa de processar coleções de dados, tornando o código mais legível e permitindo o processamento paralelo com menos esforço.

```
Public List<String> getOrderDescriptions(List<Order> orders) {

   Return orders.stream()

      .filter(order -> order.getTotal() > 100)

      .sorted(Comparator.comparing(Order::getDate))

      .map(Order::getDescription)
```



```
        .collect(Collectors.toList());

}
```

9. Gerenciamento de Dependências com Maven e Gradle

O uso adequado de ferramentas de gerenciamento de dependências, como Maven e Gradle, garante que as bibliotecas externas sejam bem versionadas, seguras e rastreáveis, além de facilitar a integração contínua.

Xml

```
<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-lang3</artifactId>
    <version>3.12.0</version>
</dependency>
```

Conclusão

Seguir as melhores práticas descritas acima é crucial para garantir que o código desenvolvido em Java 17 atenda aos mais altos padrões de qualidade, escalabilidade e eficiência. A EducaCiência FastCode reforça esses princípios em seus conteúdos e práticas pedagógicas, promovendo a excelência técnica na construção de soluções robustas e de alta performance.