

Table of contents

Primeiros passos	3
Introdução: Instalar o IntelliJ	5
Introdução: Criando e Executando seu Primeiro Aplicativo Java	8
Sintaxe: Escrevendo e lendo informações	12
Sintaxe: Variáveis e tipos primitivos	15
Sintaxe: Condicionais em Java e Operadores Ternários	17
Sintaxe: Iteradores (loops)	20
OO: Introdução	25
OO: Classes	29
OO: Abstração	38
OO: Herança e Composição	43
OO: Polimorfismo, Overload e Override	50
POJOs, Entidades e Value Objects no DDD	59
OO: Classes Abstratas	71
Enums (Enumeradores)	83
Stream API e Manipulação de Listas e Arrays	97
Interfaces em Java: Guia para Iniciantes	108
Generics com Interfaces em Java: Guia para Iniciantes	112
Projeto Maven e Maven Repository: Guia para Iniciantes	116
Log4j com Interfaces e Generics em Java: Guia para Iniciantes	119
Guia de IO em Java: Operações Simples, Arquivos Grandes e Manipulação de JSON ..	123
Implementação de CRUD em Java Console Usando JDBC para Oracle	132
Introdução a JAX-RS com GRIZZLY	140
Resolução: API REST com JAX-RS e Grizzly	143
Tutorial Java: DDD Entities e Repositories vs. Beans/DAO	150
Clean Code e Aplicação em Java com Reflections e Anotações	154
Infraestrutura Comum de uma Aplicação Enterprise: CRUD, DDD, Oracle, Java JAX- RS e React	162
Guia de Instalação: Visual Studio e JetBrains Rider	165
Tutorial: Criando sua Primeira Aplicação Console em C#	169
Tutorial: Web Scraping em C# com HtmlAgilityPack	178
ASP.NET Core MVC: Introdução	181
ASP.NET Core MVC: Partial Views	188

Introdução Minimal API	198
OpenAPI e Swagger com Minimal API	211
Idempotência no .NET	216
Implementando JWT em uma Minimal API com ASP.NET Core	220

Primeiros passos

Olá mundo! Bem-vindo aos Tutoriais de Java! Este guia foi projetado para ajudá-lo a começar com a programação em Java e servir como referência para estudantes durante suas aulas de Java.

Pré-requisitos

Antes de mergulhar nos tutoriais, certifique-se de ter os seguintes pré-requisitos:

- Kit de Desenvolvimento Java (JDK) instalado em sua máquina
- Ambiente de Desenvolvimento Integrado (IDE) como Eclipse ou IntelliJ IDEA

Se você ainda não instalou o JDK ou uma IDE, consulte a documentação oficial para obter instruções.

Tópicos Abordados

Os Tutoriais de Java abrangem uma ampla gama de tópicos, incluindo:

1. Introdução ao Java
2. Variáveis e Tipos de Dados
3. Declarações de Fluxo de Controle
4. Programação Orientada a Objetos
5. Tratamento de Exceções
6. Manipulação de Arquivos
7. JDBC com oracle (CRUD)
8. Desenvolvimento de API's com JAX-RS
9. Clean Code

Cada tópico é explicado em detalhes com exemplos de código e exercícios para reforçar

seu entendimento.

Como Usar Este Guia

Para aproveitar ao máximo este guia, siga estas etapas:

1. Comece com o tutorial "Introdução ao Java" para se familiarizar com o básico.
2. Progrida pelos tutoriais em ordem sequencial para construir uma base sólida.
3. Pratique escrever código e experimentar diferentes conceitos.
4. Use os exemplos de código fornecidos em cada tutorial como referência durante suas aulas.
5. Se tiver alguma dúvida ou precisar de esclarecimentos adicionais, não hesite em perguntar ao seu instrutor ou colegas de classe.

Conclusão

Ao seguir este guia, você obterá uma compreensão sólida da programação em Java e estará bem preparado para suas aulas de Java. Vamos começar!

Feliz codificação!

Introdução: Instalar o IntelliJ

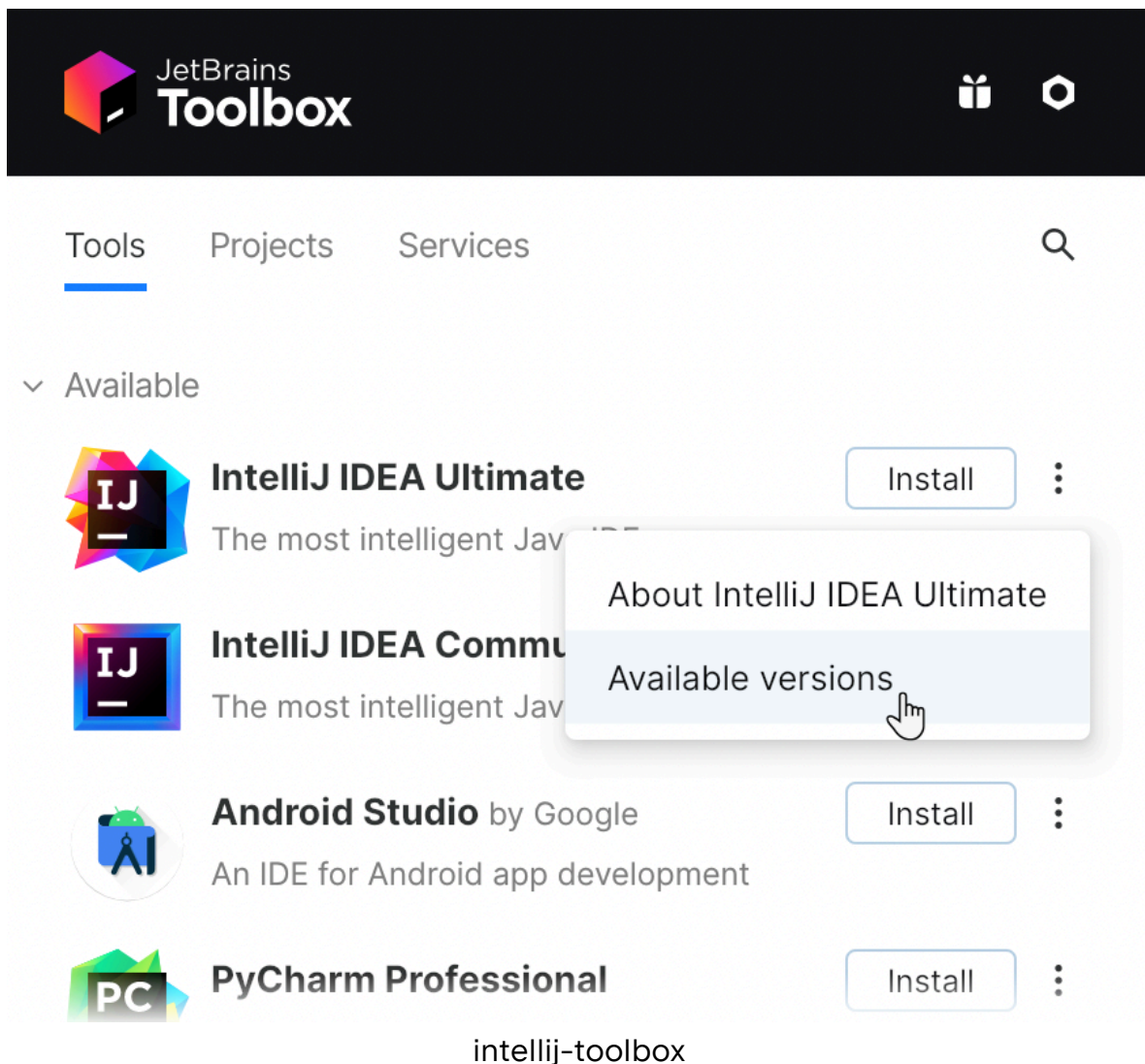
Baixando o IntelliJ para Projetos Java

Neste tutorial, vamos aprender como baixar o IntelliJ Ultimate ou Community Edition para desenvolver projetos Java.

Passo 1: Acessar o site do IntelliJ

Primeiro, acesse o site oficial do IntelliJ em <https://www.jetbrains.com/pt-br/idea/> (<https://www.jetbrains.com/pt-br/idea/>) ou pelo toolbox

<https://www.jetbrains.com/toolbox-app/> (<https://www.jetbrains.com/toolbox-app/>), recomende-se pelo toolbox para facilitar instalação

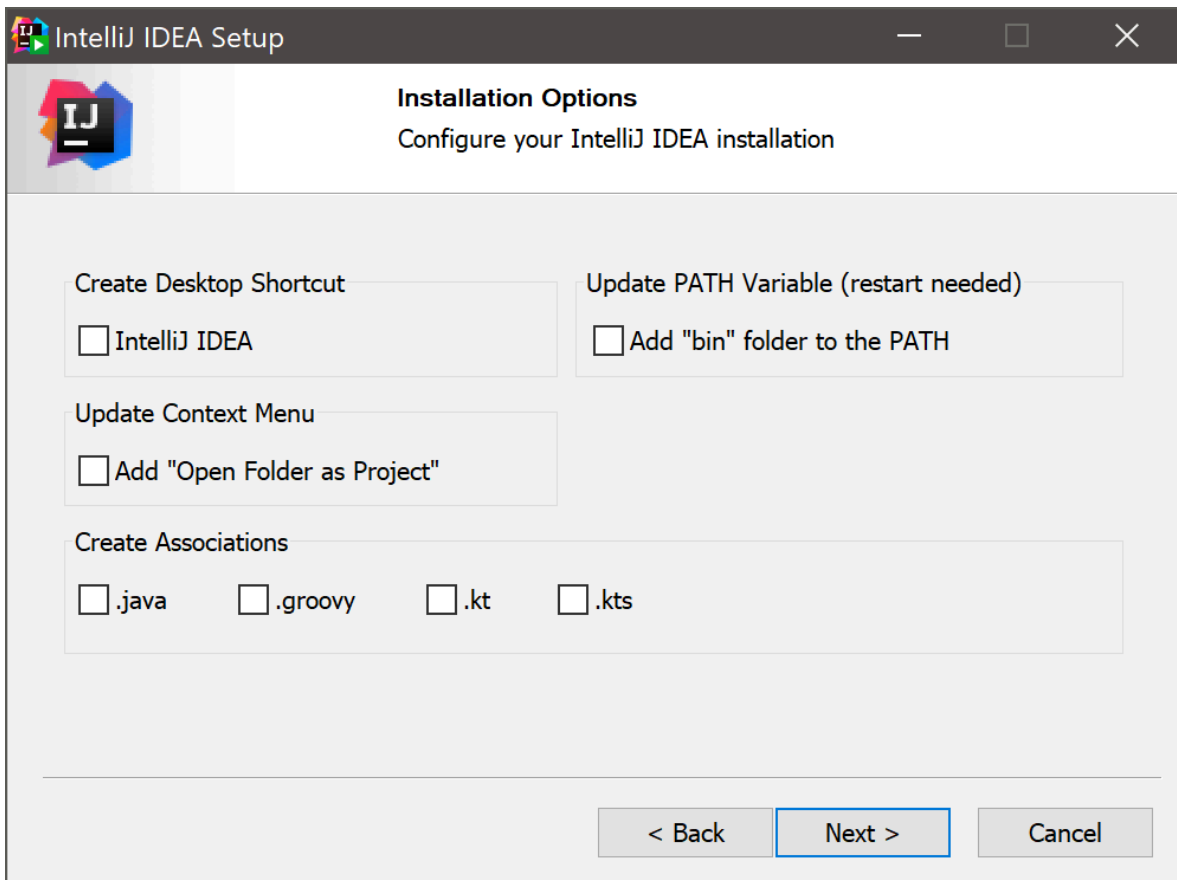


Passo 2: Escolher a edição

Na página inicial do site, você encontrará as opções para baixar o IntelliJ Ultimate e o IntelliJ Community Edition. Escolha a edição que melhor atenda às suas necessidades e clique no respectivo botão de download.

Passo 3: Executar o instalador

Após o download, execute o instalador do IntelliJ. Siga as instruções do assistente de instalação para concluir o processo. selecione todas as opções



alt text

Passo 4: Configurar o IntelliJ

Após a instalação, abra o IntelliJ e siga as etapas de configuração inicial, como a seleção do tema, plugins e configurações de idioma.

Passo 5: Começar a desenvolver

Agora você está pronto para começar a desenvolver projetos Java no IntelliJ. Explore os recursos e ferramentas disponíveis para tornar sua experiência de programação mais

produtiva.

See also

Outros Links

Site oficial do IntelliJ (<https://www.jetbrains.com/idea/download/?section=mac>)

Como instalar o IntelliJ (<https://www.jetbrains.com/help/idea/installation-guide.html>)

Introdução: Criando e Executando seu Primeiro Aplicativo Java

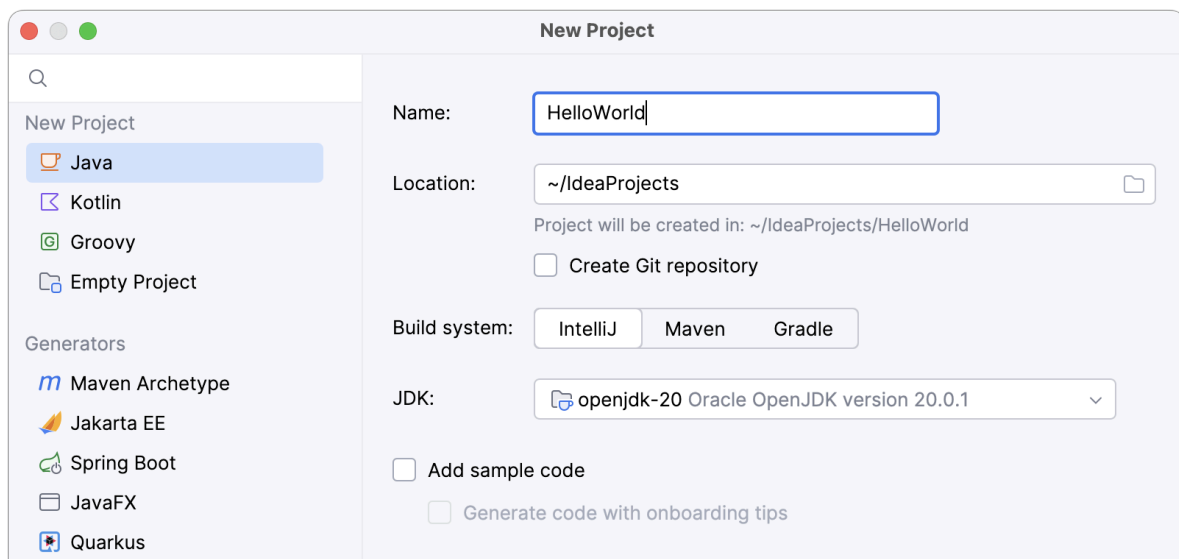
Neste tutorial, vamos aprender como criar e executar seu primeiro aplicativo Java usando o IntelliJ IDEA.

Pré-requisitos

Antes de começar, certifique-se de ter o IntelliJ IDEA instalado em seu sistema. Se você ainda não o tem, você pode baixá-lo aqui (<https://www.jetbrains.com/idea/download/>).

Passo 1: Criando um Novo Projeto

1. Abra o IntelliJ IDEA e clique em "Create New Project" na tela inicial.
2. Selecione "Java" na lista de opções à esquerda e clique em "Next".
3. Escolha um local para o seu projeto (ex: HelloWorld) e defina um nome para ele. Clique em "Finish" para criar o projeto.



new_java_project.png

Estrutura de Pasta de um Novo Projeto IntelliJ Java

Ao criar um novo projeto Java no IntelliJ IDEA, a estrutura de pasta padrão é criada automaticamente. Aqui está uma explicação do propósito de cada pasta:

- **src:** Esta pasta é onde você deve colocar todos os arquivos de código-fonte do seu projeto Java. Por padrão, ela contém uma pasta chamada "main" e uma pasta chamada "test". A pasta "main" é onde você coloca o código principal do seu aplicativo, enquanto a pasta "test" é onde você coloca os testes unitários.
- **out:** Esta pasta é onde os arquivos compilados do seu projeto são armazenados. Ela contém uma pasta chamada "production" que contém os arquivos compilados do código principal do seu aplicativo e uma pasta chamada "test" que contém os arquivos compilados dos testes unitários.
- **.idea:** Esta pasta contém as configurações do projeto do IntelliJ IDEA. Ela inclui arquivos como o arquivo de configuração do projeto (.iml), o arquivo de configuração do módulo (.xml) e outros arquivos de configuração específicos do IntelliJ IDEA.
- **.git:** Esta pasta é usada pelo sistema de controle de versão Git para armazenar os arquivos relacionados ao controle de versão do seu projeto. Ela inclui arquivos como o diretório de repositório (.git) e outros arquivos relacionados ao Git.
- **out:** Esta pasta é usada para armazenar arquivos de saída gerados pelo IntelliJ IDEA. Ela inclui arquivos como logs, relatórios de análise estática e outros arquivos de saída gerados durante o desenvolvimento do projeto.
- **.iml:** Este é o arquivo de configuração do módulo do IntelliJ IDEA. Ele contém informações sobre o módulo do projeto, como as dependências, as configurações de compilação e outras configurações específicas do módulo.

Essa é a estrutura de pasta básica de um novo projeto Java no IntelliJ IDEA. Você pode personalizar essa estrutura de acordo com as necessidades do seu projeto.

Passo 2: A classe main em Java

A classe `main` é uma parte fundamental de qualquer aplicativo Java. É nela que o programa começa a ser executado.

No exemplo fornecido, a classe `HelloWorld` contém o método `main`. Esse método é o ponto de entrada do programa, onde a execução começa. Ele possui a seguinte

assinatura:

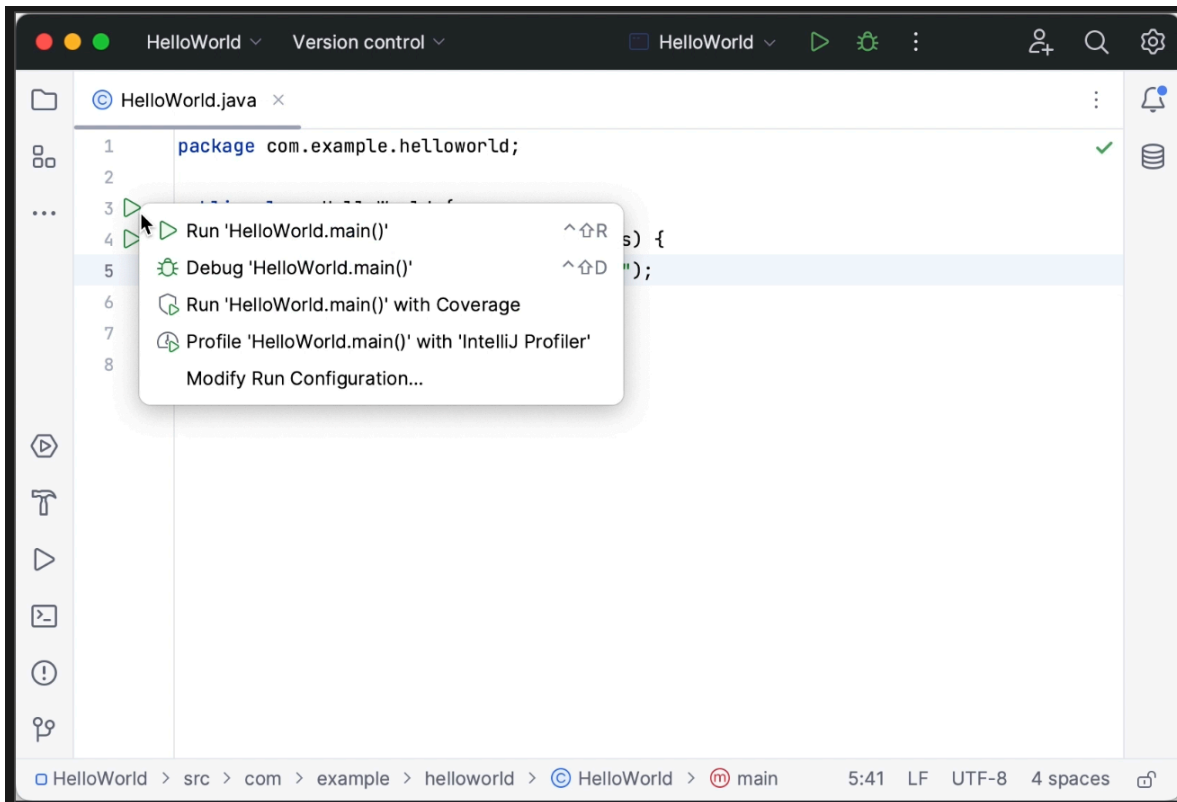
Aqui está uma explicação de cada parte da assinatura do método main:

- **public:** É um modificador de acesso que indica que o método pode ser acessado de qualquer lugar.
- **static:** É um modificador que indica que o método pertence à classe em si, e não a uma instância específica da classe.
- **void:** É o tipo de retorno do método main. Nesse caso, o método não retorna nenhum valor.
- **main:** É o nome do método. Esse nome é fixo e é o ponto de entrada padrão para a execução do programa.
- **String[] args:** É o parâmetro do método main. Ele permite que você passe argumentos para o programa quando ele é executado a partir da linha de comando. Neste exemplo, não estamos usando os argumentos, mas você pode acessá-los dentro do método se precisar.

Passo 3: Rodar o projeto

Para executar o código no IntelliJ IDEA, siga estas etapas:

1. Certifique-se de que o projeto esteja aberto no IntelliJ IDEA.
2. Abra o arquivo que contém a classe main que você deseja executar.
3. Clique com o botão direito do mouse dentro do corpo do método main.
4. Selecione a opção “Run ‘meu_primeiro_projeto_java.main()’”. Isso iniciará a execução do programa.



alt text

Você também pode executar o código pressionando a combinação de teclas Shift + F10 ou clicando no botão “Run” na barra de ferramentas superior.

Após a execução, você verá a saída do programa no painel de saída do IntelliJ IDEA. No exemplo fornecido, a saída será “Hello World!”.

See also

Outros Links

Tutorial Java (<https://www.jetbrains.com/help/idea/creating-and-running-your-first-java-application.html>)

Documentação do IntelliJ IDEA (<https://www.jetbrains.com/help/idea/>)

Sintaxe: Escrevendo e lendo informações

Os tutoriais são artigos orientados para aprendizado que ajudam os usuários a passar por um processo e alcançar um resultado final. Comece com uma introdução: para quem é este tutorial e o que o leitor irá alcançar ao lê-lo. Responda à pergunta: "Por que devo seguir este tutorial?"

Forneça um breve resumo do tutorial. Neste tutorial, você aprenderá como usar `println`, `printf` e `Scanner` em Java.

Parte 1: Usando println

Nesta parte, você aprenderá como usar o método `println` para imprimir saída no console.

1. Abra seu IDE Java ou editor de texto.
2. Crie um novo arquivo de classe Java.
3. Dentro do método `main`, adicione o seguinte código:

```
public class HelloWorld {  
    public static void main(String[] args) {  
        System.out.println("Olá, Mundo!");  
    }  
}
```

4. Salve o arquivo e compile-o.
5. Execute o programa e observe a saída no console.

Parte 2: Usando printf

Nesta parte, você aprenderá como usar o método `printf` para formatar e imprimir saída.

1. Abra seu IDE Java ou editor de texto.

2. Crie um novo arquivo de classe Java.
3. Dentro do método `main`, adicione o seguinte código:

```
public class HelloWorld {  
    public static void main(String[] args) {  
        String nome = "João";  
        int idade = 25;  
        double altura = 1.75;  
  
        System.out.printf("Nome: %s, Idade: %d, Altura: %.2f",  
            nome, idade, altura);  
    }  
}
```

4. Salve o arquivo e compile-o.
5. Execute o programa e observe a saída formatada no console.

Parte 3: Usando Scanner

Nesta parte, você aprenderá como usar a classe `Scanner` para ler entrada do usuário.

1. Abra seu IDE Java ou editor de texto.
2. Crie um novo arquivo de classe Java.
3. Dentro do método `main`, adicione o seguinte código:

```
import java.util.Scanner;  
  
public class HelloWorld {  
    public static void main(String[] args) {  
        Scanner scanner = new Scanner(System.in);  
  
        System.out.print("Digite seu nome: ");  
        String nome = scanner.nextLine();  
  
        System.out.print("Digite sua idade: ");
```

```
        int idade = scanner.nextInt();

        System.out.printf("Nome: %s, Idade: %d", nome, idade);
    }
}
```

4. Salve o arquivo e compile-o.

5. Execute o programa e digite seu nome e idade quando solicitado. Observe a saída no console.

O que você aprendeu

Neste tutorial, você aprendeu como usar `println`, `printf` e `Scanner` em Java para imprimir saída e ler entrada. Agora você tem o conhecimento básico para começar a construir programas Java mais complexos.

See also

Outros Links

Tutorial de escrita e leitura de informações em Java no Baeldung

(<https://www.baeldung.com/java-io>)

Documentação do IntelliJ sobre escrita e leitura de informações em Java

(<https://www.jetbrains.com/help/idea/working-with-files.html>)

Sintaxe: Variáveis e tipos primitivos

Este tutorial fornece uma visão geral de como as variáveis funcionam em Java, os tipos primitivos disponíveis e como armazená-los usando o Scanner.

Visão geral

Nesta parte do tutorial, você aprenderá sobre os tipos primitivos em Java e como declarar e inicializar variáveis. Os tipos primitivos incluem:

- `int`: para armazenar números inteiros.
- `double`: para armazenar números de ponto flutuante.
- `boolean`: para armazenar valores verdadeiro ou falso.
- `char`: para armazenar caracteres individuais.
- `String`: para armazenar sequências de caracteres.

Armazenando variáveis usando o Scanner

O Scanner é uma classe em Java que permite ler entrada do usuário. Aqui está um exemplo de como usar o Scanner para armazenar valores em variáveis:

1. Importe a classe Scanner no início do seu código:

```
import java.util.Scanner;
```

2. Crie uma instância do Scanner para ler a entrada do usuário:

```
Scanner scanner = new Scanner(System.in);
```

3. Solicite ao usuário que insira um valor e armazene-o em uma variável:

```
System.out.print("Digite um número inteiro: ");  
int numero = scanner.nextInt();
```

4. Repita o processo para outros tipos de variáveis, como `double`, `boolean`, `char` e `String`.

O que você aprendeu

Neste tutorial, você aprendeu sobre os tipos primitivos em Java, como declarar e inicializar variáveis e como armazenar valores usando o Scanner.

See also

Outros Links

Tipos de Dados em Java

(<https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html>)

Curso de Introdução ao Java na Codecademy

(<https://www.codecademy.com/learn/learn-java>)

Documentação Oficial do Java sobre Variáveis

(<https://docs.oracle.com/javase/specs/jls/se8/html/jls-4.html>)

Convenções de Nomenclatura em Java

(<https://www.oracle.com/java/technologies/javase/codeconventions-namingconventions.html>)

Java Variable Types (<https://www.geeksforgeeks.org/variables-in-java/>)

Sintaxe: Condicionais em Java e Operadores Ternários

Introdução

Este tutorial vai ensinar você a usar condicionais em Java, incluindo a estrutura if-else e o operador ternário. As condicionais permitem que seu programa execute diferentes partes do código com base em certas condições.

Objetivos

- Compreender a estrutura if-else em Java
- Utilizar operadores ternários para simplificar expressões condicionais

Estrutura if-else

O que é uma condicional if-else?

A estrutura if-else permite que seu programa execute um bloco de código se uma condição específica for verdadeira e outro bloco de código se a condição for falsa.

Sintaxe

```
if (condição) {  
    // código a ser executado se a condição for verdadeira  
} else {  
    // código a ser executado se a condição for falsa  
}
```

Exemplo

Vamos considerar um exemplo onde verificamos se uma pessoa é maior de idade:

```
int idade = 18;  
  
if (idade >= 18) {
```

```
        System.out.println("Você é maior de idade.");
    } else {
        System.out.println("Você é menor de idade.");
    }
}
```

Exercício Prático

Tente criar um programa que verifique se um número é positivo, negativo ou zero.

Operadores Ternários

O que é um operador ternário?

O operador ternário é uma forma concisa de escrever uma expressão if-else. Ele usa o símbolo `?` para separar a condição da expressão a ser executada se a condição for verdadeira e `:` para separar a expressão a ser executada se a condição for falsa.

Sintaxe

```
variável = (condição) ? expressãoSeVerdadeira : expressãoSeFalsa;
```

Exemplo

Vamos reescrever o exemplo da maioridade usando um operador ternário:

```
int idade = 18;
String mensagem = (idade >= 18) ? "Você é maior de idade." : "Você é menor de idade.";
System.out.println(mensagem);
```

Exercício Prático

Tente criar um programa que utilize o operador ternário para determinar se um número é par ou ímpar.

Conclusão

Neste tutorial, aprendemos sobre condicionais em Java e como usar o operador ternário para simplificar expressões. Pratique criando suas próprias condicionais e utilizando operadores ternários para fortalecer sua compreensão.

See also

Outros Links

Documentação Oficial do Java (<https://docs.oracle.com/javase/8/docs/api/>)

Tutorial de Estruturas de Controle em Java (<https://www.geeksforgeeks.org/decision-making-java-if-else-switch-break-continue-jump/>)

Curso de Java na Codecademy (<https://www.codecademy.com/learn/learn-java>)

Sintaxe: Iteradores (loops)

Loops são uma estrutura fundamental em qualquer linguagem de programação, permitindo a execução repetida de um bloco de código enquanto uma condição específica for verdadeira. Em Java, temos vários tipos de loops: `while`, `do-while`, `for`, e mais recentemente, expressões lambdas para manipulações mais funcionais.

1. Loop while

O loop `while` continua executando um bloco de código enquanto a condição fornecida é verdadeira.

Estrutura Básica

```
while (condição) {  
    // Código a ser executado repetidamente  
}
```

Exemplo

```
public class ExemploWhile {  
    public static void main(String[] args) {  
        int contador = 0;  
        while (contador < 5) {  
            System.out.println("Contador: " + contador);  
            contador++;  
        }  
    }  
}
```

Explicação

1. **Declaração da Condição:** O loop começa verificando a condição `contador < 5`.
2. **Execução do Bloco de Código:** Se a condição for verdadeira, o bloco de código dentro do loop é executado.

3. **Atualização da Condição:** Após a execução, o contador é incrementado.
4. **Repetição:** O loop verifica novamente a condição e repete os passos até que a condição seja falsa.

2. Loop do-while

O loop `do-while` é similar ao `while`, mas garante que o bloco de código seja executado pelo menos uma vez antes de verificar a condição.

Estrutura Básica

```
do {  
    // Código a ser executado repetidamente  
} while (condição);
```

Exemplo

```
public class ExemploDoWhile {  
    public static void main(String[] args) {  
        int contador = 0;  
        do {  
            System.out.println("Contador: " + contador);  
            contador++;  
        } while (contador < 5);  
    }  
}
```

Explicação

1. **Execução Inicial:** O bloco de código dentro do `do` é executado uma vez antes de qualquer verificação.
2. **Verificação da Condição:** Após a execução inicial, a condição `contador < 5` é verificada.
3. **Repetição:** Se a condição for verdadeira, o bloco de código é executado novamente.

3. Loop for

O loop `for` é frequentemente usado quando o número de iterações é conhecido. Ele combina inicialização, condição e atualização em uma única linha.

Estrutura Básica

```
for (inicialização; condição; atualização) {  
    // Código a ser executado repetidamente  
}
```

Exemplo

```
public class ExemploFor {  
    public static void main(String[] args) {  
        for (int i = 0; i < 5; i++) {  
            System.out.println("Contador: " + i);  
        }  
    }  
}
```

Explicação

1. **Inicialização:** `int i = 0` inicializa a variável de controle.
2. **Condição:** `i < 5` é verificada antes de cada iteração.
3. **Atualização:** `i++` é executado após cada iteração.

4. Novidades: Expressões Lambdas

Com a introdução de expressões lambdas no Java 8, manipulamos coleções de forma mais funcional e declarativa.

Estrutura Básica

```
lista.forEach(elemento -> {  
    // Código a ser executado para cada elemento  
})
```

```
});
```

Exemplo com Lambdas

```
import java.util.Arrays;
import java.util.List;

public class ExemploLambda {
    public static void main(String[] args) {
        List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5);
        numeros.forEach(numero -> System.out.println("Número: " +
numero));
    }
}
```

Explicação

1. **Criação da Lista:** `Arrays.asList(1, 2, 3, 4, 5)` cria uma lista de inteiros.
2. **Uso de `forEach`:** `numeros.forEach(numero -> ...)` itera sobre cada elemento da lista.
3. **Expressão Lambda:** `numero -> System.out.println("Número: " + numero)` é a função que será executada para cada elemento.

Conclusão

Loops são essenciais para automatizar tarefas repetitivas. Entender `while`, `do-while`, `for`, e expressões lambdas é fundamental para escrever código eficiente e elegante em Java. Experimente esses exemplos e adapte-os para suas necessidades!

See also

Outros Links

Tutorial de Java - W3Schools (https://www.w3schools.com/java/java_while_loop.asp)

Expressões Lambdas em Java - Oracle

(<https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html>)

Documentação Oficial do Java sobre Loops

(<https://docs.oracle.com/javase/tutorial/java/nutsandbolts/while.html>)

Java Loops - GeeksforGeeks (<https://www.geeksforgeeks.org/loops-in-java/>)

Java 8 Lambdas - Baeldung (<https://www.baeldung.com/java-8-lambda-expressions-tips>)

OO: Introdução

Introdução

A orientação a objetos (OO) é um paradigma de programação que surgiu na década de 1960 com a linguagem Simula. Esse paradigma organiza o software em torno de "objetos" que combinam dados e comportamentos. A linguagem de programação Java, criada pela Sun Microsystems em 1995, é uma das linguagens mais populares que utiliza esse paradigma.

Aplicações da Orientação a Objetos

A orientação a objetos é amplamente utilizada em diversas áreas, incluindo:

- **Desenvolvimento de Software Comercial:** ERP, CRM e sistemas de gerenciamento.
- **Desenvolvimento de Jogos:** Frameworks e motores de jogos que permitem criar personagens, cenários e interações complexas.
- **Aplicações Web e Móveis:** Aplicações robustas que necessitam de manutenção e escalabilidade.

Os 4 Pilares da Orientação a Objetos

Os quatro pilares fundamentais da orientação a objetos são: Abstração, Encapsulamento, Herança e Polimorfismo. Vamos explorar cada um deles em detalhe.

1. Abstração

A abstração foca em esconder os detalhes complexos e mostrar apenas as características essenciais de um objeto. Em outras palavras, abstração permite que você modele objetos do mundo real em termos de classes e atributos.

Exemplo

```
abstract class Animal {  
    abstract void fazerSom();  
}
```

```
class Cachorro extends Animal {  
    void fazerSom() {  
        System.out.println("O cachorro late");  
    }  
}
```

2. Encapsulamento

O encapsulamento é o mecanismo de restringir o acesso a alguns dos componentes de um objeto e pode ser alcançado usando modificadores de acesso. Isso melhora a segurança e a manutenção do código.

Exemplo

```
class Pessoa {  
    private String nome;  
    private int idade;  
  
    public String getNome() {  
        return nome;  
    }  
  
    public void setNome(String nome) {  
        this.nome = nome;  
    }  
  
    public int getIdade() {  
        return idade;  
    }  
  
    public void setIdade(int idade) {  
        this.idade = idade;  
    }  
}
```

3. Herança

A herança permite que uma classe (subclasse) herde atributos e métodos de outra classe (superclasse). Isso promove a reutilização do código.

Exemplo

```
class Animal {
    void dormir() {
        System.out.println("O animal está dormindo");
    }
}

class Cachorro extends Animal {
    void latir() {
        System.out.println("O cachorro está latindo");
    }
}
```

4. Polimorfismo

O polimorfismo permite que objetos de diferentes classes sejam tratados como objetos da mesma classe base. Existem dois tipos de polimorfismo: estático (sobrecarga) e dinâmico (sobrescrita).

Exemplo de Sobrecarga

```
class Calculadora {
    int somar(int a, int b) {
        return a + b;
    }

    double somar(double a, double b) {
        return a + b;
    }
}
```

Exemplo de Sobrescrita

```
class Animal {
    void fazerSom() {
        System.out.println("O animal faz um som");
    }
}
```

```
class Cachorro extends Animal {  
    void fazerSom() {  
        System.out.println("O cachorro late");  
    }  
}
```

Conclusão

A orientação a objetos é um paradigma poderoso que permite criar software modular, reutilizável e fácil de manter. Compreender os pilares da OO é fundamental para qualquer desenvolvedor Java. Experimente os exemplos e veja como esses conceitos se aplicam no desenvolvimento de software.

See also

Outros Links

Documentação Oficial do Java sobre Orientação a Objetos

(<https://docs.oracle.com/javase/tutorial/java/concepts/index.html>)

Pilares da Orientação a Objetos - GeeksforGeeks (<https://www.geeksforgeeks.org/four-pillars-of-object-oriented-programming/>)

Java OOP - W3Schools (https://www.w3schools.com/java/java_oop.asp)

Encapsulamento em Java - Baeldung (<https://www.baeldung.com/java-encapsulation>)

Polimorfismo em Java - Oracle

(<https://docs.oracle.com/javase/tutorial/java/landl/polymorphism.html>)

OO: Classes

Introdução

Em Java, uma classe é um modelo (ou blueprint) a partir do qual objetos são criados. Uma classe define atributos e comportamentos que os objetos dessa classe terão.

Estrutura de uma Classe

Uma classe em Java é composta por:

- **Atributos:** Também chamados de variáveis de instância, eles representam as propriedades de um objeto.
- **Métodos:** Também chamados de funções, eles definem os comportamentos que um objeto pode realizar.
- **Construtores:** Métodos especiais usados para inicializar objetos.
- **Getters e Setters:** Métodos para acessar e modificar os atributos de um objeto.
- **Controles de Acesso:** Definem a visibilidade dos atributos e métodos (público, privado, protegido).

Estrutura Básica

```
public class NomeDaClasse {  
    // Atributos  
    tipo atributo1;  
    tipo atributo2;  
  
    // Construtor  
    public NomeDaClasse(tipo atributo1, tipo atributo2) {  
        this.atributo1 = atributo1;  
        this.atributo2 = atributo2;  
    }  
  
    // Métodos
```

```

    public void metodo1() {
        // Implementação
    }

    public tipo getAtributo1() {
        return atributo1;
    }

    public void setAtributo1(tipo atributo1) {
        this.atributo1 = atributo1;
    }
}

```

Construtores

Construtores são métodos especiais que são chamados quando um objeto é instanciado. Eles têm o mesmo nome da classe e não têm um tipo de retorno.

Exemplo de construtor

```

public class Musica {
    // Atributos
    private String titulo;
    private String artista;
    private int duracao; // duração em segundos

    // Construtor
    public Musica(String titulo, String artista, int duracao) {
        this.titulo = titulo;
        this.artista = artista;
        this.duracao = duracao;
    }

    // Métodos
    // ...
}

```

Getters e Setters

Getters e setters são métodos que permitem o acesso e a modificação dos atributos de um objeto. Eles seguem uma convenção de nomenclatura específica: get para obter o valor e set para modificar o valor.

```
// Getters
public String getTitulo() {
    return titulo;
}

public String getArtista() {
    return artista;
}

public int getDuracao() {
    return duracao;
}

// Setters
public void setTitulo(String titulo) {
    this.titulo = titulo;
}

public void setArtista(String artista) {
    this.artista = artista;
}

public void setDuracao(int duracao) {
    this.duracao = duracao;
}

// Outros métodos
// ...
```

Exemplo 1: Classe Musica

Vamos criar uma classe `Musica` que tem atributos como `titulo`, `artista`, `duracao` e métodos para obter essas informações.

Definição da Classe Musica

```
public class Musica {
    // Atributos
    private String titulo;
    private String artista;
    private int duracao; // duração em segundos

    // Construtor
    public Musica(String titulo, String artista, int duracao) {
        this.titulo = titulo;
        this.artista = artista;
        this.duracao = duracao;
    }

    // Métodos
    public String getTitulo() {
        return titulo;
    }

    public void setTitulo(String titulo) {
        this.titulo = titulo;
    }

    public String getArtista() {
        return artista;
    }

    public void setArtista(String artista) {
        this.artista = artista;
    }

    public int getDuracao() {
        return duracao;
    }

    public void setDuracao(int duracao) {
```



```

        this.duracao = duracao;
    }

    public void tocar() {
        System.out.println("Tocando " + titulo + " de " +
artista);
    }
}

```

Uso da Classe Musica

```

public class TesteMusica {
    public static void main(String[] args) {
        Musica musica = new Musica("Bohemian Rhapsody", "Queen",
354);
        musica.tocar();
        System.out.println("Título: " + musica.getTitulo());
        System.out.println("Artista: " + musica.getArtista());
        System.out.println("Duração: " + musica.getDuracao() + "
segundos");
    }
}

```

Exemplo 2: Classe CartaMagic

Nome do card


Linha de tipo

Informa o tipo do card: artefato, criatura, encantamento, mágica instantânea, terreno, feitiço ou planeswalker. Se o card tem um *subtipo* ou *supertipo*, isso também é listado aqui. Por exemplo, Piromante Pródigo é uma criatura, e seus subtipos são os tipos de criatura Humano e Mago.

Caixa de texto

É aí que aparecerão as *habilidades* do card. Você também pode encontrar texto ilustrativo impresso em *itálico* (*assim*) que traz alguma informação sobre o mundo de **Magic**. O texto ilustrativo não tem efeito no jogo. Algumas habilidades têm um *texto explicativo* em *itálico* para ajudá-lo a lembrar de seus efeitos.

Custo de mana

Mana é o principal recurso do jogo. É produzido pelos terrenos e você pode gastá-lo para jogar *mágicas*. Os símbolos no canto superior direito de um card informam o custo para jogar aquela mágica. Se o custo de mana indica , você deve pagar dois manas de qualquer tipo mais um mana vermelho (de uma Montanha) para jogá-la.

Símbolo da expansão

Esse símbolo indica à qual coleção de **Magic** aquele card pertence. Por exemplo, o símbolo da expansão *Décima Edição* é **X**. A cor do símbolo mostra a *raridade* do card: preto para os cards comuns, prateado para os incomuns, dourado para os raros e alaranjado para os míticos raros.

Número na coleção

O número na coleção facilita a organização dos seus cards. Por exemplo, "221/383" significa que o card é o 221º de 383 cards na coleção.

Poder e resistência

Todas as criaturas têm uma caixa especial com seu poder e sua resistência. O poder de uma criatura (o primeiro número) é a quantidade de dano que ela causa em combate. A sua resistência (o segundo número) é a quantidade de dano que ela deve sofrer em um único turno para ser destruída.



magic_card.png

Vamos criar uma classe `CartaMagic` que tem atributos como `nome`, `tipo`, `custo`, `poder`, `resistencia` e métodos para obter essas informações.

Definição da Classe `CartaMagic`

```
public class CartaMagic {  
    // Atributos  
    private String nome;  
    private String tipo;  
    private int custo;  
    private int poder;  
    private int resistencia;  
  
    // Construtor  
    public CartaMagic(String nome, String tipo, int custo, int  
poder, int resistencia) {  
        this.nome = nome;  
    }  
}
```

```
        this.tipo = tipo;
        this.custo = custo;
        this.poder = poder;
        this.resistencia = resistencia;
    }

    // Métodos
    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public String getTipo() {
        return tipo;
    }

    public void setTipo(String tipo) {
        this.tipo = tipo;
    }

    public int getCusto() {
        return custo;
    }

    public void setCusto(int custo) {
        this.custo = custo;
    }

    public int getPoder() {
        return poder;
    }

    public void setPoder(int poder) {
        this.poder = poder;
    }
}
```

```

    public int getResistencia() {
        return resistencia;
    }

    public void setResistencia(int resistencia) {
        this.resistencia = resistencia;
    }

    public void exibirDetalhes() {
        System.out.println(nome + " - " + tipo);
        System.out.println("Custo: " + custo);
        System.out.println("Poder: " + poder);
        System.out.println("Resistência: " + resistencia);
    }
}

```

Uso da Classe CartaMagic

```

public class TesteCartaMagic {
    public static void main(String[] args) {
        CartaMagic carta = new CartaMagic("Serra Angel",
        "Criatura", 5, 4, 4);
        carta.exibirDetalhes();
    }
}

```

Conclusão

Classes são a base da programação orientada a objetos em Java. Elas permitem a criação de objetos com atributos e comportamentos específicos. Compreender como definir e usar classes é fundamental para qualquer desenvolvedor Java. Experimente os exemplos e veja como esses conceitos se aplicam no desenvolvimento de software.

See also

Outros Links

Métodos em Java - W3Schools (https://www.w3schools.com/java/java_methods.asp)

Construtores em Java - GeeksforGeeks (<https://www.geeksforgeeks.org/constructors-in-java/>)

Encapsulamento em Java - Baeldung (<https://www.baeldung.com/java-encapsulation>)

Documentação Oficial do Java sobre Classes

(<https://docs.oracle.com/javase/tutorial/java/javaOO/classes.html>)

OO: Abstração

Introdução

A abstração é um dos pilares fundamentais da programação orientada a objetos. Em termos simples, abstração é o processo de esconder os detalhes complexos de uma implementação e mostrar apenas as funcionalidades essenciais de um objeto. Ela permite que os desenvolvedores trabalhem em um nível mais alto de complexidade sem se preocupar com os detalhes internos.

O Conceito de Abstração

Definição

Abstração se refere ao ato de representar características essenciais sem incluir a complexidade de fundo. Em Java, isso é frequentemente feito usando classes abstratas e interfaces.

Exemplos de Abstração

- **Abstração tradicional:** Uma classe com as propriedades e métodos implementadas é realizada a abstração.
- **Classe Abstrata:** Uma classe que não pode ser instanciada diretamente e pode conter métodos abstratos que não têm implementação.
- **Interface:** Um contrato que define um conjunto de métodos que uma classe deve implementar.

Abstração em Java

Vimos no outro capítulo o processo de abstrair algo do mundo real para o mundo virtual.

Classe Abstrata

Uma classe abstrata é uma classe que não pode ser instanciada. Ela pode conter métodos abstratos (sem implementação) e métodos concretos (com implementação).

```
abstract class Animal {  
    // Método abstrato
```

```

    abstract void fazerSom();

    // Método concreto
    void dormir() {
        System.out.println("O animal está dormindo");
    }
}

class Cachorro extends Animal {
    void fazerSom() {
        System.out.println("O cachorro late");
    }
}

```

Interface

Uma interface é um contrato que especifica um conjunto de métodos que uma classe deve implementar. Diferente das classes abstratas, interfaces não podem ter qualquer implementação de métodos (até Java 8, que introduziu métodos padrão e estáticos).

```

interface Animal {
    void fazerSom();
}

class Cachorro implements Animal {
    public void fazerSom() {
        System.out.println("O cachorro late");
    }
}

```

Dicas e Recomendações para Fazer Abstração

Dicas Gerais

1. **Foque nas Funcionalidades Essenciais:** Concentre-se em representar apenas as características e comportamentos essenciais do objeto.
2. **Use Interfaces para Contratos Comuns:** Utilize interfaces para definir contratos que

podem ser implementados por diferentes classes.

3. Utilize Classes Abstratas para Compartilhar Código: Use classes abstratas quando precisar compartilhar código comum entre várias classes.

4. Evite o Uso Excessivo de Abstrações: Embora a abstração seja poderosa, o uso excessivo pode tornar o código difícil de entender e manter. Use-a de maneira equilibrada.

Recomendações de Autores Renomados

Robert C. Martin (Uncle Bob)

No livro "Clean Code: A Handbook of Agile Software Craftsmanship", Robert C. Martin enfatiza a importância de manter o código limpo e fácil de entender. Ele sugere:

- **Mantenha as Abstrações Simples:** Não complique demais suas classes abstratas ou interfaces.
- **Prefira Interfaces a Classes Abstratas:** Sempre que possível, use interfaces para definir contratos.

Joshua Bloch

Em "Effective Java", Joshua Bloch oferece diversas recomendações sobre o uso de abstrações:

- **Use Interfaces para Definir Tipos:** Utilize interfaces para definir tipos que podem ter múltiplas implementações.
- **Evite Quebra de Contratos:** Certifique-se de que todas as classes que implementam uma interface seguem o contrato definido pela interface.

Martin Fowler

No livro "Patterns of Enterprise Application Architecture", Martin Fowler discute a importância dos padrões de design e abstrações:

- **Use Abstrações para Flexibilidade:** Abstrações podem tornar o sistema mais flexível e adaptável a mudanças futuras.

- **Cuidado com Overengineering:** Evite criar abstrações desnecessárias que podem complicar o sistema.

Exemplo Completo de Abstração

Exemplo de Classe Abstrata

```
abstract class Veiculo {
    abstract void acelerar();

    void frear() {
        System.out.println("O veículo está freando");
    }
}

class Carro extends Veiculo {
    void acelerar() {
        System.out.println("O carro está acelerando");
    }
}
```

Exemplo de Interface

```
interface Pagamento {
    void processarPagamento(double valor);
}

class PagamentoCartaoCredito implements Pagamento {
    public void processarPagamento(double valor) {
        System.out.println("Processando pagamento de R$" + valor +
            " via cartão de crédito");
    }
}

class PagamentoPaypal implements Pagamento {
    public void processarPagamento(double valor) {
        System.out.println("Processando pagamento de R$" + valor +
            " via Paypal");
    }
}
```

```
}  
}
```

Conclusão

A abstração é uma ferramenta poderosa na programação orientada a objetos, permitindo que os desenvolvedores se concentrem nos aspectos essenciais de um sistema enquanto escondem a complexidade. Ao seguir as dicas e recomendações de autores renomados, você pode utilizar abstração de maneira eficaz para criar sistemas flexíveis e fáceis de manter.

See also

Outros Links

Patterns of Enterprise Application Architecture - Martin Fowler

(<https://www.amazon.com/Patterns-Enterprise-Application-Architecture-Martin/dp/0321127420>)

Documentação Oficial do Java sobre Abstração

(<https://docs.oracle.com/javase/tutorial/java/concepts/index.html>)

Clean Code - Robert C. Martin (<https://www.amazon.com/Clean-Code-Handbook-Software-Craftsmanship/dp/0132350882>)

Effective Java - Joshua Bloch (<https://www.amazon.com/Effective-Java-Joshua-Bloch/dp/0134685997>)

OO: Herança e Composição

Introdução

Em programação orientada a objetos (POO), **herança** e **composição** são conceitos fundamentais para modelar relações entre classes e promover a reutilização de código. Embora ambos permitam que uma classe utilize funcionalidades de outra, eles diferem significativamente em sua semântica e aplicação.

Este guia explora as diferenças entre herança e composição em Java, destacando suas características, usos adequados e fornecendo exemplos práticos, incluindo composições com `ArrayList`.

Herança

O que é Herança?

Herança é um mecanismo que permite que uma classe (subclasse) adquira as propriedades e métodos de outra classe (superclasse). Isso estabelece uma relação hierárquica entre as classes, promovendo a reutilização e extensão do código existente.

Características da Herança

- **Relação "é-um" (is-a):** A subclasse é um tipo especializado da superclasse.
- **Herança de implementação:** A subclasse herda atributos e comportamentos da superclasse.
- **Sobrescrita de métodos:** A subclasse pode modificar o comportamento herdado.

Exemplo de Herança

```
// Superclasse
public class Veiculo {
    public void mover() {
        System.out.println("O veículo está se movendo.");
    }
}
```

```
// Subclasse
public class Carro extends Veiculo {
    public void abrirPorta() {
        System.out.println("A porta do carro está aberta.");
    }

    @Override
    public void mover() {
        System.out.println("O carro está se movendo.");
    }
}

// Uso
public class Main {
    public static void main(String[] args) {
        Carro carro = new Carro();
        carro.mover();           // Saída: O carro está se movendo.
        carro.abrirPorta();      // Saída: A porta do carro está
aberta.
    }
}
```

Quando Usar Herança?

- Quando existe uma relação natural "é-um" entre classes.
- Para reutilizar código e comportamentos comuns.
- Quando deseja-se polimorfismo com classes base.

Composição

O que é Composição?

Composição é uma forma de estruturar classes onde uma classe contém instâncias de outras classes como membros, estabelecendo uma relação de "tem-um" (has-a). Isso permite que uma classe reutilize funcionalidades de outras sem estabelecer uma hierarquia.

Características da Composição

- **Relação "tem-um" (has-a):** A classe possui instâncias de outras classes.
- **Encapsulamento:** Os detalhes das classes componentes são ocultados.
- **Flexibilidade:** Componentes podem ser alterados sem afetar a classe que os utiliza.

Exemplo de Composição com ArrayList

```
import java.util.ArrayList;

// Classe Componente
public class Item {
    private String nome;

    public Item(String nome) {
        this.nome = nome;
    }

    public String getNome() {
        return nome;
    }
}

// Classe que compõe uma lista de Itens
public class Inventario {
    private ArrayList<Item> itens;

    public Inventario() {
        itens = new ArrayList<>();
    }

    public void adicionarItem(Item item) {
        itens.add(item);
    }

    public void listarItens() {
```

```

        for (Item item : itens) {
            System.out.println(item.getNome());
        }
    }
}

// Uso
public class Main {
    public static void main(String[] args) {
        Inventario inventario = new Inventario();
        inventario.adicionarItem(new Item("Espada"));
        inventario.adicionarItem(new Item("Escudo"));
        inventario.adicionarItem(new Item("Poção"));

        inventario.listarItens();
        // Saída:
        // Espada
        // Escudo
        // Poção
    }
}

```

Quando Usar Composição?

- Quando classes não possuem relação "é-um", mas "tem-um".
- Para criar objetos complexos a partir de objetos mais simples.
- Quando se busca maior modularidade e baixo acoplamento.

Diferenças Semânticas Entre Herança e Composição

Relação Entre Classes

- **Herança:** Estabelece uma relação hierárquica rígida.
- **Composição:** Estabelece uma relação de propriedade ou uso.

Flexibilidade e Acoplamento

- **Herança:** Acoplamento forte; mudanças na superclasse afetam subclasses.
- **Composição:** Acoplamento fraco; componentes podem ser alterados independentemente.

Reutilização de Código

- **Herança:** Reutiliza código através da extensão de classes existentes.
- **Composição:** Reutiliza código ao delegar responsabilidades para componentes.

Design e Manutenibilidade

- **Herança:** Pode levar a hierarquias complexas e difíceis de manter.
- **Composição:** Favorece a composição de comportamentos, facilitando a manutenção.

Composição com ArrayList em Detalhe

Usar `ArrayList` em composição permite que uma classe gerencie uma coleção dinâmica de objetos. Isso é útil para representar relações "um-para-muitos" ou "muitos-para-muitos".

Exemplo: Gerenciador de Cursos e Alunos

```
import java.util.ArrayList;

// Classe Aluno
public class Aluno {
    private String nome;

    public Aluno(String nome) {
        this.nome = nome;
    }

    public String getNome() {
        return nome;
    }
}
```

```

    }
}

// Classe Curso que contém uma lista de Alunos
public class Curso {
    private String nome;
    private ArrayList<Aluno> alunos;

    public Curso(String nome) {
        this.nome = nome;
        alunos = new ArrayList<>();
    }

    public void matricularAluno(Aluno aluno) {
        alunos.add(aluno);
    }

    public void listarAlunos() {
        System.out.println("Alunos matriculados no curso " + nome
+ ":");
        for (Aluno aluno : alunos) {
            System.out.println(aluno.getNome());
        }
    }
}

// Uso
public class Main {
    public static void main(String[] args) {
        Curso cursoJava = new Curso("Java Avançado");

        Aluno aluno1 = new Aluno("Maria");
        Aluno aluno2 = new Aluno("João");

        cursoJava.matricularAluno(aluno1);
        cursoJava.matricularAluno(aluno2);

        cursoJava.listarAlunos();
    }
}

```



```
        // Saída:  
        // Alunos matriculados no curso Java Avançado:  
        // Maria  
        // João  
    }  
}
```

Boas Práticas

- **Preferir Composição sobre Herança:** A composição oferece maior flexibilidade e reduz o acoplamento.
- **Usar Herança para Especialização:** Apenas quando há uma relação clara de especialização.
- **Manter Hierarquias Simples:** Evitar heranças profundas que dificultam a manutenção.
- **Encapsular Componentes:** Proteger os componentes internos da classe.

Conclusão

Entender as diferenças semânticas entre herança e composição é crucial para o design eficaz de software orientado a objetos. A escolha entre eles deve ser guiada pela natureza da relação entre as classes e pelos objetivos de flexibilidade, reutilização e manutenção do código.

Ao utilizar composições com `ArrayList`, podemos construir estruturas de dados poderosas e flexíveis, capazes de modelar cenários complexos de maneira organizada e eficiente.

See also

Outros Links

<https://docs.oracle.com/javase/8/docs/api/>

OO: Polimorfismo, Overload e Override

Introdução

O **polimorfismo** é um dos pilares da programação orientada a objetos (POO), juntamente com encapsulamento, herança e abstração. Em termos simples, polimorfismo permite que objetos de diferentes classes sejam tratados como objetos de uma classe comum. Em Java, o polimorfismo é alcançado principalmente através de **overloading** (sobrecarga) e **overriding** (sobrescrita).

Neste guia, exploraremos em detalhes o conceito de polimorfismo, explicando a diferença entre overload e override, e forneceremos exemplos práticos, incluindo a sobrecarga de construtores (construtor completo e vazio).

O que é Polimorfismo?

Polimorfismo, do grego "muitas formas", refere-se à capacidade de uma entidade, como um método ou objeto, assumir diferentes formas. Em Java, isso permite que métodos com o mesmo nome se comportem de maneiras diferentes, dependendo do contexto ou dos tipos de dados envolvidos.

Tipos de Polimorfismo em Java

1. **Polimorfismo de Compilação (Static Binding):** Alcançado através de overloading de métodos e construtores.
2. **Polimorfismo de Execução (Dynamic Binding):** Alcançado através de overriding de métodos.

Overloading (Sobrecarga)

O que é Overloading?

Overloading é a capacidade de definir múltiplos métodos ou construtores com o mesmo nome, mas com diferentes parâmetros dentro da mesma classe. O compilador diferencia os métodos com base no número, tipo e ordem dos parâmetros.

Características do Overloading

- **Mesma classe:** A sobrecarga ocorre na mesma classe.
- **Assinaturas diferentes:** Métodos ou construtores devem ter assinaturas diferentes.
- **Resolução em tempo de compilação:** A ligação do método é determinada durante a compilação.

Exemplo de Overloading de Métodos

```
public class Calculadora {
    // Método para somar dois inteiros
    public int somar(int a, int b) {
        return a + b;
    }

    // Método para somar três inteiros
    public int somar(int a, int b, int c) {
        return a + b + c;
    }

    // Método para somar dois números de ponto flutuante
    public double somar(double a, double b) {
        return a + b;
    }
}

// Uso
public class Main {
    public static void main(String[] args) {
        Calculadora calc = new Calculadora();

        System.out.println(calc.somar(2, 3));           // Saída: 5
        System.out.println(calc.somar(2, 3, 4));        // Saída: 9
        System.out.println(calc.somar(2.5, 3.5));      // Saída:
```

6.0

```
}  
}
```

Overloading de Construtores

Sobrecarga de construtores permite que uma classe tenha múltiplos construtores com diferentes parâmetros, oferecendo diversas maneiras de instanciar objetos.

Exemplo: Construtor Vazio e Construtor Completo

```
public class Pessoa {  
    private String nome;  
    private int idade;  
  
    // Construtor vazio  
    public Pessoa() {  
        // Pode inicializar valores padrão  
    }  
  
    // Construtor completo  
    public Pessoa(String nome, int idade) {  
        this.nome = nome;  
        this.idade = idade;  
    }  
  
    // Métodos getters e setters  
    public String getNome() {  
        return nome;  
    }  
  
    public void setNome(String nome) {  
        this.nome = nome;  
    }  
  
    public int getIdade() {  
        return idade;  
    }  
  
    public void setIdade(int idade) {
```

```

        this.idade = idade;
    }
}

// Uso
public class Main {
    public static void main(String[] args) {
        // Usando o construtor vazio
        Pessoa pessoa1 = new Pessoa();
        pessoa1.setNome("Carlos");
        pessoa1.setIdade(30);

        // Usando o construtor completo
        Pessoa pessoa2 = new Pessoa("Ana", 25);

        System.out.println(pessoa1.getNome() + " tem " +
pessoa1.getIdade() + " anos.");
        // Saída: Carlos tem 30 anos.

        System.out.println(pessoa2.getNome() + " tem " +
pessoa2.getIdade() + " anos.");
        // Saída: Ana tem 25 anos.
    }
}

```

Overriding (Sobrescrita)

O que é Overriding?

Overriding é a capacidade de uma subclasse fornecer uma implementação específica de um método que já é fornecido por sua superclasse ou interface. Isso permite que a subclasse adapte o comportamento do método às suas necessidades.

Características do Overriding

- **Classes diferentes:** A sobrescrita ocorre entre classes relacionadas por herança.
- **Mesma assinatura:** O método sobrescrito deve ter a mesma assinatura.

- **Resolução em tempo de execução:** A ligação do método é determinada durante a execução.

Regras para Overriding

- O método na subclasse deve ter o mesmo nome, tipo de retorno e parâmetros.
- O modificador de acesso não pode ser mais restritivo.
- Métodos marcados como `final` ou `static` não podem ser sobrescritos.

Exemplo de Overriding de Métodos

```
// Superclasse
public class Animal {
    public void emitirSom() {
        System.out.println("O animal faz um som.");
    }
}

// Subclasse
public class Cachorro extends Animal {
    @Override
    public void emitirSom() {
        System.out.println("O cachorro late.");
    }
}

// Uso
public class Main {
    public static void main(String[] args) {
        Animal meuAnimal = new Animal();
        Animal meuCachorro = new Cachorro();

        meuAnimal.emitirSom();           // Saída: O animal faz um
som.
        meuCachorro.emitirSom();         // Saída: O cachorro late.
    }
}
```

```
}  
}
```

No exemplo acima, o método `emitirSom` é sobrescrito na classe `Cachorro` para fornecer uma implementação específica.

Diferenças entre Overloading e Overriding

Aspecto	Overloading (Sobrecarga)	Overriding (Sobrescrita)
Localização	Mesma classe	Classes diferentes (herança)
Assinatura do Método	Deve ser diferente	Deve ser exatamente a mesma
Modificadores	Pode variar	Não pode ser mais restritivo
Resolução	Tempo de compilação	Tempo de execução
Uso Principal	Aumentar a legibilidade e flexibilidade	Implementar polimorfismo e comportamento específico

Quando Usar Overloading e Overriding?

Overloading

- Quando métodos ou construtores realizam operações semelhantes com diferentes tipos ou número de parâmetros.
- Para oferecer múltiplas formas de inicializar objetos ou chamar métodos.

Overriding

- Quando uma subclasse precisa modificar o comportamento de um método herdado.

- Para implementar polimorfismo e permitir que classes derivadas tenham comportamentos específicos.

Polimorfismo em Ação

Exemplo Prático

Vamos combinar overloading e overriding em um exemplo que demonstra polimorfismo.

```
// Classe base
public class Forma {
    public void desenhar() {
        System.out.println("Desenhando uma forma genérica.");
    }
}

// Subclasse
public class Circulo extends Forma {
    @Override
    public void desenhar() {
        System.out.println("Desenhando um círculo.");
    }

    // Sobrecarga de método
    public void desenhar(int raio) {
        System.out.println("Desenhando um círculo com raio " +
raio + ".");
    }
}

// Uso
public class Main {
    public static void main(String[] args) {
        Forma forma = new Forma();
        Forma circuloForma = new Circulo();
        Circulo circulo = new Circulo();

        forma.desenhar();                // Saída: Desenhando uma
```



```
forma genérica.  
    circuloForma.desenhar();    // Saída: Desenhando um  
círculo.  
    circulo.desenhar();        // Saída: Desenhando um  
círculo.  
    circulo.desenhar(5);        // Saída: Desenhando um  
círculo com raio 5.  
    }  
}
```

Neste exemplo:

- **Overriding:** O método `desenhar()` é sobrescrito na classe `Circulo`.
- **Overloading:** O método `desenhar(int raio)` é uma sobrecarga em `Circulo`.

Boas Práticas

- Usar `@Override`: Sempre que sobrescrever um método, use a anotação `@Override` para evitar erros.
- **Clareza de Código:** Use overloading quando métodos realizam ações relacionadas.
- **Consistência:** Mantenha os métodos sobrecarregados logicamente consistentes.
- **Documentação:** Documente os métodos sobrecarregados e sobrescritos para facilitar a manutenção.

Conclusão

Polimorfismo é uma ferramenta poderosa que permite que objetos sejam tratados de maneira uniforme, mesmo quando comportamentos específicos são implementados. A compreensão de overloading e overriding é essencial para aproveitar ao máximo a orientação a objetos em Java.

Ao utilizar a sobrecarga de métodos e construtores, podemos oferecer maior flexibilidade e facilidade de uso em nossas classes. A sobrescrita permite que subclasses

adaptem ou estendam o comportamento de suas superclasses, promovendo reutilização de código e extensibilidade.

Exercícios Práticos

1. Criar uma classe `Calculadora` que sobrecarrega o método `multiplicar` para diferentes tipos de dados.
2. Implementar uma hierarquia de classes `Veiculo` e `Bicicleta`, onde `Bicicleta` sobrescreve o método `mover`.
3. Experimentar a sobrecarga de construtores em uma classe `Produto`, permitindo a criação de objetos com diferentes conjuntos de informações.

See also

Outros Links

<https://docs.oracle.com/javase/8/docs/api/>(<https://docs.oracle.com/javase/8/docs/api/>

POJOs, Entidades e Value Objects no DDD

Introdução

No desenvolvimento de software, especialmente ao utilizar a abordagem de **Domain-Driven Design (DDD)**, é fundamental compreender os conceitos de **POJOs**, **Entidades** e **Value Objects**. Esses conceitos ajudam a modelar o domínio da aplicação de forma clara e consistente.

Este guia explora cada um desses conceitos, como eles se relacionam conceitualmente e fornece exemplos de código que incluem construtores vazios e completos, métodos *getters* e *setters*, e implementações dos métodos `toString`, `equals` e `hashCode`.

POJOs (Plain Old Java Objects)

O que são POJOs?

POJOs são objetos Java simples, sem qualquer restrição ou requisito especial além dos definidos pela linguagem Java. Eles não estendem classes ou implementam interfaces específicas de qualquer framework ou biblioteca, tornando-os leves e fáceis de usar.

Características dos POJOs

- **Simplicidade:** Não dependem de bibliotecas ou frameworks externos.
- **Flexibilidade:** Podem ser facilmente serializados, clonados e manipulados.
- **Reutilização:** Servem como base para entidades e value objects no DDD.

Exemplo de POJO

```
public class Endereco {  
    private String rua;  
    private String cidade;  
    private String cep;  
  
    // Construtor vazio
```

```
public Endereco() {  
}  
  
// Construtor completo  
public Endereco(String rua, String cidade, String cep) {  
    this.rua = rua;  
    this.cidade = cidade;  
    this.cep = cep;  
}  
  
// Getters e Setters  
public String getRua() {  
    return rua;  
}  
  
public void setRua(String rua) {  
    this.rua = rua;  
}  
  
public String getCidade() {  
    return cidade;  
}  
  
public void setCidade(String cidade) {  
    this.cidade = cidade;  
}  
  
public String getCep() {  
    return cep;  
}  
  
public void setCep(String cep) {  
    this.cep = cep;  
}  
  
// toString  
@Override  
public String toString() {
```

```

        return "Endereco{" +
            "rua='" + rua + '\'' +
            ", cidade='" + cidade + '\'' +
            ", cep='" + cep + '\'' +
            '}';
    }

    // equals e hashCode
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;

        Endereco endereco = (Endereco) o;

        if (!rua.equals(endereco.rua)) return false;
        if (!cidade.equals(endereco.cidade)) return false;
        return cep.equals(endereco.cep);
    }

    @Override
    public int hashCode() {
        int result = rua.hashCode();
        result = 31 * result + cidade.hashCode();
        result = 31 * result + cep.hashCode();
        return result;
    }
}

```

Entidades no DDD

O que são Entidades?

No DDD, uma **Entidade** é um objeto que possui uma identidade única e persistente ao longo do tempo. A identidade é o que distingue uma entidade de outra, mesmo que seus atributos sejam iguais. As entidades geralmente representam conceitos que possuem um ciclo de vida e mudam de estado.

Características das Entidades

- **Identidade Única:** Possuem um identificador único (como um ID).
- **Ciclo de Vida:** Podem mudar de estado ao longo do tempo.
- **Igualdade por Identidade:** A comparação entre entidades é feita com base em sua identidade.

Exemplo de Entidade

```
public class Cliente {
    private Long id; // Identidade única
    private String nome;
    private String email;
    private Endereco endereco;

    // Construtor vazio
    public Cliente() {
    }

    // Construtor completo
    public Cliente(Long id, String nome, String email, Endereco
endereco) {
        this.id = id;
        this.nome = nome;
        this.email = email;
        this.endereco = endereco;
    }

    // Getters e Setters
    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }
}
```

```

// Outros getters e setters omitidos para brevidade

// toString
@Override
public String toString() {
    return "Cliente{" +
        "id=" + id +
        ", nome='" + nome + '\'' +
        ", email='" + email + '\'' +
        ", endereco=" + endereco +
        '}';
}

// equals e hashCode baseados na identidade
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;

    Cliente cliente = (Cliente) o;

    return id.equals(cliente.id);
}

@Override
public int hashCode() {
    return id.hashCode();
}
}

```

Value Objects no DDD

O que são Value Objects?

Value Objects são objetos que são definidos não por sua identidade, mas por seus atributos. Eles são imutáveis e representam um conceito ou descrição, como uma quantidade, endereço ou intervalo de datas.

Características dos Value Objects

- **Imutabilidade:** Uma vez criado, seu estado não pode ser alterado.
- **Igualdade por Valor:** A comparação é feita com base nos valores dos atributos.
- **Sem Identidade Própria:** Não possuem um identificador único.

Exemplo de Value Object

```
public final class Dinheiro {
    private final double valor;
    private final String moeda;

    // Construtor completo (sem construtor vazio devido à
    imutabilidade)
    public Dinheiro(double valor, String moeda) {
        this.valor = valor;
        this.moeda = moeda;
    }

    // Getters (sem setters)
    public double getValor() {
        return valor;
    }

    public String getMoeda() {
        return moeda;
    }

    // toString
    @Override
    public String toString() {
        return "Dinheiro{" +
            "valor=" + valor +
            ", moeda='" + moeda + '\'' +
            '}';
    }
}
```



```

// equals e hashCode baseados nos atributos
@Override
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;

    Dinheiro dinheiro = (Dinheiro) o;

    if (Double.compare(dinheiro.valor, valor) != 0) return
false;
    return moeda.equals(dinheiro.moeda);
}

@Override
public int hashCode() {
    int result;
    long temp;
    temp = Double.doubleToLongBits(valor);
    result = (int) (temp ^ (temp >>> 32));
    result = 31 * result + moeda.hashCode();
    return result;
}
}

```

Relação Conceitual entre POJOs, Entidades e Value Objects

- **POJOs** são a base para criar **Entidades** e **Value Objects**. Eles fornecem uma estrutura simples sem depender de frameworks.
- **Entidades** são POJOs com identidade única, geralmente representando objetos do mundo real que possuem um ciclo de vida e podem mudar de estado.
- **Value Objects** são POJOs que representam valores e não possuem identidade própria. São imutáveis e comparados por seus atributos.

Implementação de equals e hashCode

Importância

- **Entidades:** Devem implementar `equals` e `hashCode` baseados na identidade (ID), pois duas entidades com o mesmo ID são consideradas iguais.
- **Value Objects:** Devem implementar `equals` e `hashCode` baseados nos atributos, pois são comparados pelo valor.

Boas Práticas

- Sempre sobrescrever `equals` e `hashCode` juntos.
- Usar todas as propriedades relevantes para comparação no caso de Value Objects.
- Manter a consistência entre `equals` e `hashCode`.

Exemplo Prático Integrando Entidade e Value Object

```
public class Produto {
    private Long id; // Identidade única
    private String nome;
    private Dinheiro preco; // Value Object
    private String descricao;

    // Construtor vazio
    public Produto() {
    }

    // Construtor completo
    public Produto(Long id, String nome, Dinheiro preco, String
descricao) {
        this.id = id;
        this.nome = nome;
        this.preco = preco;
        this.descricao = descricao;
    }
}
```

```

    }

    // Getters e Setters
    // ...

    // toString
    @Override
    public String toString() {
        return "Produto{" +
            "id=" + id +
            ", nome='" + nome + '\'' +
            ", preco=" + preco +
            ", descricao='" + descricao + '\'' +
            '}';
    }

    // equals e hashCode baseados na identidade
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;

        Produto produto = (Produto) o;

        return id.equals(produto.id);
    }

    @Override
    public int hashCode() {
        return id.hashCode();
    }
}

```

Neste exemplo, `Produto` é uma entidade que possui um `Dinheiro` como atributo, representando o preço. `Dinheiro` é um Value Object.

Uso de Construtores, Getters e Setters

- **Construtor Vazio:** Necessário para algumas frameworks (como Hibernate) que instanciam objetos via reflexão.
- **Construtor Completo:** Facilita a criação de objetos com todos os atributos definidos.
- **Getters e Setters:** Permitem acessar e modificar os atributos, respeitando o encapsulamento.

Exemplo de Uso

```
public class Main {  
    public static void main(String[] args) {  
        // Usando construtor completo  
        Dinheiro preco = new Dinheiro(99.90, "BRL");  
        Produto produto = new Produto(1L, "Teclado Mecânico",  
preco, "Teclado para jogos");  
  
        // Usando construtor vazio e setters  
        Cliente cliente = new Cliente();  
        cliente.setId(1L);  
        cliente.setNome("João Silva");  
        cliente.setEmail("joao@example.com");  
  
        // Exibindo informações  
        System.out.println(produto);  
        System.out.println(cliente);  
    }  
}
```

Implementação de toString

- Facilita o log e depuração exibindo uma representação textual do objeto.
- Útil para imprimir objetos em logs, consoles ou interfaces gráficas.

Implementação de Imutabilidade em Value Objects

- Declarar a classe como `final` para evitar extensão.
- Não fornecer setters.
- Todos os atributos devem ser `final` e inicializados no construtor.

Conclusão

Compreender e aplicar os conceitos de **POJOs**, **Entidades** e **Value Objects** é essencial para um design de software orientado ao domínio efetivo. Eles permitem uma modelagem clara e consistente do domínio da aplicação, facilitando a manutenção e evolução do sistema.

Ao implementar corretamente construtores, getters, setters, e os métodos `toString`, `equals` e `hashCode`, garantimos que nossos objetos se comportem de maneira previsível e adequada às necessidades da aplicação.

Exercícios Práticos

1. Criar uma classe `Pedido` como Entidade, que possui uma lista de `ItemPedido` como Value Objects.
2. Implementar os métodos `equals` e `hashCode` em `ItemPedido` considerando todos os atributos.
3. Criar um Value Object chamado `CPF` que valida o formato do CPF no construtor e é imutável.

Observações Finais

- **Entidades e Value Objects** são conceitos fundamentais no DDD que ajudam a modelar o domínio de forma precisa.
- **POJOs** são a base para construir essas estruturas, mantendo o código limpo e desacoplado.
- **Imutabilidade** em Value Objects evita efeitos colaterais e facilita a manutenção.
- **Identidade** é crucial em Entidades para garantir que cada objeto seja único no

sistema.

OO: Classes Abstratas

Introdução

Em programação orientada a objetos, as **classes abstratas** desempenham um papel fundamental na criação de hierarquias de classes e na definição de contratos para subclasses. Elas permitem modelar conceitos que não fazem sentido serem instanciados diretamente, mas que compartilham características comuns que devem ser implementadas pelas subclasses.

Neste guia, exploraremos o conceito de classes abstratas em Java, suas características, quando e como usá-las. Faremos isso através de exemplos práticos utilizando uma classe abstrata `Midia`, que é herdada pelas classes concretas `Musica` e `Podcast`.

O que são Classes Abstratas?

Uma **classe abstrata** é uma classe que não pode ser instanciada diretamente e pode conter métodos abstratos (sem implementação) e métodos concretos (com implementação). Ela serve como um modelo ou base para outras classes que a estendem.

Características das Classes Abstratas

- **Não podem ser instanciadas:** Não é possível criar objetos diretamente de uma classe abstrata.
- **Podem conter métodos abstratos e concretos:** Métodos abstratos são declarados sem implementação, enquanto métodos concretos possuem implementação.
- **Podem conter atributos:** Tanto estáticos quanto de instância.
- **Servem como base para subclasses:** As subclasses devem implementar os métodos abstratos ou também serem abstratas.

Sintaxe em Java

```
public abstract class NomeDaClasse {  
    // Atributos
```

```

    // Métodos concretos
    public void metodoConcreto() {
        // Implementação
    }

    // Métodos abstratos
    public abstract void metodoAbstrato();
}

```

Por que Usar Classes Abstratas?

- **Modelagem de Conceitos Genéricos:** Representar conceitos que não fazem sentido serem instanciados por si só, mas que compartilham características com subclasses.
- **Definição de Contratos:** Forçar subclasses a implementar métodos específicos.
- **Reutilização de Código:** Compartilhar atributos e métodos comuns entre subclasses.
- **Polimorfismo:** Permitir que objetos de subclasses sejam tratados como objetos da classe abstrata.

Exemplo Prático: Classe Midia e Subclasses Musica e Podcast

Vamos criar uma hierarquia de classes para representar mídias de áudio, como músicas e podcasts.

Classe Abstrata Midia

```

public abstract class Midia {
    protected String titulo;
    protected String autor;
    protected int duracao; // em segundos

    // Construtor
    public Midia(String titulo, String autor, int duracao) {
        this.titulo = titulo;
    }
}

```



```

        this.autor = autor;
        this.duracao = duracao;
    }

    // Método concreto
    public void reproduzir() {
        System.out.println("Reproduzindo: " + titulo + " de " +
autor);
    }

    // Método abstrato
    public abstract void exibirDetalhes();
}

```

Explicação:

- `titulo`, `autor` e `duracao` são atributos comuns a todas as mídias.
- `reproduzir()` é um método concreto que pode ser compartilhado.
- `exibirDetalhes()` é um método abstrato que cada subclass deverá implementar.

Subclasse Musica

```

public class Musica extends Midia {
    private String album;
    private String genero;

    // Construtor
    public Musica(String titulo, String autor, int duracao, String
album, String genero) {
        super(titulo, autor, duracao);
        this.album = album;
        this.genero = genero;
    }

    // Implementação do método abstrato
    @Override
    public void exibirDetalhes() {

```

```

        System.out.println("Música: " + titulo);
        System.out.println("Artista: " + autor);
        System.out.println("Álbum: " + album);
        System.out.println("Gênero: " + genero);
        System.out.println("Duração: " + duracao + " segundos");
    }
}

```

Subclasse Podcast

```

public class Podcast extends Midia {
    private String descricao;
    private int episodio;

    // Construtor
    public Podcast(String titulo, String autor, int duracao,
String descricao, int episodio) {
        super(titulo, autor, duracao);
        this.descricao = descricao;
        this.episodio = episodio;
    }

    // Implementação do método abstrato
    @Override
    public void exibirDetalhes() {
        System.out.println("Podcast: " + titulo);
        System.out.println("Host: " + autor);
        System.out.println("Descrição: " + descricao);
        System.out.println("Episódio: " + episodio);
        System.out.println("Duração: " + duracao + " segundos");
    }
}

```

Uso das Classes

```

public class Main {
    public static void main(String[] args) {

```

```

        Midia musica = new Musica("Imagine", "John Lennon", 183,
"Imagine", "Rock");
        Midia podcast = new Podcast("Tecnologia Hoje", "Maria
Silva", 3600, "Discussões sobre tecnologia", 42);

        musica.reproduzir();
        musica.exibirDetalhes();

        System.out.println();

        podcast.reproduzir();
        podcast.exibirDetalhes();
    }
}

```

Saída:

```

Reproduzindo: Imagine de John Lennon
Música: Imagine
Artista: John Lennon
Álbum: Imagine
Gênero: Rock
Duração: 183 segundos

Reproduzindo: Tecnologia Hoje de Maria Silva
Podcast: Tecnologia Hoje
Host: Maria Silva
Descrição: Discussões sobre tecnologia
Episódio: 42
Duração: 3600 segundos

```

Análise do Exemplo

- Classe Abstrata **Midia**:
 - Não pode ser instanciada diretamente.

- Define atributos e métodos comuns.
- Contém o método abstrato `exibirDetalhes()`.
- **Subclasses `Musica` e `Podcast`:**
 - Estendem `Midia` e herdam seus atributos e métodos.
 - Implementam o método abstrato `exibirDetalhes()` de forma específica.
 - Podem adicionar atributos e métodos adicionais.
- **Polimorfismo:**
 - As variáveis `musica` e `podcast` são do tipo `Midia`, mas armazenam objetos de `Musica` e `Podcast`.
 - Chamadas aos métodos `reproduzir()` e `exibirDetalhes()` utilizam o polimorfismo.

Regras e Considerações sobre Classes Abstratas

1. Instanciação:

- Não é possível criar instâncias de classes abstratas diretamente.
- Exemplo inválido: `Midia minhaMidia = new Midia(...);`

2. Métodos Abstratos:

- Devem ser implementados em subclasses concretas.
- Uma subclasse pode ser abstrata e optar por não implementar os métodos abstratos.

3. Construtores:

- Classes abstratas podem ter construtores.
- Os construtores são chamados através do `super()` nas subclasses.

4. Modificadores de Acesso:

- Métodos e atributos podem ter qualquer modificador de acesso (`public`, `protected`, `private`).

- Métodos abstratos devem ser declarados como `public` ou `protected`.

5. Interfaces vs. Classes Abstratas:

- Classes abstratas podem conter implementação, enquanto interfaces (antes do Java 8) não podiam.
- Uma classe pode implementar múltiplas interfaces, mas só pode estender uma única classe (abstrata ou não).

Quando Usar Classes Abstratas?

- **Modelagem de Hierarquias:**
 - Quando há uma relação clara de herança e um comportamento padrão ou estado a ser compartilhado.
- **Definição de Contratos Parciais:**
 - Quando deseja fornecer alguma implementação comum, mas deixar detalhes para as subclasses.
- **Evitar Repetição de Código:**
 - Compartilhando código comum entre classes relacionadas.
- **Polimorfismo:**
 - Quando deseja tratar diferentes objetos de subclasses como objetos da classe base.

Comparação entre Classes Abstratas e Interfaces

Característica	Classe Abstrata	Interface
Instanciação	Não pode ser instanciada	Não pode ser instanciada
Métodos Abstratos	Pode conter métodos abstratos e concretos	Até Java 7, todos os métodos eram abstratos e públicos
Atributos	Pode ter atributos de instância	Até Java 7, apenas constantes (public static final)
Herança	Uma classe pode estender uma única classe abstrata	Uma classe pode implementar múltiplas interfaces
Uso	Quando há relação de herança e comportamento comum	Para definir contratos sem implementação

Nota: A partir do Java 8, interfaces podem ter métodos `default` (com implementação) e métodos estáticos.

Extensão do Exemplo: Adicionando Mais Subclasses

Subclasse Audiobook

```
public class Audiobook extends Midia {
    private String narrador;
    private int capitulos;

    // Construtor
    public Audiobook(String titulo, String autor, int duracao,
String narrador, int capitulos) {
        super(titulo, autor, duracao);
        this.narrador = narrador;
        this.capitulos = capitulos;
    }
}
```

```

// Implementação do método abstrato
@Override
public void exibirDetalhes() {
    System.out.println("Audiobook: " + titulo);
    System.out.println("Autor: " + autor);
    System.out.println("Narrador: " + narrador);
    System.out.println("Capítulos: " + capitulos);
    System.out.println("Duração: " + duracao + " segundos");
}
}

```

Uso Atualizado

```

public class Main {
    public static void main(String[] args) {
        Midia musica = new Musica("Imagine", "John Lennon", 183,
"Imagine", "Rock");
        Midia podcast = new Podcast("Tecnologia Hoje", "Maria
Silva", 3600, "Discussões sobre tecnologia", 42);
        Midia audiobook = new Audiobook("1984", "George Orwell",
7200, "Carlos Alberto", 24);

        Midia[] biblioteca = {musica, podcast, audiobook};

        for (Midia midia : biblioteca) {
            midia.reproduzir();
            midia.exibirDetalhes();
            System.out.println();
        }
    }
}

```

Saída:

```

Reproduzindo: Imagine de John Lennon
Música: Imagine
Artista: John Lennon

```

Álbum: Imagine

Gênero: Rock

Duração: 183 segundos

Reproduzindo: Tecnologia Hoje de Maria Silva

Podcast: Tecnologia Hoje

Host: Maria Silva

Descrição: Discussões sobre tecnologia

Episódio: 42

Duração: 3600 segundos

Reproduzindo: 1984 de George Orwell

Audiobook: 1984

Autor: George Orwell

Narrador: Carlos Alberto

Capítulos: 24

Duração: 7200 segundos

Boas Práticas com Classes Abstratas

- **Nomeação Clara:** Utilize nomes que reflitam a natureza abstrata da classe (e.g., `Midia`, `Veiculo`, `Animal`).
- **Visibilidade Adequada:** Use `protected` para permitir que subclasses acessem atributos e métodos, mantendo o encapsulamento.
- **Abstração Adequada:** Não implemente detalhes específicos na classe abstrata; deixe para as subclasses.
- **Documentação:** Comente os métodos abstratos para indicar o que se espera das implementações.

Considerações Finais

- **Não Exagerar:** Nem todos os cenários requerem classes abstratas; avalie se é realmente necessário.

- **Combinação com Interfaces:** Pode ser útil implementar interfaces na classe abstrata para definir contratos adicionais.
- **Atualizações Futuras:** Ao adicionar novos métodos abstratos, lembre-se de que todas as subclasses precisarão implementá-los.

Exemplos Adicionais de Uso de Classes Abstratas

Modelo de Funcionários em uma Empresa

- **Classe Abstrata Funcionario:**
 - Atributos: nome, salario.
 - Método abstrato: calcularBonus().
- **Subclasses:**
 - Gerente: Implementa calcularBonus() com uma lógica específica.
 - Desenvolvedor: Implementa calcularBonus() de outra forma.

Exercícios Práticos

1. Criar uma classe abstrata FormaGeometrica:

- Atributos: nome.
- Métodos abstratos: calcularArea(), calcularPerimetro().
- Crie subclasses Circulo, Retangulo, Triangulo e implemente os métodos abstratos.

2. Desenvolver uma hierarquia de veículos:

- Classe abstrata Veiculo com método abstrato mover().
- Subclasses Carro, Bicicleta, Aviao implementando mover() de forma específica.

3. Implementar um sistema de pagamentos:

- Classe abstrata Pagamento com método abstrato processarPagamento().

- Subclasses `PagamentoCredito`, `PagamentoDebito`, `PagamentoPix` implementando o método.

Conclusão

As classes abstratas são uma ferramenta poderosa em Java para modelar hierarquias e definir contratos parciais para subclasses. Elas permitem a reutilização de código comum e fornecem um meio de garantir que certas funcionalidades sejam implementadas nas subclasses.

Neste guia, exploramos o conceito de classes abstratas através de exemplos práticos com `Midia`, `Musica` e `Podcast`. Compreender quando e como usar classes abstratas é fundamental para escrever código orientado a objetos eficaz e manter sistemas bem estruturados e fáceis de manter.

Observações Finais

- **Prática:** Experimente criar suas próprias hierarquias utilizando classes abstratas para consolidar o conhecimento.
- **Leitura Adicional:** Estude sobre interfaces, polimorfismo e outros conceitos relacionados para expandir sua compreensão.
- **Comunidade:** Participe de fóruns e comunidades de desenvolvedores para discutir e esclarecer dúvidas.

Enums (Enumeradores)

Introdução

Em Java, os **enumeradores** (ou **Enums**) são um tipo de dado especial que permite definir um conjunto fixo de constantes nomeadas. Eles são utilizados para representar um grupo de valores constantes e facilitar a legibilidade e manutenção do código.

Este guia explora o conceito de enumeradores em Java, como declará-los e utilizá-los, suas características avançadas, e fornece exemplos práticos para ilustrar seu uso.

O que são Enumeradores?

Um **Enum** é um tipo de dado que consiste em um conjunto fixo de constantes, conhecidas como **constantes enum**, que são, por padrão, públicas, estáticas e finais. Enums são tipicamente usados quando se trabalha com valores que não mudam, como dias da semana, direções, estados de um processo, etc.

Características dos Enums

- **Tipo Seguro:** Enums fornecem segurança de tipo, evitando valores inválidos.
- **Legibilidade:** Melhoram a legibilidade do código ao usar nomes significativos.
- **Funcionalidades Adicionais:** Podem conter métodos, construtores e atributos.
- **Imutabilidade:** As constantes são imutáveis e não podem ser modificadas após a criação.

Sintaxe Básica

```
public enum NomeDoEnum {  
    CONSTANTE1,  
    CONSTANTE2,  
    CONSTANTE3  
}
```

Criando e Usando Enums

Exemplo 1: Dias da Semana

```
public enum DiaDaSemana {  
    SEGUNDA,  
    TERCA,  
    QUARTA,  
    QUINTA,  
    SEXTA,  
    SABADO,  
    DOMINGO  
}  
  
public class ExemploEnum {  
    public static void main(String[] args) {  
        DiaDaSemana hoje = DiaDaSemana.SEGUNDA;  
  
        if (hoje == DiaDaSemana.SEGUNDA) {  
            System.out.println("Hoje é segunda-feira!");  
        } else {  
            System.out.println("Hoje não é segunda-feira.");  
        }  
    }  
}
```

Saída:

```
Hoje é segunda-feira!
```

Métodos e Funcionalidades Padrão

Método values()

Retorna um array contendo todas as constantes do Enum.

```
for (DiaDaSemana dia : DiaDaSemana.values()) {  
    System.out.println(dia);  
}
```

Saída:

```
SEGUNDA  
TERCA  
QUARTA  
QUINTA  
SEXTA  
SABADO  
DOMINGO
```

Método valueOf(String name)

Retorna a constante do Enum com o nome especificado.

```
DiaDaSemana dia = DiaDaSemana.valueOf("QUARTA");  
System.out.println(dia);
```

Saída:

```
QUARTA
```

Método ordinal()

Retorna a posição ordinal da constante, começando em zero.

```
DiaDaSemana dia = DiaDaSemana.SEXTA;  
System.out.println(dia.ordinal());
```

Saída:

```
4
```

Enums com Atributos e Métodos

Os Enums em Java não se limitam a constantes simples; eles podem ter atributos, construtores e métodos, permitindo comportamentos mais complexos.

Exemplo 2: Códigos de Países com Código Telefônico

```
public enum Pais {
    BRASIL("Brasil", "+55"),
    ESTADOS_UNIDOS("Estados Unidos", "+1"),
    PORTUGAL("Portugal", "+351");

    private String nome;
    private String codigoTelefone;

    // Construtor
    Pais(String nome, String codigoTelefone) {
        this.nome = nome;
        this.codigoTelefone = codigoTelefone;
    }

    // Métodos getters
    public String getNome() {
        return nome;
    }

    public String getCodigoTelefone() {
        return codigoTelefone;
    }
}

public class ExemploPaisEnum {
    public static void main(String[] args) {
        for (Pais pais : Pais.values()) {
            System.out.println("País: " + pais.getNome() + ",
Código: " + pais.getCodigoTelefone());
        }
    }
}
```

Saída:

País: Brasil, Código: +55

País: Estados Unidos, Código: +1

País: Portugal, Código: +351

Enums com Comportamentos Específicos

É possível definir métodos abstratos dentro do Enum e fornecer implementações específicas para cada constante.

Exemplo 3: Operações Matemáticas

```
public enum Operacao {  
    SOMA {  
        @Override  
        public double calcular(double x, double y) {  
            return x + y;  
        }  
    },  
    SUBTRACAO {  
        @Override  
        public double calcular(double x, double y) {  
            return x - y;  
        }  
    },  
    MULTIPLICACAO {  
        @Override  
        public double calcular(double x, double y) {  
            return x * y;  
        }  
    },  
    DIVISAO {  
        @Override  
        public double calcular(double x, double y) {  
            return x / y;  
        }  
    }  
};
```

```
// Método abstrato
public abstract double calcular(double x, double y);
}
```

Uso em Código

```
public class ExemploOperacaoEnum {
    public static void main(String[] args) {
        double x = 10;
        double y = 5;

        for (Operacao op : Operacao.values()) {
            System.out.println(x + " " + op.name() + " " + y + " = "
                + op.calcular(x, y));
        }
    }
}
```

Saída:

```
10.0 SOMA 5.0 = 15.0
10.0 SUBTRACAO 5.0 = 5.0
10.0 MULTIPLICACAO 5.0 = 50.0
10.0 DIVISAO 5.0 = 2.0
```

Enums e Switch Case

Enums podem ser usados em instruções `switch`, tornando o código mais legível.

Exemplo 4: Direções

```
public enum Direcao {
    NORTE,
    SUL,
    LESTE,
    OESTE
}
```


Uso em Switch

```
public class ExemploDirecaoEnum {  
    public static void main(String[] args) {  
        Direcao direcao = Direcao.LESTE;  
  
        switch (direcao) {  
            case NORTE:  
                System.out.println("Indo para o Norte");  
                break;  
            case SUL:  
                System.out.println("Indo para o Sul");  
                break;  
            case LESTE:  
                System.out.println("Indo para o Leste");  
                break;  
            case OESTE:  
                System.out.println("Indo para o Oeste");  
                break;  
        }  
    }  
}
```

Saída:

```
Indo para o Leste
```

Enums e Interfaces

Enums podem implementar interfaces, permitindo adicionar comportamento consistente entre as constantes.

Exemplo 5: Interface Operavel

```
public interface Operavel {  
    double calcular(double x, double y);  
}
```

Enum Implementando a Interface

```
public enum Operacao implements Operavel {
    SOMA {
        @Override
        public double calcular(double x, double y) {
            return x + y;
        }
    },
    SUBTRACAO {
        @Override
        public double calcular(double x, double y) {
            return x - y;
        }
    }
    // ... demais operações
}
```

Boas Práticas com Enums

- **Nomes em LETRAS MAIÚSCULAS:** Por convenção, os nomes das constantes do Enum são escritos em maiúsculas.
- **Imutabilidade:** Não altere o estado interno das constantes do Enum após a inicialização.
- **Uso Adequado:** Utilize Enums para representar conjuntos fixos de constantes.
- **Evitar Abuso:** Não use Enums para simular funcionalidades que não se encaixam no conceito de constantes nomeadas.

EnumSet e EnumMap

O Java fornece coleções especializadas para trabalhar com Enums de forma eficiente.

EnumSet

Uma implementação de `Set` otimizada para trabalhar com Enums.

```
import java.util.EnumSet;

public class ExemploEnumSet {
    public static void main(String[] args) {
        EnumSet<DiaDaSemana> diasUteis =
EnumSet.range(DiaDaSemana.SEGUNDA, DiaDaSemana.SEXTA);
        System.out.println("Dias úteis: " + diasUteis);
    }
}
```

Saída:

```
Dias úteis: [SEGUNDA, TERCA, QUARTA, QUINTA, SEXTA]
```

EnumMap

Uma implementação de `Map` otimizada para Enums como chaves.

```
import java.util.EnumMap;
import java.util.Map;

public class ExemploEnumMap {
    public static void main(String[] args) {
        EnumMap<Pais, String> capitais = new EnumMap<>
(Pais.class);

        capitais.put(Pais.BRASIL, "Brasília");
        capitais.put(Pais.PORTUGAL, "Lisboa");

        for (Map.Entry<Pais, String> entry : capitais.entrySet())
        {
            System.out.println("País: " + entry.getKey().getNome()
+ ", Capital: " + entry.getValue());
        }
    }
}
```

Saída:

País: Brasil, Capital: Brasília

País: Portugal, Capital: Lisboa

Serialização de Enums

Enums são serializáveis por padrão. Durante a serialização, apenas o nome da constante é serializado, garantindo que durante a desserialização, a mesma constante seja recuperada.

Exemplo de Serialização

```
import java.io.*;

public class ExemploSerializacaoEnum {
    public static void main(String[] args) {
        try {
            // Serialização
            ObjectOutputStream oos = new ObjectOutputStream(new
            FileOutputStream("direcao.ser"));
            oos.writeObject(Direcao.NORTE);
            oos.close();

            // Desserialização
            ObjectInputStream ois = new ObjectInputStream(new
            FileInputStream("direcao.ser"));
            Direcao direcao = (Direcao) ois.readObject();
            ois.close();

            System.out.println("Direção desserializada: " +
            direcao);

        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

```
}  
}
```

Saída:

Direção desserializada: NORTE

Usando Enums em Anotações

Enums podem ser utilizados em anotações para restringir os valores possíveis.

Exemplo de Anotação Personalizada

```
public enum NivelLog {  
    DEBUG,  
    INFO,  
    WARN,  
    ERROR  
}  
  
@Retention(RetentionPolicy.RUNTIME)  
@Target(ElementType.METHOD)  
public @interface Loggable {  
    NivelLog nivel() default NivelLog.INFO;  
}
```

Uso da Anotação

```
public class Servico {  
  
    @Loggable(nivel = NivelLog.DEBUG)  
    public void executar() {  
        // Código do método  
    }  
}
```

Considerações Finais

- **Comparação:** Use `==` para comparar Enums, pois eles são singletons.
- **Thread-Safety:** Enums são thread-safe por natureza.
- **Singleton:** Enums podem ser usados para implementar o padrão Singleton de forma segura.

Exemplo de Singleton com Enum

```
public enum GerenciadorDeConexao {  
    INSTANCIA;  
  
    // Métodos e atributos do singleton  
    public void conectar() {  
        System.out.println("Conectado!");  
    }  
}
```

Uso

```
public class ExemploSingletonEnum {  
    public static void main(String[] args) {  
        GerenciadorDeConexao conexao =  
        GerenciadorDeConexao.INSTANCIA;  
        conexao.conectar();  
    }  
}
```

Saída:

```
Conectado!
```

Exercícios Práticos

1. Criar um Enum `TipoDocumento`:

- Constantes: `CPF`, `CNPJ`.

- Adicionar um método `validar(String valor)` que verifica se o número do documento é válido.

2. Implementar um Enum `EstadoProcesso`:

- Constantes: `INICIADO`, `EM_ANDAMENTO`, `FINALIZADO`.
- Adicionar um atributo `descricao` e métodos para acessar a descrição.

3. Utilizar `EnumSet`:

- Criar um programa que pergunta ao usuário quais dias da semana ele está disponível e armazena as respostas em um `EnumSet<DiaDaSemana>`.

Conclusão

Enumeradores em Java são uma poderosa ferramenta para representar conjuntos fixos de constantes, fornecendo segurança de tipo, legibilidade e funcionalidades avançadas. Eles vão além de simples constantes, permitindo a adição de métodos, atributos e comportamentos específicos para cada constante.

Compreender e utilizar Enums efetivamente pode melhorar significativamente a qualidade e a manutenção do seu código Java.

Observações Finais

- **Prática:** Experimente criar seus próprios Enums e utilizar as funcionalidades avançadas.
- **Leitura Adicional:** Estude como Enums podem interagir com outras partes do Java, como anotações e coleções.
- **Comunidade:** Participe de fóruns e grupos de discussão para compartilhar conhecimentos e tirar dúvidas.

See also

Outros Links

<https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html>

<https://docs.oracle.com/javase/specs/jls/se8/html/jls-8.html#jls-8.9>

Stream API e Manipulação de Listas e Arrays

Introdução

A **Stream API** introduzida no Java 8 revolucionou a forma como manipulamos coleções de dados. Ela permite processar conjuntos de dados de maneira declarativa, expressando o "**o quê**" ao invés do "**como**". Isso resulta em um código mais conciso, legível e fácil de manter.

Neste guia, exploraremos como a Stream API funciona, como manipulá-la com **listas** e **arrays**, e forneceremos exemplos práticos para ilustrar seu uso.

O que é a Stream API?

A **Stream API** é um conjunto de classes e interfaces que permite processar dados de coleções (como List, Set, Map) e arrays de forma funcional e paralela. Uma **Stream** representa uma sequência de elementos suportando operações sequenciais e paralelas.

Características Principais

- **Declaratividade:** Permite descrever **o que** fazer, não **como** fazer.
- **Pipeline de Operações:** As Streams permitem encadear operações intermediárias e terminais.
- **Lazy Evaluation:** Operações intermediárias são avaliadas apenas quando necessário.
- **Possibilidade de Paralelismo:** Fácil conversão para processamento paralelo.

Como Funcionam as Streams?

As Streams funcionam em três etapas principais:

1. **Fonte (Source):** Origem dos dados, como coleções ou arrays.
2. **Operações Intermediárias:** Transformam a Stream em outra Stream (e.g., `filter`, `map`).

3. **Operação Terminal:** Finaliza o processamento e retorna um resultado (e.g., `collect`, `forEach`).

Fluxo Básico

Source -> Operações Intermediárias -> Operação Terminal

Manipulação de Listas com Stream API

Exemplo 1: Filtrar e Mapear uma Lista de Números

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class ExemploStream {
    public static void main(String[] args) {
        List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5, 6);

        List<Integer> numerosParesAoQuadrado = numeros.stream()
            .filter(n -> n % 2 == 0)           // Operação
Intermediária: Filtrar números pares
            .map(n -> n * n)                   // Operação
Intermediária: Elevar ao quadrado
            .collect(Collectors.toList());    // Operação
Terminal: Coletar em uma lista

        System.out.println(numerosParesAoQuadrado); // Saída: [4,
16, 36]
    }
}
```

Operações Comuns

- **filter(Predicate T predicate):** Seleciona elementos que correspondem a um predicado.
- **map(Function<T, R> mapper):** Transforma cada elemento em outro.

- **collect(Collector<T, A, R> collector):** Reúne os elementos finais em uma coleção ou outro tipo de resultado.

Manipulação de Arrays com Stream API

Exemplo 2: Processar um Array de Strings

```
import java.util.Arrays;

public class ExemploStreamArray {
    public static void main(String[] args) {
        String[] nomes = {"Ana", "Bruno", "Carlos", "Amanda",
                           "Beatriz"};

        Arrays.stream(nomes)
                .filter(nome -> nome.startsWith("A")) // Filtrar
nomes que começam com 'A'
                .sorted()                               // Ordenar
                .forEach(System.out::println);          // Imprimir
cada nome

        // Saída:
        // Amanda
        // Ana
    }
}
```

Criando Streams a partir de Arrays

- **Arrays.stream(array):** Cria uma Stream a partir de um array.

Operações Intermediárias em Detalhe

filter()

Filtra elementos com base em um predicado.

```
List<String> palavras = Arrays.asList("java", "stream", "api",
    "lambda");
palavras.stream()
    .filter(p -> p.length() > 4)
    .forEach(System.out::println);
// Saída:
// stream
// lambda
```

map()

Transforma elementos em outros tipos ou formatos.

```
List<String> palavras = Arrays.asList("java", "stream", "api");
palavras.stream()
    .map(String::toUpperCase)
    .forEach(System.out::println);
// Saída:
// JAVA
// STREAM
// API
```

flatMap()

Desenvolve uma função que retorna uma Stream para cada elemento e as concatena.

```
List<List<Integer>> listas = Arrays.asList(
    Arrays.asList(1, 2),
    Arrays.asList(3, 4),
    Arrays.asList(5, 6)
);

listas.stream()
    .flatMap(List::stream)
    .forEach(System.out::println);
// Saída:
// 1 2 3 4 5 6
```

sorted()

Ordena os elementos.

```
List<String> palavras = Arrays.asList("banana", "abacate",  
    "laranja");  
palavras.stream()  
    .sorted()  
    .forEach(System.out::println);  
// Saída:  
// abacate  
// banana  
// laranja
```

Operações Terminais em Detalhe

collect()

Coleta os elementos em uma coleção ou outro tipo de resultado.

```
List<String> palavras = Arrays.asList("java", "stream", "api");  
List<String> maiusculas = palavras.stream()  
    .map(String::toUpperCase)  
    .collect(Collectors.toList());
```

forEach()

Executa uma ação para cada elemento.

```
List<Integer> numeros = Arrays.asList(1, 2, 3);  
numeros.stream()  
    .forEach(n -> System.out.println(n));  
// Saída:  
// 1  
// 2  
// 3
```

reduce()

Combina os elementos em um único resultado.

```
List<Integer> numeros = Arrays.asList(1, 2, 3, 4);
int soma = numeros.stream()
    .reduce(0, Integer::sum);
System.out.println(soma); // Saída: 10
```

anyMatch(), allMatch(), noneMatch()

Testam se algum, todos ou nenhum elemento correspondem a um predicado.

```
List<String> palavras = Arrays.asList("java", "stream", "api");
boolean algumComecaComA = palavras.stream()
    .anyMatch(p -> p.startsWith("a"));
System.out.println(algumComecaComA); // Saída: false
```

Processamento Paralelo com Streams

É possível paralelizar o processamento simplesmente chamando `parallelStream()` ao invés de `stream()`.

Exemplo

```
List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5, 6);

numeros.parallelStream()
    .map(n -> n * n)
    .forEach(System.out::println);
```

Nota: O processamento paralelo pode melhorar a performance em conjuntos de dados grandes, mas deve ser usado com cuidado devido a questões de thread safety.

Exemplo Prático Completo

Cenário: Processar uma Lista de Produtos

Vamos supor que temos uma lista de produtos e queremos aplicar descontos, filtrar produtos em promoção e coletar os nomes em uma lista.

```

import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class Produto {
    private String nome;
    private double preco;
    private boolean promocao;

    // Construtor
    public Produto(String nome, double preco, boolean promocao) {
        this.nome = nome;
        this.preco = preco;
        this.promocao = promocao;
    }

    // Getters
    public String getNome() {
        return nome;
    }

    public double getPreco() {
        return preco;
    }

    public boolean isPromocao() {
        return promocao;
    }
}

public class ExemploProduto {
    public static void main(String[] args) {
        List<Produto> produtos = Arrays.asList(
            new Produto("Notebook", 3000.0, false),
            new Produto("Smartphone", 2000.0, true),
            new Produto("TV", 2500.0, true),
            new Produto("Mouse", 150.0, false)
        );
    }
}

```

```

    );

    List<String> nomesEmPromocao = produtos.stream()
        .filter(Produto::isPromocao)           // Filtrar
produtos em promoção
        .map(Produto::getNome)                 // Obter
nomes dos produtos
        .collect(Collectors.toList());        // Coletar
em uma lista

    System.out.println("Produtos em promoção: " +
nomesEmPromocao);
    // Saída: Produtos em promoção: [Smartphone, TV]
}
}

```

Convertendo Arrays em Listas e vice-versa

De Array para Lista

```

String[] array = {"A", "B", "C"};
List<String> lista = Arrays.asList(array);

```

De Lista para Array

```

List<String> lista = Arrays.asList("A", "B", "C");
String[] array = lista.toArray(new String[0]);

```

Usando Streams

```

// De Array para Stream
String[] array = {"A", "B", "C"};
Stream<String> stream = Arrays.stream(array);

// De Lista para Stream

```



```
List<String> lista = Arrays.asList("A", "B", "C");  
Stream<String> streamLista = lista.stream();
```

Boas Práticas

- **Evitar Estados Mutáveis:** Evite modificar o estado de objetos dentro das operações de Stream.
- **Operações de Curto-Circuito:** Use `findFirst`, `findAny`, `anyMatch`, etc., para eficiência.
- **Limpar e Conciso:** Mantenha as expressões lambda simples e legíveis.
- **Evitar Processamento Paralelo Desnecessário:** Use `parallelStream` apenas quando apropriado.

Comparação com Abordagem Imperativa

Abordagem Imperativa

```
List<String> nomes = Arrays.asList("Ana", "Bruno", "Carlos",  
    "Amanda", "Beatriz");  
List<String> nomesComA = new ArrayList<>();  
  
for (String nome : nomes) {  
    if (nome.startsWith("A")) {  
        nomesComA.add(nome);  
    }  
}  
  
Collections.sort(nomesComA);  
  
for (String nome : nomesComA) {  
    System.out.println(nome);  
}
```

Abordagem Declarativa com Streams

```
nomes.stream()
    .filter(nome -> nome.startsWith("A"))
    .sorted()
    .forEach(System.out::println);
```

Vantagens da Abordagem Declarativa:

- Código mais conciso e legível.
- Facilita o paralelismo.
- Expressa a intenção sem detalhes de implementação.

Exercícios Práticos

1. Filtrar Números Ímpares e Multiplicar por 3:

Dada uma lista de números inteiros, filtre os números ímpares, multiplique por 3 e coleciona em uma lista.

2. Contar Ocorrências de Palavras:

Dada uma lista de palavras, conte quantas vezes cada palavra aparece usando Streams.

3. Agrupar Pessoas por Idade:

Dada uma lista de objetos `Pessoa` com nome e idade, agrupe as pessoas por idade usando `Collectors.groupingBy`.

Conclusão

A Stream API é uma ferramenta poderosa que permite processar coleções de dados de forma eficiente e expressiva. Compreender como utilizá-la para manipular listas e arrays é essencial para escrever código Java moderno e de alta qualidade.

Referências

- **Java Documentation:** Stream API
(<https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html>)
- **Oracle Tutorial:** Processing Data with Java SE 8 Streams
(<https://www.oracle.com/technical-resources/articles/java/ma14-java-se-8-streams.html>)
- **Livro:** *Java 8 in Action* - Raoul-Gabriel Urma, Mario Fusco, Alan Mycroft

Observações Finais

- Pratique escrevendo código com Streams para se familiarizar com as operações.
- Lembre-se de que nem todas as operações precisam ser feitas com Streams; escolha a abordagem que torna o código mais claro.
- Esteja atento ao desempenho e à legibilidade do código ao usar Streams.

Interfaces em Java: Guia para Iniciantes

O que é uma Interface?

Em Java, uma interface é uma referência de tipo, semelhante a uma classe, que pode conter apenas constantes, assinaturas de métodos, métodos padrão, métodos estáticos e tipos aninhados. Interfaces não podem conter implementações de métodos, exceto para métodos padrão e métodos estáticos.

Interfaces são usadas para definir um contrato para as classes que a implementam. Uma classe que implementa uma interface deve fornecer implementações para todos os métodos abstratos declarados pela interface.

Vantagens de Usar Interfaces

- **Separação de especificação e implementação:** Interfaces permitem que você separe o que algo faz da forma como o faz.
- **Polimorfismo:** Permite que diferentes classes sejam tratadas da mesma maneira.
- **Flexibilidade e escalabilidade:** Facilitam a modificação e a expansão de sistemas.

Desvantagens de Usar Interfaces

- **Restrições de design:** Você pode precisar refatorar o código se os requisitos mudarem significativamente.
- **Curva de aprendizado:** Pode ser um conceito difícil para novos programadores.

Etapa 1: Definindo uma Interface para CRUD

Vamos começar definindo uma interface simples para operações CRUD (Create, Read, Update, Delete) em uma lista em memória.

```
public interface CrudRepository<T> {
    void create(T item);
    T readOne(long id);
    List<T> readAll();
    void update(long id, T item);
    void delete(long id);
}
```

Neste exemplo, `T` é um tipo genérico que representa o tipo de objetos que o repositório irá gerenciar.

Etapa 2: Implementando a Interface

Vamos implementar a interface `CrudRepository` com uma classe que gerencia uma lista de objetos em memória.

```
public class InMemoryCrudRepository<T> implements
    CrudRepository<T> {
    private Map<Long, T> database = new HashMap<>();
    private long currentId = 0;

    @Override
    public void create(T item) {
        database.put(currentId++, item);
    }

    @Override
    public T readOne(long id) {
        return database.get(id);
    }

    @Override
    public List<T> readAll() {
        return new ArrayList<>(database.values());
    }

    @Override
```

```

    public void update(long id, T item) {
        database.put(id, item);
    }

    @Override
    public void delete(long id) {
        database.remove(id);
    }
}

```

Etapa 3: Usando a Implementação

Agora, você pode usar a implementação `InMemoryCrudRepository` para gerenciar qualquer tipo de objeto.

```

public class Main {
    public static void main(String[] args) {
        CrudRepository<User> userRepository = new
        InMemoryCrudRepository<>();

        // Criar um novo usuário
        userRepository.create(new User("John Doe",
        "johndoe@example.com"));

        // Ler e exibir todos os usuários
        List<User> users = userRepository.readAll();
        for (User user : users) {
            System.out.println(user);
        }

        // Atualizar um usuário
        userRepository.update(0, new User("Jane Doe",
        "janedoe@example.com"));

        // Deletar um usuário
        userRepository.delete(0);
    }
}

```

```
}  
}
```

Conclusão

Interfaces em Java são uma maneira poderosa de definir contratos dentro do seu código. Elas permitem que você escreva código mais modular e reutilizável. Embora possam parecer complicadas no início, entender como usar interfaces pode torná-lo um desenvolvedor Java mais eficaz e versátil.

Referências

- Oracle Java Documentation (<https://docs.oracle.com/javase/tutorial/java/landl/createinterface.html>)
- GeeksforGeeks Java Interfaces (<https://www.geeksforgeeks.org/interfaces-in-java/>)

Generics com Interfaces em Java: Guia para Iniciantes

O que são Generics?

Em Java, Generics são uma funcionalidade que permite a você escrever e usar classes e métodos que podem operar em qualquer tipo de objeto. Você pode usar tipos genéricos para criar classes, interfaces e métodos que automaticamente operam com diferentes tipos de dados.

Vantagens de Usar Generics

- **Segurança de tipo:** Generics proporcionam segurança de tipo em tempo de compilação e eliminam a necessidade de conversões manuais de tipos.
- **Reutilização de código:** Com generics, você pode escrever um método ou classe que pode ser utilizado com diferentes tipos de dados.
- **Clareza e robustez:** Código usando generics é mais claro e robusto, pois os erros são capturados em tempo de compilação.

Desvantagens de Usar Generics

- **Complexidade:** Pode tornar o código mais complexo e difícil de entender para novatos.
- **Limitações:** Algumas características de generics podem ser limitadas devido à compatibilidade retroativa.
- **Erros de compilação:** Iniciantes podem achar difícil interpretar os erros de compilação relacionados a generics.

Etapa 1: Definindo uma Interface Genérica para Repositório

Podemos usar generics para definir uma interface de repositório que pode ser usada com qualquer tipo de objeto.

```
public interface Repository<T> {  
    void add(T item);  
    T get(Long id);  
    List<T> getAll();  
    void update(Long id, T item);  
    void remove(Long id);  
}
```

Aqui, **T** é um parâmetro de tipo que será substituído pelo tipo de objeto real quando a interface for implementada.

Etapa 2: Implementando a Interface com Generics

Vamos implementar a interface **Repository** usando generics. Isso permitirá que a mesma classe seja usada para diferentes tipos de dados.

```
public class GenericRepository<T> implements Repository<T> {  
    private Map<Long, T> storage = new HashMap<>();  
    private long currentId = 1;  
  
    @Override  
    public void add(T item) {  
        storage.put(currentId++, item);  
    }  
  
    @Override  
    public T get(Long id) {  
        return storage.get(id);  
    }  
  
    @Override  
    public List<T> getAll() {  
        return new ArrayList<>(storage.values());  
    }  
}
```

```

@Override
public void update(Long id, T item) {
    storage.put(id, item);
}

@Override
public void remove(Long id) {
    storage.remove(id);
}
}

```

Etapa 3: Usando a Implementação Genérica

Agora você pode usar a implementação `GenericRepository` para qualquer tipo de objeto.

```

public class Main {
    public static void main(String[] args) {
        Repository<User> userRepository = new GenericRepository<>
();

        // Adicionar novos usuários
        userRepository.add(new User("John Doe",
"johndoe@example.com"));
        userRepository.add(new User("Jane Doe",
"janedoe@example.com"));

        // Obter e exibir todos os usuários
        List<User> users = userRepository.getAll();
        for (User user : users) {
            System.out.println(user);
        }

        // Atualizar um usuário
        userRepository.update(1L, new User("Johnathan Doe",
"johnathandoe@example.com"));

        // Remover um usuário
        userRepository.remove(1L);
    }
}

```

```
}  
}
```

Conclusão

Generics adicionam flexibilidade e segurança de tipo ao uso de interfaces em Java. Apesar de sua complexidade inicial, eles oferecem vantagens significativas, como reutilização de código e redução de erros de tempo de execução. Entender como usar generics com interfaces pode tornar seu código Java mais versátil e robusto.

Referências

- Oracle Java Documentation (<https://docs.oracle.com/javase/tutorial/java/generics/index.html>)
- GeeksforGeeks Java Generics (<https://www.geeksforgeeks.org/generics-in-java/>)

Projeto Maven e Maven Repository: Guia para Iniciantes

O que é Maven?

Maven é uma ferramenta de automação de build utilizada principalmente para projetos Java. Ele é usado para compilar o código do projeto, empacotá-lo, testá-lo e gerenciar suas dependências e documentação.

Vantagens de Usar Maven

- **Gerenciamento de Dependências:** Simplifica a inclusão e o gerenciamento de bibliotecas.
- **Padronização de Projetos:** Fornece uma estrutura padrão para todos os projetos Maven.
- **Automação de Build:** Compila, testa e empacota o projeto com comandos simples.

Desvantagens de Usar Maven

- **Curva de Aprendizado:** Pode ser complicado para iniciantes.
- **Configuração:** Requer um bom entendimento do arquivo `pom.xml`.
- **Velocidade:** Pode ser mais lento em comparação com outras ferramentas devido à sua abrangência.

Estrutura de um Projeto Maven

A estrutura padrão de um projeto Maven inclui:

- `src/main/java`: Código fonte do projeto.
- `src/main/resources`: Recursos do projeto (propriedades, configuração).

- `src/test/java`: Código de teste.
- `pom.xml`: Arquivo de configuração do Maven.

O que é o Maven Repository?

Um repositório Maven é uma localização central onde as dependências do projeto (bibliotecas, plugins) são armazenadas e recuperadas.

Tipos de Repositórios Maven:

- **Local**: Uma pasta no computador do desenvolvedor.
- **Central**: O repositório central do Maven, acessado publicamente.
- **Remoto**: Outros repositórios personalizados ou empresariais.

Etapa 1: Instalando e Configurando Maven

1. Baixe e instale o Maven a partir do site oficial.
2. Configure a variável de ambiente `MAVEN_HOME` e atualize o `PATH`.

Etapa 2: Criando um Projeto Maven

Execute o seguinte comando para criar um novo projeto Maven:

```
mvn archetype:generate -DgroupId=com.exemplo -DartifactId=projeto-maven -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

Etapa 3: Entendendo o arquivo pom.xml

O `pom.xml` é o coração de um projeto Maven, contendo informações sobre o projeto e listas de dependências. Aqui você define o que precisa e Maven cuida do resto.

Etapa 4: Gerenciando Dependências

Para adicionar uma dependência, insira as coordenadas da dependência dentro da tag `<dependencies>` no `pom.xml`:

```
<dependencies>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.3.2</version>
  </dependency>
</dependencies>
```

Etapa 5: Construindo e Testando o Projeto

Use os comandos `mvn compile`, `mvn test` e `mvn package` para compilar, testar e empacotar seu projeto, respectivamente.

Conclusão

Maven é uma ferramenta poderosa que ajuda no gerenciamento de projetos Java. Apesar de sua curva de aprendizado, oferece vantagens significativas, como o gerenciamento de dependências e a padronização de projetos.

Referências

- Maven Official Site (<https://maven.apache.org/>)
- Maven Repository (<https://mvnrepository.com/>)

Log4j com Interfaces e Generics em Java: Guia para Iniciantes

O que é Log4j?

Log4j é uma biblioteca de logging popular em Java que permite aos desenvolvedores registrar mensagens de log em vários destinos, como arquivos, consoles, sockets de rede e muito mais. Log4j é flexível e configurável, tornando-o uma escolha ideal para adicionar logging a aplicações Java.

Vantagens de Usar Log4j

- **Flexibilidade:** Diferentes níveis de log e múltiplos destinos de log.
- **Desempenho:** Projetado para ser rápido e ter um pequeno impacto no desempenho.
- **Configurável:** Configurações através de arquivos XML, JSON, YAML ou propriedades.

Desvantagens de Usar Log4j

- **Configuração:** Pode ser complexo configurar corretamente.
- **Curva de aprendizado:** Requer algum tempo para aprender e utilizar efetivamente.
- **Dependência:** Adiciona uma dependência externa ao seu projeto.

Etapa 1: Configurando Log4j

Primeiro, você precisa adicionar a dependência Log4j ao seu projeto. Se estiver usando Maven, adicione o seguinte ao seu `pom.xml`:

```
<dependency>
  <groupId>org.apache.logging.log4j</groupId>
  <artifactId>log4j-core</artifactId>
```

```
<version>2.14.1</version> <!-- Use a versão mais recente -->
</dependency>
```

Crie um arquivo `log4j2.xml` na pasta `src/main/resources` do seu projeto com a seguinte configuração básica:

```
<?xml version="1.0" encoding="UTF-8" ?>
<Configuration status="WARN">
  <Appenders>
    <File name="File" fileName="app.log" append="true">
      <PatternLayout pattern="%d{yyyy-MM-dd HH:mm:ss} [%t]
%-5level %logger{36} - %msg %n" />
    </File>
    <Console name="Console" target="SYSTEM_OUT">
      <PatternLayout pattern="%d{yyyy-MM-dd HH:mm:ss} [%t]
%-5level %logger{36} - %msg %n" />
    </Console>
  </Appenders>
  <Loggers>
    <Root level="info">
      <AppenderRef ref="File" />
      <AppenderRef ref="Console" />
    </Root>
  </Loggers>
</Configuration>
```

Na classe `main`, podemos testar o `log4j`, verificando se o arquivo será criado ou o console irá mostrar os registros de log.

```
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

import java.util.Arrays;
import java.util.Scanner;

public class Main {
    public static Logger logger =
```

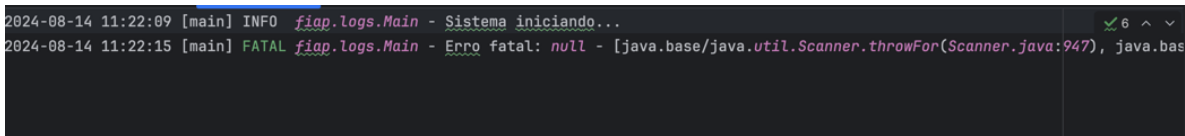


```

LogManager.getLogger(Main.class);
    public static void main(String[] args){
        try{
            logger.info("Sistema iniciando...");

            var scanner = new Scanner(System.in);
            System.out.println("Digite um numero:");
            int numero = scanner.nextInt();
        }
        catch (Exception e){
            logger.fatal("Erro fatal: " + e.getMessage() + " - " +
                Arrays.toString(e.getStackTrace()));
        }
    }
}

```



```

2024-08-14 11:22:09 [main] INFO fiap.logs.Main - Sistema iniciando...
2024-08-14 11:22:15 [main] FATAL fiap.logs.Main - Erro fatal: null - [java.base/java.util.Scanner.throwFor(Scanner.java:947), java.bas

```

logfile.png

Etapa 2: Implementando Logging com Interfaces e Generics

Podemos criar uma interface genérica com métodos de logging padrão e depois implementá-la em nossas classes.

```

public interface Loggable<T> {
    Logger LOGGER = LogManager.getLogger(Loggable.class);

    default void logInfo(T message) {
        LOGGER.info(message);
    }

    default void logError(T message) {
        LOGGER.error(message);
    }
}

```

```
        default void logDebug(T message) {  
            LOGGER.debug(message);  
        }  
    }  
}
```

Etapa 3: Usando a Interface Loggable

Agora, suas classes podem implementar `Loggable<String>` e usar métodos de log diretamente.

```
public class UserRepository implements Loggable<String> {  
    public void addUser(String user) {  
        logInfo("Adding user: " + user);  
        // Código para adicionar usuário  
    }  
  
    public void removeUser(String user) {  
        logInfo("Removing user: " + user);  
        // Código para remover usuário  
    }  
}
```

Conclusão

Usar Log4j com interfaces e generics em Java pode aumentar a reusabilidade do código de logging e manter sua aplicação bem organizada e mais fácil de manter. Apesar de algumas desvantagens, os benefícios de ter um sistema de logging robusto como o Log4j superam os contras.

Referências

- Log4j 2 Apache (<https://logging.apache.org/log4j/2.x/>)
- Baeldung Log4j 2 Tutorial (<https://www.baeldung.com/log4j2>)

Guia de IO em Java: Operações Simples, Arquivos Grandes e Manipulação de JSON

Este guia oferece uma visão passo a passo sobre como realizar operações básicas de entrada e saída (IO) em Java, incluindo a escrita e leitura de arquivos de texto simples, o gerenciamento de arquivos grandes com buffers e o trabalho com JSON.

Sumário

1. Operações Simples de IO

- Escrita em Arquivo TXT
- Leitura de Arquivo TXT

2. Trabalhando com Arquivos Grandes

- Escrita de Arquivos Grandes com Buffer
- Escrita em Modo Append

3. Manipulação de Arquivos JSON

- Escrita de JSON em Arquivo
- Leitura de JSON de Arquivo
- Escrita em Modo Append em JSON

Operações Simples de IO

Escrita em Arquivo TXT

Para escrever dados em um arquivo de texto, usamos as classes `FileWriter` e `BufferedWriter`.

Exemplo:

```

import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;

public class ExportarArquivo {
    public static void main(String[] args) {
        String conteudo = "Este é o conteúdo que será salvo no
arquivo de texto.";
        String caminhoDoArquivo = "saida.txt";

        try (BufferedWriter writer = new BufferedWriter(new
FileWriter(caminhoDoArquivo))) {
            writer.write(conteudo);
            System.out.println("Arquivo exportado com sucesso!");
        } catch (IOException e) {
            System.out.println("Ocorreu um erro ao exportar o
arquivo: " + e.getMessage());
        }
    }
}

```

Explicação:

- **FileWriter:** Escreve dados no arquivo.
- **BufferedWriter:** Melhora a performance ao armazenar temporariamente os dados antes de gravá-los.

Leitura de Arquivo TXT

Para ler o conteúdo de um arquivo de texto, utilizamos `FileReader` e `BufferedReader`.

Exemplo:

```

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class LerArquivo {

```

```

public static void main(String[] args) {
    String caminhoDoArquivo = "saida.txt";

    try (BufferedReader reader = new BufferedReader(new
FileReader(caminhoDoArquivo))) {
        String linha;
        while ((linha = reader.readLine()) != null) {
            System.out.println(linha);
        }
    } catch (IOException e) {
        System.out.println("Ocorreu um erro ao ler o arquivo:
" + e.getMessage());
    }
}
}

```

Explicação:

- **FileReader:** Lê o arquivo.
- **BufferedReader:** Envolve o `FileReader` para leitura eficiente.

Trabalhando com Arquivos Grandes

Escrita de Arquivos Grandes com Buffer

Para escrever arquivos grandes eficientemente, podemos usar `StringBuilder` para construir o conteúdo e `BufferedWriter` para escrevê-lo.

Exemplo:

```

import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;

public class EscreverArquivoGrande {
    public static void main(String[] args) {
        StringBuilder conteudoGrande = new StringBuilder();
        for (int i = 0; i < 1000000; i++) {
            conteudoGrande.append("Esta é a linha número

```

```

    ").append(i).append("\n");
    }

    String caminhoDoArquivo = "arquivo_grande.txt";

    try (BufferedWriter writer = new BufferedWriter(new
    FileWriter(caminhoDoArquivo))) {
        writer.write(conteudoGrande.toString());
        System.out.println("Arquivo grande escrito com
    sucesso!");
    } catch (IOException e) {
        System.out.println("Ocorreu um erro ao escrever o
    arquivo: " + e.getMessage());
    }
}
}

```

Explicação:

- **StringBuilder:** Eficiência na construção de grandes strings.
- **BufferedWriter:** Grava os dados de forma eficiente usando um buffer.

Escrita em Modo Append

Para adicionar conteúdo a um arquivo existente sem sobrescrever o conteúdo anterior, use `FileWriter` no modo append.

Exemplo:

```

import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;

public class AppendArquivo {
    public static void main(String[] args) {
        String conteudoAdicional = "Este é o novo conteúdo que
    será adicionado ao final do arquivo.\n";
        String caminhoDoArquivo = "arquivo_grande.txt";
    }
}

```

```

        try (BufferedWriter writer = new BufferedWriter(new
FileWriter(caminhoDoArquivo, true))) {
            writer.write(conteudoAdicional);
            System.out.println("Conteúdo adicionado ao arquivo com
sucesso!");
        } catch (IOException e) {
            System.out.println("Ocorreu um erro ao adicionar
conteúdo ao arquivo: " + e.getMessage());
        }
    }
}

```

Explicação:

- **Modo append:** O parâmetro `true` no `FileWriter` permite adicionar dados ao final do arquivo.

Manipulação de Arquivos JSON

Dependências

Primeiro, adicione a dependência Gson ao seu `pom.xml` se você estiver usando Maven:

```

<dependencies>
    <dependency>
        <groupId>com.google.code.gson</groupId>
        <artifactId>gson</artifactId>
        <version>2.9.0</version>
    </dependency>
</dependencies>

```

Escrita de JSON em Arquivo

Usamos a biblioteca **Gson** para converter objetos Java em JSON e escrevê-los em arquivos.

Exemplo:

```

import com.google.gson.Gson;
import com.google.gson.GsonBuilder;

```

```

import java.io.FileWriter;
import java.io.IOException;

class Pessoa {
    private String nome;
    private int idade;

    public Pessoa() {
    }

    public Pessoa(String nome, int idade) {
        this.nome = nome;
        this.idade = idade;
    }

    // Getters e Setters
}

public class EscritaJson {
    public static void main(String[] args) {
        Pessoa pessoa = new Pessoa("João", 30);
        Gson gson = new
GsonBuilder().setPrettyPrinting().create();

        try (FileWriter writer = new FileWriter("pessoa.json")) {
            gson.toJson(pessoa, writer);
            System.out.println("Arquivo JSON escrito com
sucesso!");
        } catch (IOException e) {
            System.out.println("Ocorreu um erro ao escrever o
arquivo JSON: " + e.getMessage());
        }
    }
}

```

Explicação:

- **Gson:** Facilita a conversão de objetos Java para JSON.
- **setPrettyPrinting:** Gera JSON formatado de maneira legível.

Leitura de JSON de Arquivo

Podemos ler um arquivo JSON e convertê-lo de volta em um objeto Java.

Exemplo:

```
import com.google.gson.Gson;

import java.io.FileReader;
import java.io.IOException;

public class LeituraJson {
    public static void main(String[] args) {
        Gson gson = new Gson();

        try (FileReader reader = new FileReader("pessoa.json")) {
            Pessoa pessoa = gson.fromJson(reader, Pessoa.class);
            System.out.println("Nome: " + pessoa.getNome());
            System.out.println("Idade: " + pessoa.getIdade());
        } catch (IOException e) {
            System.out.println("Ocorreu um erro ao ler o arquivo
JSON: " + e.getMessage());
        }
    }
}
```

Explicação:

- **fromJson:** Converte JSON de volta para um objeto Java.

Escrita em Modo Append em JSON

Para adicionar mais dados a um arquivo JSON sem sobrescrever, primeiro lemos o conteúdo existente, adicionamos novos dados e depois escrevemos de volta.

Exemplo:

```

import com.google.gson.Gson;
import com.google.gson.GsonBuilder;
import com.google.gson.reflect.TypeToken;

import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.lang.reflect.Type;
import java.util.ArrayList;
import java.util.List;

public class AppendJson {
    public static void main(String[] args) {
        Gson gson = new
GsonBuilder().setPrettyPrinting().create();
        Type listType = new TypeToken<List<Pessoa>>()
{}.getType();
        List<Pessoa> listaPessoas = new ArrayList<>();

        // Tentar ler o conteúdo atual do arquivo, se existir
        try (FileReader reader = new FileReader("pessoa.json")) {
            listaPessoas = gson.fromJson(reader, listType);
        } catch (IOException e) {
            System.out.println("Ocorreu um erro ao ler o arquivo
JSON: " + e.getMessage());
        }

        // Adicionar uma nova pessoa à lista
        Pessoa novaPessoa = new Pessoa("Maria", 25);
        listaPessoas.add(novaPessoa);

        // Escrever a lista de volta no arquivo JSON
        try (FileWriter writer = new FileWriter("pessoa.json")) {
            gson.toJson(listaPessoas, writer);
            System.out.println("Novo conteúdo adicionado ao
arquivo JSON com sucesso!");
        } catch (IOException e) {

```

```
        System.out.println("Ocorreu um erro ao escrever o  
arquivo JSON: " + e.getMessage());  
    }  
}  
}
```

Explicação:

- **TypeToken:** Utilizado para lidar com tipos genéricos, como `List<Pessoa>`.
- **Escrita em modo append:** Lê o conteúdo existente do arquivo, adiciona novos dados e grava de volta no arquivo.

Este guia cobre os fundamentos de IO em Java, incluindo operações básicas de leitura e escrita, gerenciamento eficiente de arquivos grandes, e manipulação de dados JSON usando a biblioteca Gson. Essas técnicas são essenciais para o desenvolvimento de aplicações robustas que precisam interagir com sistemas de arquivos e dados estruturados.

Implementação de CRUD em Java Console Usando JDBC para Oracle

Dependências

Para começar, você precisará incluir a dependência do JDBC para Oracle em seu pom.xml:

```
<dependency>
  <groupId>com.oracle.database.jdbc</groupId>
  <artifactId>ojdbc8</artifactId>
  <version>19.8.0.0</version>
</dependency>
```

Conexão com o Banco de Dados

DatabaseConfig

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class DatabaseConfig {
    private static final String URL =
"jdbc:oracle:thin:@oracle.fiap.com.br:1521:ORCL";
    private static final String USER = "your_username";
    private static final String PASS = "your_password";

    public static Connection getConnection() throws SQLException {
        return DriverManager.getConnection(URL, USER, PASS);
    }
}
```

Modelo

User

```
public class User {  
    private Long id;  
    private String name;  
    private String email;  
  
    // Getters, setters, construtores e outros métodos  
}
```

Inicializar banco

```
import java.sql.Connection;  
import java.sql.Statement;  
  
public class DatabaseInitialization {  
  
    public static void initialize() {  
        // SQL para criar a tabela "users"  
        String createTableSQL =  
            "CREATE TABLE users (" +  
            "id NUMBER(12) PRIMARY KEY," +  
            "name VARCHAR2(255) NOT NULL," +  
            "email VARCHAR2(255) NOT NULL UNIQUE" +  
            ")";  
  
        try (Connection conn = DatabaseConfig.getConnection();  
            Statement stmt = conn.createStatement()) {  
  
            // Criar a tabela  
            stmt.execute(createTableSQL);  
            System.out.println("Tabela 'users' criada com  
sucesso!");  
  
        } catch (Exception e) {  
            System.out.println("Erro ao criar a tabela: " +  
e.getMessage());  
        }  
    }  
}
```

```

    }
}

public static void main(String[] args) {
    initialize();
}
}

```

Repositório

Classe UserRepository

```

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.util.ArrayList;
import java.util.List;

public class UserRepository {

    // Insere um novo usuário
    public void addUser(String name, String email) {
        String insertSQL = "INSERT INTO users (name, email) VALUES"
            "(?, ?)";

        try (Connection conn = DatabaseConfig.getConnection();
            PreparedStatement pstmt =
                conn.prepareStatement(insertSQL)) {

            pstmt.setString(1, name);
            pstmt.setString(2, email);

            pstmt.executeUpdate();
            System.out.println("Usuário inserido com sucesso!");

        } catch (Exception e) {
            System.out.println("Erro ao inserir usuário: " +

```

```

e.getMessage());
    }
}

// Recupera um usuário pelo ID
public User getUserById(int id) {
    String selectSQL = "SELECT * FROM users WHERE id = ?";
    User user = null;

    try (Connection conn = DatabaseConfig.getConnection();
        PreparedStatement pstmt =
conn.prepareStatement(selectSQL)) {

        pstmt.setInt(1, id);
        ResultSet rs = pstmt.executeQuery();

        if (rs.next()) {
            user = new User(rs.getInt("id"),
rs.getString("name"), rs.getString("email"));
        }

    } catch (Exception e) {
        System.out.println("Erro ao obter usuário: " +
e.getMessage());
    }

    return user;
}

// Atualiza um usuário pelo ID
public void updateUser(int id, String newName, String
newEmail) {
    String updateSQL = "UPDATE users SET name = ?, email = ?
WHERE id = ?";

    try (Connection conn = DatabaseConfig.getConnection();
        PreparedStatement pstmt =
conn.prepareStatement(updateSQL)) {

```

```

        pstmt.setString(1, newName);
        pstmt.setString(2, newEmail);
        pstmt.setInt(3, id);

        pstmt.executeUpdate();
        System.out.println("Usuário atualizado com sucesso!");

    } catch (Exception e) {
        System.out.println("Erro ao atualizar usuário: " +
e.getMessage());
    }
}

// Deleta um usuário pelo ID
public void deleteUser(int id) {
    String deleteSQL = "DELETE FROM users WHERE id = ?";

    try (Connection conn = DatabaseConfig.getConnection();
        PreparedStatement pstmt =
conn.prepareStatement(deleteSQL)) {

        pstmt.setInt(1, id);

        pstmt.executeUpdate();
        System.out.println("Usuário excluído com sucesso!");

    } catch (Exception e) {
        System.out.println("Erro ao excluir usuário: " +
e.getMessage());
    }
}

// Lista todos os usuários
public List<User> listAllUsers() {
    List<User> userList = new ArrayList<>();
    String selectAllSQL = "SELECT * FROM users";

```



```

        try (Connection conn = DatabaseConfig.getConnection();
            PreparedStatement pstmt =
conn.prepareStatement(selectAllSQL)) {

            ResultSet rs = pstmt.executeQuery();

            while (rs.next()) {
                User user = new User(rs.getInt("id"),
rs.getString("name"), rs.getString("email"));
                userList.add(user);
            }

        } catch (Exception e) {
            System.out.println("Erro ao listar usuários: " +
e.getMessage());
        }

        return userList;
    }
}

```

Nota: Estou presumindo que existe uma classe User com atributos básicos (id, name e email) e os respectivos getters e setters, além do construtor utilizado nos métodos acima.

Esse repositório contém operações básicas CRUD. A utilização de PreparedStatement ajuda a evitar injeção SQL e torna a interação com o banco de dados mais segura e eficiente.

Aplicação Console

Main

```

import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        initialize();
    }
}

```

```

Scanner scanner = new Scanner(System.in);
UserRepository userRepository = new UserRepository();

while (true) {
    System.out.println("Escolha uma opção: ");
    System.out.println("1. Listar todos os usuários");
    System.out.println("2. Adicionar usuário");

    int choice = scanner.nextInt();

    switch (choice) {
        case 1:
            List<User> users = userRepository.findAll();
            for (User user : users) {
                System.out.println(user);
            }
            break;
        case 2:
            System.out.println("Digite o nome: ");
            String name = scanner.next();
            System.out.println("Digite o email: ");
            String email = scanner.next();
            User user = new User(name, email);
            userRepository.save(user);
            break;
    }
}

```

Notas:

Estou usando a classe DatabaseConfig que foi fornecida anteriormente para obter uma conexão com o banco de dados. A função initialize cria a tabela "users". Você pode estender essa função para criar outras tabelas ou fazer outras configurações iniciais do

banco de dados. O método main é um ponto de entrada simples que chama initialize. Você pode executar esta classe uma vez para configurar seu banco de dados. Lembre-se de ter cuidado ao executar scripts de inicialização, pois eles podem destruir dados existentes ou causar erros se a tabela já existir. Você pode querer adicionar verificações para ver se a tabela já existe antes de tentar criá-la.

Introdução a JAX-RS com GRIZZLY

No JAX-RS (Java API for RESTful Web Services), um "resource" é um componente chave que é modelado para corresponder a um recurso no mundo real que pode ser manipulado via HTTP. Aqui está o passo a passo para entender como um recurso funciona no JAX-RS:

1. Anotações de Recurso

O JAX-RS utiliza anotações para definir recursos e mapear métodos Java para operações HTTP.

@Path: Usada para definir a rota URI para o recurso. @GET, @POST, @PUT, @DELETE: Indicam o tipo de operação HTTP que o método manipulará.

2. Definindo o Recurso

Você define um recurso criando uma classe Java e anotando-a com as anotações do JAX-RS.

Exemplo:

```
@Path("/books")
public class BookResource {

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public List<Book> getAllBooks() {
        // Retorna todos os livros
    }
}
```

3. Métodos de Recurso

Os métodos dentro da classe de recurso são mapeados para operações HTTP através de anotações.

Cada método é responsável por lidar com uma operação específica (ex: GET, POST). Os métodos podem retornar uma resposta que será automaticamente convertida para o

formato apropriado (ex: JSON, XML).

4. Parâmetros de Recurso

Os parâmetros dos métodos de recurso podem ser anotados para puxar informações do pedido HTTP.

- `@PathParam`: Obtém parâmetros da URI.
- `@QueryParam`: Obtém parâmetros de consulta da URL.
- `@HeaderParam`: Obtém informações do cabeçalho HTTP.

```
@GET
@Path("/{id}")
@Produces(MediaType.APPLICATION_JSON)
public Book getBook(@PathParam("id") int id) {
    // Retorna o livro com o ID específico
}
```

5. Resposta

Os métodos de recurso devem retornar uma resposta que será enviada ao cliente.

Você pode retornar um objeto diretamente que será automaticamente convertido (ex: para JSON). Ou você pode usar a classe `Response` para construir uma resposta HTTP detalhada.

```
@POST
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public Response addBook(Book book) {
    // Adiciona um novo livro
    return
    Response.status(Response.Status.CREATED).entity(book).build();
}
```

6. Registro de Recursos

O s recursos precisam ser registrados para serem disponibilizados via HTTP.

Isso pode ser feito através da configuração do aplicativo ou automaticamente se você estiver usando um servidor de aplicativos que suporta a descoberta automática de recursos.

7. Testando o Recurso

Uma vez definido e registrado, o recurso pode ser acessado via HTTP.

Você pode usar ferramentas como curl, Postman, ou um navegador para acessar e testar o recurso.

Resumo: Crie a classe de recurso com métodos que representam operações HTTP. Use anotações JAX-RS para mapear métodos para URIs e operações HTTP. Retorne respostas que podem ser convertidas automaticamente para formatos como JSON ou XML.

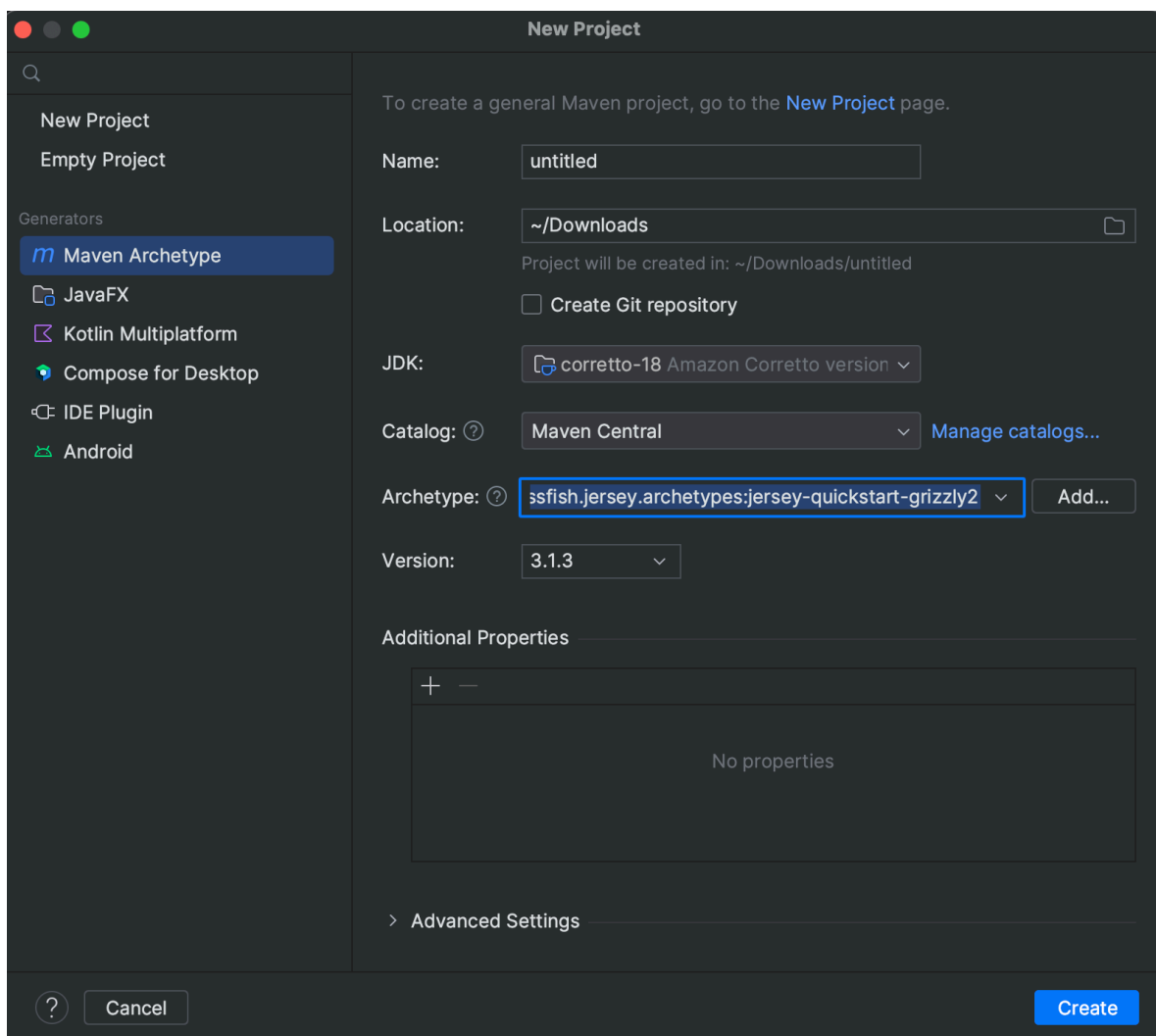
Registre o recurso para torná-lo acessível via HTTP. Teste o recurso usando ferramentas HTTP ou testes automatizados.

Resolução: API REST com JAX-RS e Grizzly

1. Configuração Inicial:

Criando o Projeto Maven:

No IntelliJ, selecione um novo projeto, escolha o tipo Maven Archetype, o catálogo Maven Central e o Arquetipo: `org.glassfish.jersey.archetypes:jersey-quickstart-grizzly2`



jax_rs_template.png

Adicionar Dependência:

No arquivo pom.xml, adicione a dependência do JDBC para Oracle, lembrem-se que outra versão pode ser baixada no link: <https://mvnrepository.com>

```
<dependency>
  <groupId>com.oracle.database.jdbc</groupId>
  <artifactId>ojdbc8</artifactId>
  <version>19.8.0.0</version>
</dependency>
```

2. Modelo: Classe Livro

```
public class Livro {
    private int id;
    private String titulo;
    private String autor;
    private String sinopse;

    // Construtores, getters e setters aqui...
}
```

3. Resource:

```
import javax.ws.rs.*;
import javax.ws.rs.core.MediaType;
import java.util.List;

@Path("/livros")
public class LivroRecurso {

    private LivroRepositorio repositorio = new LivroRepositorio();

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public List<Livro> getLivros() {
        return repositorio.findAll();
    }
}
```



```

@GET
@Path("/{id}")
@Produces(MediaType.APPLICATION_JSON)
public Livro getLivro(@PathParam("id") int id) {
    return repositorio.findById(id).orElse(null);
}

@POST
@Consumes(MediaType.APPLICATION_JSON)
public void addLivro(Livro livro) {
    repositorio.add(livro);
}

@PUT
@Path("/{id}")
@Consumes(MediaType.APPLICATION_JSON)
public void updateLivro(@PathParam("id") int id, Livro livro)
{
    if (repositorio.findById(id).isPresent()) {
        livro.setId(id);
        repositorio.update(livro);
    }
}

@DELETE
@Path("/{id}")
public void deleteLivro(@PathParam("id") int id) {
    repositorio.delete(id);
}
}

```

4. Repositório

os componentes principais do JDBC (Java Database Connectivity), que é uma API Java para conectar e executar operações de banco de dados. Os componentes essenciais

incluem `Connection`, `PreparedStatement` e métodos como `executeQuery()` e `executeUpdate()`.

4.1. Connection

Função: A `Connection` é uma interface no JDBC que fornece métodos para se conectar a um banco de dados, realizar transações, etc.

Como é Utilizada: É obtida através do `DriverManager` ou um `DataSource`. Uma vez obtida, a conexão pode ser usada para criar instâncias de `Statement` ou `PreparedStatement`.

```
Connection connection = DriverManager.getConnection(URL, USERNAME,
PASSWORD);
```

Detalhes: É sempre recomendado fechar a conexão depois de usá-la para liberar recursos.

2. PreparedStatement

Função: É uma interface que estende `Statement`. É usado para executar SQL parametrizado, o que ajuda a prevenir ataques de injeção SQL.

Como é Utilizada: Criado a partir de um objeto `Connection`. Parâmetros são definidos antes de o SQL ser executado. Utilizado para executar consultas SQL parametrizadas e atualizações.

Detalhes: Melhora a performance e segurança em comparação com o `Statement` regular.

```
String sql = "SELECT * FROM users WHERE username = ?";
PreparedStatement statement = connection.prepareStatement(sql);
statement.setString(1, "alice");
ResultSet resultSet = statement.executeQuery();
```

3. executeQuery()

Função: Este método é utilizado para executar consultas SQL que retornam um `ResultSet`, geralmente `SELECT`s.

Como é Utilizada: Invocado em um objeto `Statement` ou `PreparedStatement`. Retorna um `ResultSet` que pode ser usado para acessar os resultados da consulta.

```

ResultSet resultSet = statement.executeQuery();
while (resultSet.next()) {
    String username = resultSet.getString("username");
    // Processa resultado...
}

```

Segue a classe completa de repositório:

```

import java.util.ArrayList;
import java.util.List;
import java.util.Optional;

public class LivroRepositorio {

    public List<Livro> findAll() throws Exception {
        List<Livro> resultList = new ArrayList<>();
        String sql = "SELECT id, titulo, autor, sinopse FROM livros";

        try (Connection conn = DatabaseConfig.getConnection();
            PreparedStatement stmt = conn.prepareStatement(sql);
            ResultSet rs = stmt.executeQuery()) {

            while (rs.next()) {
                resultList.add(new Livro(rs.getInt("id"),
                    rs.getString("titulo"), rs.getString("autor"),
                    rs.getString("sinopse")));
            }

        }

        return resultList;
    }

    public Optional<Livro> findById(int id) throws Exception {
        String sql = "SELECT id, titulo, autor, sinopse FROM livros
        WHERE id = ?";
        Livro livro = null;
    }

```

```

        try (Connection conn = DatabaseConfig.getConnection();
            PreparedStatement stmt = conn.prepareStatement(sql)) {
            stmt.setInt(1, id);
            try (ResultSet rs = stmt.executeQuery()) {
                if (rs.next()) {
                    livro = new Livro(rs.getInt("id"),
rs.getString("titulo"), rs.getString("autor"),
rs.getString("sinopse"));
                }
            }
        }

        return Optional.ofNullable(livro);
    }

    public Livro add(Livro livro) throws Exception {
        String sql = "INSERT INTO livros (titulo, autor, sinopse)
VALUES (?, ?, ?)";

        try (Connection conn = DatabaseConfig.getConnection();
            PreparedStatement stmt = conn.prepareStatement(sql, new
String[]{"ID"})) {
            stmt.setString(1, livro.getTitulo());
            stmt.setString(2, livro.getAutor());
            stmt.setString(3, livro.getSinopse());
            stmt.executeUpdate();

            try (ResultSet generatedKeys = stmt.getGeneratedKeys()) {
                if (generatedKeys.next()) {
                    livro.setId(generatedKeys.getInt(1));
                }
            }
        }

        return livro;
    }

    public void update(Livro livro) throws Exception {

```

```

        String sql = "UPDATE livros SET titulo = ?, autor = ?, sinopse
= ? WHERE id = ?";

        try (Connection conn = DatabaseConfig.getConnection();
            PreparedStatement stmt = conn.prepareStatement(sql)) {
            stmt.setString(1, livro.getTitulo());
            stmt.setString(2, livro.getAutor());
            stmt.setString(3, livro.getSinopse());
            stmt.setInt(4, livro.getId());
            stmt.executeUpdate();
        }
    }

    public void delete(int id) throws Exception {
        String sql = "DELETE FROM livros WHERE id = ?";

        try (Connection conn = DatabaseConfig.getConnection();
            PreparedStatement stmt = conn.prepareStatement(sql)) {
            stmt.setInt(1, id);
            stmt.executeUpdate();
        }
    }
}

```

Links Úteis:

<https://www.treinaweb.com.br/blog/criando-uma-api-restful-com-a-jax-rs-api>

<https://www.baeldung.com/jax-rs-spec-and-implementations>

Tutorial Java: DDD Entities e Repositories vs. Beans/DAO

Introdução

Neste tutorial, vamos explorar os conceitos de Domain-Driven Design (DDD) focando em Entities e Repositories, e como eles diferem do tradicional modelo de Beans e DAO (Data Access Object) em aplicações Java.

O que é DDD?

Domain-Driven Design é uma abordagem para o desenvolvimento de software centrada no domínio e na lógica do negócio. Ela propõe a modelagem baseada no domínio real para o qual o software está sendo desenvolvido..

Entidades (Entities) em DDD

Em DDD, uma Entidade é um objeto que é identificado não pelo seu atributo, mas pela sua identidade única (ex: ID ou chave única).

```
public class Pedido {  
    private Long id;  
    private Date dataPedido;  
    private List<ItemPedido> itens;  
  
    // Getters e Setters, construtores, toString(),  
    equalsAndHashCode(), clone()  
}
```

Entidade Base

A `EntidadeBase` é uma classe abstrata que contém os atributos comuns a todas as outras entidades do sistema, como `id`. Ela serve como uma base para as outras classes, evitando a repetição de código.

Exemplo:

```
public abstract class EntidadeBase {
    protected int id;
    protected bool published;

    // Construtores, getters e setters
}
```

Repositórios (Repositories) em DDD

Repositories são abstrações que gerenciam o armazenamento e recuperação de entidades, sem expor detalhes de implementação do banco de dados.

```
public interface PedidoRepository {
    Pedido findById(Long id);
    void save(Pedido pedido);
    void delete(Pedido pedido);
}
```

Repositório Genérico

A interface RepositorioGenerico T define os métodos CRUD básicos que serão implementados pelos repositórios específicos das entidades.

```
public interface RepositorioGenerico<T> {
    void adicionar(T entidade);
    void atualizar(T entidade);
    void remover(int id);
    List<T> listar();
}
```

Camada de Serviço (Service)

A camada de serviço é responsável por validar os dados das entidades antes de serem persistidos no repositório.

Validador de Entidades

A classe ValidadorEntidades fornece métodos estáticos para validar atributos específicos das entidades, como verificar se o nome não está vazio ou se o ano de

lançamento é válido.

```
public class ProdutoService {  
    public static void validarProduto(Produto produto) {  
        // Validações para o produto  
    }  
}
```

Modelo Beans/DAO

Tradicionalmente, em aplicações Java, usamos o padrão Beans para representar objetos e DAO para interagir com o banco de dados.

Beans

São simples objetos Java que seguem uma convenção de nomenclatura e têm propriedades privadas acessadas por getters e setters.

```
public class PedidoBean {  
    private Long id;  
    private Date dataPedido;  
    // Getters e Setters  
}
```

DAO (Data Access Object)

DAO é um padrão de design que fornece uma interface para interagir com o banco de dados, separando a lógica de acesso ao banco de dados da lógica de negócios.

```
public class PedidoDAO {  
    public PedidoBean findById(Long id) {  
        // Implementação do acesso ao banco de dados  
    }  
}
```

Diferenças entre DDD e Modelo Beans/DAO

1. **Foco no Domínio vs. Foco na Persistência:** DDD se concentra no domínio e na lógica de negócios, enquanto Beans/DAO se concentra em persistência e acesso a dados.
2. **Identidade vs. Estrutura:** Entidades em DDD são definidas por sua identidade, enquanto Beans são definidos por sua estrutura.
3. **Abstração vs. Concretização:** Repositories em DDD abstraem o acesso a dados, enquanto DAOs são implementações concretas de acesso a dados.

Por que usar DDD?

1. **Complexidade do Domínio:** DDD é útil em sistemas complexos onde a lógica do negócio e as regras do domínio são mais importantes e complicadas.
2. **Evolução e Manutenção:** DDD facilita a evolução e manutenção do software, permitindo mudanças no domínio de forma mais ágil.
3. **Colaboração:** Facilita a comunicação entre desenvolvedores e especialistas do domínio através da modelagem baseada no domínio real.

Conclusão

Neste tutorial, abordamos os conceitos básicos de Entities e Repositories em DDD e como eles se diferenciam do modelo Beans/DAO. O uso de DDD pode proporcionar um design mais limpo, focado no domínio e facilitar a manutenção e evolução do software.

Clean Code e Aplicação em Java com Reflections e Anotações

Sumário

1. Introdução
2. Capítulo 1: Clean Code
3. Capítulo 2: Significância dos Nomes
4. Capítulo 3: Funções
5. Aplicação em Java com Reflections e Anotações
6. Comparação de Código: Antes e Depois

Introdução

Este tutorial apresenta os conceitos dos primeiros capítulos do livro "Clean Code" de Robert C. Martin e demonstra como aplicá-los em Java, utilizando reflexões e anotações para simplificar um código grande de uma implementação pura de JDBC. Usaremos uma classe de CRUD chamada `ProdutoRepository` e criaremos uma anotação para facilitar a execução de queries.

Capítulo 1: Clean Code

O primeiro capítulo de "Clean Code" aborda a importância de escrever um código limpo, que é fácil de ler, entender e manter. Robert C. Martin destaca que um código limpo é aquele que se parece com uma leitura natural, tem uma estrutura clara e é escrito de maneira que outros desenvolvedores possam compreendê-lo facilmente.

Princípios de Código Limpo:

- **Legibilidade:** O código deve ser fácil de ler.
- **Simplicidade:** Mantenha o código o mais simples possível.

- **Clareza:** Evite ambiguidades no código.
- **Eliminação de Redundância:** Evite duplicação de código.

Capítulo 2: Significância dos Nomes

Neste capítulo, é enfatizada a importância de escolher nomes significativos para variáveis, funções, classes e outros elementos do código. Nomes bem escolhidos podem tornar o código muito mais legível e compreensível.

Dicas para Nomes Significativos:

- **Use Nomes Descritivos:** Nomes devem descrever claramente a intenção.
- **Evite Abreviações:** Use palavras completas para maior clareza.
- **Consistência:** Use um padrão de nomenclatura consistente em todo o código.

Capítulo 3: Funções

Funções devem ser pequenas e ter uma única responsabilidade. Este capítulo detalha como escrever funções que seguem esses princípios.

Boas Práticas para Funções:

- **Pequenas e Concisas:** Funções devem ser curtas e fazer apenas uma coisa.
- **Nome Descritivo:** O nome da função deve descrever claramente o que ela faz.
- **Evitar Efeitos Colaterais:** Funções não devem ter efeitos colaterais inesperados.

Aplicação em Java com Reflections e Anotações

Reflexões

A reflexão em Java permite que um programa inspecione e modifique seu comportamento em tempo de execução. Isso é útil para criar códigos mais flexíveis e reutilizáveis.

Anotações

Anotações são usadas para fornecer metadados sobre o programa. Elas podem ser processadas em tempo de compilação ou em tempo de execução.

Comparação de Código: Antes e Depois

Código Sem Clean Code

Vamos começar com uma implementação típica de CRUD em JDBC sem aplicar os princípios de Clean Code.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class ProdutoRepository {

    private static final String URL =
"jdbc:mysql://localhost:3306/seu_banco";
    private static final String USER = "seu_usuario";
    private static final String PASSWORD = "sua_senha";

    public Produto findById(int id) {
        Produto produto = null;
        try (Connection connection =
DriverManager.getConnection(URL, USER, PASSWORD)) {
            String query = "SELECT * FROM produtos WHERE id = ?";
            try (PreparedStatement statement =
connection.prepareStatement(query)) {
                statement.setInt(1, id);
                try (ResultSet resultSet =
statement.executeQuery()) {
                    if (resultSet.next()) {
                        produto = new
Produto(resultSet.getInt("id"), resultSet.getString("nome"),
resultSet.getDouble("preco"));
                    }
                }
            }
        }
    }
}
```

```

        }
    }
    } catch (SQLException e) {
        e.printStackTrace();
    }
    return produto;
}

public void save(String nome, double preco) {
    try (Connection connection =
DriverManager.getConnection(URL, USER, PASSWORD)) {
        String query = "INSERT INTO produtos (nome, preco)
VALUES (?, ?)";
        try (PreparedStatement statement =
connection.prepareStatement(query)) {
            statement.setString(1, nome);
            statement.setDouble(2, preco);
            statement.executeUpdate();
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
}

public void update(int id, String nome, double preco) {
    try (Connection connection =
DriverManager.getConnection(URL, USER, PASSWORD)) {
        String query = "UPDATE produtos SET nome = ?, preco =
? WHERE id = ?";
        try (PreparedStatement statement =
connection.prepareStatement(query)) {
            statement.setString(1, nome);
            statement.setDouble(2, preco);
            statement.setInt(3, id);
            statement.executeUpdate();
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
}

```

```

    }
}

public void delete(int id) {
    try (Connection connection =
DriverManager.getConnection(URL, USER, PASSWORD)) {
        String query = "DELETE FROM produtos WHERE id = ?";
        try (PreparedStatement statement =
connection.prepareStatement(query)) {
            statement.setInt(1, id);
            statement.executeUpdate();
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
}

```

Código Com Clean Code e Anotações

Agora vamos reescrever a classe `ProdutoRepository` aplicando os princípios de Clean Code e utilizando anotações para simplificar a execução das queries.

Passo 1: Criação da Anotação `@Query`

```

import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

@Retention(RetentionPolicy.RUNTIME)
public @interface Query {
    String value();
}

```

Passo 2: refatoração do repository:

```

public class ProdutoRepository {

    @Query("SELECT * FROM produtos WHERE id = ?")
    public Produto findById(int id) {

```

```

        // Implementação será simplificada pelo QueryProcessor
        return null;
    }

    @Query("INSERT INTO produtos (nome, preco) VALUES (?, ?)")
    public void save(String nome, double preco) {
        // Implementação será simplificada pelo QueryProcessor
    }

    @Query("UPDATE produtos SET nome = ?, preco = ? WHERE id = ?")
    public void update(int id, String nome, double preco) {
        // Implementação será simplificada pelo QueryProcessor
    }

    @Query("DELETE FROM produtos WHERE id = ?")
    public void delete(int id) {
        // Implementação será simplificada pelo QueryProcessor
    }
}

```

Passo 3: Processamento de Anotações com Reflexões

```

import java.lang.reflect.Method;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;

public class QueryProcessor {

    private static final String URL =
        "jdbc:mysql://localhost:3306/seu_banco";
    private static final String USER = "seu_usuario";
    private static final String PASSWORD = "sua_senha";

    public static void executeAnnotatedMethods(Object obj,
        Object... params) throws Exception {
        Class<?> clazz = obj.getClass();
    }
}

```

```

        for (Method method : clazz.getDeclaredMethods()) {
            if (method.isAnnotationPresent(Query.class)) {
                Query query = method.getAnnotation(Query.class);
                executeQuery(query.value(), params);
            }
        }
    }

    private static void executeQuery(String query, Object...
params) throws Exception {
        try (Connection connection =
DriverManager.getConnection(URL, USER, PASSWORD);
            PreparedStatement statement =
connection.prepareStatement(query)) {

            for (int i = 0; i < params.length; i++) {
                statement.setObject(i + 1, params[i]);
            }

            if (query.trim().toUpperCase().startsWith("SELECT")) {
                try (ResultSet resultSet =
statement.executeQuery()) {
                    while (resultSet.next()) {
                        // Processar resultado
                    }
                }
            } else {
                statement.executeUpdate();
            }
        }
    }

    public static void main(String[] args) throws Exception {
        ProdutoRepository repository = new ProdutoRepository();
        executeAnnotatedMethods(repository, 1); // Exemplo para
encontrar produto por ID
    }

```



```
}  
}
```

Comparação e Benefícios

Antes (Sem Clean Code)

Código repetitivo: Cada método do repositório contém código similar para abrir conexões, preparar statements e executar queries. Baixa reutilização: Código para manipulação de banco de dados não é reutilizável. Difícil manutenção: Mudanças na lógica de conexão ou tratamento de exceções requerem alterações em múltiplos lugares.

Depois (Com Clean Code)

Redução de Redundância: A lógica para abrir conexões e executar queries é centralizada no QueryProcessor. Aumento da Legibilidade: Uso de anotações @Query deixa claro o propósito de cada método sem se preocupar.

Infraestrutura Comum de uma Aplicação Enterprise: CRUD, DDD, Oracle, Java JAX-RS e React

Introdução

Este documento descreve os conceitos fundamentais e a infraestrutura comum envolvida na construção de uma aplicação enterprise usando Domain-Driven Design (DDD), Java com JAX-RS para o backend, Oracle como banco de dados, e React para o frontend.

Componentes da Infraestrutura

Banco de Dados Oracle

Oracle é um sistema de gerenciamento de banco de dados relacional robusto e amplamente utilizado em ambientes empresariais. Suporta grandes volumes de dados e transações, tornando-o ideal para aplicações empresariais.

Características do Oracle:

- Escalabilidade e desempenho.
- Recursos avançados de segurança.
- Suporte para transações complexas.

Backend Java com JAX-RS

Java é uma linguagem de programação orientada a objetos usada para desenvolver o backend de aplicações. JAX-RS (Java API for RESTful Web Services) é uma especificação Java para a criação de serviços web RESTful.

Características do Jax-RS:

- Portabilidade e robustez.
- Suporte a múltiplas bibliotecas e frameworks.

- Integração com DDD e acesso ao banco de dados Oracle.

Domain-Driven Design (DDD)

DDD é uma abordagem de design focada no domínio e na lógica do negócio. No contexto de aplicações enterprise, DDD ajuda a criar uma arquitetura limpa e modular.

Elementos:

- Entidades (Entities): Objetos com identidade única.
- Objetos de Valor (Value Objects): Objetos imutáveis que descrevem características.
- Agregados: Grupo de entidades e objetos de valor.
- Repositórios: Interfaces para acessar entidades.

Frontend com React

React é uma biblioteca JavaScript para construir interfaces de usuário. É comumente usado para desenvolver o frontend de aplicações web de página única (SPAs).

Características:

- Componentes reutilizáveis e declarativos.
- Estado e ciclo de vida dos componentes.
- Integração com APIs RESTful.

Arquitetura da Aplicação

Uma aplicação enterprise típica inclui:

1. **Camada de Apresentação (Frontend):** Desenvolvida com React, apresenta a interface do usuário e interage com o backend através de APIs RESTful.
2. **Camada de Aplicação (Backend):** Implementa a lógica de negócio usando Java e JAX-RS. Adota os princípios do DDD para organizar o código.
3. **Camada de Domínio:** Contém a lógica e as regras de negócio do domínio da aplicação.
4. **Camada de Infraestrutura:** Inclui o acesso ao banco de dados Oracle, integrações

externas e outros serviços de suporte.

Conclusão

A combinação de DDD, Java com JAX-RS, Oracle e React oferece uma infraestrutura sólida para o desenvolvimento de aplicações enterprise. Esta abordagem não apenas facilita a gestão da complexidade do negócio, mas também proporciona uma aplicação escalável, segura e de alto desempenho.

Guia de Instalação: Visual Studio e JetBrains Rider

Este guia oferece instruções detalhadas para a instalação do Visual Studio da Microsoft e do Rider da JetBrains, duas IDEs amplamente utilizadas para desenvolvimento de software.

Índice

1. Instalando o Visual Studio

- Passos de Instalação

2. Instalando o JetBrains Rider

- Passos de Instalação

Instalando o Visual Studio

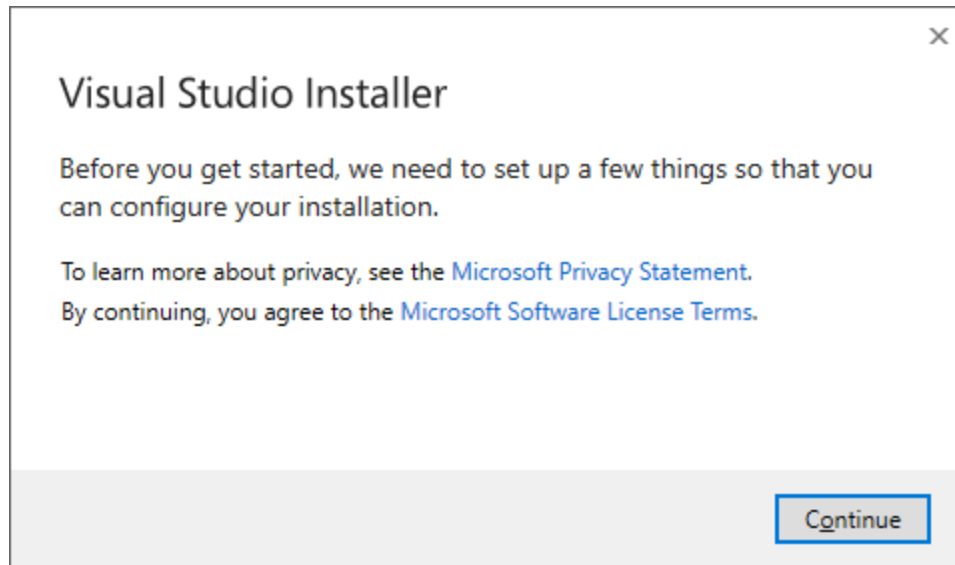
Passos de Instalação

1. Baixe o Instalador:

- Acesse o site oficial do Visual Studio (<https://visualstudio.microsoft.com/downloads/>).
- Escolha a edição community que não exige nenhuma licença

2. Execute o Instalador:

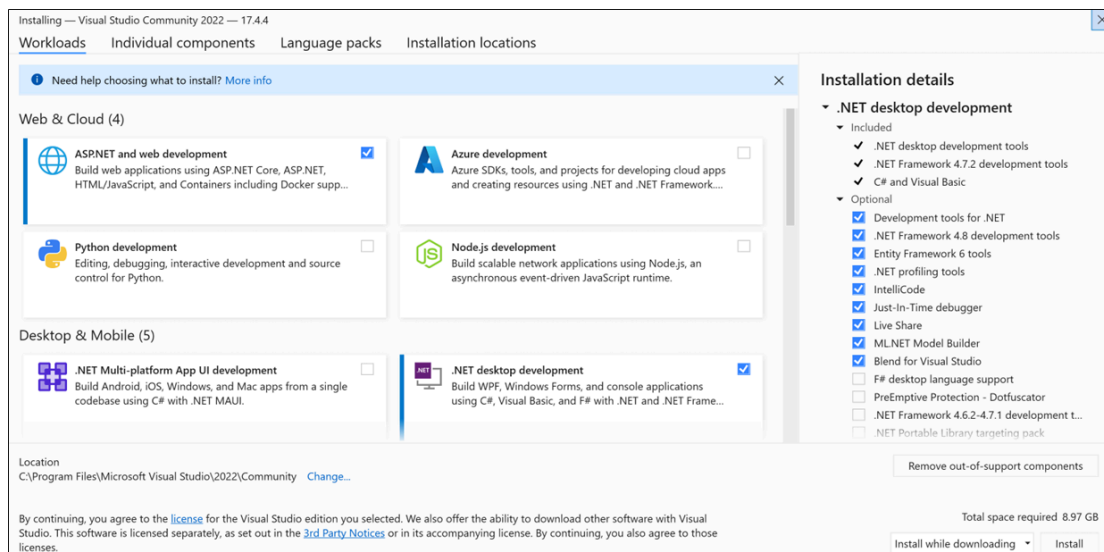
- Localize o arquivo `.exe` baixado e execute-o.
- Se for solicitado pela Controle de Conta de Usuário, clique em "Sim" para continuar.



vs_installer

3. Escolha as Cargas de Trabalho:

- O instalador permitirá que você selecione as "cargas de trabalho" que deseja instalar, como Desenvolvimento Web, Desktop, ou C++.
- Você pode personalizar ainda mais a instalação selecionando componentes individuais na aba correspondente.



vs_cargas_de_trabalho.png

4. Inicie a Instalação:

- Após selecionar as cargas de trabalho e componentes desejados, clique em "Instalar".
- O processo pode levar algum tempo, dependendo da velocidade de sua conexão e do número de componentes escolhidos.

5. Finalização:

- Ao final da instalação, você pode iniciar o Visual Studio e começar a desenvolver seus projetos.

Para mais informações, consulte a documentação oficial do Visual Studio (<https://learn.microsoft.com/pt-br/visualstudio/install/install-visual-studio?view=vs-2022>).

Instalando o JetBrains Rider

Passos de Instalação JetBrains Rider

1. Baixe o Instalador:

- Visite a página de downloads do Rider (<https://www.jetbrains.com/rider/download/>) e escolha a versão correspondente ao seu sistema operacional.

2. Instale o Rider:

- No Windows, execute o instalador `.exe`.
- No macOS, abra o arquivo `.dmg` e arraste o Rider para a pasta Aplicativos.
- No Linux, extraia o arquivo `.tar.gz` para o diretório desejado e execute o script `rider.sh`.

3. Configuração Inicial:

- Ao iniciar o Rider pela primeira vez, configure o ambiente de acordo com suas preferências. Você pode importar configurações de outras IDEs da JetBrains se desejar.

4. Ativação:

- Entre com sua conta JetBrains para ativar o Rider ou use a licença de avaliação gratuita.

5. Conclusão:

- Após a configuração, o Rider estará pronto para ser usado em seus projetos.

See also

Outros Links

<https://learn.microsoft.com/pt-br/visualstudio/install/install-visual-studio?view=vs-2022>
https://www.jetbrains.com/help/rider/Installation_guide.html

Tutorial: Criando sua Primeira Aplicação Console em C#

Este tutorial irá guiá-lo na criação de uma aplicação console simples em C# usando o Visual Studio.

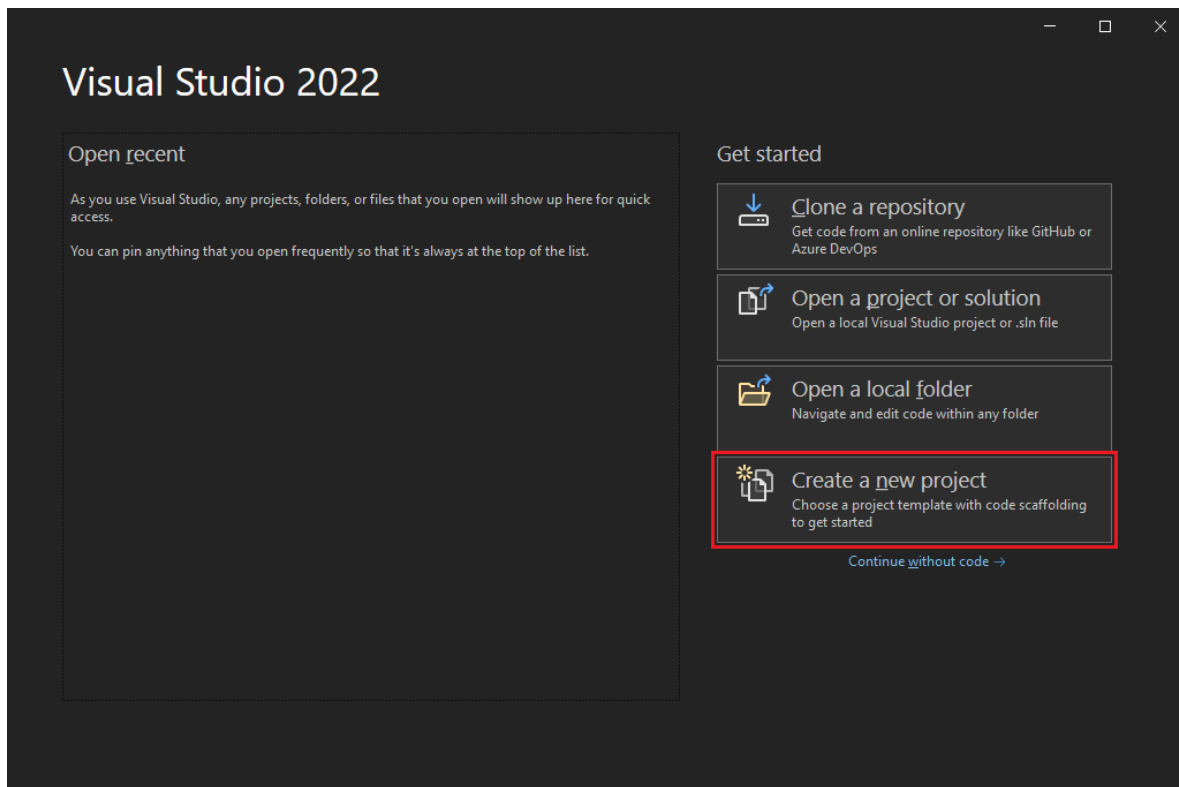
Pré-requisitos

- **Visual Studio 2022:** Certifique-se de ter o Visual Studio 2022 instalado. Se você ainda não o instalou, siga o guia de instalação (<https://learn.microsoft.com/pt-br/visualstudio/install/install-visual-studio?view=vs-2022>).

Passos para Criar uma Aplicação Console

Passo 1: Criar um Novo Projeto

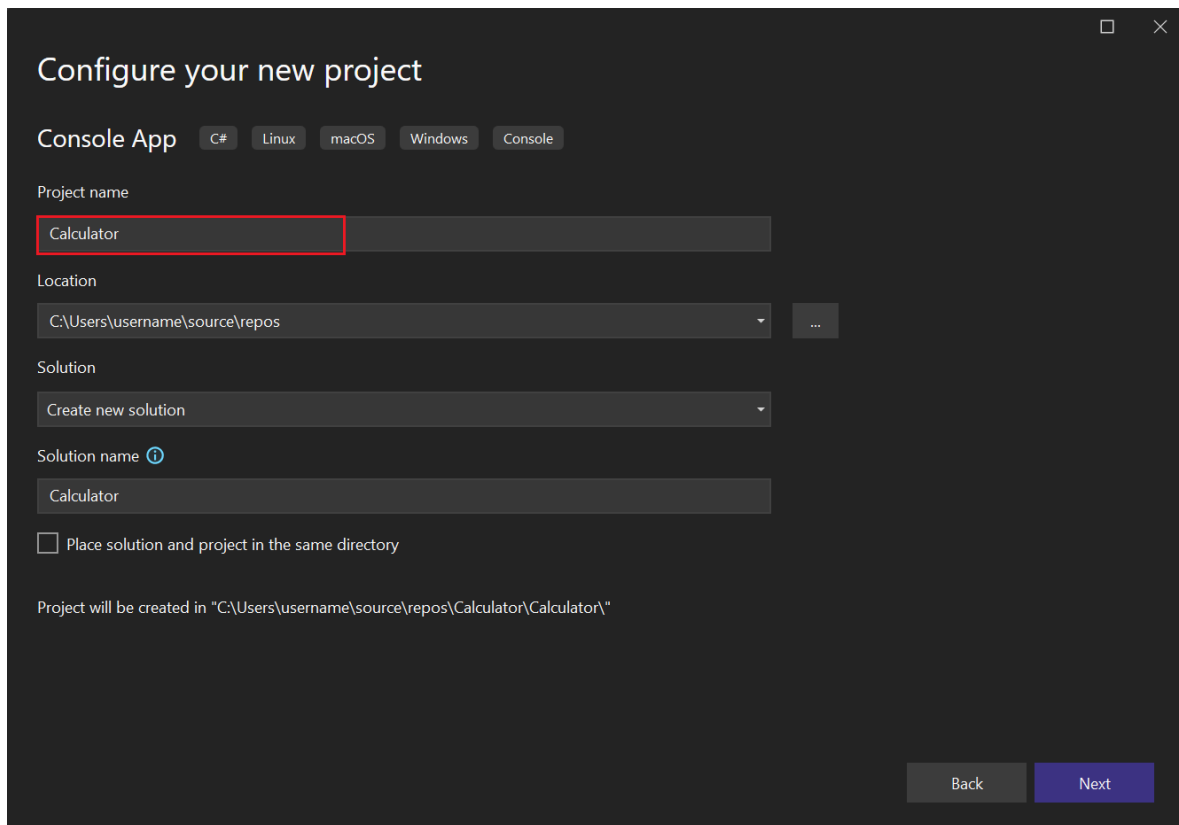
1. **Abrir o Visual Studio:** Inicie o Visual Studio



vs_new_project.png

2. Criar um novo projeto:

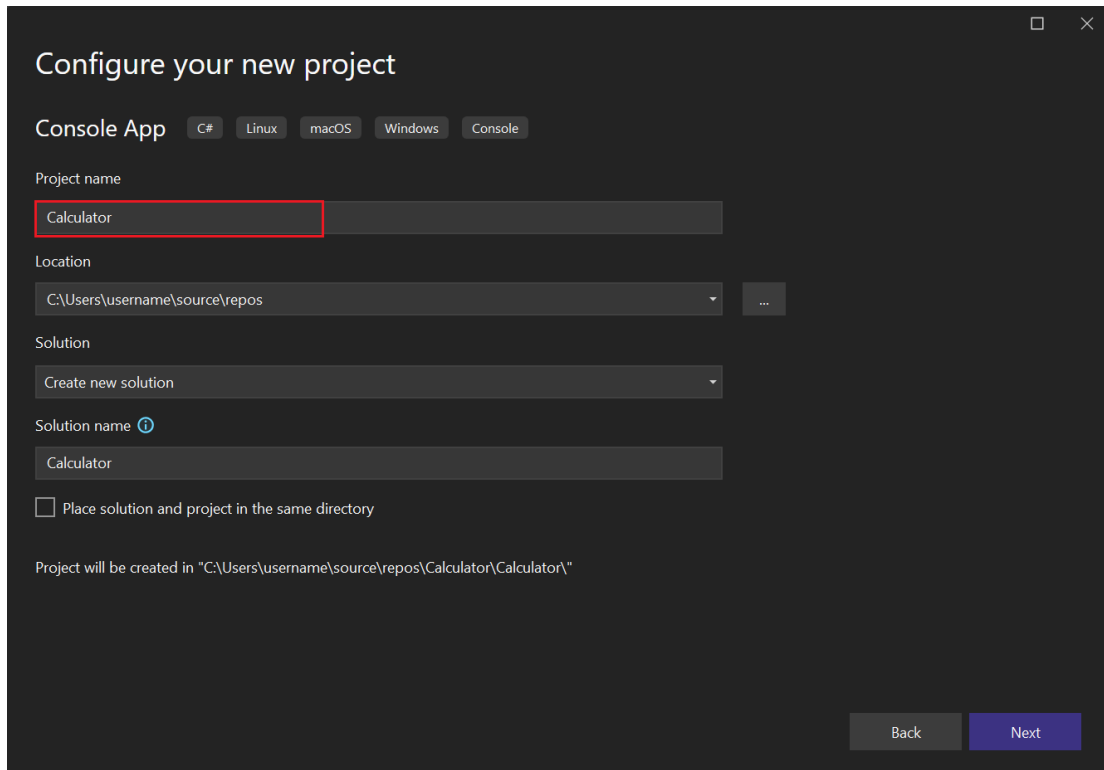
- Na janela inicial, clique em "**Criar um novo projeto**".
- No campo de busca, digite "**Aplicativo de Console**" e selecione "**Aplicativo de Console C#**".
- Clique em "**Avançar**".
- obs: certifique-se de ter instalado as cargas de trabalho de .NET



vs_console_new_app.png

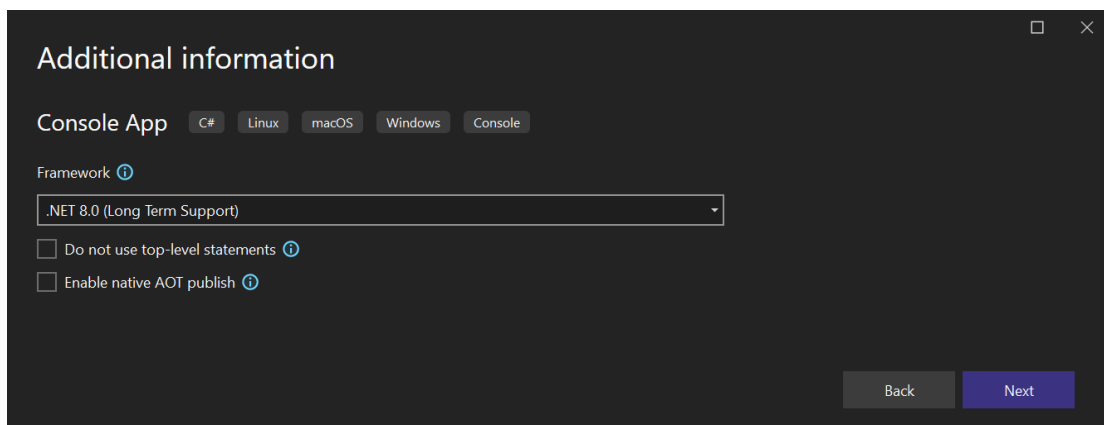
3. Configurar o novo projeto:

- Dê um nome ao projeto, por exemplo, "Calculadora".



vs_calculator_app.png

- Escolha um local para salvar o projeto.
- Clique em "Avançar".
- Na janela "Informações adicionais", selecione **.NET 8.0 (Long-term support)** como o framework de destino.
- Clique em "Criar".



net_version.png

Passo 2: Escrever o Código

1. Abrir o arquivo Program.cs:

- No **Gerenciador de Soluções**, clique duas vezes em **Program.cs** para abrir o arquivo no editor de código.

2. Substituir o código padrão:

- Apague o código existente e substitua pelo seguinte:

```
using System;

namespace Calculadora
{
    class Program
    {
        static void Main(string[] args)
        {
            int a = 42;
            int b = 119;
            int c = a + b;
            Console.WriteLine(c);
            Console.ReadKey();
        }
    }
}
```

Passo 3: Executar a Aplicação

1. Compilar e Executar:

- Para compilar e executar a aplicação, pressione **F5** ou clique na seta verde "Iniciar" na barra de ferramentas superior.
- Uma janela de console abrirá mostrando o resultado da soma de $42 + 119$, que é 161.

Passo 4: Adicionar Funcionalidade de Calculadora

Para estender a funcionalidade e transformar a aplicação em uma calculadora interativa, substitua o código em **Program.cs** pelo seguinte:

```
using System;

namespace Calculadora
{
    class Program
    {
        static void Main(string[] args)
        {
            bool endApp = false;
            Console.WriteLine("Console Calculator in C#\r");
            Console.WriteLine("-----\n");

            while (!endApp)
            {
                string numInput1 = "";
                string numInput2 = "";
                double result = 0;

                Console.WriteLine("Type a number, and then press
Enter");
                numInput1 = Console.ReadLine();

                double cleanNum1 = 0;
                while (!double.TryParse(numInput1, out cleanNum1))
                {
                    Console.Write("This is not valid input. Please
enter an integer value: ");
                    numInput1 = Console.ReadLine();
                }

                Console.WriteLine("Type another number, and then
press Enter");
                numInput2 = Console.ReadLine();
```

```

        double cleanNum2 = 0;
        while (!double.TryParse(numInput2, out cleanNum2))
        {
            Console.WriteLine("This is not valid input. Please
enter an integer value: ");
            numInput2 = Console.ReadLine();
        }

        Console.WriteLine("Choose an option from the
following list:");
        Console.WriteLine("\ta - Add");
        Console.WriteLine("\ts - Subtract");
        Console.WriteLine("\tm - Multiply");
        Console.WriteLine("\td - Divide");
        Console.Write("Your option? ");

        string op = Console.ReadLine();

        try
        {
            result = Calculator.DoOperation(cleanNum1,
cleanNum2, op);
            if (double.IsNaN(result))
            {
                Console.WriteLine("This operation will
result in a mathematical error.\n");
            }
            else Console.WriteLine("Your result:
{0:0.##}\n", result);
        }
        catch (Exception e)
        {
            Console.WriteLine("Oh no! An exception
occurred trying to do the math.\n - Details: " + e.Message);
        }

        Console.WriteLine("-----\n");

```

```

        Console.WriteLine("Press 'n' and Enter to close the
app, or press any other key and Enter to continue: ");
        if (Console.ReadLine() == "n") endApp = true;

        Console.WriteLine("\n");
    }
    return;
}

class Calculator
{
    public static double DoOperation(double num1, double num2,
string op)
    {
        double result = double.NaN;

        switch (op)
        {
            case "a":
                result = num1 + num2;
                break;
            case "s":
                result = num1 - num2;
                break;
            case "m":
                result = num1 * num2;
                break;
            case "d":
                if (num2 != 0)
                {
                    result = num1 / num2;
                }
                break;
            default:
                break;
        }
        return result;
    }
}

```



```
}  
    }  
}
```

See also

Outros Links

<https://learn.microsoft.com/pt-br/visualstudio/get-started/csharp/tutorial-console-part-2?view=vs-2022>

<https://learn.microsoft.com/pt-br/visualstudio/get-started/csharp/tutorial-console?view=vs-2022>

Tutorial: Web Scraping em C# com HtmlAgilityPack

Introdução

Web scraping é a prática de extrair dados de websites. Em C#, a biblioteca **HtmlAgilityPack** é amplamente utilizada para essa tarefa. Este guia irá te mostrar como configurar um projeto de web scraping em C# e como paralelizar as tarefas para aumentar a eficiência.

Configurando o Projeto

Instalando HtmlAgilityPack

Para começar, você precisa instalar o pacote **HtmlAgilityPack**. Isso pode ser feito usando o NuGet Package Manager no Visual Studio:

```
Install-Package HtmlAgilityPack
```

Criando o Documento HTML

Com o **HtmlAgilityPack**, você pode carregar uma página HTML e manipulá-la como um documento:

```
var web = new HtmlWeb();  
var doc = web.Load("URL_DA_PAGINA");
```

Extraindo Dados

Você pode extrair dados utilizando **XPath** ou **manipulação de nós**:

```
var nodes =  
doc.DocumentNode.SelectNodes("//tagname[@attribute='value']");  
foreach (var node in nodes)  
{
```

```
        Console.WriteLine(node.InnerText);  
    }
```

Paralelizando o Web Scraping

Por que Paralelizar?

Se você precisa processar várias páginas, fazer isso em série pode ser ineficiente. Em vez disso, podemos paralelizar as requisições para economizar tempo.

Usando Parallel.ForEach

O C# oferece a classe `Parallel` para executar loops em paralelo:

```
var urls = new List<string> { "URL1", "URL2", "URL3" };  
Parallel.ForEach(urls, url =>  
{  
    var web = new HtmlWeb();  
    var doc = web.Load(url);  
    // Processa o documento aqui  
});
```

Gerenciando o Nível de Paralelismo

É importante controlar o nível de paralelismo para evitar sobrecarregar o sistema ou ser bloqueado pelo site:

```
var options = new ParallelOptions { MaxDegreeOfParallelism = 4 };  
Parallel.ForEach(urls, options, url =>  
{  
    var web = new HtmlWeb();  
    var doc = web.Load(url);  
    // Processa o documento aqui  
});
```

Considerações Finais

- Respeite o `robots.txt` dos sites: Nem todos os sites permitem web scraping. Verifique

o arquivo `robots.txt` do site para entender suas políticas.

- **Gestão de Erros:** Sempre inclua tratamento de exceções para gerenciar erros de rede ou problemas na extração de dados.
- **Intervalos entre Requisições:** Considere implementar um intervalo entre as requisições para evitar sobrecarregar o servidor.

See also

Outros Links

HtmlAgilityPack Documentation (<https://html-agility-pack.net/>)

Artigo: How to make webscrapping using C# (<https://dev.to/leonardogr/how-to-make-webscrapping-using-c-597b>)

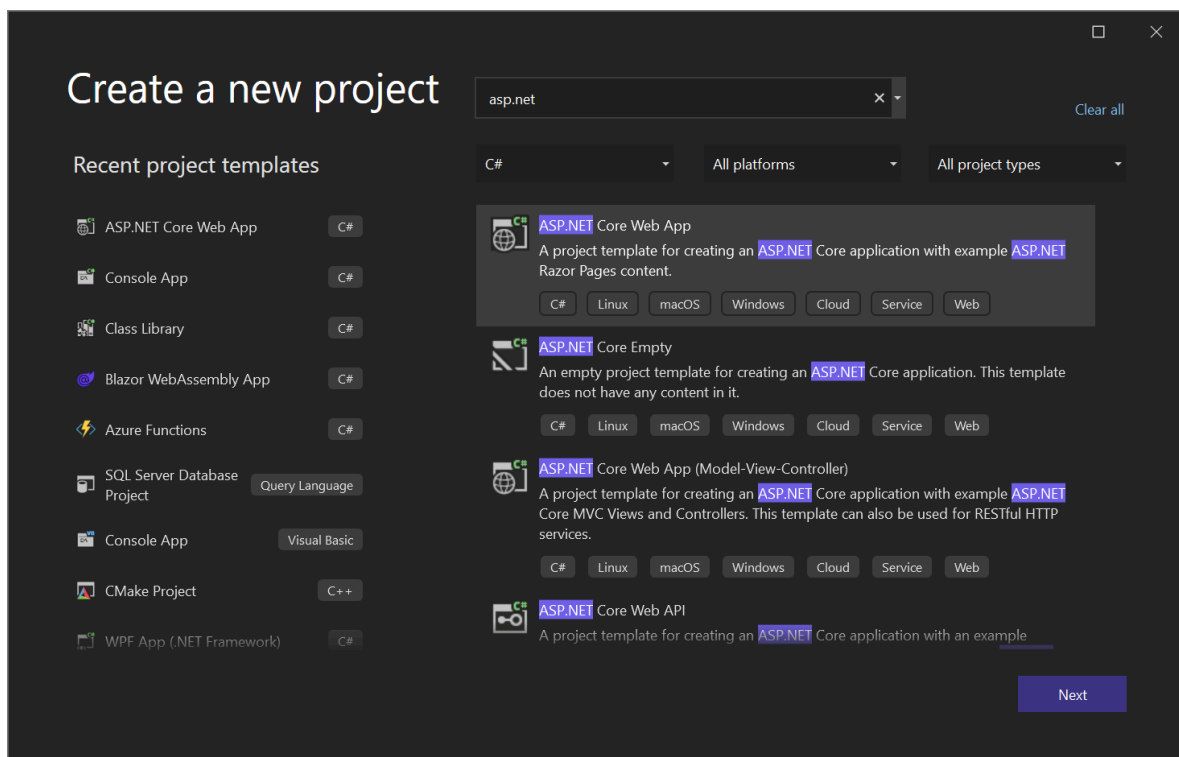
Artigo: Speed up web scrapping using C# (<https://dev.to/leonardogr/parallelizing-web-scrapping-using-c-1d1p>)

Artigo: Agility Pack Geeks for Geeks (<https://www.geeksforgeeks.org/how-to-use-html-agility-pack/>)

ASP.NET Core MVC: Introdução

Introdução

ASP.NET Core MVC é um framework robusto para construir aplicações web modernas e escaláveis usando o padrão Model-View-Controller (MVC). Este guia explorará os componentes fundamentais do ASP.NET Core MVC, incluindo o Razor, os controllers, os results e como as models são usadas dentro das views.



vs_projects_templates.png

1. Arquitetura MVC no ASP.NET Core

Antes de mergulharmos nos detalhes, é essencial entender o padrão MVC:

- **Model (Modelo):** Representa os dados da aplicação e a lógica de negócios.
- **View (Visão):** Responsável pela apresentação dos dados ao usuário.
- **Controller (Controlador):** Intermediário que manipula as

solicitações do usuário, interage com o modelo e seleciona a view adequada para renderizar.

2. Razor: O Motor de Visualização

O Razor é a linguagem de marcação usada nas views do ASP.NET Core MVC. Ele permite misturar código C# com HTML de forma fluida e expressiva.

Sintaxe Básica

Usa o símbolo @ para transicionar entre HTML e C#.

```
<h1>@ViewData["Title"]</h1>
<p>Bem-vindo, @Model.Nome!</p>
```

Helpers do Razor

Facilita a geração de elementos HTML com funcionalidades do servidor.

```
@Html.ActionLink("Home", "Index", "Home")
```

Layouts e Seções

Permite criar layouts comuns para várias views, promovendo reutilização.

```
@{
    Layout = "_Layout";
}
```

3. Controllers (Controladores)

Os controllers são responsáveis por manipular as solicitações HTTP e retornar respostas apropriadas.

Criando um Controller

Herda da classe Controller.

```
public class HomeController : Controller
{
    public IActionResult Index()
```

```
{  
    return View();  
}
```

Actions (Ações)

Métodos dentro do controller que correspondem a endpoints.

```
public IActionResult Sobre()  
{  
    return View();  
}
```

Recebendo Dados do Usuário

Via parâmetros de método, query strings, formulários, etc.

```
[HttpPost]  
public IActionResult Contato(FormularioContato model)  
{  
    if (ModelState.IsValid)  
    {  
        // Processar o formulário  
    }  
    return View(model);  
}
```

4. Results (Resultados)

Os métodos de ação retornam objetos que derivam de `IActionResult`, indicando como o framework deve processar a resposta.

Tipos Comuns de Resultados

```
//ViewResult: Renderiza uma view.  
return View();
```

```
//RedirectResult: Redireciona para outra ação ou URL.
```

```
return RedirectToAction("Index");

//JsonResult: Retorna dados em formato JSON.
return Json(novoObjeto);

//ContentResult: Retorna conteúdo textual.
return Content("Olá, mundo!");

//StatusCodeResult: Retorna um código HTTP específico.
return StatusCode(404);
```

5. Models dentro das Views

As models contêm os dados que serão exibidos nas views.

Passando Model para a View

Do controller para a view.

```
public IActionResult Detalhes(int id)
{
    var produto = _context.Produtos.Find(id);
    return View(produto);
}
```

Definindo a Model na View

No topo da view, especifica-se o tipo de model.

```
@model MeuProjeto.Models.Produto

<h2>@Model.Nome</h2>
<p>Preço: @Model.Preco</p>
```

ViewData e ViewBag

Alternativas para passar dados dinâmicos.

```
ViewData["Mensagem"] = "Bem-vindo!";
```



```
ViewBag.Saudacao = "Olá!";
```

```
<p>@ViewData["Mensagem"]</p>  
<p>@ViewBag.Saudacao</p>
```

6. Exemplo Prático

Model: Produto

```
public class Produto  
{  
    public int Id { get; set; }  
    public string Nome { get; set; }  
    public decimal Preco { get; set; }  
}
```

Controller: ProdutosController

```
public class ProdutosController : Controller  
{  
    private readonly List<Produto> _produtos = new List<Produto>  
    {  
        new Produto { Id = 1, Nome = "Caneta", Preco = 2.00M },  
        new Produto { Id = 2, Nome = "Caderno", Preco = 15.00M },  
        new Produto { Id = 3, Nome = "Borracha", Preco = 1.50M },  
    };  
  
    public IActionResult Index()  
    {  
        return View(_produtos);  
    }  
  
    public IActionResult Detalhes(int id)  
    {  
        var produto = _produtos.FirstOrDefault(p => p.Id == id);  
        if (produto == null)  
            return NotFound();  
    }  
}
```

```
        return View(produto);
    }
}
```

View: Index.cshtml

```
@model IEnumerable<MeuProjeto.Models.Produto>

<h1>Lista de Produtos</h1>
<ul>
    @foreach (var produto in Model)
    {
        <li>
            @Html.ActionLink(produto.Nome, "Detalhes", new { id =
produto.Id })
        </li>
    }
</ul>
```

View: Details.cshtml

```
@model MeuProjeto.Models.Produto

<h1>@Model.Nome</h1>
<p>Preço: @Model.Preco.ToString("C")</p>
<p>@Html.ActionLink("Voltar", "Index")</p>
```

7. Conclusão

Neste guia, exploramos os fundamentos do ASP.NET Core MVC:

- Razor: Combinação elegante de C# e HTML para gerar conteúdo dinâmico.
- Controllers: Manipulam solicitações e determinam as respostas.
- Results: Diferentes formas de retornar respostas ao cliente.

- **Models e Views:** Como os dados são estruturados e apresentados ao usuário.

Compreender esses conceitos é essencial para desenvolver aplicações web eficientes com o ASP.NET Core MVC.

See also

Outros Links

<https://docs.microsoft.com/pt-br/aspnet/core/mvc>

<https://docs.microsoft.com/pt-br/aspnet/core/tutorials/first-mvc-app/start-mvc>

ASP.NET Core MVC: Partial Views

Introdução

Em aplicações web, frequentemente encontramos a necessidade de reutilizar trechos de código HTML em diferentes partes do site. Para evitar a repetição de código e facilitar a manutenção, o **ASP.NET Core MVC** oferece o conceito de **Partial Views**. Elas permitem encapsular porções da interface do usuário em componentes reutilizáveis e modulares.

Este guia irá explicar o que são Partial Views, como e quando utilizá-las, e fornecer exemplos práticos de implementação em um projeto ASP.NET Core MVC.

O que são Partial Views?

Uma **Partial View** é um arquivo de visualização (view) que pode ser renderizado dentro de outra view. Elas funcionam como componentes reutilizáveis que ajudam a manter o código organizado e promovem a separação de responsabilidades na camada de apresentação.

Características das Partial Views

- **Reutilização de Código:** Evita duplicação de código em diferentes views.
- **Modularidade:** Permite dividir a interface em componentes menores e gerenciáveis.
- **Facilita a Manutenção:** Alterações em um componente refletem em todas as views que o utilizam.
- **Separação de Responsabilidades:** Cada Partial View pode se responsabilizar por uma parte específica da interface.

Quando Utilizar Partial Views?

- **Elementos Comuns:** Cabeçalhos, rodapés, barras de navegação que aparecem em várias páginas.
- **Componentes Reutilizáveis:** Formulários, listas, tabelas ou qualquer componente que seja utilizado em múltiplos lugares.

- **Organização:** Para manter o código limpo e organizado, especialmente em views complexas.
- **Atualizações Dinâmicas:** Quando é necessário atualizar partes da página via AJAX sem recarregar toda a view.

Criando e Utilizando Partial Views

Passo 1: Criar a Partial View

Por convenção, os arquivos de Partial Views começam com um underscore (_). Isso ajuda a identificar que o arquivo é uma Partial View.

Exemplo: Criando uma Partial View para o menu de navegação.

```
<!-- Views/Shared/_Menu.cshtml -->
<ul>
  <li><a href="/">Home</a></li>
  <li><a href="/Produtos">Produtos</a></li>
  <li><a href="/Sobre">Sobre</a></li>
  <li><a href="/Contato">Contato</a></li>
</ul>
```

Passo 2: Incluir a Partial View na View Principal

Para renderizar a Partial View dentro de uma view, você pode utilizar os métodos `Partial` ou `RenderPartialAsync`.

Usando `@Html.Partial`

```
<!-- Views/Shared/_Layout.cshtml -->
<body>
  <header>
    @Html.Partial("_Menu")
  </header>
  <div>
    @RenderBody()
  </div>
</body>
```

Usando @await Html.RenderPartialAsync

```
<body>
  <header>
    @await Html.RenderPartialAsync("_Menu")
  </header>
  <div>
    @RenderBody()
  </div>
</body>
```

Passando Dados para Partial Views

Usando Modelos Fortemente Tipados

Você pode passar um modelo para a Partial View da mesma forma que faz com uma view comum.

Exemplo: Passando um objeto `Usuario` para a Partial View.

```
// Modelo Usuario
public class Usuario
{
    public string Nome { get; set; }
    public bool IsAdmin { get; set; }
}
```

Partial View

```
<!-- Views/Shared/_UsuarioInfo.cshtml -->
@model MeuProjeto.Models.Usuario

<div>
  <p>Bem-vindo, @Model.Nome!</p>
  @if (Model.IsAdmin)
  {
    <p><a href="/Admin">Área Administrativa</a></p>
```

```
}  
</div>
```

View Principal

```
@model MeuProjeto.Models.ViewModels.HomeViewModel  
  
<!-- Passando o modelo para a Partial View -->  
@Html.Partial("_UsuarioInfo", Model.UsuarioAtual)
```

Usando ViewData ou ViewBag

Outra forma de passar dados é através do `ViewData` ou `ViewBag`.

Partial View

```
<!-- Views/Shared/_Aviso.cshtml -->  
@{  
    var mensagem = ViewBag.Mensagem as string;  
}  
  
@if (!string.IsNullOrEmpty(mensagem))  
{  
    <div class="alert alert-warning">  
        @mensagem  
    </div>  
}
```

View Principal

```
@{  
    ViewBag.Mensagem = "Este é um aviso importante!";  
}  
  
<!-- Renderizando a Partial View -->  
@Html.Partial("_Aviso")
```

Usando Partial Views com AJAX

Partial Views são especialmente úteis quando combinadas com AJAX para atualizar partes da página sem recarregar toda a interface.

Exemplo: Atualizar uma lista de comentários em um post de blog.

Passo 1: Criar a Partial View para a Lista de Comentários

```
<!-- Views/Shared/_Comentarios.cshtml -->
@model IEnumerable<MeuProjeto.Models.Comentario>

<div id="comentarios">
    @foreach (var comentario in Model)
    {
        <div class="comentario">
            <p><strong>@comentario.Autor</strong> disse:</p>
            <p>@comentario.Mensagem</p>
        </div>
    }
</div>
```

Passo 2: Criar a Action no Controller que Retorna a Partial View

```
public class PostsController : Controller
{
    private readonly MeuDbContext _context;

    public PostsController(MeuDbContext context)
    {
        _context = context;
    }

    public IActionResult ObterComentarios(int postId)
    {
        var comentarios = _context.Comentarios
            .Where(c => c.PostId == postId)
            .OrderByDescending(c => c.Data)
            .ToList();
    }
}
```



```

        return PartialView("_Comentarios", comentarios);
    }
}

```

Passo 3: Chamar a Partial View via AJAX

```

<!-- Views/Posts/Detalhes.cshtml -->
<div id="area-comentarios">
    <!-- Comentários serão carregados aqui -->
</div>

<script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.5.1/jquery.min
.js"></script>
<script>
    function carregarComentarios() {
        $.ajax({
            url: '@Url.Action("ObterComentarios", "Posts", new {
postId = Model.PostId })',
            type: 'GET',
            success: function (result) {
                $('#area-comentarios').html(result);
            }
        });
    }

    // Carrega os comentários quando a página é carregada
$(document).ready(function () {
    carregarComentarios();
});
</script>

```

Dicas e Boas Práticas

- **Organização de Arquivos:** Mantenha suas Partial Views na pasta **Views/Shared** ou em pastas específicas, facilitando a localização.

- **Nomenclatura:** Utilize nomes claros e, por convenção, inicie com underscore (`_NomeDaPartial.cshtml`).
- **Evite Lógica Complexa:** Mantenha a lógica de negócios no Controller ou em ViewModels, não na Partial View.
- **Reutilização:** Sempre que identificar código repetido em várias views, considere extrair esse código para uma Partial View.
- **Desempenho:** Use `RenderPartialAsync` para melhorar o desempenho em aplicações com alto tráfego.
- **View Components:** Para componentes mais complexos que requerem lógica adicional ou injeção de dependências, considere utilizar **View Components**.

Comparação entre Partial Views e View Components

- **Partial Views:** Ideais para conteúdo estático ou que depende apenas de dados passados diretamente. Não suportam injeção de dependências.
- **View Components:** Mais poderosos, permitem injeção de dependências e lógica mais complexa. Funcionam de forma semelhante a controllers miniaturizados.

Exemplo Completo: Criando um Layout com Partial Views

Arquitetura do Projeto

- Views
 - Shared
 - `_Layout.cshtml`
 - `_Menu.cshtml`
 - `_Footer.cshtml`
 - Home
 - `Index.cshtml`

_Layout.cshtml

```
<!DOCTYPE html>
<html>
<head>
    <title>@ViewData["Title"] - Meu Site</title>
    <link rel="stylesheet" href="~/css/site.css" />
</head>
<body>
    <header>
        @await Html.RenderPartialAsync("_Menu")
    </header>
    <main role="main" class="container">
        @RenderBody()
    </main>
    <footer>
        @await Html.RenderPartialAsync("_Footer")
    </footer>
    @RenderSection("Scripts", required: false)
</body>
</html>
```

_Menu.cshtml

```
<nav>
    <ul>
        <li><a href="/">Início</a></li>
        <li><a href="/Produtos">Produtos</a></li>
        <li><a href="/Servicos">Serviços</a></li>
        <li><a href="/Contato">Contato</a></li>
    </ul>
</nav>
```

_Footer.cshtml

```
<footer>
    <p>&copy; @DateTime.Now.Year - Meu Site</p>
```

```
</footer>
```

Index.cshtml

```
@{  
    ViewData["Title"] = "Página Inicial";  
}  
  
<h1>Bem-vindo ao Meu Site!</h1>  
<p>Esta é a página inicial.</p>
```

Considerações sobre Layouts e Section Rendering

- **Layouts:** Usados para definir a estrutura comum de páginas (header, footer, etc.).
- **RenderBody:** Marca o local onde o conteúdo específico de cada view será renderizado.
- **RenderSection:** Permite definir seções opcionais ou obrigatórias nas views, como scripts ou estilos adicionais.

Resumo

- **Partial Views** ajudam a modularizar a interface do usuário.
- Podem receber dados via modelo, `ViewData`, `ViewBag` ou parâmetros anônimos.
- São úteis para reutilização de código e atualização parcial de páginas com AJAX.
- Devem ser usadas para componentes simples; para lógica mais complexa, considere View Components.

Recursos Adicionais

- **Documentação Oficial:** Partial views in ASP.NET Core (<https://docs.microsoft.com/pt-br/aspnet/core/mvc/views/partial>)

- **Tutorial Vídeo:** Busque por vídeos que demonstrem o uso de Partial Views no ASP.NET Core MVC.
- **Comunidades e Fóruns:** Participe de comunidades como Stack Overflow para tirar dúvidas e compartilhar conhecimento.

Exercícios Práticos

1. **Crie uma Partial View para exibir mensagens de erro** que possa ser reutilizada em diferentes formulários.
2. **Implemente uma Partial View para um formulário de login** que aparece em várias páginas do seu site.
3. **Utilize AJAX para atualizar uma Partial View** que exibe notificações ou alertas em tempo real.

Espero que este guia tenha esclarecido o uso de Partial Views no ASP.NET Core MVC e como elas podem ser aplicadas para melhorar a organização e eficiência das suas aplicações web.

Introdução Minimal API

Minimal API é um estilo de construção de APIs que visa simplificar o processo de criação de aplicações web, especialmente em termos de estrutura e configuração. O termo ganhou destaque no contexto do .NET, mais especificamente no ASP.NET Core 6.0 e versões posteriores, onde a Microsoft introduziu o conceito de Minimal APIs.

Aqui estão os principais aspectos de uma Minimal API:

Simplicidade: Diferente de arquiteturas mais complexas (como MVC), uma Minimal API foca em um design mais enxuto e direto. O código de uma aplicação pode ser definido com poucas linhas, sem a necessidade de uma estrutura extensa.

Menos Configuração: Em vez de precisar configurar roteamento, controladores, e outras dependências, a Minimal API permite definir endpoints diretamente no arquivo principal da aplicação (geralmente o Program.cs).

Ideal para Microserviços: Minimal APIs são especialmente adequadas para aplicações de menor escala, como microserviços, onde você só precisa de endpoints HTTP simples para executar tarefas específicas.

Performance: Com uma estrutura mais leve, a Minimal API tende a ser mais rápida e eficiente do que APIs tradicionais, especialmente em cenários de alto desempenho.

Pré-requisitos

Para criar um projeto Minimal API no .NET 8, você precisará cumprir alguns pré-requisitos essenciais, tanto em termos de ferramentas quanto de configuração. Aqui estão os principais pré-requisitos:

- **.NET 8 SDK Instalado**
 - **Requisito Fundamental:** Ter o .NET 8 SDK instalado na sua máquina é essencial. O SDK contém todas as ferramentas necessárias para compilar e executar aplicações .NET, incluindo as Minimal APIs.
 - **Download:** Você pode baixar o .NET 8 SDK no site oficial da Microsoft [aqui](#).
- **IDE (Ambiente de Desenvolvimento Integrado)**

- **Visual Studio 2022 (versão 17.8 ou superior):** O Visual Studio é uma das IDEs mais populares para desenvolvimento em .NET. Certifique-se de que você tenha a versão mais recente, que suporte o .NET 8.
- Durante a instalação, selecione a carga de trabalho "ASP.NET e desenvolvimento web".
- **Visual Studio Code:** Outra opção leve, que também suporta o desenvolvimento com Minimal APIs, embora requeira a instalação de extensões, como o C# Extension.
- **JetBrains Rider:** Se você preferir uma IDE alternativa, o Rider é uma opção compatível com .NET e suporta o .NET 8.
- **Conhecimento Básico de C# e ASP.NET Core**
 - Familiaridade com C# e ASP.NET Core será útil, pois as Minimal APIs são desenvolvidas com base no framework ASP.NET Core e na linguagem C#.
- **Bibliotecas e Dependências**
 - Ao iniciar um projeto de Minimal API, o template incluirá as bibliotecas necessárias para a criação da API. Entretanto, para alguns recursos avançados, você pode precisar instalar pacotes adicionais do NuGet.
- **SDK de Desenvolvimento Web Instalado**
 - O SDK de desenvolvimento web do .NET, que faz parte do .NET 8 SDK, deve estar instalado para que você possa criar APIs e aplicações web.
- **Ferramenta de Linha de Comando (Opcional)**
 - Caso prefira trabalhar com a linha de comando, o CLI do .NET (dotnet) permite criar, compilar e executar projetos sem uma IDE. Certifique-se de que o caminho do SDK esteja corretamente configurado.
 - **Exemplo de criação via CLI:**

```
dotnet new webapi --minimal -n MinhaMinimalAPI
```

Passo a passo para criar projeto de Minimal API no Visual Studio

1. Abra o Visual Studio 2022 e selecione a opção ASP.NET Core Web API.
2. Forneça um nome e escolha um local para o seu projeto.
3. Selecione o framework desejado (certifique-se de que está usando .NET 8).
4. Marque as opções "Configurar para HTTPS" e "Habilitar suporte OpenAPI".
5. Deixe a caixa de seleção "Usar controladores" desmarcada.
6. Clique no botão "Criar".

Exploração do arquivo Program.cs

Vamos agora explorar o arquivo `Program.cs`, onde os diversos componentes da sua Minimal API se integram. Esse arquivo é responsável por orquestrar a configuração dos serviços e a definição dos endpoints da API.

```
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddControllers();
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();
builder.Services.AddSingleton<IBookService, BookService>();

var app = builder.Build();

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}

app.UseHttpsRedirection();
```



```

// Hello World
app.MapGet("/", () => "Hello, World!");

// Get all books
app.MapGet("/books", (IBookService bookService) =>
    TypedResults.Ok(bookService.GetBooks()))
    .WithName("GetBooks");

// Get a specific book by ID
app.MapGet("/books/{id}", Results<Ok<Book>, NotFound>(IBookService
bookService, int id) =>
{
    var book = bookService.GetBook(id);
    return book is not null ? TypedResults.Ok(book) :
TypedResults.NotFound();
}).WithName("GetBookById");

// Add a new book with validation
app.MapPost("/books", (IBookService bookService, Book newBook) =>
{
    if (string.IsNullOrEmpty(newBook.Title))
    {
        return TypedResults.BadRequest("Title cannot be empty");
    }

    bookService.AddBook(newBook);
    return TypedResults.Created($" /books/{newBook.Id}", newBook);
}).WithName("AddBook");

// Update an existing book with validation
app.MapPut("/books/{id}", (IBookService bookService, int id, Book
updatedBook) =>
{
    if (string.IsNullOrEmpty(updatedBook.Title))
    {
        return TypedResults.BadRequest("Title cannot be empty");
    }
}

```

```

        var existingBook = bookService.GetBook(id);
        if (existingBook is null)
        {
            return TypedResults.NotFound();
        }

        bookService.UpdateBook(id, updatedBook);
        return TypedResults.Ok();
    }).WithName("UpdateBook");

// Delete a book by ID
app.MapDelete("/books/{id}", (IBookService bookService, int id) =>
{
    var existingBook = bookService.GetBook(id);
    if (existingBook is null)
    {
        return TypedResults.NotFound();
    }

    bookService.DeleteBook(id);
    return TypedResults.NoContent();
}).WithName("DeleteBook");

app.Run();

```

Exemplo de BookService e IBookService

O código a seguir demonstra uma classe chamada `BookService`, responsável por gerenciar uma coleção de livros, enquanto a interface `IBookService` define o contrato para interação com esse serviço. Este é um exemplo simples de um serviço que executa operações CRUD em uma coleção de livros.

```

using System.Collections.Generic;
using System.Linq;

namespace MinimalApi
{

```

```

public interface IBookService
{
    IReadOnlyList<Book> GetBooks();
    Book? GetBook(int id);
    void AddBook(Book book);
    void UpdateBook(int id, Book updatedBook);
    void DeleteBook(int id);
}

public class BookService : IBookService
{
    private readonly Dictionary<int, Book> _books;
    private int _nextId;

    public BookService()
    {
        _books = new Dictionary<int, Book>();
        _nextId = 1;

        // Initial books
        AddBook(new Book { Id = _nextId++, Title = "Clean
Code: A Handbook of Agile Software Craftsmanship", Author =
"Robert C. Martin" });
        AddBook(new Book { Id = _nextId++, Title = "Design
Patterns: Elements of Reusable Object-Oriented Software", Author =
"Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides" });
        AddBook(new Book { Id = _nextId++, Title =
"Refactoring: Improving the Design of Existing Code", Author =
"Martin Fowler" });
        AddBook(new Book { Id = _nextId++, Title = "Code
Complete: A Practical Handbook of Software Construction", Author =
"Steve McConnell" });
    }

    public IReadOnlyList<Book> GetBooks()
    {
        return _books.Values.ToList();
    }
}

```

```

public Book? GetBook(int id)
{
    _books.TryGetValue(id, out var book);
    return book;
}

public void AddBook(Book book)
{
    if (string.IsNullOrEmpty(book.Title) ||
string.IsNullOrEmpty(book.Author))
        throw new ArgumentException("Book title and author
cannot be empty.");

    book.Id = _nextId++;
    _books[book.Id] = book;
}

public void UpdateBook(int id, Book updatedBook)
{
    if (string.IsNullOrEmpty(updatedBook.Title) ||
string.IsNullOrEmpty(updatedBook.Author))
        throw new ArgumentException("Book title and author
cannot be empty.");

    if (_books.ContainsKey(id))
    {
        updatedBook.Id = id; // Preserve the existing ID
        _books[id] = updatedBook;
    }
    else
    {
        throw new KeyNotFoundException("Book not found.");
    }
}

public void DeleteBook(int id)
{

```

```

        if (!_books.Remove(id))
        {
            throw new KeyNotFoundException("Book not found.");
        }
    }

    public class Book
    {
        public int Id { get; set; }
        public string Title { get; set; } = string.Empty;
        public string Author { get; set; } = string.Empty;
    }
}

```

Tópicos Complementares

1. API Rate Limiting

- **O que é:** Limita o número de solicitações que um cliente pode fazer a uma API em um período de tempo específico.
- **Como implementar:** Pode ser feito usando middleware ou bibliotecas como `AspNetCoreRateLimit`. Configura-se o limite de requisições no `Startup.cs` ou `Program.cs`.

```

builder.Services.AddMemoryCache();
builder.Services.AddInMemoryRateLimiting();
builder.Services.AddSingleton<IRateLimitConfiguration,
RateLimitConfiguration>();

```

2. Error Handling

- **O que é:** Tratamento de exceções e envio de mensagens de erro apropriadas para os clientes.

- **Como implementar:** Pode ser feito através de middleware de tratamento de erros.

```
app.UseExceptionHandler("/error");
```

```
app.Map("/error", (HttpContext httpContext) =>
{
    var exception =
httpContext.Features.Get<IExceptionHandlerFeature>();
    return TypedResults.Problem(title: "An error occurred",
detail: exception?.Error.Message);
});
```

3. Caching

- **O que é:** Armazenamento de dados temporários para reduzir a carga em recursos e melhorar o desempenho.
- **Como implementar:** Usando middleware de cache ou serviços de cache como Redis ou Memcached.

```
builder.Services.AddResponseCaching();
app.UseResponseCaching();
```

```
app.MapGet("/books", async (IBookService bookService, HttpContext
httpContext) =>
{
    var cacheKey = "allBooks";
    var cache =
httpContext.RequestServices.GetRequiredService<IMemoryCache>();
    if (!cache.TryGetValue(cacheKey, out List<Book> books))
    {
        books = bookService.GetBooks();
        var cacheEntryOptions = new
MemoryCacheEntryOptions().SetSlidingExpiration(TimeSpan.FromMinute
s(1));
```

```
        cache.Set(cacheKey, books, cacheEntryOptions);
    }
    return TypedResults.Ok(books);
});
```

4. Request and Response Compression

- **O que é:** Compressão dos dados de solicitação e resposta para reduzir o uso de largura de banda e melhorar o desempenho.
- **Como implementar:** Usando middleware de compressão.

```
builder.Services.AddResponseCompression(options =>
{
    options.EnableForHttps = true;
    options.MimeTypes =
ResponseCompressionDefaults.MimeTypes.Concat(new[] {
"application/json" });
});
app.UseResponseCompression();
```

5. Authentication and Authorization

- **O que é:** Controle de acesso à API, garantindo que apenas usuários autenticados e autorizados possam acessar certos recursos.
- **Como implementar:** Usando middleware de autenticação e autorização, e configurando esquemas de autenticação (e.g., JWT).

```
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(options =>
    {
        options.TokenValidationParameters = new
TokenValidationParameters
```

```

        {
            ValidateIssuer = true,
            ValidateAudience = true,
            ValidateLifetime = true,
            ValidateIssuerSigningKey = true,
            ValidIssuer = "yourIssuer",
            ValidAudience = "yourAudience",
            IssuerSigningKey = new
SymmetricSecurityKey(Encoding.UTF8.GetBytes("yourSecretKey"))
        };
    });
builder.Services.AddAuthorization();
app.UseAuthentication();
app.UseAuthorization();

```

Principais tópicos abordados

1. Configuração do WebApplication:

- Criação e configuração do aplicativo usando `WebApplication.CreateBuilder` e `builder.Build()`.
- Adição de serviços ao contêiner com `builder.Services.Add...`

2. Mapeamento de Endpoints:

- Definição de rotas HTTP com `app.MapGet`, `app.MapPost`, `app.MapPut` e `app.MapDelete`.
- Uso de métodos de resultados tipados como `TypedResults.Ok()`, `TypedResults.Created()`, `TypedResults.NoContent()`, e `TypedResults.NotFound()`.

3. Injeção de Dependências:

- Registro de serviços no contêiner usando `builder.Services.AddSingleton` ou outras variações.

4. Ambiente de Desenvolvimento:

- Configuração condicional para ambientes de desenvolvimento (`app.Environment.IsDevelopment()`).
- Configuração de Swagger para documentação da API.

5. Configuração de Middleware:

- Configuração do pipeline de middleware com métodos como `app.UseHttpsRedirection()`.

6. Definição de Modelos:

- Criação de classes de modelo (`Book`) e interfaces (`IBookService`).

7. Manipulação de Dados:

- Implementação de serviços para manipulação de dados com métodos para adicionar, atualizar, e excluir dados.

8. Respostas HTTP e Erros:

- Manipulação de erros e validações usando retornos apropriados e tratamento de exceções.

9. Uso de `ReadOnlyList`:

- Implementação de coleções imutáveis para garantir a integridade dos dados.

10. Uso de `Dictionary` para Armazenamento de Dados:

- Utilização de `Dictionary` para armazenamento e acesso rápido a dados com base em IDs.

11. API Rate Limiting:

- Limitação do número de solicitações de um cliente usando middleware ou bibliotecas como `AspNetCoreRateLimit`.

12. Error Handling:

- Tratamento de exceções e envio de mensagens de erro apropriadas através de middleware.

13. Caching:

- Armazenamento temporário de dados para melhorar o desempenho usando middleware de cache ou serviços como Redis ou Memcached.

14. Request and Response Compression:

- Compressão de dados de solicitação e resposta para reduzir o uso de largura de banda e melhorar o desempenho.

15. Authentication and Authorization:

- Controle de acesso à API usando middleware de autenticação e autorização, e configuração de esquemas de autenticação (e.g., JWT).

See also

Outros Links

<https://learn.microsoft.com/pt-br/aspnet/core/performance/rate-limit?view=aspnetcore-8.0>

<https://medium.com/@alibenchabene/minimal-api-in-net-8-a-simplified-approach-to-build-web-apis-6b772059f17c>

OpenAPI e Swagger com Minimal API

1. Introdução ao OpenAPI e Swagger:

OpenAPI: Um padrão de especificação para definir APIs RESTful, permitindo a descrição de endpoints, parâmetros, respostas e mais.

Swagger: Conjunto de ferramentas que implementa a especificação OpenAPI, oferecendo funcionalidades para documentação e testes de APIs.

2. Configuração do Swagger em uma Minimal API:

Adicionar pacotes necessários: Inclua os pacotes NuGet necessários para Swagger no seu projeto.

```
dotnet add package Swashbuckle.AspNetCore
```

Configurar Swagger no Program.cs: Adicione a configuração do Swagger no pipeline de serviços e no middleware.

```
var builder = WebApplication.CreateBuilder(args);

// Adiciona serviços ao contêiner.
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();

var app = builder.Build();

// Configura o pipeline de requisições HTTP.
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}
```

```
app.UseHttpsRedirection();  
// Outros middlewares e endpoints  
  
app.Run();
```

Personalização da Documentação Swagger:

Configurar opções do Swagger: Personalize a documentação gerada adicionando informações sobre o título, descrição e versão da API.

```
builder.Services.AddSwaggerGen(options =>  
{  
    options.SwaggerDoc("v1", new OpenApiInfo  
    {  
        Title = "Minha API",  
        Version = "v1",  
        Description = "Uma descrição detalhada da minha API",  
        Contact = new OpenApiContact  
        {  
            Name = "Nome do projeto",  
            Email = "contato@example.com"  
        }  
    });  
});
```

Configurar exemplos e modelos: forneça exemplos de requisições e respostas para melhorar a documentação.

```
options.ExampleFilters();
```

4. Uso do Swagger UI:

Acesso à UI: Quando o Swagger UI é configurado, ele está disponível em localhost:port/swagger por padrão. Você pode visualizar e testar a API diretamente na interface web.

Testar Endpoints: Utilize a interface interativa para enviar solicitações aos endpoints da

API e ver as respostas diretamente no navegador.

5. Gerar Documentação OpenAPI:

Gerar e Exportar Especificação: Swagger gera uma especificação OpenAPI que pode ser exportada em formatos como JSON ou YAML. A URL para a especificação JSON geralmente é localhost:port/swagger/v1/swagger.json.

Integrar com Ferramentas: Use ferramentas como Swagger Editor ou SwaggerHub para editar e compartilhar a especificação OpenAPI.

6. Personalização da Documentação Swagger:

6.1 Remover Endpoint da Definição Swagger:

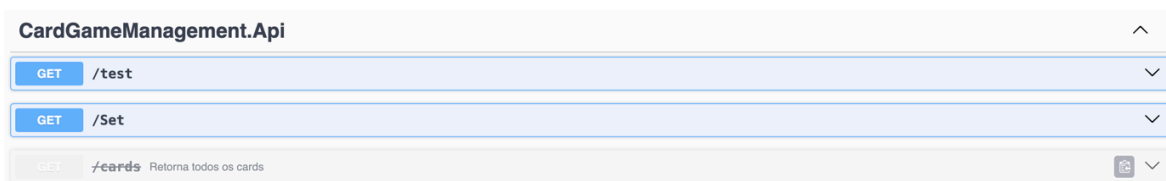
Para excluir um endpoint da definição do Swagger, é necessário utilizar a extensão `ExcludeFromDescription` no método do endpoint:

```
app.MapGet("/remover-da-doc", () => "Este endpoint não aparecerá na documentação.").ExcludeFromDescription();
```

6.2 Marcar Endpoint como Obsoleto:

Para marcar um endpoint como obsoleto no Swagger, utilize o atributo `Deprecated` nas opções da rota:

```
pp.MapGet("/rota-obsoleta", () => "Esta rota é obsoleta.").WithOpenApi(operation => new(operation)
{
    Deprecated = true
})
```



swagger_obsolete.png

6.3 Adicionar Descrição das Rotas e Retornos:

Para adicionar descrições detalhadas de rotas e seus retornos no Swagger, você pode usar o método `WithMetadata()` e fornecer informações adicionais:

```
app.MapGet("/descricao", () => "Exemplo de descrição.")
    .WithName("DescriçãoExemplo")
    .WithMetadata(new SwaggerOperationAttribute(summary: "Resumo
da operação", description: "Descrição detalhada da rota e seu
retorno."));
```

6.4 Colocar Informações Básicas da API no Swagger UI (Contato, Website):

Para incluir informações básicas da API, como nome de contato e website, configure o Swagger no `Program.cs` conforme o exemplo:

```
builder.Services.AddSwaggerGen(options =>
{
    options.SwaggerDoc("v1", new OpenApiInfo
    {
        Title = "Minha API",
        Version = "v1",
        Description = "Uma descrição detalhada da minha API",
        Contact = new OpenApiContact
        {
            Name = "Nome do Projeto",
            Email = "contato@example.com",
            Url = new Uri("https://www.example.com")
        }
    });
});
```

6.5 Adicionar Tags às Rotas:

As tags podem ser adicionadas para agrupar endpoints e melhorar a organização da documentação Swagger. Use a seguinte abordagem para adicionar tags às rotas:

```
app.MapGet("/minha-rota", () => "Conteúdo da rota.")
    .WithTags("Grupo de Rotas 1");
```

7. Benefícios da Integração com OpenAPI e Swagger:

Documentação Automática: Geração automática de documentação baseada no código.

Testes Interativos: Capacidade de testar a API diretamente na UI do Swagger. Facilidade

de Integração: Integração com outras ferramentas e serviços baseados na especificação OpenAPI.

See also

Outros Links

<https://learn.microsoft.com/en-us/aspnet/core/fundamentals/openapi/aspnetcore-openapi>

Idempotência no .NET

O que é Idempotência?

Idempotência refere-se à propriedade de que o resultado de uma operação não muda quando aplicada uma ou mais vezes. Em termos de desenvolvimento de software, um método idempotente é aquele que sempre retorna o mesmo resultado, independentemente de quantas vezes seja chamado.

Exemplo:

- A requisição `GET /produto/3` sempre retornará o produto com ID 3, sendo, portanto, idempotente.

Por que Idempotência é Importante?

A importância da idempotência se destaca em **arquiteturas distribuídas**. Nessas arquiteturas, a mesma requisição pode ser enviada várias vezes devido a falhas de rede, perdas de requisição ou problemas de coordenação entre serviços. Operações idempotentes garantem que o estado do sistema permaneça consistente, mesmo quando uma requisição é repetida.

Exemplo de service:

- Imagine um serviço de pagamento. Se o cliente enviar várias confirmações de pagamento por falha de rede, um método de pagamento idempotente garantirá que o pagamento seja processado apenas uma vez, mesmo que a requisição seja enviada várias vezes.

Implementando Idempotência no .NET

Passos:

1. **Identificar Requisições Unicamente:** Cada requisição precisa ser única para saber quais já foram processadas.

- Utilize um parâmetro exclusivo no cabeçalho da requisição (chave de idempotência).
2. **Salvar o Parâmetro:** Ao processar a requisição, salve esse parâmetro para identificar se ela já foi tratada.
 3. **Verificação de Idempotência:** Verifique o parâmetro em cada nova requisição. Dependendo da lógica da aplicação, retorne a resposta já processada ou uma nova.

Detalhes:

- **Cuidado com a Criação de Chaves:** Chaves duplicadas podem causar problemas, especialmente em transações críticas, como pagamentos.
- **Cache Distribuído:** Em APIs intensivas, o uso de cache distribuído para armazenar a chave pode ser a melhor escolha.
- **Expiração do Cache:** O tempo de vida do cache deve ser equilibrado, de forma a não criar sobrecarga nem expirar antes do necessário.

Usando a Biblioteca IdempotentAPI

IdempotentAPI (<https://www.nuget.org/packages/IdempotentAPI/>) é uma biblioteca open-source que facilita a implementação de APIs idempotentes no ASP.NET Core, lidando com operações HTTP como POST e PATCH.

Implementação com IdempotentAPI:

1. Instalar a Biblioteca:

```
PM> Install-Package IdempotentAPI -Version 2.1.0
```

2. Registrar os Serviços em Program.cs:

```
builder.Services.AddIdempotentAPI();
```

3. Adicionar o Cache Distribuído:

```
builder.Services.AddIdempotentMinimalAPI(new  
IdempotencyOptions());
```

```
builder.Services.AddDistributedMemoryCache();
builder.Services.AddIdempotentAPIUsingDistributedCache();
```

4. Marcar Objetos de Resposta como [Serializable]:

```
[Serializable]
public record CardAddOrUpdateModel()
{
    public string? Name { get; init; }
    public string? Description { get; init; }
    public string? ImageUrl { get; init; }
};
```

5. Marcar o Controller como Idempotente (opcional):

```
[Idempotent(ExpireHours = 48)]
[ApiController]
[Route("[controller]")]
public class WeatherForecastController : ControllerBase { }
```

6. Marcar rotas em minimal API:

```
cardGroup.MapPost("/", (CardAddOrUpdateModel model) =>
{
    cards.Add(model.MapToCard());
    return Results.Created("/cards", cards);
})
    .Accepts<Card>("application/json")
    .AddEndpointFilter<IdempotentAPIEndpointFilter>()
    .WithName("Add card")
    .WithOpenApi();
```

Testando a API:

1. Faça uma requisição sem a chave `IdempotencyKey` e você verá uma exceção.
2. Adicione a chave no cabeçalho, por exemplo `1234`, e execute a requisição.

3. Execute novamente com a mesma chave e veja que o resultado é o mesmo.

Conclusão

Idempotência é essencial em arquiteturas distribuídas para manter a consistência do sistema. A biblioteca `IdempotentAPI` facilita a implementação desse padrão no ASP.NET Core, especialmente para métodos POST e PATCH, garantindo que requisições repetidas não causem resultados inesperados.

Referências:

- GitHub do IdempotentAPI (<https://github.com/leventozturk/IdempotentAPI>)
- NuGet Package (<https://www.nuget.org/packages/IdempotentAPI/>)

Implementando JWT em uma Minimal API com ASP.NET Core

Este guia passo a passo irá ajudá-lo a implementar a autenticação JWT em uma Minimal API. Você aprenderá a adicionar uma chave secreta, configurar a autenticação e criar endpoints protegidos.

Passo 1: Criar um novo projeto de Minimal API

Crie um novo projeto usando o comando:

```
dotnet new web -o MinhaApiJwtStart typing here...
```

Passo 2: Adicionar o pacote de autenticação JWT

Navegue até o diretório do projeto e adicione o pacote necessário:

```
cd MinhaApiJwt
dotnet add package Microsoft.AspNetCore.Authentication.JwtBearer
```

Passo 3: Configurar a chave secreta no appsettings.json

```
{
  "Jwt": {
    "Key": "sua-chave-secreta-super-segura",
    "Issuer": "sua-issuer",
    "Audience": "sua-audience"
  }
}
```

Passo 4: Configurar os serviços de autenticação no Program.cs

Edite o arquivo Program.cs para configurar a autenticação JWT:

```

using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.IdentityModel.Tokens;
using System.Text;

var builder = WebApplication.CreateBuilder(args);

// Adiciona a autenticação JWT
builder.Services.AddAuthentication(options =>
{
    options.DefaultAuthenticateScheme =
JwtBearerDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme =
JwtBearerDefaults.AuthenticationScheme;
})
.AddJwtBearer(options =>
{
    var key =
Encoding.UTF8.GetBytes(builder.Configuration["Jwt:Key"]);
    options.SaveToken = true;
    options.TokenValidationParameters = new
TokenValidationParameters
    {
        ValidateIssuer = true,
        ValidateAudience = true,
        ValidateLifetime = true,
        ValidateIssuerSigningKey = true,
        ValidIssuer = builder.Configuration["Jwt:Issuer"],
        ValidAudience = builder.Configuration["Jwt:Audience"],
        IssuerSigningKey = new SymmetricSecurityKey(key)
    };
});

var app = builder.Build();

// Usa autenticação e autorização

```

```
app.UseAuthentication();
app.UseAuthorization();
```

Explicação das opções de configuração:

- **Key (Chave):** Esta é a chave secreta usada para assinar o token JWT. Deve ser uma string complexa e segura para evitar que seja facilmente descoberta.
- **Issuer (Emissor):** Representa o emissor do token. Geralmente é o nome ou URL da sua aplicação ou serviço.
- **Audience (Audiência):** Indica quem pode usar ou consumir o token. Pode ser o nome da sua aplicação, serviço ou clientes específicos.

Essas configurações são usadas tanto na geração quanto na validação dos tokens JWT.

Passo 5: Criar um endpoint para gerar o token JWT

Ainda no Program.cs, adicione um endpoint POST que gera o token:

```
using System.IdentityModel.Tokens.Jwt;
using System.Security.Claims;

app.MapPost("/gerar-token", (IConfiguration config) =>
{
    // Normalmente, você validaria as credenciais do usuário aqui
    // Para simplificar, vamos assumir que o usuário é válido

    var key = new
    SymmetricSecurityKey(Encoding.UTF8.GetBytes(config["Jwt:Key"]));
    var creds = new SigningCredentials(key,
    SecurityAlgorithms.HmacSha256);

    var claims = new[]
    {
        new Claim(JwtRegisteredClaimNames.Sub, "usuarioTeste"),
        new Claim(JwtRegisteredClaimNames.Jti,
        Guid.NewGuid().ToString())
    }
```

```
};

var token = new JwtSecurityToken(
    issuer: config["Jwt:Issuer"],
    audience: config["Jwt:Audience"],
    claims: claims,
    expires: DateTime.UtcNow.AddHours(1),
    signingCredentials: creds);

var tokenString = new
JwtSecurityTokenHandler().WriteToken(token);

return Results.Ok(new { token = tokenString });
});
```

Passo 6: Proteger endpoints com [Authorize] ou RequireAuthorization()

Adicione um endpoint protegido:

```
using Microsoft.AspNetCore.Authorization;

app.MapGet("/dados-seguros", [Authorize] () =>
{
    return "Este é um dado protegido por JWT!";
});
```

```
using Microsoft.AspNetCore.Authorization;

app.MapGet("/dados-seguros", () =>
{
    return "Este é um dado protegido por JWT!";
}).RequireAuthorization();
```

Passo 7: Executar e testar a API

1. Obtenha o token JWT: Faça uma requisição POST para `https://localhost:5001/gerar-token` usando o Postman, Insomnia ou curl. Você receberá um token JWT.
2. Acesse o endpoint protegido: Use o token recebido para fazer uma requisição GET para `https://localhost:5001/dados-seguros`, adicionando o header Authorization: Bearer . Exemplo com curl:

```
curl -H "Authorization: Bearer {seu_token}"  
https://localhost:5001/dados-seguros
```

Passo 8: Integrar o JWT ao Swagger

Para testar endpoints protegidos via Swagger, é necessário configurar o Swagger para suportar autenticação JWT.

8.1 Adicionar pacotes necessários

Adicione os pacotes do Swagger ao projeto:

```
dotnet add package Swashbuckle.AspNetCore
```

8.2 Configurar o Swagger no Program.cs

Adicione as seguintes linhas no Program.cs:

Após adicionar os serviços de autenticação:

```
builder.Services.AddEndpointsApiExplorer();  
builder.Services.AddSwaggerGen(options =>  
{  
    // Configura o Swagger para usar o JWT  
    options.AddSecurityDefinition("Bearer", new  
Microsoft.OpenApi.Models.OpenApiSecurityScheme  
    {  
        Name = "Authorization",  
        Type = Microsoft.OpenApi.Models.SecuritySchemeType.Http,  
        Scheme = "Bearer",  
        BearerFormat = "JWT",  
        In = Microsoft.OpenApi.Models.ParameterLocation.Header,
```



```

        Description = "Insira o token JWT no formato Bearer {seu token}"
    });
    options.AddSecurityRequirement(new
Microsoft.OpenApi.Models.OpenApiSecurityRequirement
    {
        {
            new Microsoft.OpenApi.Models.OpenApiSecurityScheme
            {
                Reference = new
Microsoft.OpenApi.Models.OpenApiReference
            {
                Type =
Microsoft.OpenApi.Models.ReferenceType.SecurityScheme,
                Id = "Bearer"
            },
            new string[] {}
        }
    });
});

if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}

```

1. Acesse o Swagger UI: Navegue até <https://localhost:5001/swagger/index.html>.
2. Obtenha o token JWT: Use o Swagger para fazer uma requisição POST em /gerar-token e copie o token recebido.
3. Autentique-se no Swagger: Clique no botão Authorize no Swagger UI, insira o token no campo Authorization no formato Bearer e clique em Authorize.
4. Acesse o endpoint protegido: Agora, você pode testar o endpoint /dados-seguros diretamente pelo Swagger UI.

Conclusão

Você configurou com sucesso a autenticação JWT em sua Minimal API. Agora, você tem um endpoint que gera tokens e um endpoint protegido que requer um token válido para acesso.