



Manual da UFCD: 10791 - Desenvolvimento de aplicações web em JAVA

Luís Cunha

27 de novembro de 2024

UFCD: 10791 - Desenvolvimento de aplicações web em JAVA

Versão v0.1 (documento em construção)

Índice

Índice	3
Condições de utilização do manual	7
Objetivos	8
Capítulo 1 – Introdução à Linguagem Java	9
Ambiente de Desenvolvimento.....	9
Estrutura Básica de um Programa Java	9
Variáveis e Tipos de Dados	10
Tipos Primitivos	10
Tipos de Referência	10
Conversão de Dados	11
Alargamento:	11
Redução:	11
Conversão de valores numéricos para texto (String):	11
Conversão de String (texto) para valores numéricos:	11
Operadores.....	12
Strings (Texto) e Métodos com Strings.....	14
Strings (texto) em Java	14
Atenção à comparação de Strings: não usamos “==” mas sim “equals”:	14
Alguns métodos (apenas exemplos) que se podem usar com Strings:	14
Métodos em Java	15
Definição de um Método:	15
Chamada de um Método:	15
Métodos com void:	16
Métodos com múltiplos parâmetros:	16
Exemplo - Jogo “Adivinha um Número”:	17
Estruturas de Controlo e Iteração	20
Estrutura de seleção if-else:	20

Estrutura de seleção switch-case:	21
Estrutura de repetição while:	23
Estrutura de repetição do-while:	24
Estrutura de repetição for:	24
Estrutura de repetição for-each:	25
Arrays e Listas	26
Arrays:	26
Listas (ArrayList):	26
Exemplo A - Gestão de Contactos:	28
Exemplo B- Conversor de Temperatura:	30
“Exceções”	32
Manipulação de Ficheiros	34
Leitura de Ficheiros:	34
Escrita de Ficheiros:	35
Sobre o uso de Maiúsculas e Minúsculas em Java:	38
Capítulo 2 – Programação Orientada por Objetos com Java	41
Introdução à Programação Orientada a Objetos (OOP)	41
Classes e Objetos	42
Criar uma Classe	43
Instanciar Objetos	44
Noção de construtor	46
Encapsulamento	51
Getters e Setters	52
Herança	56
Polimorfismo	58
Composição	60
Interfaces	62
Classes Abstratas	67
O uso de “static” em Java	73
O uso de “this” em Java	75
O uso de “super” em Java	77
Gráficos: uso das bibliotecas Swing e AWT	79

Exceções e Tratamento de Erros.....	82
Coleções em Java (<i>Java Collections Framework</i>).....	84
Multithreading em Java	87
Método 1: Estender a classe Thread	87
Método 2: Implementar a interface Runnable	88
Sincronização de Threads	88
Bloco synchronized	89
Métodos synchronized	90
Padrões de Projeto (tópico avançado / opcional).....	92
Singleton:	92
Factory:	93
Observer	101
Capítulo 3 – Introdução à Programação Web em Java.....	104
Vaadin Flow.....	104
O que é possível fazer com o Vaadin Flow	104
Passos para Criar uma UI com Vaadin Flow:	104
Como configurar um projeto para usar Vaadin Flow (baseado em Spring Boot):	106
1. Criação do Projeto	106
2. Estrutura do Projeto	106
3. Adicionar Dependências.....	107
4. Definir a Classe Principal	109
5. Criar Vistas com Vaadin.....	109
6. Executar a Aplicação.....	109
7. Implementar Funcionalidades Adicionais.....	109
Layouts	110
Eventos de UI	110
Navegação e Layout Principal	112
Implementação de Manipuladores de Eventos.....	113
Eventos Personalizados	114
Filtragem e Debouncing de Eventos	115
Passos para Implementar a Vista Principal:	116
Definição de Rotas com <i>@Route</i>	119
Parâmetros de Rota	120

Localização dos Recursos em Projetos Spring Boot	122
Definição de Rotas com <i>@Route</i>	124
Parâmetros de Rota	124
Definição de Parâmetros de Rota	126
Parâmetros Opcionais	126
Atualização de Parâmetros de URL sem Navegação	128
Eventos do Ciclo de Vida da Navegação	131
Redirecionamento e Encaminhamento durante a Navegação	132
Redirecionamento para Vistas de Erro	134
Obtenção de Todas as Rotas Registadas	135
Utilização da Anotação <i>@PageTitle</i>	136
Glossário de termos	138
Bibliografia	141

Condições de utilização do manual

Em construção

Objetivos

Nota:

Deve haver coerência interna entre objetivos, conteúdos formativos, materiais e respetivos suportes (áudio, vídeo e multimédia) desenvolvidos pelo formador/a.

Em construção

Capítulo 1 – Introdução à Linguagem Java

A linguagem Java é uma linguagem de programação de alto nível, criada pela Sun Microsystems em 1995, e atualmente mantida pela Oracle Corporation. É uma linguagem versátil e amplamente utilizada no desenvolvimento de aplicações, desde simples programas para computadores pessoais até grandes sistemas empresariais. A linguagem Java é orientada a objetos, o que facilita a organização e a reutilização de código.

Ambiente de Desenvolvimento

Nas aulas é usado o VSCode. Consultar o documento no Moodle sobre como instalar este editor de código.

Estrutura Básica de um Programa Java

Um programa Java é composto por classes. Cada classe tem um nome e pode conter variáveis e métodos. Os métodos são blocos de código que realizam uma tarefa específica. Um programa Java deve ter pelo menos uma classe com um método chamado "main", que é o ponto de entrada do programa. O método "main" é onde o programa começa a ser executado.

Aqui está um exemplo de uma classe Java simples chamada "OlaMundo":

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

Variáveis e Tipos de Dados

Variáveis são usadas para armazenar valores temporariamente durante a execução do programa. Em Java, cada variável tem um tipo de dado específico, que determina o tipo de valor que pode ser armazenado nela. Os tipos de dados em Java são divididos em dois grupos: tipos primitivos e tipos de referência.

Tipos Primitivos

Os tipos primitivos são os tipos de dados básicos do Java. Incluem:

byte: Um número inteiro de 8 bits (-128 a 127)

short: Um número inteiro de 16 bits (-32.768 a 32.767)

int: Um número inteiro de 32 bits (-2.147.483.648 a 2.147.483.647)

long: Um número inteiro de 64 bits (-9.223.372.036.854.775.808 a 9.223.372.036.854.775.807)

float: Um número decimal de precisão simples de 32 bits

double: Um número decimal de precisão dupla de 64 bits

char: Um único caractere Unicode de 16 bits

boolean: Um valor verdadeiro (true) ou falso (false)

Grupo	Tipo	Tamanho	Intervalo de Valores	Valor Default
Inteiros	int	4 bytes	-2.147.483.648 até 2.147.483.647	0
	short	2 bytes	-32.768 até 32.767	0
	long	8 bytes	-9.223.372.036.854.775.808L até 9.223.372.036.854.775.807L	0L
	byte	1 byte	-128 até 127	0
Ponto Flutuante	float	4 bytes	+ - 3,40282347E+38F (6-7 dígitos significativos)	0.0f
	double	8 bytes	+ - 1,79769313486231570E+308 (15 dígitos significativos)	0.0d
	char	2 bytes	representa um Unicode	'\u0000';
	boolean	1 bit	true ou false	false

Tipos de Referência

Os tipos de referência são usados para armazenar objetos em Java. Estes incluem objetos criados a partir de classes, interfaces e arrays. **As variáveis de referência armazenam a localização na memória** onde o objeto / array está armazenado, e não o próprio objeto (daí o nome “referência”: é a referência para a posição de memória onde está o objeto).

Conversão de Dados

O processo de alterar um valor de um tipo de dados para outro é conhecido como **conversão de tipo de dados**. A conversão de tipo de dados é de dois tipos:

Alargamento:

O tipo de dados de menor tamanho é convertido num tipo de dados de maior tamanho sem perda de informação.

```
// Alargamento (byte < short < int < long < float < double)
```

```
// Por exemplo: conversão automática de tipo int -> long:
```

```
int i = 10;
```

```
long l = i;
```

Como **long > int**, não há perda de informação, e a conversão é automática (**não é preciso pedir explicitamente** ao compilador para converter).

Redução:

O tipo de dados de maior tamanho é convertido num tipo de dados de menor tamanho com perda de informação.

```
// Redução:
```

```
double d = 10.02;
```

```
long l = (long) d; // "casting" (pedir para converter de forma explícita)
```

Neste caso, **temos de pedir explicitamente a conversão, usando "casting"**, porque **long < double**.

Conversão de valores numéricos para texto (String):

```
String str = String.valueOf(value);
```

Conversão de String (texto) para valores numéricos:

```
int i = Integer.parseInt(str); double d = Double.parseDouble(str);
```

Operadores

Operadores são símbolos que realizam operações em variáveis e valores. Em Java, existem vários tipos de operadores, como:

Operadores aritméticos: +, -, *, /, % (soma, subtração, multiplicação, divisão e módulo)

Operadores de comparação: ==, !=, <, >, <=, >= (igual, diferente, menor, maior, menor ou igual, maior ou igual)

Operadores lógicos: &&, ||, ! (E, OU, NÃO)

Operadores de atribuição: =, +=, -=, *=, /=, %= (atribuição, adição, subtração, multiplicação, divisão e módulo)

Operadores aritméticos

Operador	Operação	Exemplo	Resultado
+	adição	x=1+2;	x=3
-	subtração	x=3-1;	x=2
*	multiplicação	x=2*3;	x=6
/	divisão	x=6/2;	x=3
%	módulo (resto da divisão inteira)	x=7%2;	x=1
++	incremento (equivale a x=x+1)	x=1; x++;	x=2
	equivale a y=x e x=x+1	x=1; y=0; y=x++;	x=2, y=1
	equivale a x=x+1 e y=x	x=1; y=0; y=++x;	x=2, y=2
--	decremento (equivale a x=x-1)	x=1; x--;	x=0
	equivale a y=x e x=x-1	x=1; y=0; y=x--;	x=0; y=1
	equivale a x=x-1 e y=x	x=1; y=0; y=--x;	x=0, y=0
+=	Soma e atribui (equivale a i=i+2)	i=1; i+=2;	i=3
-=	Subtrai e atribui (equivale a i=i-2)	i=1; i-=2;	i=-1
*=	Multiplica e atribui (equivale a i=i*2)	i=1; i*=2;	i=2
/=	Divide e atribui (equivale a i=i/2)	i=2; i/=2;	i=1
%=	Módulo e atribui (equivale a i=i%2)	i=1; i%=2;	i=1

Operadores relacionais

Operador	Operação
==	Igual a
!=	Diferente de
>	Maior que
<	Menor que
>=	Maior ou igual que
<=	Menor ou igual que

Nota: os operadores acima usam-se para comparar duas variáveis

O resultado é sempre do tipo **boolean**, e pode ser **true** ou **false**

Exemplos:

```
int a = 1;
int b = 1;
int c = a + b;

boolean teste1 = (a == b); // teste1 ficará com o valor true
boolean teste2 = (a == c); // teste2 ficará com o valor false
boolean teste3 = (a != c); // teste3 ficará com o valor true
boolean teste4 = (c >= a); // teste4 ficará com o valor true

if (teste1) { System.out.println("Verdade"); } else { System.out.println("Falso"); } // imprime "Verdade"

// estes dois "if's" são equivalentes:
if (a == c) { System.out.println("Verdade"); } else { System.out.println("Falso"); } // imprime "Falso"
if (teste2) { System.out.println("Verdade"); } else { System.out.println("Falso"); } // imprime "Falso"
```

Strings (Texto) e Métodos com Strings

Strings (texto) em Java

Duas formas de declarar uma String (como “literal” ou com “new”):

```
String str1 = "Informática no ISMT"; // Utilizando literal  
String str2 = new String("Informática no ISMT"); // Utilizando a palavra-chave new
```

Atenção à comparação de Strings: não usamos “==” mas sim “equals”:

```
str1 == str2 // compara endereços;  
  
String newStr = str1.equals(str2); // compara os valores  
  
String newStr = str1.equalsIgnoreCase(); // compara ignorando maiúscula/minúscula
```

Alguns métodos (apenas exemplos) que se podem usar com Strings:

```
newStr = str1.length(); // calcula o comprimento  
newStr = str1.charAt(i); // extrai o i-ésimo caracter  
newStr = str1.toUpperCase(); // retorna a String em MAIÚSCULAS  
newStr = str1.toLowerCase(); // retorna a String em minúsculas  
newStr = str1.replace(oldVal, newVal); // pesquisa e substitui  
newStr = str1.trim(); // elimina espaços em branco circundantes  
newStr = str1.contains("parte"); // verifica a presença de partes de texto  
newStr = str1.toCharArray(); // converte String para Array do tipo character  
newStr = str1.isEmpty(); // verifica se a String está vazia  
newStr = str1.endsWith(); // verifica se a String termina com o sufixo dado
```

Métodos em Java

Métodos são blocos de código que executam uma tarefa específica e podem ser chamados quando necessário. Eles são usados para organizar e modularizar o código, tornando-o mais fácil de entender, manter e reutilizar.

Definição de um Método:

Em Java, um método é definido com a seguinte estrutura:

```
tipo_de_retorno nome_do_método (lista_de_parâmetros) {  
  
    // Corpo do método  
  
}
```

Onde:

tipo_de_retorno: É o tipo de dado que o método irá retornar. Se não retornar nenhum valor, use a palavra-chave `void`.

nome_do_método: É o nome que identifica o método. Deve começar com letra minúscula e seguir o padrão `camelCase`.

lista_de_parâmetros: São as variáveis que o método recebe como entrada. Cada parâmetro é definido com um tipo e um nome, e múltiplos parâmetros são separados por vírgulas. Se o método não precisar de parâmetros, deixe os parênteses vazios.

Exemplo:

```
int soma(int a, int b) {  
  
    int resultado = a + b;  
  
    return resultado;  
  
}
```

Chamada de um Método:

Para utilizar um método, é necessário invocá-lo (chamar) no código. A chamada do método consiste em utilizar o seu nome seguido de parênteses, que contêm os argumentos a serem passados para os parâmetros do método.

Exemplo:

```
int resultado = soma(10, 20); // Invocação do método soma
```

Neste exemplo, estamos a chamar o método `soma` e a passar os valores 10 e 20 como argumentos. O método irá executar o seu código e retornar o resultado da soma, que será armazenado na variável `resultado`.

Métodos com void:

Quando um método não retorna um valor, utiliza-se a palavra-chave `void`. Estes métodos costumam realizar uma ação sem precisar de devolver um resultado.

Exemplo:

```
void mostrarMensagem(String mensagem) {  
  
    System.out.println(mensagem);  
  
}
```

Para chamar um método `void`, basta usar o seu nome seguido dos argumentos necessários:

```
mostrarMensagem("Olá, mundo!"); // A mensagem "Olá, mundo!" será exibida no  
ecrã
```

Métodos com múltiplos parâmetros:

Um método pode receber vários parâmetros, separados por vírgulas. Ao chamar o método, deve-se passar os argumentos na mesma ordem dos parâmetros.

Exemplo:

```
double calcularMedia(double valor1, double valor2, double valor3) {  
  
    double media = (valor1 + valor2 + valor3) / 3;  
  
    return media;  
  
}
```

Chamada do método:

```
double media = calcularMedia(8.5, 9.0, 7.5); // A média dos valores será  
calculada e armazenada na variável media
```

Com estas informações sobre métodos em Java, já pode começar a criar programas que utilizem métodos para organizar e simplificar o código. É importante praticar a criação e a chamada de métodos com diferentes tipos de retorno e parâmetros para aprimorar a compreensão e habilidade na linguagem Java.

Exemplo - Jogo “Adiviha um Número”:

Nas duas páginas seguintes, apresenta-se o código para um jogo simples e divertido, em que o jogador deve adivinhar um número entre 1 e 100. São usados métodos para organizar o código e facilitar a compreensão do programa.

Tente compreender bem o seu funcionamento.

```
import java.util.Scanner;

import java.util.Random;

public class JogoAdivinhacao {

    // Método para gerar um número aleatório entre 1 e 100

    public static int gerarNumeroAleatorio() {

        Random random = new Random();

        return random.nextInt(100) + 1;

    }

    // Método para receber o palpite do jogador

    public static int receberPalpite(Scanner teclado) {

        System.out.println("Introduza o seu palpite (1-100):");

        return teclado.nextInt();

    }

    // Método para verificar se o palpite está correto, e dar dicas ao jogador

    public static boolean verificarPalpite(int palpite, int numeroSecreto) {

        if (palpite == numeroSecreto) {

            System.out.println("Parabéns! Acertou o número secreto!");

            return true;

        } else if (palpite < numeroSecreto) {

            System.out.println("O número secreto é maior. Tente novamente!");

        } else {

            System.out.println("O número secreto é menor. Tente novamente!");

        }

        return false;

    }

}
```

```
public static void main(String[] args) {  
  
    Scanner teclado = new Scanner(System.in);  
  
    int numeroSecreto = gerarNumeroAleatorio();  
  
    boolean acertou = false;  
  
  
    System.out.println("Bem-vindo ao Jogo de Adivinhação de Número!");  
  
  
    while (!acertou) {  
  
        int palpite = receberPalpite(teclado);  
  
        acertou = verificarPalpite(palpite, numeroSecreto);  
  
    }  
  
    teclado.close();  
  
}
```

Estruturas de Controlo e Iteração

Nesta página, vamos abordar as estruturas de controlo e iteração em Java, que permitem executar diferentes partes do código consoante determinadas condições, bem como repetir a execução de partes do código.

Estrutura de seleção `if-else`:

A estrutura `if-else` é utilizada para tomar decisões no código, executando um bloco de instruções se uma condição for verdadeira e outro bloco se for falsa.

Exemplo:

```
if (condição) {  
    // Bloco de código a executar se a condição for verdadeira  
} else {  
    // Bloco de código a executar se a condição for falsa  
}
```

Por exemplo, para verificar se um número é par ou ímpar:

```
int numero = 5;  
  
if (numero % 2 == 0) {  
    System.out.println("O número é par.");  
} else {  
    System.out.println("O número é ímpar.");  
}
```

Estrutura de seleção switch-case:

A estrutura switch-case permite verificar múltiplas condições, sendo útil para simplificar o código quando se tem várias opções a verificar.

Exemplo:

```
switch (variável) {  
  
    case valor1:  
  
        // Bloco de código a executar se variável == valor1  
  
        break;  
  
    case valor2:  
  
        // Bloco de código a executar se variável == valor2  
  
        break;  
  
    default:  
  
        // Bloco de código a executar se nenhum dos valores anteriores corresponder  
  
        break;  
  
}
```

Um caso concreto: verificar o dia da semana:

```
int dia = 3;  
  
switch (dia) {  
  
    case 1:  
  
        System.out.println("Segunda-feira");  
  
        break;  
  
    case 2:  
  
        System.out.println("Terça-feira");  
  
        break;  
  
    case 3:  
  
        System.out.println("Quarta-feira");  
  
}
```

```
        break;

    case 4:

        System.out.println("Quinta-feira");

        break;

    case 5:

        System.out.println("Sexta-feira");

        break;

    case 6:

        System.out.println("Sábado");

        break;

    case 7:

        System.out.println("Domingo");

        break;

    default:

        System.out.println("Dia inválido");

}
```

Estrutura de repetição `while`:

A estrutura `while` permite repetir um bloco de código enquanto uma condição for verdadeira.

Exemplo:

```
while (condição) {  
    // Bloco de código a executar enquanto a condição for verdadeira  
}
```

Por exemplo, para imprimir os números de 1 a 5:

```
int contador = 1;  
  
while (contador <= 5) {  
    System.out.println(contador);  
    contador++;  
}
```

Estrutura de repetição do-while:

A estrutura do-while é semelhante ao while, mas garante que o bloco de código seja executado pelo menos uma vez, independentemente da condição.

Exemplo:

```
do {  
    // Bloco de código a executar  
} while (condição);
```

Por exemplo, para ler números até que o utilizador insira um número negativo:

```
int numero;  
  
Scanner teclado = new Scanner(System.in);  
  
do {  
    System.out.println("Introduza um número (negativo para sair):");  
    numero = teclado.nextInt();  
} while (numero >= 0);
```

Estrutura de repetição for:

A estrutura for é uma forma compacta de criar um loop, sendo especialmente útil quando se sabe quantas vezes o bloco de código deve ser repetido.

Exemplo:

```
for (inicialização; condição; atualização) {  
    // Bloco de código a executar  
}
```

inicialização: É executada uma vez antes do loop começar. Normalmente, é usada para declarar e inicializar a variável de controlo do loop.

condição: É avaliada antes de cada iteração. Se for verdadeira, o loop continua; se for falsa, o loop termina.

atualização: É executada após cada iteração, geralmente para atualizar a variável de controlo.

Por exemplo, para imprimir os números de 1 a 10:

```
for (int i = 1; i <= 10; i++) {  
    System.out.println(i);  
}
```

Estrutura de repetição **for-each**:

O loop **for-each** é uma forma simplificada do loop **for**, utilizado para iterar sobre elementos de um array ou de uma coleção (por exemplo, um **ArrayList**).

Exemplo:

```
for (tipo elemento : colecao) {  
    // Bloco de código a executar  
}
```

Por exemplo, para iterar e imprimir os elementos de um array:

```
int[] numeros = {1, 2, 3, 4, 5};  
for (int numero : numeros) { System.out.println(numero); }
```

Arrays e Listas

Nesta secção, vamos abordar os conceitos de arrays e listas em Java, que são estruturas de dados utilizadas para armazenar múltiplos elementos do mesmo tipo.

Arrays:

Arrays são estruturas de dados que armazenam uma quantidade fixa de elementos do mesmo tipo. Para criar um array, deve-se especificar o tipo de dado seguido de colchetes e atribuir um novo array com a palavra-chave `new` e o tamanho desejado.

Exemplo:

```
int[] numeros = new int[5]; // Array para armazenar 5 inteiros
```

Para atribuir valores a posições específicas do array, utiliza-se o índice da posição (começando em 0):

```
numeros[0] = 10;  
numeros[1] = 20;  
numeros[2] = 30;  
numeros[3] = 40;  
numeros[4] = 50;
```

Também é possível criar e inicializar um array diretamente com os valores:

```
int[] numeros = {10, 20, 30, 40, 50};
```

Para aceder aos valores armazenados no array, utiliza-se o índice da posição:

```
int primeiroNumero = numeros[0]; // A variável primeiroNumero receberá o  
valor 10
```

Listas (ArrayList):

Listas são estruturas de dados flexíveis que permitem armazenar uma quantidade variável de elementos do mesmo tipo. Em Java, uma das implementações mais comuns de lista é a `ArrayList`. Para utilizar listas, é necessário importar a classe `java.util.ArrayList`.

Exemplo:

```
import java.util.ArrayList;
```

Para criar uma lista, especifica-se o tipo de dado entre parênteses angulares (`<>`) e atribui-se uma nova instância de `ArrayList`:

```
ArrayList<Integer> numeros = new ArrayList<>(); // Lista para armazenar números inteiros
```

Para adicionar elementos à lista, utiliza-se o método `add`:

```
numeros.add(10);
```

```
numeros.add(20);
```

```
numeros.add(30);
```

Para aceder a elementos da lista, utiliza-se o método `get` e o índice da posição (começando em 0):

```
int primeiroNumero = numeros.get(0); // A variável primeiroNumero receberá o valor 10
```

Para alterar um elemento na lista, utiliza-se o método `set` e o índice da posição:

```
numeros.set(0, 99); // O primeiro elemento da lista será alterado para 99
```

Para remover um elemento da lista, utiliza-se o método `remove` e o índice da posição:

```
numeros.remove(1); // O segundo elemento da lista será removido
```

Para obter o tamanho da lista, utiliza-se o método `size`:

```
int tamanho = numeros.size(); // A variável tamanho receberá o valor 3
```

Com estas informações sobre arrays e listas em Java, já pode criar programas que manipulem coleções de dados de forma eficiente. É importante praticar a criação e manipulação de arrays e listas, bem como a interação entre eles e as estruturas de controlo e iteração apresentadas nas páginas anteriores.

Seguem-se alguns programas de exemplo de aplicação dos conceitos dados até agora. Serão usados arrays (ou `ArrayList`), estruturas de controle e iteração e métodos.

Exemplo A - Gestão de Contactos:

Utilizaremos um ArrayList, estruturas de controle e iteração e métodos para criar um programa que permite adicionar, listar e remover contactos.

```
import java.util.ArrayList;

import java.util.Scanner;

public class GestaoContactos {

    public static void menu() {

        System.out.println("\nGestão de Contactos");

        System.out.println("1. Adicionar contacto");

        System.out.println("2. Listar contactos");

        System.out.println("3. Remover contacto");

        System.out.println("4. Sair");

        System.out.print("Escolha uma opção: ");

    }

    public static void main(String[] args) {

        ArrayList<String> contactos = new ArrayList<>();

        Scanner teclado = new Scanner(System.in);

        int opcao;

        do {

            menu();

            opcao = teclado.nextInt();

            teclado.nextLine(); // Limpar o buffer do teclado

            switch (opcao) {

                case 1:

                    System.out.print("Introduza o nome do contacto: ");

                    String nome = teclado.nextLine();

                    contactos.add(nome);

                    System.out.println("Contacto adicionado com sucesso!");

                    break;
```

case 2:

```
System.out.println("\nLista de Contactos:");  
  
for (int i = 0; i < contactos.size(); i++) {  
    System.out.println((i + 1) + "." + contactos.get(i));  
}  
  
break;
```

case 3:

```
System.out.print("Introduza o número do contacto a remover: ");  
  
int indice = teclado.nextInt() - 1;  
  
if (indice >= 0 && indice < contactos.size()) {  
    contactos.remove(indice);  
    System.out.println("Contacto removido com sucesso!");  
} else {  
    System.out.println("Número de contacto inválido!");  
}  
  
break;
```

case 4:

```
System.out.println("A sair do programa...");  
  
break;
```

default:

```
System.out.println("Opção inválida! Tente novamente.");
```

```
}
```

```
} while (opcao != 4);
```

```
teclado.close();
```

```
}
```

```
}
```

Exemplo B- Conversor de Temperatura:

Neste exemplo, criaremos um programa simples para converter temperaturas entre graus Celsius e Fahrenheit:

```
import java.util.Scanner;

public class ConversorTemperatura {

    public static double celsiusParaFahrenheit(double celsius) {

        return (celsius * 9.0 / 5.0) + 32;

    }

    public static double fahrenheitParaCelsius(double fahrenheit) {

        return (fahrenheit - 32) * 5.0 / 9.0;

    }

    public static void menu() {

        System.out.println("\nConversor de Temperatura");

        System.out.println("1. Converter Celsius para Fahrenheit");

        System.out.println("2. Converter Fahrenheit para Celsius");

        System.out.println("3. Sair");

        System.out.print("Escolha uma opção: ");

    }

    public static void main(String[] args) {

        Scanner teclado = new Scanner(System.in);

        int opcao;

        do {

            menu();

            opcao = teclado.nextInt();

        }

    }

}
```

```
switch (opcao) {  
  
    case 1:  
  
        System.out.print("Introduza a temperatura em Celsius: ");  
  
        double tempCelsius = teclado.nextDouble();  
  
        double tempFahrenheit1 = celsiusParaFahrenheit(tempCelsius);  
  
        System.out.printf("%.2f°C = %.2f°F\n", tempCelsius, tempFahrenheit1);  
  
        break;  
  
    case 2:  
  
        System.out.print("Introduza a temperatura em Fahrenheit: ");  
  
        double tempFahrenheit2 = teclado.nextDouble();  
  
        double tempCelsius1 = fahrenheitParaCelsius(tempFahrenheit2);  
  
        System.out.printf("%.2f°F = %.2f°C\n", tempFahrenheit2, tempCelsius1);  
  
        break;  
  
    case 3:  
  
        System.out.println("A sair do programa...");  
  
        break;  
  
    default:  
  
        System.out.println("Opção inválida! Tente novamente.");  
  
}  
  
} while (opcao != 3);  
  
teclado.close();  
  
}  
  
}
```

“Exceções”

Nas próximas páginas, vamos abordar os conceitos de exceções e manipulação de ficheiros em Java, que são importantes para lidar com situações de erro e realizar operações de entrada e saída de dados.

Exceções:

Exceções são eventos que ocorrem durante a execução de um programa e que interrompem o seu fluxo normal. Em Java, as exceções são representadas por objetos de classes derivadas da classe `Throwable`. Existem duas categorias principais de exceções: as verificadas (checked) e as não verificadas (unchecked).

Para lidar com exceções, utiliza-se uma combinação das palavras-chave `try`, `catch` e `finally`.

Exemplo 1:

```
try {  
    // Bloco de código onde pode ocorrer uma exceção  
} catch (TipoDeExcecao1 e) {  
    // Bloco de código a executar se ocorrer a exceção TipoDeExcecao1  
} catch (TipoDeExcecao2 e) {  
    // Bloco de código a executar se ocorrer a exceção TipoDeExcecao2  
} finally {  
    // Bloco de código que será sempre executado, independentemente de  
    // ocorrer uma exceção ou não  
}
```


Exemplo 2:

Neste exemplo, um programa solicita ao utilizador dois números inteiros e realiza a divisão do primeiro pelo segundo. Utilizamos o tratamento de exceções para lidar com a situação de divisão por zero.

```
import java.util.Scanner;

public class ExemploExcecoes {

    public static void main(String[] args) {

        Scanner teclado = new Scanner(System.in);

        try {

            System.out.println("Introduza o primeiro número:");

            int num1 = teclado.nextInt();

            System.out.println("Introduza o segundo número:");

            int num2 = teclado.nextInt();

            int resultado = num1 / num2;

            System.out.println("Resultado da divisão: " + resultado);

        } catch (ArithmeticException e) {

            System.out.println("Erro: Divisão por zero não é permitida.");

        } finally {

            teclado.close();

            System.out.println("Fim do programa.");

        }

    }

}
```

Manipulação de Ficheiros

Manipulação de Ficheiros:

Em Java, é possível ler e escrever ficheiros utilizando diversas classes, como `File`, `FileReader`, `FileWriter`, `BufferedReader` e `BufferedWriter`. Nesta secção, abordaremos a utilização das classes `FileReader`, `BufferedReader`, `FileWriter` e `BufferedWriter` para a leitura e escrita de ficheiros.

Primeiro, é necessário importar as classes necessárias:

```
java

import java.io.BufferedReader;

import java.io.FileReader;

import java.io.FileWriter;

import java.io.BufferedWriter;

import java.io.IOException;
```

Leitura de Ficheiros:

Para ler um ficheiro, cria-se um objeto `FileReader` e um objeto `BufferedReader`. Utiliza-se o método `readLine` do objeto `BufferedReader` para ler as linhas do ficheiro.

Exemplo:

```
try {

    FileReader fileReader = new FileReader("caminho_do_ficheiro.txt");

    BufferedReader bufferedReader = new BufferedReader(fileReader);

    String linha;

    while ((linha = bufferedReader.readLine()) != null) {

        System.out.println(linha);

    }

    bufferedReader.close();

} catch (IOException e) {

    System.out.println("Erro ao ler o ficheiro: " + e.getMessage());

}
```

Escrita de Ficheiros:

Para escrever num ficheiro, cria-se um objeto `FileWriter` e um objeto `BufferedWriter`. Utiliza-se o método `write` do objeto `BufferedWriter` para escrever no ficheiro.

Exemplo:

```
try {  
  
    FileWriter fileWriter = new FileWriter("caminho_do_ficheiro.txt");  
  
    BufferedWriter bufferedWriter = new BufferedWriter(fileWriter);  
  
    bufferedWriter.write("Linha 1");  
  
    bufferedWriter.newLine();  
  
    bufferedWriter.write("Linha 2");  
  
    bufferedWriter.close();  
} catch (IOException e) {  
  
    System.out.println("Erro ao escrever no ficheiro: " + e.getMessage());  
}
```

Exemplo Suplementar (com operações de leitura e de escrita):

No exemplo apresentado na próxima página, um programa lê o conteúdo de um ficheiro chamado "ficheiro_entrada.txt" e cria uma cópia chamada "ficheiro_saida.txt".

Estes exemplos demonstram como lidar com exceções e realizar operações de leitura e escrita de ficheiros em Java. É importante praticar e adaptar estes exemplos a diferentes situações para desenvolver habilidades sólidas na manipulação de ficheiros e no tratamento de exceções.

```
import java.io.BufferedReader;

import java.io.BufferedWriter;

import java.io.FileReader;

import java.io.FileWriter;

import java.io.IOException;

public class ExemploFicheiros {

    public static void main(String[] args) {

        String ficheiroEntrada = "ficheiro_entrada.txt";

        String ficheiroSaida = "ficheiro_saida.txt";

        try {

            // Leitura do ficheiro

            FileReader fileReader = new FileReader(ficheiroEntrada);

            BufferedReader bufferedReader = new BufferedReader(fileReader);

            // Escrita do ficheiro

            FileWriter fileWriter = new FileWriter(ficheiroSaida);

            BufferedWriter bufferedWriter = new BufferedWriter(fileWriter);

            String linha;

            while ((linha = bufferedReader.readLine()) != null) {

                bufferedWriter.write(linha);

                bufferedWriter.newLine();

            }

        }
```

// Fechar os recursos utilizados

```
bufferedReader.close();
```

```
bufferedWriter.close();
```

```
System.out.println("Ficheiro copiado com sucesso!");
```

```
} catch (IOException e) {
```

```
System.out.println("Erro ao copiar o ficheiro: " + e.getMessage());
```

```
}
```

```
}
```

```
}
```

Sobre o uso de Maiúsculas e Minúsculas em Java:

A convenção "camelCase" é um estilo de escrita amplamente utilizado em Java e outras linguagens de programação para nomear variáveis, métodos e instâncias de objetos. Neste estilo, o nome é composto por várias palavras juntas, onde a primeira letra de cada palavra subsequente é maiúscula, e a primeira letra da primeira palavra é minúscula.

O nome "camelCase" vem da semelhança das letras maiúsculas no meio das palavras com as corcovas de um camelo. A ideia por trás dessa convenção é facilitar a leitura e compreensão dos nomes ao torná-los mais descritivos e distinguíveis.

Aqui estão alguns exemplos de nomes usando a convenção camelCase:

nomeDoUtilizador

dataDeNascimento

calcularSalario()

Para nomes de classes, é comum utilizar o "PascalCase", que é semelhante ao camelCase, mas a primeira letra da primeira palavra também é maiúscula. Por exemplo:

ContaBancaria

CarroEletrico

ConversorTemperatura

Adotar convenções de nomenclatura como camelCase e PascalCase melhora a legibilidade do código e facilita a manutenção ao seguir um padrão consistente em todo o projeto.

Exemplo (uso de maiúsculas e minúsculas em Java):

```
// Nome da classe em PascalCase

public class ConversorTemperatura {

    // Variável de tipo primitivo em camelCase

    private double factorConversao;

    // Método em camelCase

    public double celsiusParaFahrenheit(double tempCelsius) {

        // Variável de tipo primitivo em camelCase

        double tempFahrenheit = (tempCelsius * 9.0 / 5.0) + 32;

        return tempFahrenheit;

    }

    public double fahrenheitParaCelsius(double tempFahrenheit) {

        // Variável de tipo primitivo em camelCase

        double tempCelsius = (tempFahrenheit - 32) * 5.0 / 9.0;

        return tempCelsius;

    }

    public static void main(String[] args) {

        // Variável de tipo primitivo em camelCase

        double temperaturaCelsius = 25.0;

        double temperaturaFahrenheit = 77.0;

        // Objecto da classe em camelCase

        ConversorTemperatura conversor = new ConversorTemperatura ();
```

```
// Array em camelCase

double[] temperaturasCelsius = {0, 10, 20, 30, 40, 50};

double[] temperaturasFahrenheit = new double[temperaturasCelsius.length];

for (int i = 0; i < temperaturasCelsius.length; i++) {

    temperaturasFahrenheit[i] =
    conversor.celsiusParaFahrenheit(temperaturaCelsius[i]);

}

System.out.println("Temperaturas em Celsius:");

for (double tempCelsius : temperaturasCelsius) {

    System.out.printf("%.2f°C ", tempCelsius);

}

System.out.println("\nTemperaturas em Fahrenheit:");

for (double tempFahrenheit : temperaturasFahrenheit) {

    System.out.printf("%.2f°F ", tempFahrenheit);

}

// Uso do método em camelCase

double tempConvertidaCelsius =
conversor.fahrenheitParaCelsius(temperaturaFahrenheit);

double tempConvertidaFahrenheit =
conversor.celsiusParaFahrenheit(temperaturaCelsius);

System.out.printf("\n\n%.2f°C = %.2f°F", temperaturaCelsius,
tempConvertidaFahrenheit);

System.out.printf("\n\n%.2f°F = %.2f°C", temperaturaFahrenheit,
tempConvertidaCelsius);

}

}
```


Capítulo 2 – Programação Orientada por Objetos com Java

Introdução à Programação Orientada a Objetos (OOP)

A Programação Orientada a Objetos (OOP) é um paradigma de programação que utiliza objetos e suas interações para projetar e implementar aplicações. Neste paradigma, os objetos são instâncias de classes que encapsulam dados (atributos) e comportamentos (métodos). A OOP tem como objetivo facilitar a organização e modularização do código, tornando-o mais fácil de entender, manter e reutilizar. Os conceitos fundamentais da OOP são:

Classe: Uma classe é um modelo ou "blueprint" a partir do qual os objetos são criados. Define os atributos e métodos que os objetos dessa classe terão.

Objeto: Um objeto é uma instância de uma classe. Ele possui atributos (estado) e métodos (comportamento) definidos pela classe à qual pertence.

Herança: A herança é um mecanismo que permite que uma classe herde os atributos e métodos de outra classe, facilitando a reutilização e organização do código.

Encapsulamento: O encapsulamento é o processo de ocultar detalhes de implementação de uma classe e expor apenas uma interface segura e simples para interagir com ela.

Polimorfismo: O polimorfismo é a capacidade de tratar diferentes objetos como se fossem do mesmo tipo, permitindo que um método ou variável seja usada de várias maneiras, dependendo do tipo de objeto com o qual está a trabalhar.

Classes e Objetos

Vamos criar uma classe simples chamada `Carro` e **instanciar** alguns objetos dessa classe:

```
public class Carro {  
    // Atributos  
  
    String marca;  
  
    String modelo;  
  
    int ano;  
  
    // Método  
  
    void buzinar() {  
        System.out.println("Buzina do " + marca + " " + modelo + "!");  
    }  
}
```

```
public class TesteCarro {  
    public static void main(String[] args) {  
        // Instanciar objetos da classe Carro  
  
        Carro carro1 = new Carro();  
        carro1.marca = "Toyota";  
        carro1.modelo = "Corolla";  
        carro1.ano = 2020;  
  
        Carro carro2 = new Carro();  
        carro2.marca = "Ford";  
        carro2.modelo = "Fiesta";  
        carro2.ano = 2018;
```

```
// Utilizar métodos e atributos dos objetos

carro1.buzinar(); // Buzina do Toyota Corolla!

carro2.buzinar(); // Buzina do Ford Fiesta!

}

}
```

Neste exemplo, a classe Carro define atributos como marca, modelo e cor, além de métodos como acelerar e travar. Cada objeto criado a partir da classe Carro terá suas próprias características e comportamentos específicos.

Quando criamos um objeto, a partir da classe Carro, dizemos que estamos a instanciar um objeto dessa classe.

Como acabámos de ver, uma classe em Java é um modelo a partir do qual os objetos são criados. As classes definem os atributos (características) e métodos (comportamentos) comuns aos objetos dessa classe. Um objeto é uma instância de uma classe, representando uma entidade no mundo real com características e comportamentos específicos.

Criar uma Classe

Vamos criar uma outra classe simples chamada "Carro" (parecida mas não igual à anterior) para treinar o modo de como definir uma classe em Java:

```
public class Carro {

    // Atributos (características)

    String marca;

    String modelo;

    String cor;

    int velocidade;

    // Métodos (comportamentos)

    void acelerar(int incremento) {

        velocidade += incremento;

    }

    void travar(int decremento) {

        velocidade -= decremento;

    }

}
```

}

Neste exemplo, a classe Carro possui atributos como marca, modelo, cor e velocidade. Além disso, possui métodos como acelerar e travar que alteram a velocidade do carro.

Instanciar Objetos

Para criar objetos a partir de uma classe, utilizamos a palavra-chave "new" e o construtor da classe. Veja como criar objetos a partir da classe Carro:

```
public class TesteCarro {  
  
    public static void main(String[] args) {  
  
        // Criar objetos (instanciar a classe Carro)  
  
        Carro carro1 = new Carro();  
  
        Carro carro2 = new Carro();  
  
  
        // Definir características do carro1  
        carro1.marca = "Toyota";  
        carro1.modelo = "Corolla";  
        carro1.cor = "Preto";  
        carro1.velocidade = 0;  
  
  
        // Definir características do carro2  
        carro2.marca = "Honda";  
        carro2.modelo = "Civic";  
        carro2.cor = "Vermelho";  
        carro2.velocidade = 0;  
  
  
        // Utilizar métodos dos objetos  
        carro1.acelerar(20);  
  
        System.out.println("A velocidade do carro1 é: " + carro1.velocidade + "  
km/h");  
    }  
}
```

```
carro2.travar(10);

System.out.println("A velocidade do carro2 é: " + carro2.velocidade + "
km/h");

}

}
```

Neste exemplo, criamos duas instâncias da classe Carro chamadas "carro1" e "carro2". Em seguida, definimos os atributos de cada objeto e utilizamos os métodos acelerar e travar. Ao executar o programa, ele mostrará a velocidade atual de cada carro após as operações.

Noção de construtor

Um construtor é um bloco de código especial usado para inicializar objetos em uma classe. Ele é chamado automaticamente quando um objeto é criado e tem o mesmo nome que a classe à qual pertence. Os construtores são usados para definir valores iniciais para os atributos dos objetos, e podem ter parâmetros ou não.

Aqui estão dois exemplos simples de construtores em Java:

Construtor sem parâmetros (ver comentário no código):

```
public class Estudante {  
  
    String nome;  
  
    int idade;  
  
    // Construtor sem parâmetros  
    public Estudante() {  
        nome = "Desconhecido";  
        idade = 0;  
    }  
}
```

Construtor com parâmetros (ver comentário no código):

```
public class Estudante {  
  
    String nome;  
  
    int idade;  
  
    // Construtor com parâmetros  
    public Estudante(String nome, int idade) {  
        this.nome = nome;  
        this.idade = idade;  
    }  
}
```

Como usar os construtores (ver comentário no código):

```
public class Principal {  
  
    public static void main(String[] args) {  
  
        // Usando o construtor sem parâmetros  
  
        Estudante estudante1 = new Estudante();  
  
        System.out.println("Nome: " + estudante1.nome + ", Idade: " +  
estudante1.idade);  
  
  
        // Usando o construtor com parâmetros  
  
        Estudante estudante2 = new Estudante("Ana", 20);  
  
        System.out.println("Nome: " + estudante2.nome + ", Idade: " +  
estudante2.idade);  
  
    }  
}
```

No exemplo anterior, criámos dois objetos da classe `Estudante` usando os dois construtores diferentes e imprimimos os valores dos seus atributos.

Quando escrevemos um construtor com parâmetros para uma classe, o construtor padrão (sem parâmetros) fornecido automaticamente pelo Java deixa de estar disponível. Isso acontece porque o compilador Java cria um construtor padrão apenas se você não fornecer nenhum construtor para a classe.

Vamos usar a classe `Estudante` como exemplo:

```
public class Estudante {  
  
    String nome;  
  
    int idade;  
  
    // Construtor com parâmetros  
  
    public Estudante(String nome, int idade) {  
  
        this.nome = nome;  
  
        this.idade = idade;  
  
    }  
}
```

Neste exemplo, escrevemos um construtor com parâmetros para a classe Estudante. Como resultado, o construtor padrão deixa de estar disponível e não será criado automaticamente pelo compilador Java. Se tentarmos criar um objeto Estudante usando o construtor padrão, receberemos um erro de compilação:

```
public class Principal {  
  
    public static void main(String[] args) {  
  
        // Tentando usar o construtor padrão - causará um erro de compilação  
  
        Estudante estudante1 = new Estudante(); // Erro: O construtor Estudante() não está  
definido  
  
    }  
  
}
```


Para resolver esse problema e disponibilizar o construtor padrão novamente, você deve declará-lo explicitamente na classe:

```
public class Estudante {  
  
    String nome;  
  
    int idade;  
  
    // Construtor padrão (sem parâmetros)  
  
    public Estudante() {  
  
        nome = "Desconhecido";  
  
        idade = 0;  
  
    }  
  
    // Construtor com parâmetros  
  
    public Estudante(String nome, int idade) {  
  
        this.nome = nome;  
  
        this.idade = idade;  
  
    }  
  
}
```

Agora, com o construtor padrão declarado explicitamente na classe Estudante, podemos criar objetos usando tanto o construtor padrão quanto o construtor com parâmetros:

```
public class Principal {  
  
    public static void main(String[] args) {  
  
        // Usando o construtor padrão  
  
        Estudante estudante1 = new Estudante();  
  
        System.out.println("Nome: " + estudante1.nome + ", Idade: " +  
estudante1.idade);  
  
    }  
  
}
```

// Usando o construtor com parâmetros

```
Estudante estudante2 = new Estudante("Ana", 20);

    System.out.println("Nome: " + estudante2.nome + ", Idade: " +
estudante2.idade);

}

}
```

Em resumo, quando escrevemos um construtor com parâmetros para uma classe, o construtor padrão não é mais fornecido automaticamente pelo Java. Para disponibilizá-lo, você precisa declará-lo explicitamente na classe.

Encapsulamento

O encapsulamento é um dos princípios fundamentais da Programação Orientada a Objetos. É a prática de esconder os detalhes internos de uma classe e expor apenas uma interface bem definida para interagir com essa classe. Isso é alcançado usando modificadores de acesso e métodos getter e setter.

Modificadores de Acesso:

Os modificadores de acesso em Java são utilizados para controlar o acesso aos membros (atributos e métodos) de uma classe. Existem quatro tipos de modificadores de acesso:

public: O membro é acessível por qualquer classe.

private: O membro só é acessível dentro da própria classe.

protected: O membro é acessível dentro da própria classe, suas subclasses e classes no mesmo pacote.

(sem modificador): O membro é acessível dentro de classes no mesmo pacote.

Exemplo com modificadores de acesso:

```
public class Carro {  
  
    // Atributos (características)  
  
    private String marca;  
  
    private String modelo;  
  
    private String cor;  
  
    private int velocidade;  
  
  
    // Métodos (comportamentos)  
  
    public void acelerar(int incremento) {  
  
        velocidade += incremento;  
  
    }  
  
  
    public void travar(int decremento) {  
  
        velocidade -= decremento;  
  
    }  
  
}
```

Neste exemplo, os atributos da classe Carro são declarados como "private", o que significa que eles só podem ser acessados diretamente dentro da própria classe. Os métodos acelerar e travar são declarados como "public", então eles podem ser acessados por outras classes.

Getters e Setters

Para acessar e modificar os atributos private de uma classe, utilizamos métodos getter e setter. Um método getter retorna o valor de um atributo, enquanto um método setter define o valor de um atributo.

Exemplo com getters e setters:

```
public class Carro {  
  
    // Atributos (características)  
  
    private String marca;  
  
    private String modelo;  
  
    private String cor;  
  
    private int velocidade;  
  
  
    // Getters e Setters  
  
    public String getMarca() {  
  
        return marca;  
  
    }  
  
  
    public void setMarca(String marca) {  
  
        this.marca = marca;  
  
    }  
  
  
    public String getModelo() {  
  
        return modelo;  
  
    }  
  
  
    public void setModelo(String modelo) {  
  
        this.modelo = modelo;  
  
    }  
  
}
```

```
public String getCor() {  
    return cor;  
}  
  
public void setCor(String cor) {  
    this.cor = cor;  
}  
  
public int getVelocidade() {  
    return velocidade;  
}  
  
// Métodos (comportamentos)  
public void acelerar(int incremento) {  
    velocidade += incremento;  
}  
  
public void travar(int decremento) {  
    velocidade -= decremento;  
}  
}
```

Neste exemplo, adicionamos métodos getter e setter para os atributos marca, modelo e cor. Agora, outras classes podem utilizar esses métodos para acessar e modificar os atributos private da classe Carro.

Agora, para utilizar a classe Carro com encapsulamento, precisamos ajustar o código do exemplo anterior:

```
public class TesteCarro {
```

```
public static void main(String[] args) {  
  
    // Criar objetos (instanciar a classe Carro)  
  
    Carro carro1 = new Carro ();  
  
    Carro carro2 = new Carro ();  
  
  
    // Definir características do carro1 usando setters  
  
    carro1.setMarca ("Toyota");  
  
    carro1.setModelo ("Corolla");  
  
    carro1.setCor ("Preto");  
  
    carro1.setVelocidade (0);  
  
  
    // Definir características do carro2 usando setters  
  
    carro2.setMarca ("Honda");  
  
    carro2.setModelo ("Civic");  
  
    carro2.setCor ("Vermelho");  
  
    carro2.setVelocidade (0);  
  
  
    // Utilizar métodos dos objetos  
  
    carro1.acelerar (20);  
  
    System.out.println ("A velocidade do carro1 é: " + carro1.getVelocidade () + "  
km/h");  
  
  
    carro2.travar (10);  
  
    System.out.println ("A velocidade do carro2 é: " + carro2.getVelocidade () + "  
km/h");  
  
    }  
  
}
```

Neste exemplo, ajustamos o código para utilizar os métodos getters e setters em vez de aceder diretamente aos atributos da classe Carro. Ao encapsular os atributos e fornecer métodos públicos para aceder e modificar esses atributos, garantimos que o estado interno dos objetos seja protegido e consistente.

Com este exemplo, abordamos o conceito de encapsulamento em Java e como aplicá-lo usando modificadores de acesso e métodos getters e setters. No próximo tópico, exploraremos a herança, um conceito importante na Programação Orientada a Objetos que permite a reutilização de código e a organização de classes em hierarquias.

Herança

A herança é um conceito fundamental da Programação Orientada a Objetos que permite a uma classe herdar atributos e métodos de outra classe. A herança facilita a reutilização de código e a organização de classes em hierarquias.

Em Java, a herança é representada pela palavra-chave "extends". Quando uma classe herda de outra classe, a classe herdeira é chamada de subclasse e a classe herdada é chamada de superclasse.

Vamos usar um exemplo para ilustrar a herança em Java. Suponha que temos uma classe base chamada "Veiculo" e queremos criar classes específicas para Carro e Motocicleta, que herdam atributos e métodos da classe Veiculo.

Classe Veiculo (superclasse):

```
public class Veiculo {  
  
    String marca;  
  
    String modelo;  
  
    String cor;  
  
    int velocidade;  
  
    void acelerar(int incremento) {  
  
        velocidade += incremento;  
  
    }  
  
    void travar(int decremento) {  
  
        velocidade -= decremento;  
  
    }  
}
```

Classe Carro (subclasse):

```
public class Carro extends Veiculo {  
  
    int numeroDePortas;  
  
    void buzinar() {  
  
        System.out.println("O carro está a buzinar!");  
    }  
}
```



```
}  
  
}
```

Classe Motocicleta (subclasse):

```
public class Motocicleta extends Veiculo {  
  
    boolean capacete;  
  
    void empinar() {  
  
        System.out.println("A motocicleta está a empinar!");  
  
    }  
  
}
```

Neste exemplo, a classe Carro e a classe Motocicleta herdam atributos e métodos da classe Veiculo. Além disso, cada subclasse pode ter seus próprios atributos e métodos específicos, como o número de portas para o Carro e o uso de capacete para a Motocicleta.

Ao utilizar herança, podemos reutilizar código e criar classes mais específicas que herdam características e comportamentos comuns de uma classe base. No próximo tópico, abordaremos o polimorfismo, outro conceito importante na Programação Orientada a Objetos que permite tratar diferentes objetos como se fossem do mesmo tipo e simplificar a escrita de código genérico.

Polimorfismo

O polimorfismo é um conceito chave na Programação Orientada a Objetos que permite que diferentes objetos sejam tratados como se fossem do mesmo tipo. O polimorfismo facilita a escrita de código genérico, uma vez que permite tratar objetos de diferentes classes através de uma referência comum.

Em Java, o polimorfismo é implementado através do uso de herança e interfaces. Vamos usar um exemplo para ilustrar o polimorfismo em Java. Vamos criar uma interface chamada "Animal" e duas classes que implementam essa interface, "Cão" e "Gato".

Interface Animal:

```
public interface Animal {  
  
    void emitirSom();  
  
}
```

Classe Cão:

```
public class Cao implements Animal {  
  
    @Override  
  
    public void emitirSom() {  
  
        System.out.println("O cão faz: au au!");  
  
    }  
  
}
```

Classe Gato:

```
public class Gato implements Animal {  
  
    @Override  
  
    public void emitirSom() {  
  
        System.out.println("O gato faz: miau!");  
  
    }  
  
}
```

Neste exemplo, a interface `Animal` define um método chamado `"emitirSom()"`. As classes `Cão` e `Gato` implementam essa interface e fornecem suas próprias implementações do método `"emitirSom()"`.

Agora podemos usar o polimorfismo para tratar objetos das classes `Cão` e `Gato` como se fossem do mesmo tipo:

```
public class TestePolimorfismo {  
  
    public static void main(String[] args) {  
  
        Animal meuCao = new Cao();  
  
        Animal meuGato = new Gato();  
  
        meuCao.emitirSom(); // O cão faz: au au!  
  
        meuGato.emitirSom(); // O gato faz: miau!  
  
    }  
  
}
```

Neste exemplo, criamos referências do tipo `Animal` para os objetos das classes `Cão` e `Gato`. Graças ao polimorfismo, podemos chamar o método `"emitirSom()"` em ambas as referências, e a implementação correta do método será executada dependendo do tipo real do objeto.

Com este exemplo, exploramos o conceito de polimorfismo em Java e como ele permite tratar diferentes objetos como se fossem do mesmo tipo. O polimorfismo facilita a escrita de código genérico e a organização de classes em hierarquias de herança e interfaces.

Composição

A composição é outro conceito importante na Programação Orientada a Objetos, que permite construir classes complexas a partir da combinação de outras classes mais simples. A composição estabelece uma relação de "tem-um" (has-a) entre as classes, o que significa que uma classe contém uma instância de outra classe como um dos seus atributos.

A composição facilita a reutilização de código, a organização de classes em estruturas modulares e a manutenção de aplicações.

Vamos usar um exemplo para ilustrar a composição em Java. Suponha que temos uma classe "Motor" e queremos criar uma classe "Carro" que inclui um motor como um dos seus atributos.

Classe Motor:

```
public class Motor {  
  
    String tipo;  
  
    int potencia;  
  
    public Motor(String tipo, int potencia) {  
  
        this.tipo = tipo;  
  
        this.potencia = potencia;  
  
    }  
}
```

Classe Carro:

```
public class Carro {  
  
    String marca;  
  
    String modelo;  
  
    Motor motor;  
  
    public Carro(String marca, String modelo, String tipoMotor, int potenciaMotor) {  
  
        this.marca = marca;
```

```
        this.modelo = modelo;

        this.motor = new Motor(tipoMotor, potenciaMotor);

    }

}
```

Neste exemplo, a classe Carro inclui um atributo "motor" do tipo Motor. Quando criamos um objeto da classe Carro, também criamos um objeto da classe Motor e atribuímos ao atributo "motor" do carro. Isso ilustra a relação de composição entre as classes Carro e Motor.

Agora podemos criar um objeto Carro e aceder ao motor como um dos seus atributos:

```
public class TesteComposicao {

    public static void main(String[] args) {

        Carro meuCarro = new Carro("Toyota", "Corolla", "Gasolina", 130);

        System.out.println("Marca: " + meuCarro.marca);

        System.out.println("Modelo: " + meuCarro.modelo);

        System.out.println("Tipo de motor: " + meuCarro.motor.tipo);

        System.out.println("Potência do motor: " + meuCarro.motor.potencia);

    }

}
```

Neste exemplo, criamos um objeto da classe Carro e utilizamos a composição para aceder aos atributos do motor. A composição permite-nos criar classes complexas a partir da combinação de outras classes mais simples e facilita a reutilização de código e a organização de classes em estruturas modulares.

Interfaces

As interfaces são um conceito importante na Programação Orientada a Objetos que permite definir um contrato para as classes que implementam a interface. Uma interface define um conjunto de métodos que as classes devem implementar, mas não fornece uma implementação concreta para esses métodos. As interfaces permitem criar código flexível e reutilizável, pois permitem que diferentes classes sejam tratadas como se fossem do mesmo tipo, através do polimorfismo.

Em Java, as interfaces são definidas com a palavra-chave "interface". Vamos usar um exemplo para ilustrar o uso de interfaces em Java. Suponha que queremos criar uma interface chamada "Imprimivel" que define um método "imprimir()".

Interface Imprimivel:

```
public interface Imprimivel {  
  
    void imprimir();  
  
}
```

Agora, podemos criar classes que implementam a interface Imprimivel, fornecendo suas próprias implementações do método "imprimir()".

Classe Documento:

```
public class Documento implements Imprimivel {  
  
    String texto;  
  
    public Documento(String texto) {  
  
        this.texto = texto;  
  
    }  
  
    @Override  
  
    public void imprimir() {  
  
        System.out.println("Imprimindo documento: " + texto);  
  
    }  
  
}
```

Classe Foto:

```
public class Foto implements Imprimivel {  
  
    String descricao;  
  
    public Foto(String descricao) {  
  
        this.descricao = descricao;  
  
    }  
  
    @Override  
  
    public void imprimir() {  
  
        System.out.println("Imprimindo foto: " + descricao);  
  
    }  
}
```

Neste exemplo, as classes Documento e Foto implementam a interface Imprimivel e fornecem suas próprias implementações do método "imprimir()". Graças ao polimorfismo, podemos tratar objetos das classes Documento e Foto como se fossem do mesmo tipo:

```
public class TesteInterfaces {  
  
    public static void main(String[] args) {  
  
        Imprimivel documento = new Documento("Exemplo de texto");  
  
        Imprimivel foto = new Foto("Férias na praia");  
  
        documento.imprimir(); // Imprimindo documento: Exemplo de texto  
  
        foto.imprimir(); // Imprimindo foto: Férias na praia  
  
    }  
}
```

Neste exemplo, utilizamos interfaces para definir um contrato comum para as classes Documento e Foto. As interfaces permitem criar código flexível e reutilizável, pois facilitam a criação de diferentes classes que podem ser tratadas como se fossem do mesmo tipo, através do polimorfismo.

Portanto, como vimos, quando uma classe implementa uma interface, ela concorda em fornecer a implementação de todos os métodos declarados na interface.

Vamos ver mais alguns exemplos simples, para esclarecer o conceito de Interface:

Interface `Calculadora` com métodos para operações aritméticas básicas

```
public interface Calculadora {  
  
    double somar(double a, double b);  
  
    double subtrair(double a, double b);  
  
    double multiplicar(double a, double b);  
  
    double dividir(double a, double b) throws ArithmeticException;  
  
}  
  
public class CalculadoraImpl implements Calculadora {  
  
    @Override  
    public double somar(double a, double b) {  
        return a + b;  
    }  
  
    @Override  
    public double subtrair(double a, double b) {  
        return a - b;  
    }  
  
    @Override  
    public double multiplicar(double a, double b) {  
        return a * b;  
    }  
  
    @Override  
    public double dividir(double a, double b) throws ArithmeticException {  
        if (b == 0) {  
            throw new ArithmeticException ("Divisão por zero") ;  
        }  
  
        return a / b;  
    }  
}
```



```
}  
  
}
```

Neste exemplo, a interface `Calculadora` possui quatro métodos para realizar operações aritméticas básicas. A classe `CalculadoraImpl` implementa a interface e fornece implementações para todos os métodos.

Nota sobre o uso de “`@Override`”:

Chama-se a isto uma anotação (as anotações começam com o símbolo “`@`”). Esta anotação, em particular, indica que o método está a sobreescrever (ou seja, fornecendo uma implementação) um método da interface ou da classe pai. É uma forma de assinalar que se está a cumprir o “contrato” de implementar os métodos descritos na interface, ajudando também a identificar erros de digitação. Na verdade, escrever esta anotação é opcional (o programa corre, embora o editor assinale um “warning”/aviso), mas é uma boa prática escrever `@Override` nestes casos.

E ainda mais um exemplo, sobre interfaces:

Interface `Reproduzivel` para objetos que podem ser reproduzidos e pausados

```
public interface Reproduzivel {  
  
    void reproduzir();  
  
    void pausar();  
  
    void parar();  
  
}  
  
public class Musica implements Reproduzivel {  
  
    @Override  
    public void reproduzir() {  
  
        System.out.println("Reproduzindo música");  
  
    }  
  
    @Override  
    public void pausar() {  
  
        System.out.println("Pausando música");  
  
    }  
  
}
```

```
@Override

public void parar() {

    System.out.println("Parando música");

}

}

public class Video implements Reproduzivel {

    @Override

    public void reproduzir() {

        System.out.println("Reproduzindo vídeo");

    }

    @Override

    public void pausar() {

        System.out.println("Pausando vídeo");

    }

    @Override

    public void parar() {

        System.out.println("Parando vídeo");

    }

}
```

Neste exemplo, a interface `Reproduzivel` possui três métodos: `reproduzir`, `pausar` e `parar`. As classes `Musica` e `Video` implementam a interface `Reproduzivel`, fornecendo implementações para todos os métodos. Esta interface permite que objetos de diferentes tipos sejam tratados de maneira genérica quando se trata de reprodução, pausa e parada.

Classes Abstratas

Uma classe abstrata é uma classe que não pode ser instanciada diretamente e pode conter métodos abstratos, que não têm uma implementação na classe abstrata e devem ser implementados nas subclasses. As classes abstratas são usadas para fornecer uma base comum para as subclasses e encapsular a lógica comum. Aqui estão alguns exemplos simples:

Exemplo:

```
public abstract class Veiculo {  
  
    private int velocidade;  
  
    public void acelerar(int quantidade) {  
        velocidade += quantidade;  
    }  
  
    public abstract void buzinar(); // Método abstrato  
}  
  
public class Carro extends Veiculo {  
  
    @Override  
    public void buzinar() {  
        System.out.println("Beep beep!");  
    }  
}  
  
public class Bicicleta extends Veiculo {  
  
    @Override  
    public void buzinar() {  
        System.out.println("Trin trin!");  
    }  
}
```

}

Neste exemplo, temos uma classe abstrata `Veiculo` com um atributo `velocidade` e um método `acelerar()`. A classe `Veiculo` também possui um método abstrato `buzinar()` que não tem implementação. As subclasses `Carro` e `Bicicleta` estendem a classe `Veiculo` e fornecem suas próprias implementações para o método `buzinar()`.

Ainda outro exemplo de uma classe abstrata:

```
public abstract class Animal {  
  
    private String nome;  
  
    public Animal(String nome) {  
  
        this.nome = nome;  
  
    }  
  
    public String getNome() {  
  
        return nome;  
  
    }  
  
    public abstract void comunicar(); // Método abstrato  
  
}  
  
public class Cao extends Animal {  
  
    public Cao(String nome) {  
  
        super(nome);  
  
    }  
  
    @Override  
  
    public void comunicar() {  
  
        System.out.println("Au au!");  
  
    }  
  
}
```

```
}
```

```
public class Gato extends Animal {  
  
    public Gato(String nome) {  
  
        super(nome);  
  
    }  
  
    @Override  
  
    public void comunicar() {  
  
        System.out.println("Miau!");  
  
    }  
  
}
```

Neste outro exemplo, a classe abstrata `Animal` possui um atributo `nome`, um construtor e um método `getNome()`. A classe `Animal` também possui um método abstrato `comunicar()`. As subclasses `Cao` e `Gato` estendem a classe `Animal` e fornecem suas próprias implementações para o método `comunicar()`.

Quando uma classe não abstrata herda de uma classe abstrata, ela é obrigada a implementar todos os métodos declarados como abstratos na classe abstrata. Se a classe não abstrata não fornecer uma implementação para algum dos métodos abstratos, ocorrerá um erro de compilação.

Por exemplo, o código a seguir gera um erro de compilação:

```
abstract class Veiculo {  
  
    abstract void acelerar();  
  
    abstract void parar();  
  
}  
  
class Carro extends Veiculo {  
  
    @Override  
  
    void acelerar() {  
  
        System.out.println("O carro está acelerando");  
  
    }  
  
}  
  
public class ExemploErro {  
  
    public static void main(String[] args) {  
  
        Carro carro = new Carro();  
  
        carro.acelerar();  
  
    }  
  
}
```

Para resolver este erro, basta fornecer uma implementação para o método `parar()` na classe `Carro`:

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

```
class Carro extends Veiculo {  
  
    @Override  
    void acelerar() {  
        System.out.println("O carro está acelerando");  
    }  
  
    @Override  
    void parar() {  
        System.out.println("O carro está parando");  
    }  
}  
  
public class ExemploCorrigido {  
    public static void main(String[] args) {  
        Carro carro = new Carro();  
        carro.acelerar();  
        carro.parar();  
    }  
}
```

Agora, a classe `Carro` fornece implementações para ambos os métodos abstratos `acelerar()` e `parar()`, e o código compila e executa corretamente.

O uso de “static” em Java

Em Java, a palavra-chave `static` é usada para indicar que um membro (variável, método ou bloco) pertence à classe em si e não a uma instância específica da classe (objeto). Quando uma variável é declarada como `static`, existe apenas uma cópia dessa variável, que é compartilhada por todas as instâncias da classe.

As variáveis `static` são frequentemente usadas para representar constantes ou informações que devem ser compartilhadas entre todos os objetos da classe.

Vejamos um exemplo simples de uso de variáveis `static`:

```
public class Contador {  
  
    // Variável estática para manter a contagem de objetos criados  
  
    static int contador = 0;  
  
  
    // Construtor para incrementar o contador sempre que um novo objeto é criado  
  
    public Contador() {  
  
        contador++;  
  
    }  
  
  
    // Método estático para obter o valor do contador  
  
    public static int getContador() {  
  
        return contador;  
  
    }  
  
}
```

Neste exemplo, criamos uma classe chamada `Contador`. A variável `contador` é declarada como `static`, o que significa que ela pertence à classe e é compartilhada por todos os objetos dessa classe. O construtor da classe `Contador` incrementa a variável `contador` sempre que um novo objeto é criado. Além disso, o método `getContador()` também é declarado como `static`, permitindo que seja chamado diretamente através da classe, sem a necessidade de criar um objeto.

Agora, vejamos como usar a variável `static` e o método `static`:

```
public class Principal {  
  
    public static void main(String[] args) {  
  
        // Criando três objetos da classe Contador  
  
        Contador obj1 = new Contador();  
  
        Contador obj2 = new Contador();  
  
        Contador obj3 = new Contador();  
  
  
        // Acessando o valor do contador usando o método estático  
  
        System.out.println("Número de objetos criados: " + Contador.getContador());  
  
    }  
  
}
```

Neste exemplo, criamos três objetos da classe `Contador`. Como a variável `contador` é `static`, ela é compartilhada por todos os objetos e seu valor é incrementado cada vez que um novo objeto é criado. No final, usamos o método `static` `getContador()` para exibir o número total de objetos criados.

Em resumo, as variáveis `static` em Java são usadas para representar membros que pertencem à classe como um todo e não a instâncias específicas da classe. Essas variáveis são compartilhadas por todos os objetos da classe e são úteis para armazenar informações globais ou constantes.

Definição de constantes em Java (“`static final`”)

Usamos `final` para indicar que a variável é uma constante e o seu valor não pode ser alterado após a atribuição inicial.

A convenção para nomear constantes em Java é usar letras maiúsculas e separar as palavras com sublinhados (`_`). Aqui está um exemplo simples:

```
public class ExemploConstantes {  
  
    public static final int LARGURA_JANELA = 400;  
  
    public static final int ALTURA_JANELA = 300;  
  
  
    public static void main(String[] args) {
```

```
        System.out.println("A largura da janela é: " + LARGURA_JANELA);

        System.out.println("A altura da janela é: " + ALTURA_JANELA);

    }

}
```

Neste exemplo, definimos duas constantes `LARGURA_JANELA` e `ALTURA_JANELA`, que representam a largura e a altura de uma janela.

O uso de “this” em Java

A palavra-chave `this` é usada para se referir à instância atual de um objeto em Java. Ela pode ser usada para acessar atributos e métodos da instância atual e é especialmente útil para diferenciar atributos da classe de parâmetros de métodos ou construtores com nomes semelhantes.

Exemplo de uso de `this`:

```
public class Ponto {

    int x;

    int y;

    // Construtor com parâmetros

    public Ponto(int x, int y) {

        this.x = x; // Atribui o valor do parâmetro x ao atributo x da
        instância atual

        this.y = y; // Atribui o valor do parâmetro y ao atributo y da
        instância atual

    }

}
```

Neste exemplo, a classe `Ponto` tem dois atributos, `x` e `y`, e um construtor com parâmetros. A palavra-chave `this` é usada para diferenciar os atributos da instância atual dos parâmetros do construtor.

Exemplo de uso de `this ()` <- notar os parêntesis à frente de “this”

`this ()` é uma chamada especial de construtor que pode ser usada para invocar outro construtor na mesma classe. Isso pode ser útil quando você tem vários construtores e deseja reutilizar a lógica de um construtor dentro de outro.

Exemplo de uso de `this()`:

```
public class Ponto {  
  
    int x;  
  
    int y;  
  
    // Construtor padrão (sem parâmetros)  
  
    public Ponto() {  
        this(0, 0); // Chama o construtor com parâmetros passando valores  
        padrão (0, 0)  
    }  
  
    // Construtor com parâmetros  
  
    public Ponto(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
}
```

Neste exemplo, a classe `Ponto` possui dois construtores: um padrão (sem parâmetros) e um com parâmetros. No construtor padrão, a chamada `this(0, 0)` é usada para invocar o construtor com parâmetros e atribuir valores padrão aos atributos `x` e `y` da instância atual.

Em resumo, a palavra-chave `this` é usada para se referir à instância atual de um objeto em Java e diferenciar atributos da classe de parâmetros de métodos ou construtores com nomes semelhantes. `this()` é uma chamada de construtor especial que permite invocar outro construtor na mesma classe, útil para reutilizar a lógica de construtores e simplificar o código.

O uso de “super” em Java

A palavra-chave `super` é usada para se referir à superclasse imediata (classe pai) de uma classe. Ela pode ser usada para acessar atributos e métodos da superclasse que foram ocultados ou sobrescritos na subclasse (classe filha).

Exemplo de uso de `super`:

```
public class Veiculo {  
    int velocidade;  
  
    public void acelerar() {  
        velocidade += 5;  
    }  
}  
  
public class Carro extends Veiculo {  
    int velocidade;  
  
    public void acelerar() {  
        super.acelerar(); // Chama o método acelerar() da superclasse  
        // Veiculo  
        velocidade += 10;  
    }  
}
```

Neste exemplo, a classe `Carro` estende a classe `Veiculo`. Ambas as classes possuem um atributo chamado `velocidade` e um método chamado `acelerar()`. Na classe `Carro`, a palavra-chave `super` é usada para chamar o método `acelerar()` da superclasse `Veiculo`.

Exemplo de uso de `super()`:

`super()`, com parênteses a seguir à palavra, é uma chamada especial de construtor usada para invocar o construtor da superclasse imediata. Isso pode ser útil quando você deseja inicializar os atributos herdados da superclasse antes de inicializar os atributos específicos da subclasse.

Exemplo de uso de `super()`:

```
public class Veiculo {  
  
    String marca;  
  
    // Construtor com parâmetros  
  
    public Veiculo(String marca) {  
  
        this.marca = marca;  
  
    }  
  
}  
  
  
public class Carro extends Veiculo {  
  
    int portas;  
  
  
    // Construtor com parâmetros  
  
    public Carro(String marca, int portas) {  
  
        super(marca); // Chama o construtor da superclasse Veiculo passando  
o parâmetro marca  
  
        this.portas = portas;  
  
    }  
  
}
```

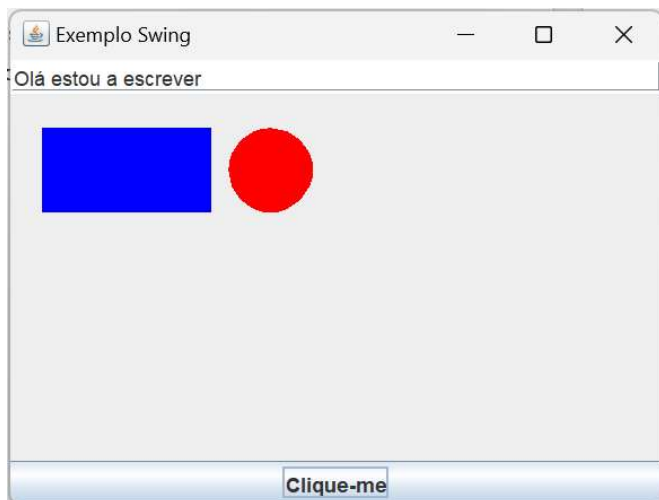
Neste exemplo, a classe `Carro` estende a classe `Veiculo`. A classe `Veiculo` possui um atributo chamado `marca` e um construtor com parâmetros. A classe `Carro` possui um atributo chamado `portas` e um construtor com parâmetros. No construtor da classe `Carro`, a chamada `super(marca)` é usada para invocar o construtor da superclasse `Veiculo` e inicializar o atributo `marca` antes de inicializar o atributo `portas`.

Em resumo, a palavra-chave `super` é usada para se referir à superclasse imediata de uma classe em Java e acessar atributos e métodos da superclasse que foram ocultados ou sobrescritos na subclasse. `super()` é uma chamada especial de construtor que permite invocar o construtor da superclasse, útil para inicializar os atributos herdados da superclasse antes de inicializar os atributos específicos da subclasse.

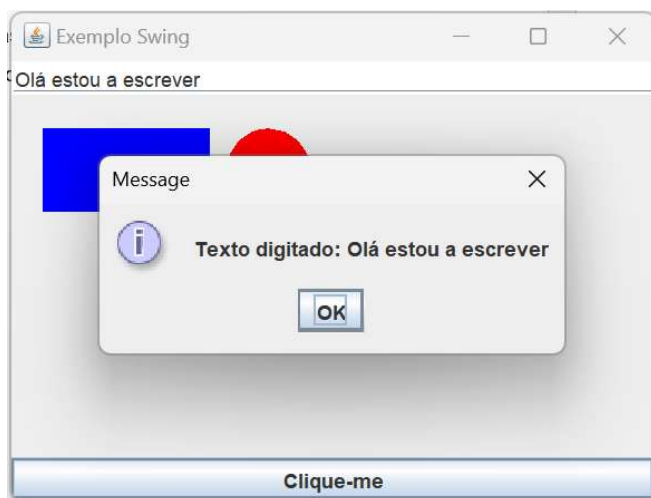
Gráficos: uso das bibliotecas Swing e AWT

O exemplo dado na página seguinte cria uma janela com botões, imagens, cores, formas geométricas e campos de texto.

Quando o programa corre, podemos visualizar a seguinte janela:



E quando carregamos no botão “Clique-me” aparece esta mensagem:



É importante constatar que todos os conceitos que temos abordado neste documento são importantes para compreender o código da página seguinte: o Java é uma linguagem que, para ser compreendida, temos de compreender as noções de programação orientada por objetos. Convido-o a ler atentamente o código e a tentar relacionar o que cada parte faz com as imagens que vê em cima.

```
import javax.swing.*;

import java.awt.*;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

public class ExemploSwing {

    public static void main(String[] args) {

        JFrame janela = new JFrame("Exemplo Swing");

        janela.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        janela.setSize(400, 300);

        JPanel painel = new JPanel();

        painel.setLayout(new BorderLayout());

        // Criar e adicionar botão

        JButton botao = new JButton("Clique-me");

        painel.add(botao, BorderLayout.SOUTH);

        // Criar e adicionar campo de texto

        JTextField campoTexto = new JTextField("Escreve aqui");

        painel.add(campoTexto, BorderLayout.NORTH);

        // Criar e adicionar imagem

        ImageIcon imagemIcon = new ImageIcon("caminho/para/imagem.jpg");

        JLabel imagemLabel = new JLabel(imagemIcon);

        painel.add(imagemLabel, BorderLayout.WEST);
```



```
// Criar e adicionar painel de desenho
```

```
JPanel painelDesenho = new JPanel() {
```

```
    @Override
```

```
    protected void paintComponent(Graphics g) {
```

```
        super.paintComponent(g);
```

```
        g.setColor(Color.BLUE);
```

```
        g.fillRect(20, 20, 100, 50);
```

```
        g.setColor(Color.RED);
```

```
        g.fillOval(130, 20, 50, 50);
```

```
    }
```

```
};
```

```
painel.add(painelDesenho, BorderLayout.CENTER);
```

```
// Adicionar ação ao botão
```

```
botao.addActionListener(new ActionListener() {
```

```
    @Override
```

```
    public void actionPerformed(ActionEvent e) {
```

```
        String texto = campoTexto.getText();
```

```
        JOptionPane.showMessageDialog(janela, "Texto digitado: " +  
texto);
```

```
    }
```

```
});
```

```
janela.getContentPane().add(painel);
```

```
janela.setVisible(true);
```

```
}
```

```
}
```

Exceções e Tratamento de Erros

Nota: embora já tenhamos abordado este tópico na Parte A deste documento (e na UC de Introdução à Programação), é importante incluir também na Parte B, para quem ler diretamente esta parte, sem ter lido antes a anterior.

Exceções são eventos que ocorrem durante a execução de um programa e que podem interromper o fluxo normal de instruções. O tratamento de erros é uma parte essencial da programação, pois ajuda a garantir que o seu programa funcione corretamente, mesmo quando ocorrem erros ou situações inesperadas.

Em Java, as exceções são representadas por objetos e são lançadas quando ocorre uma condição de erro. Existem dois tipos principais de exceções em Java:

Exceções verificadas (checked exceptions): são exceções que o programador deve prever e tratar, como por exemplo, `FileNotFoundException` e `IOException`.

Exceções não verificadas (unchecked exceptions): são exceções que ocorrem devido a erros de programação, como `NullPointerException` e `ArrayIndexOutOfBoundsException`.

Para tratar exceções em Java, utiliza-se um bloco try-catch:

```
try {  
  
    // Código que pode lançar uma exceção  
  
} catch (TipoDeExcecao excecao) {  
  
    // Código para lidar com a exceção  
  
}
```

Por exemplo, podemos usar um bloco try-catch para tratar uma exceção ao tentar ler um ficheiro:

```
import java.io.File;  
  
import java.io.FileNotFoundException;  
  
import java.util.Scanner;  
  
public class TesteExcecoes {  
  
    public static void main(String[] args) {  
  
        try {
```

```
File ficheiro = new File("arquivo_inexistente.txt");

Scanner scanner = new Scanner(ficheiro);

} catch (FileNotFoundException e) {

    System.out.println("Erro: Ficheiro não encontrado!");

}

}
```

Neste exemplo, colocamos o código que pode lançar uma `FileNotFoundException` dentro de um bloco `try` e tratamos a exceção no bloco `catch` correspondente.

Também é possível usar o bloco `finally`, que será executado independentemente de uma exceção ser lançada ou não:

```
try {

    // Código que pode lançar uma exceção

} catch (TipoDeExcecao excecao) {

    // Código para lidar com a exceção

} finally {

    // Código que será executado independentemente de uma exceção ser lançada ou não

}
```

O tratamento de erros é uma parte essencial da programação, e o uso de exceções em Java permite tratar e gerir situações inesperadas ou de erro de forma eficiente e organizada.

Coleções em Java (*Java Collections Framework*)

A *Java Collections Framework* é um conjunto de classes e interfaces que fornecem estruturas de dados e algoritmos comuns para manipular coleções de objetos. Algumas das estruturas de dados mais comuns incluem Listas, Conjuntos (Sets) e Mapas (Maps). Estas coleções são muito úteis para armazenar, manipular e organizar dados em programas Java.

Listas (List): Uma lista é uma coleção ordenada de elementos, que podem ser duplicados. A interface List é implementada pelas classes ArrayList, LinkedList, e outras. Aqui está um exemplo de como usar a classe ArrayList:

```
import java.util.ArrayList;

import java.util.List;

public class TesteColecoes {

    public static void main(String[] args) {

        List<String> listaDeNomes = new ArrayList<>();

        listaDeNomes.add("Maria");

        listaDeNomes.add("João");

        listaDeNomes.add("Pedro");

        for (String nome : listaDeNomes) {

            System.out.println(nome);

        }

    }

}
```

Conjuntos (Set): Um conjunto é uma coleção de elementos sem ordem e sem duplicados. A interface Set é implementada pelas classes HashSet, TreeSet, e outras. Aqui está um exemplo de como usar a classe HashSet:

```
import java.util.HashSet;

import java.util.Set;
```

```
public class TesteColecoes {  
    public static void main(String[] args) {  
        Set<String> conjuntoDeNomes = new HashSet<>();  
  
        conjuntoDeNomes.add("Maria");  
        conjuntoDeNomes.add("João");  
        conjuntoDeNomes.add("Pedro");  
        conjuntoDeNomes.add("Maria"); // Este elemento não será adicionado,  
        pois já existe no conjunto  
  
        for (String nome : conjuntoDeNomes) {  
            System.out.println(nome);  
        }  
    }  
}
```

Mapas (Map): Um mapa é uma coleção de pares chave-valor, onde cada chave é única. A interface Map é implementada pelas classes HashMap, TreeMap, e outras. Aqui está um exemplo de como usar a classe HashMap:

```
import java.util.HashMap;  
import java.util.Map;  
  
public class TesteColecoes {  
    public static void main(String[] args) {  
        Map<String, Integer> mapaDeIdades = new HashMap<>();  
  
        mapaDeIdades.put("Maria", 30);  
        mapaDeIdades.put("João", 25);  
        mapaDeIdades.put("Pedro", 28);  
    }  
}
```

```
for (Map.Entry<String, Integer> entrada: mapaDeIdades.entrySet()) {  
    System.out.println(entrada.getKey() + " tem " +  
    entrada.getValue() + " anos.");  
}  
  
}  
  
}
```

O Java Collections Framework fornece uma série de utilitários e algoritmos para manipular e trabalhar com coleções de dados de forma eficiente e padronizada. Além das estruturas de dados mencionadas acima, o framework também fornece métodos para ordenar, pesquisar e realizar outras operações comuns em coleções.

Aqui estão algumas das principais classes e interfaces do Java Collections Framework:

Iterable: Interface base para todas as coleções que podem ser percorridas usando um laço for-each.

Collection: Interface base para todas as coleções. Ela estende a interface Iterable.

List: Interface para listas ordenadas de elementos, que podem conter duplicados.

Set: Interface para conjuntos de elementos sem ordem e sem duplicados.

Map: Interface para coleções de pares chave-valor, onde cada chave é única.

Queue: Interface para coleções que implementam uma fila (FIFO - First In, First Out).

Deque: Interface para coleções que implementam uma fila de duas pontas (pode adicionar ou remover elementos de ambas as extremidades).

O Java Collections Framework é uma parte essencial da biblioteca padrão Java e facilita a manipulação de dados em programas Java. Compreender e usar eficientemente essas estruturas de dados e algoritmos pode melhorar significativamente a qualidade e a eficiência do seu código.

Multithreading em Java

Multithreading é um conceito que permite a execução de várias partes de um programa em simultâneo. Em Java, isso é conseguido através da criação de múltiplas "threads". Cada thread é uma sequência independente de instruções que pode ser executada em paralelo com outras threads.

O multithreading é útil em aplicações onde certas tarefas podem ser executadas de forma independente e simultânea, melhorando assim a eficiência e o desempenho do programa.

Criar uma Thread em Java

Existem duas formas principais de criar uma thread em Java:

Estender a classe `Thread`.

Implementar a interface `Runnable`.

Método 1: Estender a classe `Thread`

Para criar uma thread estendendo a classe `Thread`, siga estes passos:

Crie uma nova classe que estenda a classe `Thread`.

Substitua o método `run()` com o código que deve ser executado pela thread.

Crie uma instância da classe e chame o método `start()` para iniciar a thread.

```
class MinhaThread extends Thread {  
  
    public void run() {  
  
        for (int i = 0; i < 5; i++) {  
  
            System.out.println("Minha Thread: " + i);  
  
        }  
  
    }  
  
}
```

```
public class ExemploThread {  
  
    public static void main(String[] args) {  
  
        MinhaThread t1 = new MinhaThread();  
  
        t1.start();  
  
    }  
  
}
```

Método 2: Implementar a interface Runnable

Para criar uma thread implementando a interface `Runnable`, siga estes passos:

Crie uma nova classe que implemente a interface `Runnable`.

Implemente o método `run()` com o código que deve ser executado pela thread.

Crie uma instância da classe `Thread`, passando a instância da classe `Runnable` como argumento.

Chame o método `start()` para iniciar a thread.

```
class MinhaRunnable implements Runnable {  
  
    public void run() {  
  
        for (int i = 0; i < 5; i++) {  
  
            System.out.println("Minha Runnable: " + i);  
  
        }  
  
    }  
  
}
```

```
public class ExemploRunnable {

    public static void main(String[] args) {

        MinhaRunnable r1 = new MinhaRunnable();

        Thread t1 = new Thread(r1);

        t1.start();

    }

}
```

Sincronização de Threads

Quando várias threads acessam recursos compartilhados, podem ocorrer problemas de sincronização. Para evitar esses problemas, Java fornece o mecanismo de sincronização. O método mais comum é usar o bloco `synchronized`.

Bloco synchronized

Um bloco `synchronized` é usado para garantir que apenas uma thread possa aceder a um recurso partilhado de cada vez. Para utilizar um bloco `synchronized`, utilize a palavra-chave `synchronized` seguida de parênteses que contenham um objecto de referência e, em seguida, coloque o código que necessita de ser sincronizado entre chavetas.

Exemplo:

```
class Contador {  
  
    private int contagem;  
  
    public void incrementar() {  
        synchronized (this) {  
            contagem++;  
        }  
    }  
  
    public int getContagem() {  
        synchronized (this) {  
            return contagem;  
        }  
    }  
}  
  
class TarefaIncrementadora implements Runnable {  
  
    private Contador contador;  
  
    public TarefaIncrementadora(Contador contador) {  
        this.contador = contador;  
    }  
}
```

```
public void run() {  
  
    for (int i = 0; i < 5; i++) {  
  
        contador.incrementar();  
  
        System.out.println("Contagem: " + contador.getContagem());  
  
    }  
  
}  
  
}
```

```
public class ExemploSincronizado {  
  
    public static void main(String[] args) {  
  
        Contador contador = new Contador();  
  
        TarefaIncrementadora r1 = new TarefaIncrementadora(contador);  
  
        TarefaIncrementadora r2 = new TarefaIncrementadora(contador);  
  
  
        Thread t1 = new Thread(r1);  
  
        Thread t2 = new Thread(r2);  
  
  
        t1.start();  
  
        t2.start();  
  
    }  
  
}
```

No exemplo acima, a classe `Contador` possui dois métodos que são acedidos simultaneamente pelas threads `t1` e `t2`. Os blocos `synchronized` garantem que apenas uma thread possa aceder e modificar o valor de `contagem` de cada vez.

Métodos `synchronized`

Além do bloco `synchronized`, também é possível utilizar a palavra-chave `synchronized` diretamente num método. Neste caso, o método inteiro será sincronizado, e apenas uma thread poderá aceder a ele de cada vez.

Exemplo:

```
class Contador {  
  
    private int contagem;  
  
    public synchronized void incrementar() {  
        contagem++;  
    }  
  
    public synchronized int getContagem() {  
        return contagem;  
    }  
}
```

// A classe TarefaIncrementadora e a classe ExemploSincronizado permanecem inalteradas.

Padrões de Projeto (tópico avançado / opcional)

Nota: este tópico é avançado, e não será alvo de avaliação no teste escrito. Recomenda-se a leitura, porque é relevante para UC's futuras e para a prática profissional.

Padrões de projeto são soluções comprovadas e reutilizáveis para problemas comuns encontrados no desenvolvimento de software. Eles ajudam a melhorar a qualidade, modularidade e eficiência do seu código. Neste tópico, vamos abordar três padrões de projeto comuns: Singleton, Factory e Observer, e como aplicá-los em Java.

Estes são três padrões de desenho ("design patterns") frequentemente encontrados em projetos de software profissionais.

Singleton:

O padrão Singleton garante que uma classe tenha apenas uma instância e fornece um ponto de acesso global a essa instância. É útil quando você deseja garantir que apenas um objeto seja criado para gerenciar um recurso compartilhado.

Exemplo de implementação do Singleton:

```
public class Singleton {  
  
    private static Singleton instanciaUnica;  
  
    private Singleton() {  
        // Construtor privado para evitar instanciação fora da classe  
    }  
  
    public static synchronized Singleton getInstanciaUnica() {  
        if (instanciaUnica == null) {  
            instanciaUnica = new Singleton();  
        }  
        return instanciaUnica;  
    }  
}
```

Factory:

O padrão Factory permite a criação de objetos sem expor a lógica de criação ao cliente. Ele define uma interface comum para criar objetos em uma superclasse, permitindo que as subclasses decidam que classe instanciar.

Exemplo de implementação do Factory:

```
public interface Animal {  
  
    void falar();  
  
}  
  
public class Cao implements Animal {  
  
    @Override  
    public void falar() {  
        System.out.println("Au au!");  
    }  
}  
  
public class Gato implements Animal {  
  
    @Override  
    public void falar() {  
        System.out.println("Miau!");  
    }  
}  
  
public class AnimalFactory {  
  
    public static Animal criarAnimal(String tipo) {  
        if ("cao".equalsIgnoreCase(tipo)) {  
            return new Cao();  
        } else if ("gato".equalsIgnoreCase(tipo)) {  
            return new Gato();  
        }  
    }  
}
```

```
    }  
  
    return null;  
  
}  
  
}
```

A classe `AnimalFactory` é responsável por criar objetos das subclasses de `Animal`, com base no parâmetro `tipo`. Ao usar o padrão `Factory`, o código cliente não precisa saber detalhes sobre a criação de objetos específicos, como `Cao` e `Gato`, tornando o processo de criação mais flexível e desacoplado.

Eis como poderia ser usada a nossa `AnimalFactory`, de forma mais detalhada:

```
public class TestePadraoFactory {  
  
    public static void main(String[] args) {  
  
        // Criar um objeto Animal do tipo Cao  
  
        Animal cao = AnimalFactory.criarAnimal("cao");  
  
        cao.falar(); // Deve imprimir "Au au!"  
  
  
        // Criar um objeto Animal do tipo Gato  
  
        Animal gato = AnimalFactory.criarAnimal("gato");  
  
        gato.falar(); // Deve imprimir "Miau!"  
  
  
        // Criar uma lista de animais e fazê-los falar  
  
        List<Animal> animais = new ArrayList<>();  
  
        animais.add(AnimalFactory.criarAnimal("cao"));  
  
        animais.add(AnimalFactory.criarAnimal("gato"));  
  
        animais.add(AnimalFactory.criarAnimal("cao"));  
  
  
        for (Animal animal : animais) {  
  
            animal.falar();  
  
        }  
  
    }  
}
```

```
}  
  
// Deve imprimir:  
  
// Au au!  
  
// Miau!  
  
// Au au!  
  
}  
  
}
```

No código acima, criamos instâncias de animais usando a `AnimalFactory` e chamamos o método `falar()` em cada objeto. No primeiro exemplo, criamos um objeto `Animal` do tipo `Cao` e um objeto `Animal` do tipo `Gato`. Em seguida, criamos uma lista de animais e adicionamos três animais a ela, usando a `AnimalFactory` para criar os objetos. Por fim, iteramos sobre a lista de animais e chamamos o método `falar()` em cada animal.

Graças à `AnimalFactory`, o código cliente não precisa saber como criar instâncias específicas de animais, apenas interage com a interface `Animal` e delega a criação de objetos à `factory`.

Aqui está outro exemplo do padrão Factory, desta vez com uma classe abstrata e uma interface para a factory:

```
public interface Forma {  
  
    void desenhar();  
  
}
```

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

```
public class Retangulo implements Forma {  
  
    @Override  
    public void desenhar() {  
  
        System.out.println("Desenhando um retângulo");  
  
    }  
  
}
```

```
public abstract class FormaFactory {  
  
    public abstract Forma criarForma(String tipo);  
  
    public static FormaFactory getFactory() {  
  
        return new FormaFactoryImpl();  
  
    }  
  
}
```



```
class FormaFactoryImpl extends FormaFactory {

    @Override

    public Forma criarForma(String tipo) {

        if ("circulo".equalsIgnoreCase(tipo)) {

            return new Circulo();

        } else if ("retangulo".equalsIgnoreCase(tipo)) {

            return new Retangulo();

        }

        return null;

    }

}

public class TestePadraoFactory {

    public static void main(String[] args) {

        FormaFactory factory = FormaFactory.getFactory();

        Forma circulo = factory.criarForma("circulo");

        circulo.desenhar();

        Forma retangulo = factory.criarForma("retangulo");

        retangulo.desenhar();

    }

}
```

No exemplo apresentado, temos uma interface `Forma` e duas implementações (`Circulo` e `Retangulo`). A classe abstrata `FormaFactory` define um método abstrato `criarForma` e um método estático `getFactory` para obter uma instância concreta da factory (`FormaFactoryImpl`). A classe

FormaFactoryImpl implementa o método `criarForma`, criando objetos `Circulo` e `Retangulo` com base no parâmetro `tipo`.

O código cliente usa a factory para criar e manipular objetos `Forma` sem se preocupar com os detalhes de implementação das classes `Circulo` e `Retangulo`. Isso torna o processo de criação e uso de objetos `Forma` mais flexível e desacoplado, facilitando a adição de novas formas no futuro sem afetar o código cliente existente.

No método `main` da classe `TestePadraoFactory`, criamos instâncias de `Circulo` e `Retangulo` usando a factory e chamamos o método `desenhar()` em cada objeto. Graças ao padrão Factory, o código cliente não precisa saber como criar instâncias específicas de `Forma`, apenas interage com a interface `Forma`.

Eis uma explicação mais detalhada de como poderia ser usada a nossa FormaFactory:

```
public class TestePadraoFactory {  
  
    public static void main(String[] args) {  
  
        // Obter uma instância da FormaFactory  
  
        FormaFactory factory = FormaFactory.getFactory();  
  
  
        // Criar um objeto Forma do tipo Circulo  
  
        Forma circulo = factory.criarForma("circulo");  
  
        circulo.desenhar(); // Deve imprimir "Desenhando um círculo"  
  
  
        // Criar um objeto Forma do tipo Retangulo  
  
        Forma retangulo = factory.criarForma("retangulo");  
  
        retangulo.desenhar(); // Deve imprimir "Desenhando um retângulo"  
  
  
        // Criar uma lista de formas e desenhá-las  
  
        List<Forma> formas = new ArrayList<>();  
  
        formas.add(factory.criarForma("circulo"));  
  
        formas.add(factory.criarForma("retangulo"));  
  
        formas.add(factory.criarForma("circulo"));  
  
  
        for (Forma forma : formas) {  
  
            forma.desenhar();  
  
        }  
  
        // Deve imprimir:  
  
        // Desenhando um círculo  
  
        // Desenhando um retângulo
```

// Desenhando um círculo

}

}

Em resumo, o padrão Factory é útil para desacoplar a criação de objetos das classes concretas e simplificar a criação de novos tipos de objetos sem afetar o código cliente existente.

Observer

O padrão Observer é um padrão de projeto comportamental que permite que um objeto (chamado de "subject" ou "observável") mantenha uma lista de objetos dependentes (chamados de "observers" ou "observadores") e os notifique automaticamente sobre qualquer mudança de estado. Este padrão é útil quando você deseja manter vários objetos atualizados com as mudanças de estado de outro objeto sem acoplamento direto entre eles.

Aqui está um exemplo simples de como aplicar o padrão Observer em Java:

```
java

// Interface do observador

public interface Observador {

    void atualizar(String noticia);

}

// Classe concreta do observador

public class Assinante implements Observador {

    private String nome;

    public Assinante(String nome) {

        this.nome = nome;

    }

    @Override

    public void atualizar(String noticia) {

        System.out.println(nome + " recebeu a notícia: " + noticia);

    }

}

// Classe do sujeito/observável
```

```
public class Noticiario {  
  
    private List<Observador> observadores = new ArrayList<>();  
  
    public void adicionarObservador(Observador observador) {  
        observadores.add(observador);  
    }  
  
    public void removerObservador(Observador observador) {  
        observadores.remove(observador);  
    }  
  
    public void publicarNoticia(String noticia) {  
        for (Observador observador : observadores) {  
            observador.atualizar(noticia);  
        }  
    }  
}
```

// Classe principal para testar o padrão Observer

```
public class TestePadraoObserver {  
  
    public static void main(String[] args) {  
        Noticiario noticiario = new Noticiario();  
  
        Assinante assinante1 = new Assinante("João");  
        Assinante assinante2 = new Assinante("Maria");  
        Assinante assinante3 = new Assinante("Pedro");  
    }  
}
```

```
noticiario.adicionarObservador(assinante1);  
  
noticiario.adicionarObservador(assinante2);  
  
noticiario.adicionarObservador(assinante3);  
  
  
noticiario.publicarNoticia("Nova notícia sobre tecnologia!");  
  
  
  
noticiario.removerObservador(assinante2);  
  
  
noticiario.publicarNoticia("Nova notícia sobre ciência!");  
  
}  
  
}
```

Neste exemplo, temos uma classe `Noticiario` que representa o sujeito/observável e uma interface `Observador` com uma implementação concreta chamada `Assinante`. A classe `Noticiario` mantém uma lista de objetos `Observador` e possui métodos para adicionar, remover e notificar observadores. Quando uma notícia é publicada, todos os observadores são notificados através do método `atualizar()`.

Na classe `TestePadraoObserver`, criamos uma instância do `Noticiario` e várias instâncias de `Assinante`. Adicionamos os assinantes ao noticiário e publicamos uma notícia. Todos os assinantes são notificados e recebem a notícia. Depois, removemos um assinante e publicamos outra notícia. Os assinantes restantes são notificados, enquanto o assinante removido não recebe a notícia.

Capítulo 3 – Introdução à Programação Web em Java

Vaadin Flow

O que é possível fazer com o Vaadin Flow

Para criar uma interface de utilizador (UI) numa aplicação Vaadin Flow com Spring Boot, pode utilizar componentes Java fornecidos pelo Vaadin para construir a estrutura e funcionalidade da sua aplicação. O Vaadin Flow permite desenvolver aplicações web modernas inteiramente em Java, sem a necessidade de escrever HTML ou JavaScript diretamente.

Passos para Criar uma UI com Vaadin Flow:

1. **Definir uma Vista Principal:** Crie uma classe Java que servirá como a vista principal da sua aplicação. Utilize a anotação `@Route` para mapear esta vista a uma rota específica.

```
import com.vaadin.flow.component.button.Button;
import com.vaadin.flow.component.orderedlayout.VerticalLayout;
import com.vaadin.flow.component.textfield.TextField;
import com.vaadin.flow.router.Route;

@Route("")
public class MainView extends VerticalLayout {

    public MainView() {
        TextField nomeField = new TextField("O seu nome");
        Button saudacaoButton = new Button("Cumprimentar",
            e -> add(new Paragraph("Olá, " + nomeField.getValue() +
                "!!")));
        add(nomeField, saudacaoButton);
    }
}
```

Neste exemplo, ao aceder à raiz da aplicação ("/"), o utilizador verá um campo de texto para inserir o nome e um botão que, ao ser clicado, exibe uma saudação personalizada.

2. **Adicionar Componentes e Layouts:** Utilize componentes como *TextField*, *Button* e layouts como *VerticalLayout* para organizar a interface. Os componentes do Vaadin são altamente personalizáveis e permitem a criação de UIs interativas de forma declarativa em Java.
3. **Gerir Eventos:** Adicione *listeners* de eventos aos componentes para definir comportamentos interativos. Por exemplo, o botão no exemplo acima possui um ouvinte que adiciona um parágrafo com a saudação ao layout quando clicado.
4. **Navegação entre Vistas:** Para criar uma aplicação com múltiplas vistas, defina várias classes anotadas com *@Route*, cada uma representando uma vista diferente. Utilize o método *UI.getCurrent().navigate("rota")* para navegar entre elas.

```
@Route("outra-vista")
public class OutraVista extends VerticalLayout {
    public OutraVista() {
        add(new Text("Esta é outra vista."));
    }
}
```

Para navegar para "outra-vista":

```
Button navegarButton = new Button("Ir para Outra Vista",
    e -> UI.getCurrent().navigate("outra-vista"));
```

5. **Estilização:** Personalize o estilo da sua aplicação utilizando temas e folhas de estilo CSS. O Vaadin permite a integração de estilos para garantir que a UI corresponde à identidade visual desejada.

Ao seguir estes passos e utilizar os recursos fornecidos pelo Vaadin, pode desenvolver interfaces de utilizador modernas e responsivas de forma eficiente, aproveitando o poder do Java no desenvolvimento web.

Como configurar um projeto para usar Vaadin Flow (baseado em Spring Boot):

O Vaadin Flow é uma framework que permite aos programadores desenvolver aplicações web inteiramente em Java. Em vez de escrever HTML, CSS e JavaScript, a interface do utilizador é construída a partir de componentes de UI em Java, de forma semelhante ao que é feito com Swing e JavaFX. O código HTML, as folhas de estilo CSS e os JavaScripts ainda estão presentes e acessíveis quando necessário, mas são abstraídos por uma API rica em Java.

O Flow atua como um controlo remoto para os elementos HTML no navegador. Estes elementos HTML podem ser simples elementos `<div>` ou elementos mais complexos, como um componente web `<vaadin-grid>`. O Flow controla os atributos, propriedades e filhos desses elementos, podendo até executar invocações JavaScript personalizadas. No lado do servidor, existem objetos Java correspondentes com os quais o programador interage. O Flow gere a sincronização entre os objetos Java do lado do servidor e os elementos HTML do lado do cliente.

Ao construir uma aplicação com o Vaadin Flow, o programador pode concentrar-se na lógica da interface do utilizador em Java, enquanto o Flow trata da comunicação cliente-servidor e da renderização no navegador. Isto permite uma abordagem mais produtiva no desenvolvimento de aplicações web modernas, aproveitando a robustez e familiaridade do ecossistema Java.

Para começar a utilizar o Vaadin Flow com Spring Boot, é recomendável seguir tutoriais que abordem desde a configuração inicial do projeto até à implementação de funcionalidades específicas, garantindo uma compreensão sólida de como integrar o Vaadin Flow em aplicações Spring Boot.

Para iniciar o desenvolvimento de uma aplicação web com o Vaadin Flow e o Spring Boot, siga os passos abaixo:

1. Criação do Projeto

A forma mais simples de iniciar um projeto com Vaadin e Spring Boot é utilizar o [Vaadin Start](#). Esta ferramenta permite configurar e descarregar um projeto inicial personalizado. Alternativamente, pode utilizar o [Spring Initializr](#) para configurar um projeto Spring Boot e, em seguida, adicionar as dependências do Vaadin manualmente.

2. Estrutura do Projeto

Um projeto típico com Vaadin e Spring Boot inclui as seguintes partes:

- **Dependências:** As bibliotecas necessárias são geridas no ficheiro *pom.xml* (para projetos Maven) ou *build.gradle* (para projetos Gradle).
- **Classe Principal:** Uma classe anotada com `@SpringBootApplication` que contém o método *main* para iniciar a aplicação.

- **Vistas:** Classes Java que representam as diferentes páginas ou componentes da interface do utilizador, anotadas com `@Route`.

3. Adicionar Dependências

Para integrar o Vaadin com o Spring Boot, adicione as seguintes dependências ao seu ficheiro `pom.xml`:

(por favor, ver página seguinte, para uma visão global do ficheiro numa única página)

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>com.vaadin</groupId>
      <artifactId>vaadin-bom</artifactId>
      <version>${vaadin.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>

<dependencies>
  <dependency>
    <groupId>com.vaadin</groupId>
    <artifactId>vaadin-spring-boot-starter</artifactId>
  </dependency>
</dependencies>

<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
    <plugin>
      <groupId>com.vaadin</groupId>
      <artifactId>vaadin-maven-plugin</artifactId>
      <version>${vaadin.version}</version>
      <executions>
        <execution>
          <goals>
            <goal>prepare-frontend</goal>
            <goal>build-frontend</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

Certifique-se de substituir `#{vaadin.version}` pela versão específica do Vaadin que pretende utilizar.

4. Definir a Classe Principal

Crie uma classe principal para a sua aplicação:

```
@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

A anotação `@SpringBootApplication` ativa as configurações automáticas do Spring Boot, incluindo a configuração do Vaadin.

5. Criar Vistas com Vaadin

No Vaadin, as vistas são definidas como classes Java anotadas com `@Route`. Por exemplo:

```
@Route("")
public class MainView extends VerticalLayout {
    public MainView() {
        add(new Text("Bem-vindo ao MainView."));
    }
}
```

Esta classe define uma vista que será apresentada na raiz da aplicação.

6. Executar a Aplicação

Para iniciar a aplicação, execute o método `main` da classe principal. O Spring Boot iniciará um servidor web incorporado e a aplicação estará disponível no navegador.

7. Implementar Funcionalidades Adicionais

À medida que desenvolve a aplicação, pode adicionar mais vistas, componentes e lógica de negócio conforme necessário. O Vaadin oferece uma ampla gama de componentes de interface do utilizador que podem ser utilizados para criar interfaces ricas e interativas.

Seguindo estes passos, os alunos poderão criar e desenvolver aplicações web utilizando o Vaadin com o Spring Boot, aproveitando as vantagens de ambos os frameworks para construir interfaces de utilizador modernas e responsivas.

Para criar uma interface de utilizador (UI) com o Vaadin Flow, é fundamental compreender os componentes e layouts que o framework oferece. O Vaadin Flow permite construir UIs de forma declarativa em Java, utilizando uma vasta gama de componentes prontos para uso.

Componentes Básicos

O Vaadin fornece mais de 40 componentes de UI que facilitam a construção de aplicações. Estes componentes são elementos HTML personalizados, baseados nos padrões de Web Components do W3C, e podem ser utilizados diretamente em Java. Por exemplo, para criar um botão:

```
Button button = new Button("Clique-me!");
```

Este código cria um botão com o rótulo "Clique-me!".

Layouts

Os layouts determinam como os componentes são posicionados na janela do navegador. Os layouts mais comuns são *HorizontalLayout*, *VerticalLayout* e *Div*. Por exemplo, para adicionar componentes a um layout horizontal:

```
HorizontalLayout layout = new HorizontalLayout();  
layout.add(component1, component2);
```

Este código adiciona *component1* e *component2* ao layout horizontal, posicionando-os lado a lado.

Eventos de UI

Para adicionar funcionalidade à aplicação, é possível associar ouvintes de eventos aos componentes. Por exemplo, para adicionar um ouvinte de clique a um botão:

```
button.addClickListener(event -> {  
    // Ação a ser executada quando o botão é clicado  
});
```

Este código define uma ação a ser executada quando o botão é clicado.

Exemplo Prático: Lista de Contactos

Vamos criar uma vista que exhibe uma lista de contactos. Utilizaremos um *Grid* para mostrar os contactos e um *TextField* para filtrar a lista.

(por favor, ver página seguinte, para uma visão global do ficheiro numa única página)

```
@Route("")
@PageTitle("Contactos | Vaadin CRM")
public class ListView extends VerticalLayout {
    Grid<Contact> grid = new Grid<>(Contact.class);
    TextField filterText = new TextField();

    public ListView() {
        addClassName("list-view");
        setSizeFull();
        configureGrid();
        add(getToolbar(), grid);
    }

    private void configureGrid() {
        grid.addClassNames("contact-grid");
        grid.setSizeFull();
        grid.setColumns("firstName", "lastName", "email");
        grid.addColumn(contact ->
contact.getStatus().getName()).setHeader("Status");
        grid.addColumn(contact ->
contact.getCompany().getName()).setHeader("Empresa");
        grid.getColumns().forEach(col -> col.setAutoWidth(true));
    }

    private HorizontalLayout getToolbar() {
        filterText.setPlaceholder("Filtrar por nome...");
        filterText.setClearButtonVisible(true);
        filterText.setValueChangeMode(ValueChangeMode.LAZY);
        Button addContactButton = new Button("Adicionar contacto");
        HorizontalLayout toolbar = new HorizontalLayout(filterText,
addContactButton);
        toolbar.addClassName("toolbar");
        return toolbar;
    }
}
```

Este exemplo cria uma vista principal que exhibe uma grelha de contactos e uma barra de ferramentas com um campo de filtro e um botão para adicionar novos contactos.

Navegação e Layout Principal

Aplicações geralmente possuem uma vista principal com um menu que permite a navegação entre diferentes sub-vistas. O *AppLayout* do Vaadin facilita a criação desse layout principal, incluindo cabeçalho, menu e área de conteúdo. Por exemplo:

```
public class MainView extends AppLayout {
    private final Tabs menu;
    private H1 viewTitle;

    public MainView() {
        setPrimarySection(Section.DRAWER);
        addToNavbar(true, createHeaderContent());
        menu = createMenu();
        addToDrawer(createDrawerContent(menu));
    }

    private Component createHeaderContent() {
        HorizontalLayout layout = new HorizontalLayout();
        layout.setId("header");
        layout.setWidthFull();
        layout.setSpacing(false);
        layout.setAlignItems(FlexComponent.Alignment.CENTER);
        layout.add(new DrawerToggle());
        viewTitle = new H1();
        layout.add(viewTitle);
        layout.add(new Image("images/user.svg", "Avatar"));
        return layout;
    }

    private Tabs createMenu() {
        // Implementação do menu
    }

    private Component createDrawerContent(Tabs menu) {
        // Implementação do conteúdo do drawer
    }
}
```

Este código cria uma vista principal com um cabeçalho que inclui um botão para alternar o menu (drawer), um título de vista e um avatar de utilizador.

Seguindo estas diretrizes, irá poder construir interfaces de utilizador eficazes utilizando o Vaadin Flow com Spring Boot, aproveitando os componentes e layouts fornecidos pela framework para criar aplicações web modernas e responsivas.

Para lidar com a interação do utilizador, o Vaadin Flow utiliza programação orientada a eventos. Quando um utilizador interage com a interface no navegador, são gerados eventos que o Flow encaminha para o lado do servidor, permitindo que o código da aplicação os processe.

Implementação de Manipuladores de Eventos

Cada tipo de interação do utilizador gera um tipo específico de evento. Por exemplo, clicar num botão gera um evento `ClickEvent<Button>`. Para processar estes eventos, é necessário implementar um ouvinte que trate o evento específico. Os ouvintes de eventos podem ser implementados de várias formas:

- **Classes Locais:** Implementando a interface `ComponentEventListener` para o tipo de evento.

```
Button button = new Button("Clique-me!");
class MeuOuvinteDeClique implements
ComponentEventListener<ClickEvent<Button>> {
    int contador = 0;
    @Override
    public void onComponentEvent(ClickEvent<Button> evento) {
        evento.getSource().setText("Clicou " + (++contador) + "
vezes");
    }
}
button.addClickListener(new MeuOuvinteDeClique());
add(button);
```

- **Expressões Lambda:** Uma forma concisa de implementar ouvintes para interfaces funcionais.

```
Button button = new Button("Clique-me!", evento ->
    evento.getSource().setText("Clicado!"));
add(button);
```

- **Classes Anónimas:** Permitem implementar ouvintes de eventos sem a necessidade de classes nomeadas.

```
Button button = new Button("Clique-me!", new
ComponentEventListener<ClickEvent<Button>>() {
    int contador = 0;
    @Override
    public void onComponentEvent(ClickEvent<Button> evento) {
        evento.getSource().setText("Clicou " + (++contador) + "
vezes");
    }
});
add(button);
```

- **Referências a Métodos:** Direcionam eventos para métodos específicos.

```
class BarraDeBotoes extends HorizontalLayout {  
    public BarraDeBotoes() {  
        add(new Button("OK", this::ok));  
        add(new Button("Cancelar", this::cancelar));  
    }  
    public void ok(ClickEvent<Button> evento) {  
        evento.getSource().setText("OK!");  
    }  
    public void cancelar(ClickEvent<Button> evento) {  
        evento.getSource().setText("Cancelado!");  
    }  
}  
add(new BarraDeBotoes());
```

Estas abordagens permitem que os programadores escolham a forma mais adequada para implementar ouvintes de eventos, dependendo do contexto e das necessidades da aplicação.

Eventos Personalizados

Além dos eventos padrão fornecidos pelos componentes do Vaadin, é possível criar eventos personalizados para componentes específicos. Para isso, deve-se estender a classe *ComponentEvent* e utilizar a anotação *@DomEvent* para associar o evento a um evento DOM específico. Por exemplo:

```
@DomEvent("change")  
public class EventoDeAlteracao extends ComponentEvent<TextField> {  
    public EventoDeAlteracao(TextField source, boolean fromClient) {  
        super(source, fromClient);  
    }  
}
```

Para adicionar um ouvinte para este evento personalizado:

```
textField.addListener(EventoDeAlteracao.class, evento -> {  
    // Lógica a ser executada quando o evento ocorre  
});
```

Esta abordagem permite a criação de componentes altamente personalizados que podem responder a eventos específicos definidos pelo programador.

Filtragem e Debouncing de Eventos

Em situações onde eventos são disparados com alta frequência, como durante a digitação num campo de texto, é importante controlar a taxa de eventos enviados para o servidor. O Vaadin permite configurar filtros e definir configurações de "debounce" para limitar a frequência de eventos processados. Por exemplo:

```
@DomEvent(value = "input",
            debounce = @DebounceSettings(timeout = 500,
                                         phases = DebouncePhase.TRAILING))
public class EventoDeInput extends ComponentEvent<TextField> {
    private String valor;
    public EventoDeInput(TextField source, boolean fromClient,
                        @EventData("element.value") String valor) {
        super(source, fromClient);
        this.valor = valor;
    }
    public String getValor() {
        return valor;
    }
}
```

Neste exemplo, o evento *EventoDeInput* é enviado para o servidor 500 milissegundos após o último input do utilizador, evitando sobrecarga no processamento de eventos.

Ao compreender e aplicar estes conceitos, poderá desenvolver aplicações web interativas e responsivas utilizando o Vaadin Flow com Spring Boot, aproveitando os recursos de manipulação de eventos fornecidos pelo framework.

Para criar uma vista principal numa aplicação Vaadin Flow com Spring Boot, é comum utilizar o componente *AppLayout*. Este componente permite estruturar a aplicação com um cabeçalho, um menu de navegação e uma área de conteúdo, facilitando a navegação entre diferentes sub-vistas.

Passos para Implementar a Vista Principal:

1. **Definir a Classe da Vista Principal:** Crie uma classe que estende *AppLayout*. Esta classe servirá como a estrutura principal da aplicação.

```
@Route("")
public class MainView extends AppLayout {
    private final Tabs menu;
    private H1 viewTitle;

    public MainView() {
        setPrimarySection(Section.DRAWER);
        addToNavbar(true, createHeaderContent());
        menu = createMenu();
        addToDrawer(createDrawerContent(menu));
    }
}
```

Neste exemplo, a anotação `@Route("")` define que esta é a vista padrão da aplicação, acessível na raiz do URL.

2. **Criar o Conteúdo do Cabeçalho:** Implemente um método para criar o conteúdo do cabeçalho, incluindo elementos como o título da vista e um botão para alternar a visibilidade do menu lateral.

```
private Component createHeaderContent() {
    HorizontalLayout layout = new HorizontalLayout();
    layout.setId("header");
    layout.setWidthFull();
    layout.setSpacing(false);
    layout.setAlignItems(FlexComponent.Alignment.CENTER);
    layout.add(new DrawerToggle());
    viewTitle = new H1();
    layout.add(viewTitle);
    layout.add(new Image("images/user.svg", "Avatar"));
    return layout;
}
```

Este método cria um layout horizontal para o cabeçalho, incluindo um botão de alternância (*DrawerToggle*), um título (*H1*) e uma imagem de avatar.

3. **Configurar o Menu de Navegação:** Implemente um método para criar o menu de navegação, utilizando componentes como *Tabs* e *Tab* para representar as diferentes vistas da aplicação.

```
private Tabs createMenu() {
    final Tabs tabs = new Tabs();
    tabs.setOrientation(Tabs.Orientation.VERTICAL);
    tabs.add(createMenuItems());
    return tabs;
}

private Component[] createMenuItems() {
    return new Tab[]{
        createTab("Início", HomeView.class),
        createTab("Sobre", AboutView.class),
        createTab("Contacto", ContactView.class)
    };
}

private Tab createTab(String text, Class<? extends Component>
navigationTarget) {
    final Tab tab = new Tab();
    RouterLink link = new RouterLink(text, navigationTarget);
    link.setTabIndex(-1);
    tab.add(link);
    return tab;
}
```

Este conjunto de métodos cria um menu vertical com itens que permitem a navegação para diferentes vistas, como "Início", "Sobre" e "Contacto".

4. **Definir o Conteúdo do Menu Lateral (Drawer):** Implemente um método para criar o conteúdo do menu lateral, que incluirá o menu de navegação criado anteriormente.

```
private Component createDrawerContent(Tabs menu) {
    VerticalLayout layout = new VerticalLayout();
    layout.setSizeFull();
    layout.add(menu);
    return layout;
}
```

Este método adiciona o menu de navegação a um layout vertical que ocupa toda a altura disponível no menu lateral.

5. **Atualizar o Título da Vista:** Implemente um método para atualizar o título da vista com base na navegação atual.

```
private void updateViewTitle(String title) {  
    viewTitle.setText(title);  
}
```

Este método permite que o título exibido no cabeçalho seja atualizado conforme o utilizador navega entre diferentes vistas.

Seguindo estes passos, é possível criar uma estrutura de aplicação organizada, com um cabeçalho, menu de navegação e área de conteúdo, proporcionando uma experiência de utilizador intuitiva e eficiente.

Para implementar a navegação e o roteamento numa aplicação Vaadin Flow com Spring Boot, é essencial compreender como mapear vistas para URLs específicos e gerir a navegação entre elas. O Vaadin Flow facilita este processo através de anotações e componentes dedicados.

Definição de Rotas com `@Route`

No Vaadin, cada vista é uma classe que representa uma parte lógica da interface do utilizador. Para mapear uma vista a um URL específico, utiliza-se a anotação `@Route`. Por exemplo:

```
@Route("home")
public class HomeView extends VerticalLayout {
    public HomeView() {
        add(new Text("Bem-vindo à HomeView!"));
    }
}
```

Neste exemplo, ao navegar para `http://localhost:8080/home`, a aplicação exibirá o conteúdo definido em `HomeView`. Se a anotação `@Route` for utilizada sem parâmetros, a vista será associada à raiz do URL.

Navegação entre Vistas

A navegação entre vistas pode ser realizada de várias formas:

- **Utilizando o Componente `RouterLink`:** Este componente cria um link que aponta para uma rota específica.

```
RouterLink link = new RouterLink("Ir para Home", HomeView.class);
add(link);
```

Ao clicar no link, o utilizador será direcionado para a vista `HomeView` sem recarregar a página, e o URL no navegador será atualizado.

- **Através do Método `UI.navigate`:** Este método permite navegar programaticamente para uma rota específica.

```
Button button = new Button("Ir para Home", event ->
    UI.getCurrent().navigate("home"));
add(button);
```

Ao clicar no botão, a aplicação navegará para a rota "home".

Parâmetros de Rota

Para passar parâmetros através da URL, pode-se definir rotas com parâmetros. Por exemplo:

```
@Route("user/:userID")
public class UserView extends VerticalLayout implements BeforeEnterObserver
{
    @Override
    public void beforeEnter(BeforeEnterEvent event) {
        String userId =
event.getRouteParameters().get("userID").orElse("Desconhecido");
        add(new Text("ID do Utilizador: " + userId));
    }
}
```

Ao navegar para `http://localhost:8080/user/123`, a vista exibirá "ID do Utilizador: 123".

Layouts Aninhados e Vistas Principais

Para criar uma estrutura de navegação mais complexa, como uma aplicação com um menu e várias sub-vistas, pode-se utilizar layouts aninhados. Por exemplo:

```
@Route(value = "main", layout = MainLayout.class)
public class MainView extends VerticalLayout {
    public MainView() {
        add(new Text("Esta é a vista principal."));
    }
}
```

Neste caso, `MainLayout` é uma classe que define o layout comum (como um menu ou cabeçalho), e `MainView` é uma sub-vista que será renderizada dentro desse layout.

Ciclo de Vida da Navegação

O Vaadin fornece interfaces como `BeforeEnterObserver` e `AfterNavigationObserver` que permitem executar lógica antes ou depois da navegação para uma vista. Por exemplo:

```
@Route("secure")
public class SecureView extends VerticalLayout implements
BeforeEnterObserver {
    @Override
    public void beforeEnter(BeforeEnterEvent event) {
        if (!isUserAuthenticated()) {
            event.forwardTo("login");
        }
    }
}
```


Este exemplo verifica se o utilizador está autenticado antes de permitir o acesso à vista "secure". Se não estiver, redireciona-o para a página de login.

Compreendendo e aplicando estes conceitos, os alunos poderão criar aplicações web estruturadas e navegáveis utilizando o Vaadin Flow com Spring Boot, proporcionando uma experiência de utilizador fluida e intuitiva.

Para gerir recursos estáticos numa aplicação Vaadin Flow com Spring Boot, é fundamental compreender como armazenar e aceder a ficheiros como imagens, folhas de estilo CSS e scripts JavaScript. A estrutura do projeto e a localização dos recursos variam ligeiramente entre projetos Spring e não Spring.

Localização dos Recursos em Projetos Spring Boot

Em projetos que utilizam Spring Boot, os recursos estáticos devem ser colocados em diretórios específicos para serem servidos corretamente pela aplicação. A tabela abaixo resume onde colocar diferentes tipos de recursos:

Tipo de Ficheiro	Localização do Ficheiro no Projeto
Ficheiros CSS	<code>/src/main/frontend/my-styles/</code>
JavaScript, TypeScript e Lit templates	<code>/src/main/frontend/src/</code>
Ficheiros estáticos (imagens, ícones, documentos)	<code>/src/main/resources/META-INF/resources/</code>

Por exemplo, para adicionar uma imagem chamada *flower.jpg*, deve colocá-la em:

`/src/main/resources/META-INF/resources/img/flower.jpg`

Desta forma, a imagem estará acessível na aplicação através do caminho `/img/flower.jpg`.

Importação de Recursos no Código

Para utilizar estes recursos na aplicação, é necessário importá-los ou referenciá-los adequadamente no código Java:

- **Ficheiros CSS:** Utilize a anotação `@CssImport` para importar folhas de estilo.
`@CssImport("./my-styles/styles.css")`
- **Ficheiros JavaScript ou TypeScript:** Utilize a anotação `@JsModule` para importar scripts.
`@JsModule("./src/my-script.js")`
- **Ficheiros Estáticos (Imagens, Documentos):** Utilize componentes como *Image* ou *Anchor* para referenciar recursos estáticos.

`Image image = new Image("img/flower.jpg", "A flower");`

Ao seguir estas diretrizes, os recursos serão corretamente servidos e integrados na aplicação Vaadin com Spring Boot.

Carregamento Programático de Recursos

Para carregar recursos de forma programática, como imagens ou documentos, pode-se utilizar a classe *StreamResource* em conjunto com um *InputStreamFactory*. Isto é útil quando o conteúdo é dinâmico ou gerado em tempo real.

```
StreamResource resource = new StreamResource("flower.jpg", () ->  
getClass().getResourceAsStream("/META-INF/resources/img/flower.jpg"));  
Image image = new Image(resource, "A flower");
```

Este método permite carregar recursos diretamente do classpath da aplicação.

Considerações Finais

Ao estruturar corretamente os recursos estáticos e referenciá-los adequadamente no código, garante-se que a aplicação Vaadin Flow com Spring Boot serve e utiliza esses recursos de forma eficiente. Para mais detalhes e exemplos, consulte a [documentação oficial do Vaadin](#).

Para implementar a navegação entre vistas numa aplicação Vaadin Flow com Spring Boot, é fundamental compreender como definir rotas e gerir a transição entre elas. O Vaadin Flow oferece mecanismos para facilitar este processo, permitindo uma navegação fluida e intuitiva.

Definição de Rotas com `@Route`

Cada vista na aplicação é representada por uma classe que estende um layout, como `VerticalLayout`. Para mapear uma vista a um URL específico, utiliza-se a anotação `@Route`. Por exemplo:

```
@Route("home")
public class HomeView extends VerticalLayout {
    public HomeView() {
        add(new Text("Bem-vindo à HomeView!"));
    }
}
```

Neste exemplo, ao navegar para `http://localhost:8080/home`, o utilizador verá o conteúdo definido em `HomeView`. Se a anotação `@Route` for utilizada sem parâmetros, a vista será associada à raiz do URL.

Navegação entre Vistas

A navegação entre vistas pode ser realizada de várias formas:

- **Utilizando o Componente `RouterLink`:** Este componente cria um link que aponta para uma rota específica.

```
RouterLink link = new RouterLink("Ir para Home", HomeView.class);
add(link);
```

Ao clicar no link, o utilizador será direcionado para a vista `HomeView` sem recarregar a página, e o URL no navegador será atualizado.

- **Através do Método `UI.navigate`:** Este método permite navegar programaticamente para uma rota específica.

```
Button button = new Button("Ir para Home", event ->
    UI.getCurrent().navigate("home"));
add(button);
```

Ao clicar no botão, a aplicação navegará para a rota "home".

Parâmetros de Rota

Para passar parâmetros através da URL, pode-se definir rotas com parâmetros. Por exemplo:

```
@Route("user/{userID}")
public class UserView extends VerticalLayout implements BeforeEnterObserver
{
    @Override
```

```
public void beforeEnter(BeforeEnterEvent event) {  
    String userId =  
event.getRouteParameters().get("userID").orElse("Desconhecido");  
    add(new Text("ID do Utilizador: " + userId));  
}  
}
```

Ao navegar para `http://localhost:8080/user/123`, a vista exibirá "ID do Utilizador: 123".

Layouts Aninhados e Vistas Principais

Para criar uma estrutura de navegação mais complexa, como uma aplicação com um menu e várias sub-vistas, pode-se utilizar layouts aninhados. Por exemplo:

```
@Route(value = "main", layout = MainLayout.class)  
public class MainView extends VerticalLayout {  
    public MainView() {  
        add(new Text("Esta é a vista principal."));  
    }  
}
```

Neste caso, *MainLayout* é uma classe que define o layout comum (como um menu ou cabeçalho), e *MainView* é

Para implementar parâmetros de rota numa aplicação Vaadin Flow com Spring Boot, é fundamental compreender como definir rotas que aceitam variáveis na URL e como processar esses parâmetros nas vistas correspondentes. Os parâmetros de rota permitem que uma aplicação responda dinamicamente a diferentes entradas fornecidas diretamente na URL.

Definição de Parâmetros de Rota

No Vaadin Flow, os parâmetros de rota são definidos diretamente na anotação `@Route` ou através da implementação da interface `HasUrlParameter<T>`. Por exemplo, para criar uma rota que aceite um parâmetro de string:

```
@Route("greet")
public class GreetingComponent extends Div implements
HasUrlParameter<String> {
    @Override
    public void setParameter(BeforeEvent event, String parameter) {
        setText(String.format("Olá, %s!", parameter));
    }
}
```

Neste exemplo, ao navegar para `http://localhost:8080/greet/João`, a aplicação exibirá "Olá, João!". A interface `HasUrlParameter<String>` permite que o método `setParameter` processe o parâmetro fornecido na URL.

Parâmetros Opcionais

Para definir parâmetros de rota opcionais, utiliza-se a anotação `@OptionalParameter` no método `setParameter`:

```
@Route("greet")
public class OptionalGreeting extends Div implements HasUrlParameter<String>
{
    @Override
    public void setParameter(BeforeEvent event, @OptionalParameter String
parameter) {
        if (parameter == null) {
            setText("Bem-vindo, anónimo.");
        } else {
            setText(String.format("Bem-vindo, %s.", parameter));
        }
    }
}
```

Com esta configuração, tanto `http://localhost:8080/greet` quanto `http://localhost:8080/greet/Maria` são válidos, exibindo mensagens apropriadas conforme o parâmetro fornecido ou a ausência dele.

Parâmetros Múltiplos e Padrões Personalizados

Para rotas que requerem múltiplos parâmetros ou padrões específicos, é possível definir templates de rota diretamente na anotação `@Route`:

```
@Route("product/:identifier/:category?")
public class ProductView extends Div implements BeforeEnterObserver {
    @Override
    public void beforeEnter(BeforeEnterEvent event) {
        RouteParameters parameters = event.getRouteParameters();
        String identifier =
parameters.get("identifier").orElse("Desconhecido");
        String category = parameters.get("category").orElse("Geral");
        setText(String.format("Produto: %s, Categoria: %s", identifier,
category));
    }
}
```

Neste exemplo, ao navegar para `http://localhost:8080/product/123/Electronics`, a aplicação exibirá "Produto: 123, Categoria: Electronics". Se a categoria for omitida, como em `http://localhost:8080/product/123`, a categoria padrão "Geral" será utilizada.

Considerações Importantes

- **Precedência de Rotas:** Rotas específicas têm precedência sobre rotas com parâmetros. Por exemplo, uma rota definida como `@Route("user/new")` será resolvida antes de uma rota `@Route("user/:id")` ao navegar para `/user/new`.
- **Tipos de Parâmetros:** A interface `HasURLParameter<T>` suporta tipos como `String`, `Integer`, `Long` e `Boolean`. Para outros tipos ou padrões mais complexos, é necessário implementar lógica adicional para conversão e validação dos parâmetros.

Ao dominar a utilização de parâmetros de rota no Vaadin Flow, é possível criar aplicações web mais dinâmicas e responsivas, permitindo que diferentes URLs forneçam informações específicas às vistas correspondentes.

Para manter a integridade dos links diretos (deep linking) numa aplicação Vaadin Flow com Spring Boot, é por vezes necessário atualizar os parâmetros da URL sem efetuar uma navegação completa. Isto é particularmente útil em cenários onde o estado da aplicação deve refletir-se na URL, permitindo que os utilizadores partilhem ou guardem links que reproduzam o estado atual da interface.

Atualização de Parâmetros de URL sem Navegação

O Vaadin Flow permite a atualização dos parâmetros da URL sem recarregar a página, utilizando a API de Histórico do navegador. Ao manipular o estado do histórico, é possível modificar a URL apresentada ao utilizador sem desencadear uma navegação ou recarregamento da página.

(por favor, ver página seguinte, para uma visão global do ficheiro numa única página)

Exemplo Prático:

Considere uma vista que edita informações de um utilizador. Ao seleccionar um utilizador para edição, pretende-se que a URL reflita essa seleção, permitindo que o link seja partilhado ou guardado para acesso direto posterior.

```
@Route("user")
public class UserEditor extends VerticalLayout implements
HasUrlParameter<Integer> {

    public void editUser(User user) {
        // Lógica para editar o utilizador
        createFormForUser(user);
        // Atualizar os parâmetros da URL
        updateQueryParameters(user);
    }

    private void updateQueryParameters(User user) {
        String deepLinkingUrl = RouteConfiguration.forSessionScope()
            .getUrl(UserEditor.class, user.getId());
        // Atualiza a URL no navegador sem recarregar a página
        getUI().ifPresent(ui -> ui.getPage().getHistory()
            .replaceState(null, deepLinkingUrl));
    }

    @Override
    public void setParameter(BeforeEvent event, @OptionalParameter Integer
id) {
        if (id != null) {
            // Lógica para carregar o utilizador com o ID fornecido
            createFormForUser(new User("Utilizador " + id, id));
        }
    }

    private void createFormForUser(User user) {
        // Implementação do formulário de edição do utilizador
        add(new Paragraph("Utilizador: " + user.getId()));
    }
}
```

Neste exemplo, ao chamar o método *editUser*, a aplicação atualiza a interface para refletir o utilizador selecionado e modifica a URL no navegador para incluir o ID do utilizador. A utilização de *getPage().getHistory().replaceState(null, deepLinkingUrl)* permite alterar a URL sem recarregar a página, mantendo a experiência do utilizador fluida.

Considerações Importantes:

- **Manutenção do Estado da Aplicação:** Ao atualizar a URL para refletir o estado atual, garante-se que os utilizadores podem partilhar ou guardar links que reproduzem exatamente o mesmo estado da interface quando revisitados.
- **Utilização da API de Histórico:** A manipulação do histórico do navegador através de *replacState* ou *pushState* permite modificar a URL sem desencadear uma navegação completa, preservando o estado atual da aplicação.
- **Implementação de *HasUrlParameter*:** Ao implementar a interface *HasUrlParameter*, a vista pode processar parâmetros da URL quando o utilizador acede diretamente através de um link, assegurando que o estado correto é restaurado.

Ao aplicar estas práticas, a aplicação torna-se mais robusta e amigável, proporcionando uma experiência de utilizador consistente e permitindo a partilha eficaz de links diretos para estados específicos da aplicação.

No desenvolvimento de aplicações web com Vaadin Flow e Spring Boot, é crucial compreender o ciclo de vida da navegação para garantir uma experiência de utilizador fluida e segura. O Vaadin Flow oferece eventos específicos que permitem interagir com o processo de navegação, possibilitando a execução de ações antes e depois da transição entre vistas.

Eventos do Ciclo de Vida da Navegação

1. **BeforeLeaveEvent:** Este evento é disparado antes de a navegação sair da vista atual. Implementando a interface *BeforeLeaveObserver*, é possível controlar ou impedir a saída da vista, útil para verificar alterações não guardadas ou confirmar ações do utilizador.

```
@Route("edit")
public class EditView extends VerticalLayout implements
BeforeLeaveObserver {
    @Override
    public void beforeLeave(BeforeLeaveEvent event) {
        if (hasUnsavedChanges()) {
            // Lógica para notificar o utilizador ou cancelar a
            navegação
            event.postpone();
        }
    }
}
```

2. **BeforeEnterEvent:** Disparado antes de entrar numa nova vista, permite verificar permissões, redirecionar ou modificar o destino da navegação. Implementando a interface *BeforeEnterObserver*, pode-se controlar o acesso à vista.

```
@Route("admin")
public class AdminView extends VerticalLayout implements
BeforeEnterObserver {
    @Override
    public void beforeEnter(BeforeEnterEvent event) {
        if (!isUserAuthorized()) {
            event.forwardTo("access-denied");
        }
    }
}
```

3. **AfterNavigationEvent**: Este evento ocorre após a conclusão da navegação, sendo ideal para atualizar componentes da interface ou executar ações que dependem da vista atual. Implementando a interface *AfterNavigationObserver*, é possível reagir a mudanças na navegação.

```
@Route("dashboard")
public class DashboardView extends VerticalLayout implements
AfterNavigationObserver {
    @Override
    public void afterNavigation(AfterNavigationEvent event) {
        // Atualizar dados ou componentes após a navegação
        refreshData();
    }
}
```

Redirecionamento e Encaminhamento durante a Navegação

Durante os eventos de navegação, é possível alterar o destino utilizando métodos como *rerouteTo()* e *forwardTo()*.

- **rerouteTo()**: Altera o destino da navegação sem modificar a URL no navegador. Útil para redirecionamentos internos onde a URL visível deve permanecer inalterada.

```
@Override
public void beforeEnter(BeforeEnterEvent event) {
    if (someCondition()) {
        event.rerouteTo("alternative-view");
    }
}
```

- **forwardTo()**: Encaminha a navegação para uma nova vista e atualiza a URL no navegador. Adequado quando é necessário refletir a nova vista na URL.

```
@Override
public void beforeEnter(BeforeEnterEvent event) {
    if (anotherCondition()) {
        event.forwardTo("new-view");
    }
}
```

Compreender e utilizar estes eventos permite um controlo refinado sobre o fluxo de navegação na aplicação, assegurando que os utilizadores têm acesso adequado às vistas e que a interface responde de forma previsível às suas ações.

Para mais detalhes, consulte a [documentação oficial do Vaadin sobre o ciclo de vida da navegação](#).

No desenvolvimento de aplicações web com Vaadin Flow e Spring Boot, é essencial implementar um tratamento eficaz de exceções de roteamento para garantir uma experiência de utilizador robusta e informativa. O Vaadin Flow oferece mecanismos que permitem lidar com exceções não tratadas durante a navegação, apresentando vistas de erro apropriadas aos utilizadores.

Tratamento de Exceções de Roteamento

Quando ocorre uma exceção não tratada durante a navegação, o Vaadin Flow procura uma classe que implemente a interface *HasErrorParameter<T extends Exception>* correspondente ao tipo de exceção lançada. Estas classes atuam como alvos de exceção, funcionando de maneira semelhante aos alvos de navegação regulares, mas geralmente não possuem uma anotação *@Route* específica, pois são exibidas para URLs arbitrárias.

Exemplo de Implementação:

Considere uma exceção personalizada *AccessDeniedException* que é lançada quando um utilizador tenta aceder a uma área restrita da aplicação. Para tratar esta exceção e fornecer uma resposta adequada ao utilizador, pode-se criar uma classe que implemente *HasErrorParameter<AccessDeniedException>*:

```
@Tag(Tag.DIV)
@ParentLayout(MainLayout.class)
public class AccessDeniedExceptionHandler extends Component implements
HasErrorParameter<AccessDeniedException> {

    @Override
    public int setErrorParameter(BeforeEnterEvent event,
ErrorParameter<AccessDeniedException> parameter) {
        getElement().setText("Acesso negado: não tem permissões suficientes
para aceder a esta área.");
        return HttpServletResponse.SC_FORBIDDEN;
    }
}
```

Neste exemplo, quando uma *AccessDeniedException* é lançada durante a navegação, o utilizador é direcionado para uma vista que exibe a mensagem "Acesso negado: não tem permissões suficientes para aceder a esta área." e o código de resposta HTTP 403 (Forbidden) é retornado.

Redirecionamento para Vistas de Erro

Durante os eventos de navegação, como *BeforeEnterEvent* ou *BeforeLeaveEvent*, é possível redirecionar para uma vista de erro registada para uma exceção específica utilizando o método *rerouteToError()*. Por exemplo:

```
public class AuthenticationHandler implements BeforeEnterObserver {

    @Override
    public void beforeEnter(BeforeEnterEvent event) {
        Class<?> target = event.getNavigationTarget();
        if (!currentUserMayEnter(target)) {
            event.rerouteToError(AccessDeniedException.class);
        }
    }

    private boolean currentUserMayEnter(Class<?> target) {
        // Implementação da lógica de verificação de permissões
        return false;
    }
}
```

Neste cenário, se o utilizador não tiver permissão para aceder à vista de destino, a navegação é redirecionada para a vista de erro associada à *AccessDeniedException*.

Considerações Importantes:

- **Ordem de Correspondência de Exceções:** As exceções são correspondidas primeiro pela causa e depois pelo supertipo da exceção. O Vaadin Flow implementa por defeito o *RouteNotFoundError* para *NotFoundException* (erro 404) e o *InternalServerError* para *Exception* (erro 500).
- **Personalização de Vistas de Erro:** É possível substituir os manipuladores de exceção padrão estendendo-os e definindo layouts parentais conforme necessário. As vistas de erro podem definir *ParentLayouts* para manter a consistência da interface.
- **Eventos de Navegação:** Os eventos *BeforeNavigationEvent* e *AfterNavigationEvent* são disparados durante a navegação normal e também quando ocorre uma exceção, permitindo a execução de lógica adicional conforme necessário.

Ao implementar um tratamento de exceções de roteamento eficaz, assegura-se que a aplicação responde de forma adequada a erros, fornecendo feedback claro aos utilizadores e mantendo a integridade da experiência de navegação.

Para mais detalhes, consulte a [documentação oficial do Vaadin sobre tratamento de exceções de roteamento](#).

Para obter todas as rotas registadas numa aplicação Vaadin Flow com Spring Boot, pode utilizar a classe *RouteConfiguration*. Esta classe fornece métodos que permitem aceder às rotas disponíveis, facilitando a construção de menus de navegação dinâmicos ou outras funcionalidades que dependam das rotas definidas na aplicação.

Obtenção de Todas as Rotas Registadas

Para recuperar uma lista de todas as rotas registadas, utilize o método *getAvailableRoutes()* da seguinte forma:

```
List<RouteData> rotas =  
RouteConfiguration.forSessionScope().getAvailableRoutes();
```

O objeto *RouteData* contém informações relevantes sobre cada rota definida, como o template da rota, parâmetros e layouts parentais.

Filtragem de Rotas por Layout Parental

Se pretender obter apenas as rotas associadas a um layout parental específico, pode filtrar a lista obtida:

```
List<RouteData> rotas =  
RouteConfiguration.forSessionScope().getAvailableRoutes();  
List<RouteData> rotasDoMeuLayout = rotas.stream()  
    .filter(routeData ->  
        MeuLayoutParental.class.equals(routeData.getParentLayout()))  
    .collect(Collectors.toList());
```

Este código filtra as rotas cuja classe de layout parental corresponde a *MeuLayoutParental*.

Considerações Importantes

- **Escopo da Sessão:** O método *forSessionScope()* obtém a configuração de rotas para a sessão atual. Se necessitar de obter rotas num contexto diferente, como a nível de aplicação, utilize o método *forApplicationScope()*.
- **Construção de Menus Dinâmicos:** Ao obter a lista de rotas registadas, pode construir menus de navegação que refletem dinamicamente as vistas disponíveis na aplicação, melhorando a experiência do utilizador.

Para mais detalhes, consulte a [documentação oficial do Vaadin sobre obtenção de rotas registadas](#).

Para definir o título da página durante a navegação numa aplicação Vaadin Flow com Spring Boot, existem duas abordagens principais: utilizar a anotação `@PageTitle` ou implementar a interface `HasDynamicTitle`. Estas abordagens são mutuamente exclusivas; utilizar ambas na mesma classe resultará numa exceção em tempo de execução.

Utilização da Anotação `@PageTitle`

A forma mais simples de definir o título da página é através da anotação `@PageTitle` na classe do componente:

```
@PageTitle("Página Inicial")
@Route("inicio")
public class PaginaInicialView extends Div {
    public PaginaInicialView() {
        setText("Bem-vindo à Página Inicial");
    }
}
```

Neste exemplo, ao navegar para a rota `"inicio"`, o título da página será definido como `"Página Inicial"`. A anotação `@PageTitle` é lida apenas a partir do alvo de navegação atual; superclasses e vistas parentais não são consideradas.

Definição Dinâmica do Título da Página

Como alternativa, é possível definir o título da página em tempo de execução implementando a interface `HasDynamicTitle`:

```
@Route("artigo")
public class ArtigoView extends Div implements HasDynamicTitle,
HasUrlParameter<Long> {
    private String titulo = "";

    @Override
    public String getPageTitle() {
        return titulo;
    }

    @Override
    public void setParameter(BeforeEvent event, Long parameter) {
        if (parameter != null) {
            titulo = "Artigo #" + parameter;
        } else {
            titulo = "Lista de Artigos";
        }
    }
}
```


Neste caso, o título da página é definido dinamicamente com base no parâmetro da URL. Por exemplo, ao navegar para `/artigo/1`, o título será "Artigo #1".

Considerações Importantes

- **Exclusividade das Abordagens:** Não combine `@PageTitle` e `HasDynamicTitle` na mesma classe, pois isso causará uma exceção em tempo de execução.
- **Herança de Títulos:** A anotação `@PageTitle` não é herdada de superclasses ou layouts parentais; deve ser definida diretamente na classe do componente de navegação.

Ao aplicar estas práticas, assegura-se que o título da página reflete corretamente o conteúdo apresentado, melhorando a experiência do utilizador e a otimização para motores de busca (SEO).

Continua, em construção

Glossário de termos

Glossário de Termos

API (Application Programming Interface)

Um conjunto de definições e protocolos que permite a comunicação entre diferentes softwares.

Array

Uma estrutura de dados que armazena múltiplos valores do mesmo tipo, acessíveis através de índices numéricos.

ArrayList

Uma implementação da classe List em Java, usada para armazenar uma coleção de elementos de tamanho variável.

CamelCase

Uma convenção de nomenclatura em que cada palavra subsequente inicia com letra maiúscula, utilizada frequentemente para nomes de variáveis e métodos (ex.: minhaVariavel).

Classe

Uma definição ou modelo que especifica os atributos (dados) e métodos (comportamentos) de objetos.

Construtor

Um método especial usado para inicializar objetos de uma classe, chamado automaticamente ao criar um objeto.

Controlador (Controller)

Um componente no padrão MVC responsável por gerir as interações do utilizador e a lógica que conecta o Modelo e a Vista.

Encapsulamento

Um princípio da programação orientada a objetos que visa restringir o acesso direto aos atributos de uma classe, oferecendo métodos específicos para leitura e modificação (getters e setters).

Framework

Um conjunto de ferramentas e bibliotecas que fornece uma base estruturada para o desenvolvimento de software. Exemplos: Spring Boot e Vaadin.

Herança

Um mecanismo da programação orientada a objetos que permite que uma classe (subclasse) herde atributos e métodos de outra classe (superclasse).

Interface

Um tipo em Java que define um conjunto de métodos que uma classe deve implementar. As interfaces permitem criar código mais flexível e reutilizável.

Iteração

O processo de repetir um conjunto de instruções utilizando estruturas como for, while e do-while.

Método

Um bloco de código que realiza uma tarefa específica e pode ser invocado por outros métodos ou classes.

Modelo (Model)

A camada do padrão MVC que gere os dados, regras de negócio e a lógica da aplicação.

MVC (Model-View-Controller)

Um padrão de arquitetura de software que separa a aplicação em três componentes principais: o Modelo (dados e lógica), a Vista (interface do utilizador) e o Controlador (gestão de interações).

Objeto

Uma instância de uma classe que representa uma entidade no mundo real com estado (atributos) e comportamento (métodos).

PascalCase

Uma convenção de nomenclatura em que todas as palavras começam com letra maiúscula, utilizada comumente para nomes de classes e interfaces (ex.: MinhaClasse ou MinhaInterface).

Polimorfismo

A capacidade de usar uma interface comum para interagir com objetos de diferentes classes. Em Java, isto é frequentemente implementado por herança ou interfaces.

REST (Representational State Transfer)

Um padrão de arquitetura amplamente utilizado para a criação de APIs baseadas em HTTP.

Sobrecarga de Métodos

A definição de múltiplos métodos com o mesmo nome numa classe, mas com diferentes assinaturas (quantidade ou tipos de parâmetros).

Spring Boot

Uma plataforma baseada no framework Spring que facilita o desenvolvimento de aplicações Java, com configuração mínima e suporte integrado para diversas funcionalidades, como segurança, persistência e APIs REST.

String

Uma classe em Java usada para manipular cadeias de caracteres.

Tipo Primitivo

Um tipo básico de dados em Java, como int, double, char e boolean.

Vaadin

Um framework Java para o desenvolvimento de interfaces web modernas e interativas, utilizado em conjunto com o Spring Boot no manual.

Variável

Um espaço na memória usado para armazenar valores durante a execução do programa.

Vista (View)

A camada do padrão MVC responsável pela interface do utilizador, exibindo dados e capturando interações.

YAML (YAML Ain't Markup Language)

Um formato de ficheiro utilizado frequentemente nas configurações do Spring Boot para definir propriedades da aplicação de forma clara e organizada.

Bibliografia

Coelho, P. (2016). *Programação em Java: Curso completo* (5.^a ed. atualizada). FCA.

Martins, F. M. (2017). *Java 8: POO + construções funcionais*. FCA.

Marcelino, M. J., & Mendes, A. J. (2016). *Fundamentos de programação em Java* (4.^a ed. atualizada e aumentada). FCA.

Turnquist, G. L. (2023). *Learning Spring Boot 3.0: Build Modern, Cloud-Native, and Distributed Systems Using Spring Boot* (3rd ed.). Packt Publishing.

Continua, em construção