ENTIDADE FORMADORA CERTIFICADA
INSTITUTO DO EMPREGO E FORMAÇÃO PROFISSIONAL, IP
PRR Plano de Recuperação e Resiliência
REPÚBLICA PORTUGUESA
Financiado pela União Europeia NextGenerationEU

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

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-nésimo 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 caracter

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":

Ná página seguinte, 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;
}
```

Por exemplo, para 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%.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:

java

```
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 parenteses 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;

    }

}
```

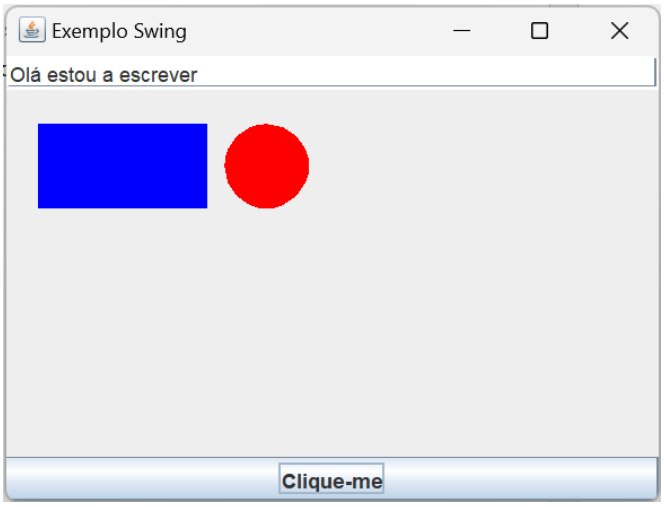
Neste exemplo, a classe `Carro` estende a classe `Veiculo`. A classe `Veiculo` possui um atributo chamado `marca` e um construtor com parâmetros. A classe `Carro` possui um atributo chamado `portas` e um construtor com parâmetros. No construtor da classe `Carro`, a chamada `super(marca)` é usada para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

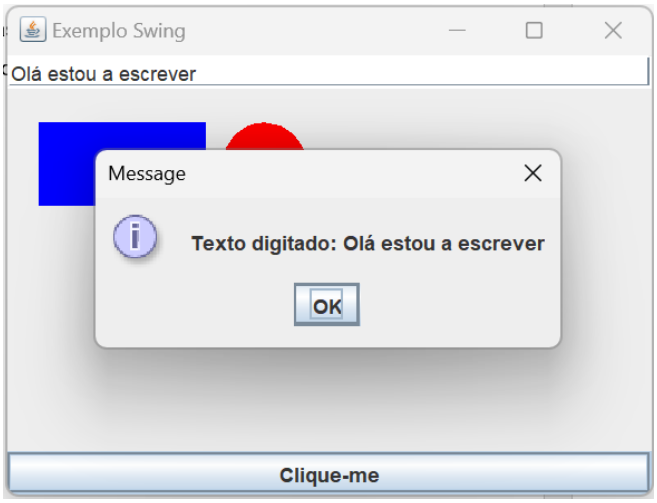
Gráficos: uso das bibliotecas Swing e AWT

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 acedem a recursos partilhados, 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

Em construção

Glossário de termos

Bibliografia