



**UNIVERSIDADE REGIONAL DE BLUMENAU - FURB**

**CIÊNCIA DA COMPUTAÇÃO**

**ALANA CRISTINA ANDREAZZA**

**LETICIA FRUET**

**PEDRO ALBUQUERQUE**

# **RELATÓRIO DE PROGRAMAÇÃO ORIENTADA A OBJETOS EM JAVA**

**BLUMENAU  
2023**

## **1. INTRODUÇÃO**

Entende-se programação processual por escrita de métodos que executam operações nos dados, enquanto a Programação Orientada a Objetos (POO) refere-se à criação de objetos que contenham dados e métodos.

A POO promove severas vantagens tanto aos usuários, quanto aos desenvolvedores. Descreve-se algumas delas abaixo.

- Fácil e rápida execução;
- Estrutura limpa para os códigos;
- Promove a reutilização de códigos, evitando assim, a repetição dos mesmos, possibilitando maior facilidade na manutenção e modificação e menor tempo de programação.

Este trabalho baseia-se na linguagem de programação JAVA e consiste na exposição de conceitos básicos de POO e utiliza imagens ilustrativas criadas pelos alunos como forma de exemplificar na prática os termos abordados para facilitar a compreensão dos mesmos.

## **2. DESENVOLVIMENTO**

### **CLASSES**

Como o próprio nome indica, a Programação Orientada a Objetos (POO) é baseada em objetos. Para criá-los, é fundamental saber a qual classe o objeto pertence, portanto, as classes são essenciais. Elas são estruturas capazes de receber diferentes tipos de variáveis (int, double, char...) e concentrar vários objetos que tenham características comuns.

As classes definem as características dos objetos e os possíveis comportamentos dos mesmos. Uma classe “Pessoa”, por exemplo, pode ter características de diferentes tipos, como nome (String), altura (double), idade (int) e também diversos comportamentos, como comer, andar e respirar. Na figura 1 é possível observar a criação de uma classe com algumas características e comportamentos:

```

public class Pessoa {

    // definindo as características (atributos)
    private String nome;
    private double altura;
    private int idade;
    private double salarioMensal;

    // definindo os comportamentos (métodos)
    public void comer(){
        // adicionar função
    }

    public void andar(){
        // adicionar função
    }

    public void respirar(){
        // adicionar função
    }

    public double calcularSalarioAnual(){
        // adicionar função
        return 0;
    }
}

```

Figura 1: exemplo de código com criação da classe Pessoa, algumas características e comportamentos

## OBJETOS

De maneira sucinta, objeto é a instância de uma classe e é caracterizado por um conjunto de atributos. Pode-se dizer que o conceito de classes, explicado anteriormente, é um conceito abstrato, servindo como um molde. Esse molde se materializa quando um objeto é criado, onde é possível colocar informações dentro das variáveis. A seguir, na figura 2, tem-se um exemplo da criação de um objeto (classe “Main”) onde um nome, altura, idade e salário são armazenados no objeto “umaPessoa”:

```

public class Main {
    Run | Debug
    public static void main(String[] args) {
        // instanciando uma classe / criando um objeto chamado "umaPessoa"
        Pessoa umaPessoa = new Pessoa(nome:"Maria", altura:1.60, idade:18, salarioMensal:2000);
    }
}

```

Figura 2: exemplo de código com criação do objeto “umaPessoa”

## MÉTODOS

Os métodos são os comportamentos/funções que o objeto de uma classe pode ter e sua declaração se baseia em: `modificadorDeAcesso + tipoRetorno + nomeMetodo + (lista de parâmetros)`. Um exemplo é apresentado, na figura 3:

```
// modificadorDeAcesso: public
// tipoRetorno: double
// nomeMetodo: calcularSalarioAnual
// parametros: -
public double calcularSalarioAnual(){
    return getSalarioMensal() * 12;
}
```

Figura 3: exemplo de código com método “calcularSalarioMensal”

Dentro do método, é possível criar variáveis que existem somente enquanto o método estiver em execução e para executar o método, deve-se “chamar” o mesmo através do nome e dos parâmetros, caso tenha algum.

O método mais comum nas classes de um programa orientado a objetos é o construtor. Ele é usado para inicializar um objeto, normalmente usado para fornecer valores iniciais para alguns ou todos os atributos da classe (nesse caso, como parâmetros deve-se colocar o tipo e nome dos atributos). Para criá-lo, é necessário saber que ele deve possuir o mesmo nome da classe e não pode possuir retorno. Além disso, é possível ter mais de um método construtor - desde que possua diferença na quantidade e/ou tipo dos parâmetros. Um método construtor já existe de forma implícita, mas nesse caso, quando chamado na inicialização de um objeto, não possuirá parâmetros.

```
// criando um método construtor com todos os atributos da classe
public Pessoa(String nome, double altura, int idade, double salarioMensal) {
    setNome(nome);
    setAltura(altura);
    setIdade(idade);
    setSalarioMensal(idade);
}
```

Figura 4: exemplo de código com método construtor

## ATRIBUTOS

Atributos representam as características de um objeto, podendo ser usados para identificar o mesmo, assim como distinguir um objeto de outro. A sua declaração é composta por três partes:

1. modificador de acesso
2. tipo do atributo
3. nome do atributo

Por exemplo:

```
// definindo as características (atributos)
private String nome; // modificadorDeAcesso: private | tipoDoAtributo: String | nomeAtributo: nome
private double altura;
private int idade;
private double salarioMensal;
```

Figura 5: exemplo de código com criação de atributos da classe “Pessoa”

## ENCAPSULAMENTO

Dar total liberdade para que qualquer alteração seja feita no código pode resultar em efeitos indesejados, por isso, controlar o acesso a atributos e métodos é essencial para uma maior segurança no código, e é exatamente para isso que o encapsulamento serve.

O encapsulamento especifica quais atributos poderão ser acessados por outros objetos (public) e quais poderão ser acessados somente em um local específico (private).

No caso do uso do modificador de acesso private, é necessário criar um método para que alguém de fora possa registrar a característica e outro para mostrá-la sem complicações. Para isso, usa-se métodos modificadores (setter) e assessores (getter), respectivamente. A declaração dos métodos getters e setters seguem uma sintaxe específica, modificando apenas o tipo e o nome do atributo que será encapsulado.

A figura 6 mostra como os métodos citados acima são criados dentro da classe e em seguida, a figura 7 mostra um exemplo de uso prático com os mesmos métodos.

```
// set: cadastra a característica
public void setNome(String nome) {
    this.nome = nome;
}

// get: mostra na tela a característica
public String getNome() {
    return nome;
}
```

Figura 6: exemplo de código com criação de métodos modificadores e assessores

```
// criando o objeto outraPessoa com o construtor sem parâmetros
Pessoa outraPessoa = new Pessoa();
// registrando/modificando o nome no objeto outraPessoa
outraPessoa.setNome(nome: "João");
// imprimindo/acessando o nome no objeto outraPessoa
System.out.println(outraPessoa.getNome());
```

Figura 7: exemplo de código com uso de métodos modificadores e assessores

O *this* (possível visualizar na figura 6) é usado quando refere-se ao objeto da classe em que o método foi chamado, evitando que outro parâmetro com o mesmo nome da variável seja chamado no lugar.

Por fim, fazer o encapsulamento de atributos e métodos é de extrema importância, pois, como já descrito anteriormente, é o encapsulamento que garante a integridade dos dados.

## TRATAMENTO DE EXCEÇÕES

Utilizar o tratamento de exceções durante a programação é essencial para que o desenvolvedor possa detectar possíveis erros, tendo assim maior facilidade para corrigi-los.

Os erros ocorridos durante a execução de um código podem ser lógicos ou de execução. Apresenta-se abaixo as definições destes termos:

1. Erros lógicos acontecem devido a forma com que o desenvolvedor cria o código, ou seja, são implicações defeituosas ou resultados inesperados, como por exemplo, quando cálculos são escritos de maneira incorreta.

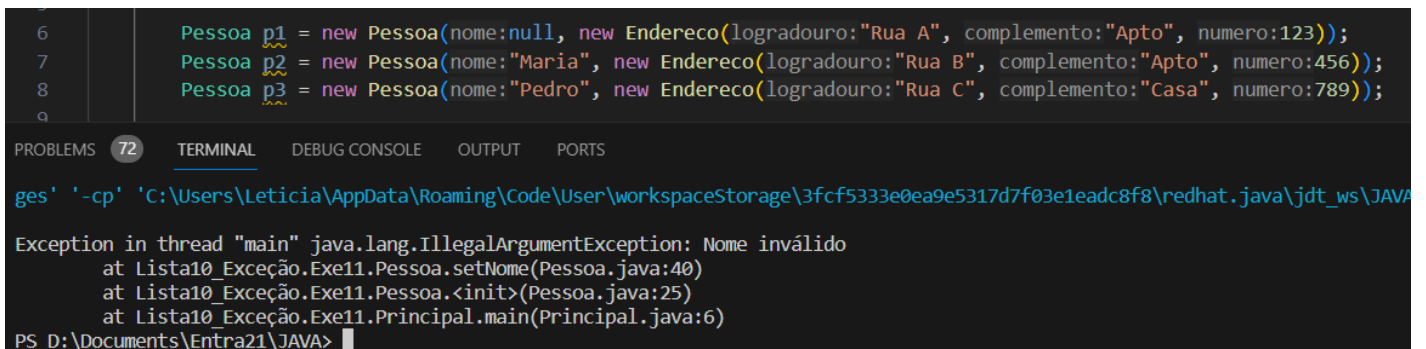
2. Já os erros de execução, como o próprio nome indica, aparecem apenas após a finalização e compilação do código, e são causados por manipulação indevida de arquivos, entrada inadequada de dados, etc.

A figura 8 ilustra como o tratamento de exceção é criado dentro do encapsulamento e a figura 9 ilustra o tratamento ocorrendo quando o nome é *null*.

```
//get: retorna a característica
public String getNome() {
    return nome;
}

//set: cadastra a característica
public void setNome(String nome) throws IllegalArgumentException {
    // verificando se o nome é nulo ou vazio
    if (nome == null || nome.trim().isEmpty()) {
        //fazendo o tratamento de exceção
        throw new IllegalArgumentException(s:"Nome inválido");
    }
    this.nome = nome;
}
```

Figura 8: exemplo de código com uso do tratamento de exceção



The screenshot shows an IDE with a Java file open. The code defines a `Pessoa` class with `nome` and `Endereco` attributes. It creates three instances: `p1` with `nome: null`, `p2` with `nome: "Maria"`, and `p3` with `nome: "Pedro"`. The terminal window shows the command `java -cp 'C:\Users\Leticia\AppData\Roaming\Code\User\workspaceStorage\3fcf5333e0ea9e5317d7f03e1eadc8f8\redhat.java\jdt_ws\JAVA' Lista10_Exceção.Exe11` and the resulting exception: `Exception in thread "main" java.lang.IllegalArgumentException: Nome inválido`. The stack trace points to `Pessoa.setNome(Pessoa.java:40)`, `Pessoa.<init>(Pessoa.java:25)`, and `Principal.main(Principal.java:6)`. The prompt at the bottom is `PS D:\Documents\Entra21\JAVA>`.

Figura 9: exemplo de lançamento de exceção no código, onde o nome é nulo

Como demonstrado no exemplo de número 8, durante a criação de um tratamento de exceção, utiliza-se:

1. *throws*: para indicar que um método pode passar uma exceção a outro método que o chamou;
2. *IllegalArgumentException*: para informar o erro que deve ser tratado;
3. *throw*: para lançar uma exceção ao método que o chamou;

4. *new*: para criar uma exceção, visto que ela é considerada um novo objeto que deve ser criado na memória;

Pode-se lançar também exceções utilizando um bloco *try-catch*. Um bloco *try* consiste na palavra-chave *try* seguida por um bloco de código entre chaves (`{}`), seguido de um ou vários *catches*, também chamados de cláusulas *catch* ou rotina de tratamento de exceção, que iniciam com a palavra-chave *catch* seguida por um parâmetro entre parênteses, chamado parâmetro de exceção, recebendo e tratando a exceção que estava no parâmetro.

Em síntese, o tratamento de exceção é realizado através de polimorfismo ou de herança, pois quando trata-se o erro na classe pai, independentemente do número de subclasses, todas recebem as mesmas validações, sendo assim, o mesmo tratamento é utilizado por várias classes.

```
try {  
    int i = input.nextInt();  
    vetor[5] = i;  
} catch (Exception e) {  
    System.out.println(e);  
}  
System.out.println("O programa continuou");  
}
```

Figura 10: exemplo de lançamento de exceção com try-catch

## RELACIONAMENTO ENTRE OBJETOS

Quando trata-se de relacionamento entre objetos, deve-se pensar que ele pode ser temporário ou duradouro. O relacionamento temporário é aquele em que um objeto necessita/depende do outro apenas durante a execução de suas responsabilidades, ou seja, vínculos não são criados entre eles. É como se algo fosse emprestado, mas devolvido logo em seguida. A figura 11 exemplifica como isso ocorre na programação:



```

//Dependência por declaração local:
//ocorre quando o objeto é declarado dentro do método
public void limpar() {
    Apagador a = new Apagador();
    a.esfregar();
}

//Dependência por parâmetro:
//ocorre quando o objeto é chamado como parâmetro do método
public void limpar(Apagador a) {
    a.esfregar();
}

```

Figura 11: exemplo de relacionamentos temporários utilizando métodos

O segundo tipo de relacionamento, o duradouro, é dividido em três tipos, sendo eles: associação, agregação e composição. A seguir, explica-se cada um deles.

1. **Associação:** Trata-se de um relacionamento onde um objeto pode usar outros objetos, dessa forma, um vínculo é criado entre eles. Um exemplo é uma associação entre uma classe Pessoa e uma classe Endereço. A imagem 12 apresenta este exemplo:

```

//um objeto de Pessoa usa um atributo de Endereco
private String nome, telefone, email;
private Endereco endereco;

```

Figura 12: exemplo de associação entre o objeto Pessoa e o atributo Endereço

2. **Agregação:** É o tipo de relacionamento em que um objeto pode conter outros objetos, onde o objeto existe sem as partes e as partes existem sem o todo, pois cada objeto é independente e serve para complementar o todo. Como exemplo utiliza-se novamente a classe Pessoa, mas desta vez, atrelada a um atributo telefone, pois um número de telefone pode existir, porém não há sentido sem que alguém o tenha e o utilize. A figura 13 explica esse relacionamento:

```

//um objeto de Pessoa existe sem um telefone,
//mas um telefone complementa uma Pessoa
private String nome, telefone, email;
private Endereco endereco;

```

Figura 13: exemplo de agregação entre o objeto Pessoa e o atributo telefone

3. **Composição:** Esse relacionamento demonstra que um objeto é formado por outros objetos e que o todo precisa das partes, assim como as partes precisam do todo, pois há uma alta dependência entre eles. Por exemplo: uma pessoa precisa de um cpf, assim como um cpf precisa de uma pessoa. A figura 14 expõe essa ideia:

```
//um objeto Pessoa necessita de um atributo cpf para existir legalmente  
//assim como um atributo cpf necessita de um objeto Pessoa para existir  
private String nome, telefone, email, cpf;  
private Endereco endereco;
```

Figura 14: exemplo de composição entre o objeto Pessoa e o atributo cpf

Desta maneira, entende-se que um objeto não existe isoladamente, pois ele sempre usará ou será formado por outros objetos, e destaca-se que um programa Orientado a Objetos (OO) possui vários objetos que interagem entre si.

## HERANÇA

Na programação a herança permite a uma classe herdar todo comportamento e atributos de outra classe. A classe herdada, conhecida como superclasse, classe pai ou classe base, costuma ser mais geral/abrangente, enquanto a classe que recebe os atributos, chamada de subclasse, classe filha ou classe derivada, é mais específica. A herança é aplicada quando não deseja-se repetir as mesmas informações, tornando o código mais limpo e simples para compreensão e manutenção. Utiliza-se como exemplo três classes: Pessoa (que possui nome, idade e sexo), PessoaFisica (que possui cpf) e PessoaJuridica (que possui cnpj), onde PessoaFisica e PessoaJuridica são subclasses de Pessoa, pois também devem apresentar os atributos nome, idade e sexo. As figuras 15, 16 E 17 ilustram essa ideia:

```

public class Pessoa {

    private String nome;

    public Pessoa(String nome) {
        setNome(nome);
    }

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) throws IllegalArgumentException {
        if (nome == null || nome.trim().isEmpty()) {
            throw new IllegalArgumentException(s:"Nome inválido");
        }
        this.nome = nome;
    }

    @Override
    public String toString() {
        return "Nome: " + getNome();
    }
}

```

Figura 15: exemplo de criação de uma superclasse Pessoa

```

public class PessoaFisica extends Pessoa {

    private String cpf;

    public PessoaFisica(String nome, String cpf) {
        super(nome);
        setCpf(cpf);
    }

    public String getCpf() {
        return cpf;
    }

    public void setCpf(String cpf) throws IllegalArgumentException {
        if (cpf == null || cpf.trim().isEmpty()) {
            throw new IllegalArgumentException(s:"CPF inválido");
        }
        this.cpf = cpf;
    }

    @Override
    public String toString() {
        return "\nCPF: " + getCpf();
    }
}

```

Figura 16: exemplo de criação de uma subclasse PessoaFisica

```

public class PessoaJuridica extends Pessoa {

    private String cnpj;

    public PessoaJuridica(String nome, String cnpj) {
        super(nome);
        setCnpj(cnpj);
    }

    public String getCnpj() {
        return cnpj;
    }

    public void setCnpj(String cnpj) throws IllegalArgumentException {
        if (cnpj == null || cnpj.trim().isEmpty()) {
            throw new IllegalArgumentException(s:"CNPJ inválido");
        }
        this.cnpj = cnpj;
    }

    @Override
    public String toString() {
        return "\nCNPJ: " + getCnpj();
    }
}

```

Figura 17: exemplo de criação de uma subclasse PessoaJuridica

Nos exemplos acima, utilizou-se a palavra reservada *extends* que é a responsável por passar todas as características da superclasse às subclasses derivadas dela.

## POLIMORFISMO

Em POO, o polimorfismo é um conceito muito abordado e é de extrema importância. É através dele que um mesmo tipo de dado pode assumir uma ou mais formas. Diferentemente do que acontece na sobrecarga de métodos, que é quando a assinatura do método é distinta em diferentes classes, o polimorfismo utiliza da sobrescrita de métodos, que trata de métodos com assinaturas necessariamente iguais, que modificam seu comportamento de acordo com a classe em que estão inseridos, assumindo diferentes formas conforme o necessário. Exemplificando isso na prática, é como uma pessoa se veste de acordo com o ambiente que ela está frequentando no momento (praia, igreja, escola, etc.). Nas imagens 18, 19 e 20 segue um exemplo prático de polimorfismo:

```
public abstract class Carro {
    public abstract String descobrirMotor();
}
```

Figura 18: exemplo de criação de declaração de método polimórfico

```
public class BMW extends Carro{
    @Override
    public String descobrirMotor() {
        return "V8";
    }
}
```

Figura 19: exemplo de criação de sobrescrita do método polimórfico da Figura 18

```
public class Ferrari extends Carro{
    @Override
    public String descobrirMotor() {
        return "V12";
    }
}
```

Figura 20: exemplo de criação de sobrescrita do método polimórfico da Figura 18

## COLEÇÕES

A API do Java fornece várias estruturas de dados pré-definidas, chamadas coleções, utilizadas para armazenar referências a outros objetos. Essas classes fornecem métodos eficientes que organizam, armazenam e recuperam seus dados sem que seja necessário conhecer como os dados são armazenados. Isso reduz o tempo de desenvolvimento de aplicativos, por exemplo.

Classes com uma coleção `ArrayList` que só pode armazenar `Strings`, classes com essa espécie de marcador de lugar que podem ser utilizadas com qualquer tipo são chamadas classes genéricas. A Figura 21 mostra alguns métodos comuns da classe `ArrayList`, que neste caso é do tipo `String`, onde o método ‘add’ adiciona um item à lista, e o ‘remove’ remove itens da lista:

```
import java.util.ArrayList;

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

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

        itens.add("Escapamento");
        itens.add("Teto Solar");

        itens.remove(0);

        for (String string : itens) {
            System.out.println(string);
        }
    }
}
```

Figura 21: exemplo de criação de uma lista da coleção de ArrayList

## INTERFACES

As Interfaces são coleções de métodos relacionados que normalmente permitem informar aos objetos o que fazer, mas não como fazer. Implementar uma interface permite que uma classe se torne mais formal sobre o comportamento que promete proporcionar.

A interface é um agrupamento de métodos com o corpo vazio, como podemos ver na Figura 22, e na figura 23 a implementação dela.

```
public interface Interface {

    public void contratarEmpregado();
    public void demitirEmpregado();
}
```

Figura 22: exemplo de criação de uma interface

```
public class Gerente implements Interface {

    @Override
    public void contratarEmpregado() {
        // TODO Auto-generated method stub
    }

    @Override
    public void demitirEmpregado() {
        // TODO Auto-generated method stub
    }

}
```

Figura 23: exemplo de implementação de uma interface

## CONCLUSÃO

Em conclusão, a POO em Java revela-se um paradigma poderoso e flexível para o desenvolvimento de software. Através dos conceitos abordados neste relatório, a linguagem Java oferece uma estrutura robusta que permite a criação de programas modulares, reutilizáveis e de fácil manutenção. Sua ampla gama de bibliotecas facilita a implementação de soluções complexas, impulsionando a eficiência e a escalabilidade dos projetos de software.

Por fim, a aplicação dos princípios da POO em Java não só simplifica o processo de desenvolvimento, mas também promove uma abordagem mais intuitiva e organizada para resolver problemas de programação. Ao entender os conceitos básicos fundamentais da POO e saber aplicá-los com destreza, os desenvolvedores podem construir sistemas mais sólidos e flexíveis, aumentando a qualidade do software desenvolvido. Assim, esse tipo de implementação desempenha um papel crucial no mundo da programação, capacitando os programadores a criar soluções inovadoras e eficazes para uma variedade de desafios.

## REFERÊNCIAS

### **Tipos de erro:**

<https://learn.microsoft.com/pt-br/dotnet/visual-basic/programming-guide/language-features/error-types>

### **POO: O que é programação orientada a objetos?:**

<https://www.alura.com.br/artigos/poo-programacao-orientada-a-objetos>

### **POO – Programação Orientada a Objetos:**

<https://docente.ifrn.edu.br/placidoneto/disciplinas/2014.1/poo/poo-05-construtores>

**Construtores em Java:** <https://www.dio.me/articles/construtores-em-java>

### **Palavras reservadas:**

<https://glysns.gitbook.io/java-basico/sintaxe/palavras-reservadas#:~:text=throw%3A%20usado%20para%20passar%20uma,que%20pode%20causar%20uma%20exce%C3%A7%C3%A3o.>

**Interface Collection<E>:** <https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html>

**Bibliografia:** Java: como programar 10ª edição - Paul Deitel e Harvey Deitel.