Paradigmas de Programação

**Paradigma Orientado a Objetos**

## Introdução

O assunto desta aula é o Paradigma Orientado a Objetos. O texto aborda as principais características desse paradigma, exemplificando-as por meio da linguagem de programação Java. Características da linguagem também estão apresentadas e exercícios para fixação do conteúdo são sugeridos no fim do texto.

## Revisão de Conceitos

Nas aulas anteriores, foram abordados os seguintes tópicos:

* Conceito de Paradigma
* Principais paradigmas
* Separação de linguagens em imperativas e declarativas
* Paradigmas Estruturado e Não Estruturado
* Paradigma Procedural
* Linguagem de Programação C
* Exercícios

É importante recordá-los para dar sequência aos estudos.

Além disso, antes de iniciar o paradigma a ser abordado neste material, aqui estão alguns exercícios relativos ao último conteúdo visto.

1. (POSCOMP 2018) Assinale a alternativa correta em relação ao Paradigma Imperativo de Linguagens de Programação.
2. Não é baseado na arquitetura de Von Neumann.
3. É paradigma de linguagens não tipadas.
4. É paradigma de linguagens orientadas a objeto.
5. É baseado na arquitetura de Harvard.
6. Descreve uma sequência de passos que mudam o estado de um programa.
7. (POSCOMP 2018) Assinale a alternativa que contém uma estrutura de controle que permite que coleções de instruções, na linguagem de programação C, sejam executadas repetidamente com o controle de execução baseado em uma expressão booleana ao invés de um contador.
8. for
9. if
10. while
11. switch
12. else
13. (POSCOMP 2018) Sobre tipos de dados, é correto afirmar que:
14. Tipos booleanos são valores que são mantidos fixos pelo compilador.
15. O double é um tipo inteiro duplo com menor precisão do que o tipo inteiro.
16. A faixa de valores dos tipos inteiros tem somente dois elementos: um para verdadeiro e outro para falso.
17. Uma conversão de tipos implícita consiste em uma modificação do tipo de dados executado, automaticamente, pelo compilador.
18. Vetores, matrizes e ponteiros são exemplos de tipos de dados primitivos (básicos).

As alternativas corretas são, respectivamente, **E**, **C** e **D**. Caso tenha obtido respostas diferentes, sugere-se a releitura dos materiais das aulas anteriores.

## Paradigma Orientado a Objetos

O Paradigma Orientado a Objetos (POO) é baseado no conceito de *objetos*, estruturas que contêm atributos/propriedades e oferecem métodos/operações, responsáveis por realizar a comunicação dos objetos e alterar os seus estados. Portanto, a resolução dos problemas se dá através da *interação entre os objetos*, também chamada de *troca de mensagens*. O que é uma mensagem?

A especificação dos objetos é feita na *classe*, que pode ser comparada a um tipo (ex: int) abstrato de dados.

Em 1967 surgiu a primeira linguagem Orientada a Objetos: Simula. Em seguida, em 1972, surgiu a linguagem SmallTalk. Outros exemplos de linguagens do Paradigma Orientado a Objetos são: Java, C#, Ruby, Object Pascal.

Existem também linguagens que originalmente não foram projetadas para serem orientadas a objetos, mas que oferecem suporte a esse paradigma. Entre elas estão PHP, JavaScript e Perl.

Na POO, diz-se que a unidade básica de trabalho é a classe.

O conceito de objetos está relacionado à classe. Diz-se que objetos são instâncias de classes.

A linguagem trabalhada neste material para demonstrar o paradigma Orientado a Objetos é Java.

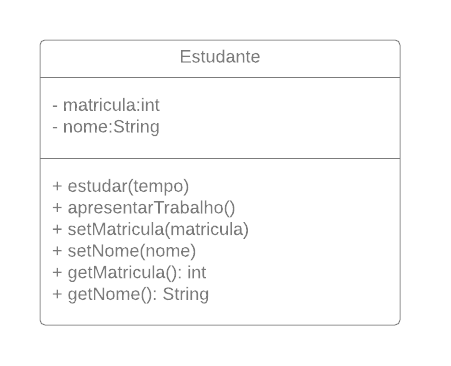
### Java

A linguagem de programação Java possui as seguintes características:

* Tipagem estática e forte.
* Utiliza *JIT compiler*, que trabalha com compilação e interpretação
* Amplamente utilizada
* Suporta operadores aritméticos +/\*-%;
* Suporta pré/pós in/decremento (utilizar com cuidado);
* Suporta operadores relacionais conhecidos;
* Suporta estruturas condicionais e de repetição semelhantes a C/C++ (if, switch, while, for, do... while).
* Comentários de uma linha são redigidos após duas barras (//) e comentários de múltiplas linhas são escritos entre /\* e \*/.
* Utiliza modificadores de acesso para definir a visibilidade de seus atributos e métodos: public, private, protected e package (default).

### Compreendendo os Conceitos de Orientação a Objetos

Para iniciar a compreensão sobre Orientação a Objetos, a Figura 1 exibe a especificação da classe Estudante.



**Figura 1**. Modelagem da classe Estudante com o Diagrama de Classes UML.

O diagrama da Figura 1 é conhecido como *Diagrama de Classes*, uma notação frequentemente utilizada para modelar a estrutura das classes de um projeto (ou algumas de suas partes) e suas relações. Este diagrama segue as regras de uma linguagem de modelagem criada para esse propósito (chamada *Unified Modeling Language* – UML) e deve ser trabalhado em mais detalhes ao longo do curso em disciplinas de Projeto de Software (ex: Engenharia de Software).

Dada a especificação da classe Estudante, é possível compreender que ela contém três seções: na primeira, mais acima, exibe-se o nome da classe. Normalmente esse nome é capitalizado desde a primeira palavra. Por exemplo, se fosse criada uma classe para modelar pessoa jurídica, um nome adequado para ela seria PessoaJuridica. Após o nome, a seção seguinte contém os atributos, propriedades comuns a todos objetos que forem daquele tipo. No exemplo, tem-se os atributos matricula e nome, de tipos int e String, respectivamente. É possível perceber que o tipo int é escrito em letras minúsculas, enquanto String é escrito de forma capitalizada. Isso ocorre porque, em Java, int é um tipo básico, enquanto String não é. Dando sequência à explicação da classe da Figura 1, tem-se a terceira e última seção, contendo os métodos. Os métodos são as operações que aquela classe deve prover. No caso em questão, todo estudante pode estudar e apresentarTrabalho, por exemplo. Os outros métodos dizem respeito às formas de obtenção e alteração dos dados e serão explicadas na sequência.

Nota-se que os nomes dos atributos foram definidos sempre com letras minúsculas, e os nomes de métodos capitalizados, porém, com a primeira palavra minúscula. Essa não é uma regra rígida (não é considerado errado), mas é uma convenção que costuma-se seguir.

Ainda sobre o exemplo da Figura 1, é possível notar também a existência de símbolos “+” e “-”. Esses símbolos representam modificadores de visibilidade e também serão explicados na sequência do material.

Compreendendo (ainda que de forma breve) a especificação da classe Estudante, é possível passar à codificação. O código do quadro a seguir, escrito em Java, é válido para modelar a classe do diagrama da Figura 1. Trechos entre as expressões /\* e \*/ representam comentários de múltiplas linhas, assim como ocorre em outras linguagens como C.

| **/\* Classe estudante, pública \*/**  **public class Estudante {**  **/\* Atributos do estudante**  **o sinal de "menos" no diagrama indica que os atributos são**  **privados. Portanto, só podem ser acessados e manipulados no**  **contexto da própria classe.**  **\*/**  **private int matricula;**  **private String nome;**  **/\* O construtor é um método especial responsável por**  **inicializar o estado interno quando uma instância pra um**  **objeto do tipo Estudante for criado. Quando não é definido,**  **um construtor vazio é criado. Neste caso, criamos**  **explicitamente um construtor vazio. \*/**  **public Estudante() {}**  **/\* Métodos em java aceitam sobrecarga.**  **Desta forma, um método pode conter o mesmo nome que outro**  **definido na mesma classe, mas a assinatura é diferente.**  **Abaixo criamos um construtor diferente para Estudante. \*/**  **public Estudante(String nome) {**  **this.matricula = 0;**  **this.nome = nome;**  **}**  **/\* Aqui novamente sobrecarregamos o construtor. \*/**  **public Estudante(int matricula, String nome) {**  **this.matricula = matricula;**  **this.nome = nome;**  **}**  **/\* Os métodos da classe Estudante estão abaixo.**  **Os métodos comumente são declarados contendo, na assinatura:**  **- visibilidade**  **- tipo de retorno**  **- identificador**  **- lista de parâmetros**  **O sinal de "mais" no diagrama indica que os métodos são**  **públicos. Portanto, podem ser acessados de qualquer lugar. \*/**  **public void estudar(int tempo) {**  **if (tempo > 0)**  **System.out.println("Estudando.");**  **}**  **public void apresentarTrabalho() {**  **System.out.println("Apresentando.");**  **}**  **/\* A criação de métodos getters e setters ajuda a padronizar**  **a forma de criação e comunicação com os objetos, bem como esconder**  **os detalhes de implementação da classe. Assim, sabemos que a forma**  **de obter valores é por meio dos métodos get, e a forma de**  **(re)definir os valores é por meio dos métodos set. Não conseguimos**  **definir diretamente nos atributos porque eles são privados. Este**  **princípio de esconder a estrutura interna e fornecer os métodos**  **como interface de comunicação é conhecido como encapsulamento. \*/**  **public void setMatricula(int matricula) {**  **/\* this é uma referência ao próprio objeto. Assim garantimos**  **que não há ambiguidade entre o atributo do objeto**  **(this.matricula) e o parâmetro do método \*/**  **this.matricula = matricula;**  **}**  **public void setNome(String nome) {**  **this.nome = nome;**  **}**  **public int getMatricula() {**  **return this.matricula;**  **}**  **public String getNome() {**  **return this.nome;**  **}**  **}** |
| --- |

Para criar objetos Estudante (objetos do tipo Estudante, ou *instâncias da classe* Estudante) e testá-los, pode-se criar uma classe de teste, tal como a classe TestaEstudante, definida no quadro que está na sequência:

| **/\* Classe de teste, onde os objetos serão criados e interagirão. \*/**  **public class TestaEstudante {**  **/\* Função estática, ponto de partida do programa. \*/**  **public static void main(String[] a) {**  **/\* Criação de três referências de tipo Estudante. \*/**  **Estudante aluno1,**  **aluno2,**  **aluno3;**  **/\* Instanciação de três referências a objetos do tipo**  **Estudante. A terceira referência aponta para a segunda. \*/**  **aluno1 = new Estudante();**  **aluno2 = new Estudante(109443, "Ricardo");**  **aluno3 = aluno2;**  **/\* Redefinição do nome do objeto por meio do método setNome. \*/**  **aluno3.setNome("João");**  **/\* Escreve os dados do objeto aluno2, do tipo Estudante. \*/**  **System.out.println("O(A) aluno(a) de matrícula "**  **+ aluno2.getMatricula() + " se chama "**  **+ aluno2.getNome() + ".");**  **}**  **}** |
| --- |

"Estudante aluno1;" é uma referência a um eventual objeto da classe Estudante que pode vir a ser instanciado, tomando como exemplo a sintaxe da linguagem Java.

Uma observação importante sobre o Diagrama de Classes: é comum encontrarmos versões resumidas. A Figura 2 mostra um exemplo disso, em que há, na esquerda, o projeto da classe utilizando o Diagrama, e na direita uma implementação (também resumida) dessa classe em Java.

Figura exibindo projeto e implementação resumidos da classe Carro. A figura contém uma seta no meio, apontando da esquerda para a direita, ou seja, do projeto para a implementação. À esquerda, a figura contém o projeto da classe Carro, em um Diagrama de Classes UML, contendo os atributos cor e velocidadeAtual e os métodos acelerar e frear, ambos sem parâmetros. À direita, a figura contém a implementação, com o código:
public class Carro {
   String cor;
   int velocidadeAtual;

   void acelerar() {}
   void frear() {}
}

**Figura 2**. Modelagem e implementação resumidas da classe Carro.

#### Encapsulamento

Um dos conceitos fundamentais da Orientação a Objetos é o Encapsulamento. Ele representa o controle de acesso aos atributos e métodos, separando a interface e a estrutura interna. O usuário só mexe no que precisa e mudanças internas não alteram a interface. Exemplos são os métodos *getters* e *setters*. No exemplo da Figura 1, setMatricula e setNome são métodos *setters*, e getMatricula e getNome são métodos *getters*. Apesar de estar explícito no nome, muitas vezes os métodos podem ser nomeados de forma diferente. Nesse caso é importante verificar o que cada método realiza.

#### Herança

Herança é um mecanismo de reaproveitamento de código de uma classe mais genérica em outra(s) mais específica(s). Também é conhecida por sua relação "é um". A classe genérica é chamada de superclasse ou classe mãe. A classe específica é chamada de subclasse ou classe filha.

Um exemplo de herança: se as classes quadrado, retângulo e forma geométrica forem criadas, qual seria a relação de herança entre elas?

A herança pode ser simples ou múltipla. Há linguagens que não aceitam herança múltipla e isso só é possível com o uso de interfaces, como é o caso da linguagem Java. Já a linguagem C++ suporta herança múltipla.

Para demonstrar o mecanismo de herança, propõe-se a elaboração da classe EstudanteBolsista, um tipo específico de estudante. Esse estudante dispõe de todas as propriedades que um estudante normal (modelado através da já implementada classe Estudante) tem, mas ainda tem uma remuneração pela bolsa. Além dos métodos para retornar e atualizar a remuneração, realiza uma ação a mais, pois precisa elaborar relatórios.

A classe EstudanteBolsista poderia ser assim:

| **/\* A palavra extends diz que EstudanteBolsista herda de Estudante. \*/**  **public class EstudanteBolsista extends Estudante {**  **private double remuneracao;**  **public EstudanteBolsista(int matricula, String nome, double remuneracao) {**  **/\* A chamada ao construtor da classe mãe precisa ser a primeira**  **instrução do construtor da classe filha. \*/**  **super(matricula, nome);**  **this.remuneracao = remuneracao;**  **}**  **public void setRemuneracao(double remuneracao) {**  **if (remuneracao > 0)**  **this.remuneracao = remuneracao;**  **}**  **public double getRemuneracao() {**  **return this.remuneracao;**  **}**  **public void elaborarRelatorio() {**  **System.out.println("Relatório elaborado.");**  **}**  **}** |
| --- |

E a classe de teste, nomeada TestaEstudante2, poderia ser assim:

| **/\* Importa a classe DecimalFormat. \*/**  **import java.text.DecimalFormat;**  **/\* Classe de teste, onde os objetos serão criados e interagirão. \*/**  **public class TestaEstudante2 {**  **/\* Função estática, ponto de partida do programa. \*/**  **public static void main(String[] a) {**  **/\* Criação de três referências de tipo Estudante. \*/**  **EstudanteBolsista aluno1,**  **aluno2,**  **aluno3;**  **/\* Instanciação de três referências a objetos do tipo**  **Estudante. A terceira referência aponta para a segunda. \*/**  **aluno1 = new EstudanteBolsista(99, "José", 500);**  **aluno2 = new EstudanteBolsista(109443, "Ricardo", 200);**  **aluno3 = aluno2;**  **/\* Redefinição do nome do objeto por meio do método setNome. \*/**  **aluno3.setNome("João");**  **/\* Formata a saída para delimitar as casas decimais. \*/**  **DecimalFormat formatoMoeda = new DecimalFormat("#.00");**  **/\* Escreve os dados do objeto aluno2, do tipo Estudante. \*/**  **System.out.println("O(A) aluno(a) de matrícula "**  **+ aluno2.getMatricula() + " se chama "**  **+ aluno2.getNome() + " e recebe uma bolsa de "**  **+ formatoMoeda.format(aluno2.getRemuneracao()) + " reais.");**  **}**  **}** |
| --- |

### Polimorfismo

Polimorfismo significa *muitas formas*. É o mecanismo através do qual é possível que a subclasses e a superclasse realizem chamadas a métodos de mesma assinatura, mas que possuam comportamentos diferentes.

Há diferentes tipos de polimorfismo:

* **Sobrecarga**: métodos com o mesmo identificador, mas assinaturas diferentes.
* **Coerção**: int x=10; double y=x; ("*casting*" automático).
* **Inclusão**: referência à superclasse, mas instancia a subclasse (País x = new PaísRico();)
* **De tipo genérico (paramétrico)**: tipos genéricos/generics (LinkedList<T>)

Após a compreensão desses conceitos, pode-se propor uma alteração na classe EstudanteBolsista, de maneira que ela seja da seguinte forma:

| **public class EstudanteBolsista2 extends Estudante {**  **private double remuneracao;**  **public EstudanteBolsista2(int matricula, String nome, double remuneracao) {**  **super(matricula, nome);**  **this.remuneracao = remuneracao;**  **}**  **public void setRemuneracao(double remuneracao) {**  **if (remuneracao > 0)**  **this.remuneracao = remuneracao;**  **}**  **public double getRemuneracao() {**  **return this.remuneracao;**  **}**  **public void estudar(int tempo) {**  **if (tempo > 0)**  **System.out.println("Bolsista estudando.");**  **}**  **public void elaborarRelatorio() {**  **System.out.println("Relatório elaborado.");**  **}**  **}** |
| --- |

Qual é a diferença para a anterior? O método estudar foi reescrito na subclasse, mas ele já está definido na classe Estudante. O que acontecerá?

A classe de teste pode ser assim:

| **public class TestaEstudante3 {**  **public static void main(String[] a) {**    **/\* Desta forma, consigo acessar as referências depois? \*/**  **new Estudante(99, "José").estudar(2);**  **new EstudanteBolsista2(109443, "Ricardo", 200).estudar(3);**  **}**  **}** |
| --- |

É possível perceber que houve chamada a um método de mesmo nome para os dois objetos, mas, olhando apenas a classe de teste, não se conhece as diferenças dos métodos.

| **Para pesquisar!**  Como seria o diagrama de classes para a classe EstudanteBolsista? |
| --- |

### Gerenciamento de Memória

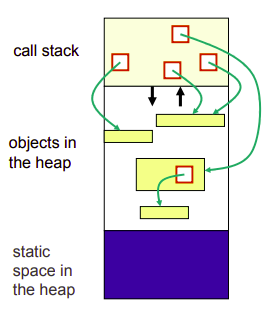
No que diz respeito ao gerenciamento de memória, há áreas distintas de memória para a alocação de cada recurso.

**Pilha**: é usada para empilhar, durante a execução, dados de métodos, tais como parâmetros, variáveis locais e valores de retorno. Quando o método retorna, o espaço de memória por ele ocupado é desalocado automaticamente. A pilha utiliza posições contíguas de memória e não é fragmentada.

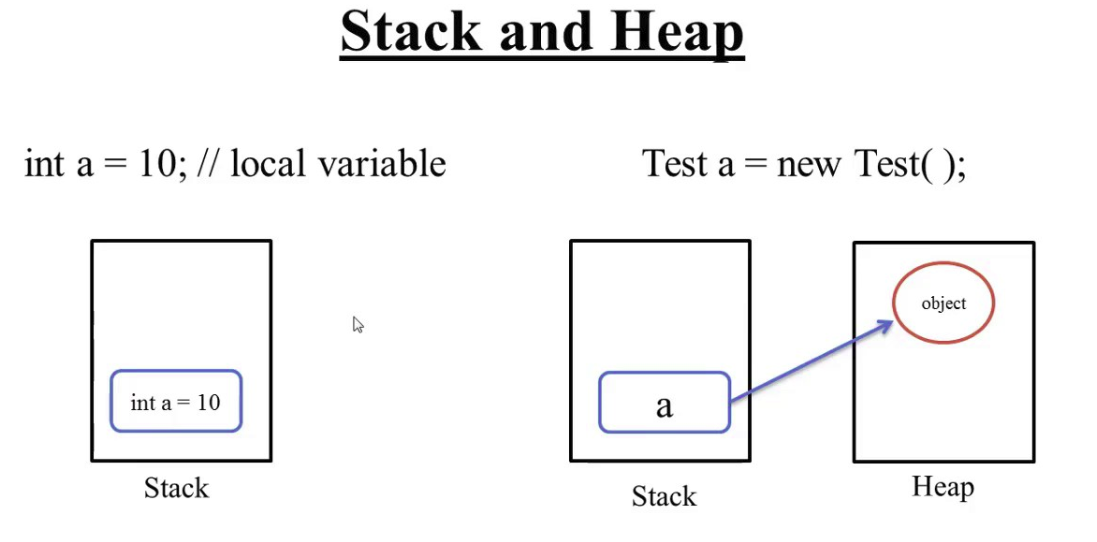
***Heap***: é usada para alocar informações sobre objetos, tais como valores de seus atributos e seu tipo (espaço de objeto), interfaces, classes, suas variáveis estáticas e métodos (espaço estático). Mesmo se não houver referência para um objeto, ele permanecerá no *heap* até que o coletor de lixo desaloque memória. O heap pode ser fragmentado e alocado aleatoriamente, sem ordem.

Embora o *Heap* seja um segmento para alocação dinâmica de memória, há uma parte estática chamada de segmento de código ou segmento de dados. Membros estáticos, chamados de membros de classe ou de contexto de classe, ficam onde são alocadas as classes (*bytecodes*), ou seja, no segmento de dados. É no segmento de dados onde o *bytecode* Java compilado fica quando ele é carregado.

As Figuras 3 e 4 ilustram o que foi explicado nesta subseção.



**Figura 3**. Stack e Heap em Programação Java. Fonte: <https://www.youtube.com/watch?v=w_xMK1ygPjo>.



**Figura 4**. Gerenciamento de memória em Java. Fonte: <https://www.csd.uwo.ca/courses/CS1027b/notes/CS1027-010-Memory-W12.pdf>.

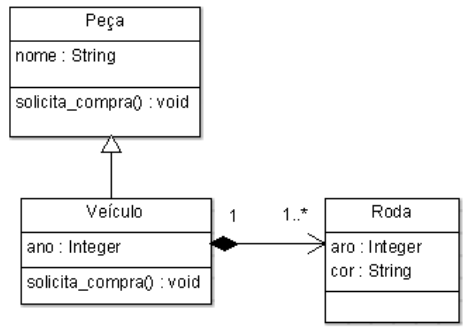
## Conclusão

O Paradigma Orientado a Objetos provê características distintas e é considerado um paradigma realístico para o desenvolvimento de programas. Foram apresentados os principais conceitos desse paradigma por meio d e exemplos criados na linguagem de programação Java.

Alguns conceitos não foram cobertos neste material, tais como o conceito de *interface*, uma espécie de *contrato* que garante os métodos que devem ser implementados na classe. Os estudantes são encorajados a aprofundarem os estudos em Orientação a Objetos e pesquisarem sobre este e outros conceitos não cobertos nessas notas de aula.

## Exercícios

1. O conceito de encapsulamento de programação orientada a objetos pode ser implementado na linguagem Java por meio de
2. métodos estáticos (static) e públicos (public).
3. métodos públicos (public), privados (private) e protegidos (protected).
4. classes abstratas (abstract) e métodos protegidos (protect).
5. interfaces (interface), métodos públicos (public) e métodos protegidos (protect).
6. herança (extends) e métodos estáticos (static)
7. De acordo com o diagrama de classes UML, assinale a alternativa que se relaciona diretamente com o conceito de Polimorfismo da Programação Orientada a Objetos.



1. A relação entre as classes “Veículo” e “Roda”.
2. O método “solicita\_compra()” das classes “Peça” e “Veículo”.
3. Os atributos “aro: Integer” e “cor: String” da classe “Roda”.
4. O atributo “nome: String” da classe “Peça”.
5. O atributo “ano: Integer” da classe “Veículo”.
6. Considere o seguinte código desenvolvido em Java.

| **public class Animal {**  **int numeroPatas;**  **public void fale (){};**  **}**  **public class Cao extends Animal {**  **public void fale() {**  **System.out.println ("au au");**  **}**  **}**  **public class Gato extends Animal {**  **public void fale() {**  **System.out.println ("miau");**  **}**  **}**  **public class GatoPersa extends Gato {**  **public void fale() {**  **System.out.println ("miauuuu");**  **}**  **}**  **public class Tigre extends Gato {**  **public void fale() {**  **super.fale();**  **System.out.println ("rrrrrr");**  **}**  **}**  **public class Principal {**  **public static void main(String[] args) {**  **Gato gato = new GatoPersa();**  **gato.fale();**  **Cao cao = new Cao();**  **cao.fale();**  **Tigre tigre = new Tigre();**  **tigre.fale();**  **}**  **}** |
| --- |

Ao executar o código, a saída impressa no console é:

1. miauuuu

au au

miau

rrrrrr

1. miauuuuu

au au

rrrrrr

1. miau

au au

miau

miau

1. miau

au au

rrrrrr

1. miau

au au

miau

rrrrrr

1. Resolva os exercícios da [Lista #3](https://docs.google.com/document/d/1c2ZSBsNb4-cf2uVPrJnzmnP1r67fAMXEAkXQ9kV6AQg/edit?usp=drive_link).

## Referências

FIGUEIREDO, Eduardo. Conceitos de Programação Orientada a Objetos. Disponível em: <https://homepages.dcc.ufmg.br/~figueiredo/disciplinas/aulas/poo-java_v01-2.pdf>. Acesso em: 27 dez. 2023.

CAVALHEIRO, Gerson G.H. Material de Apoio 1. Material didático. Universidade Federal de Pelotas. [s.d:s.l.].