Ciência da Computação

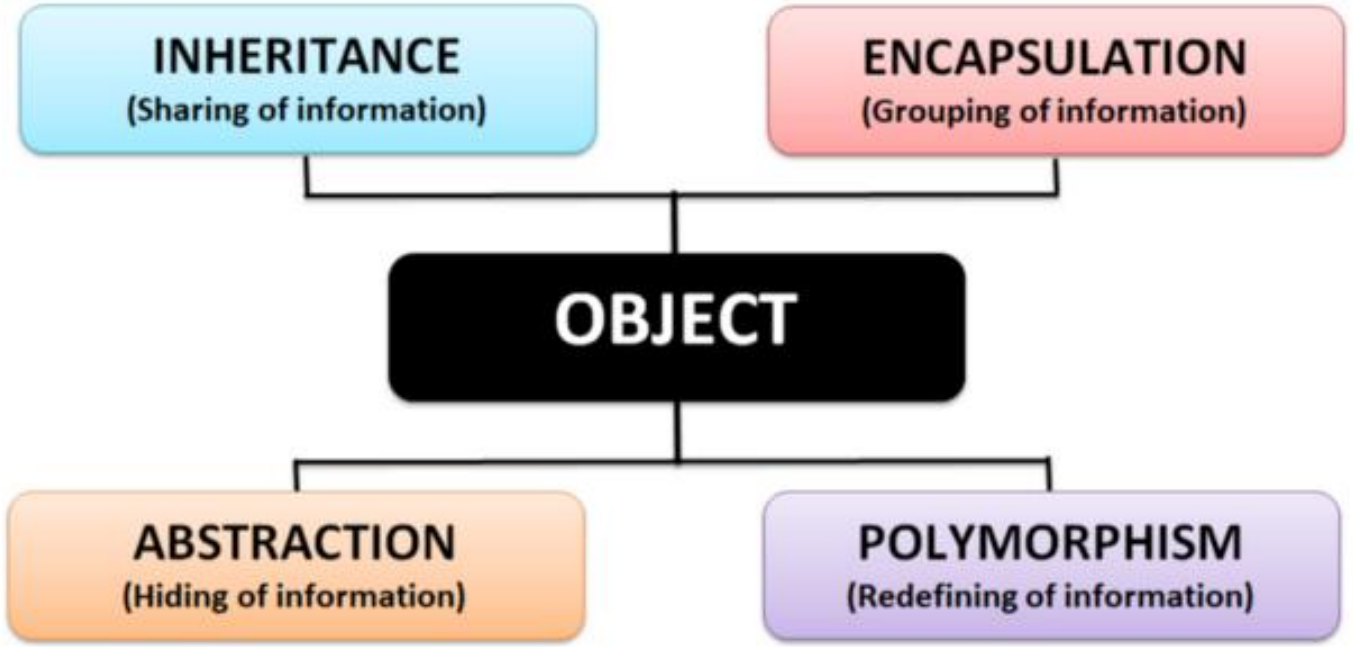
Programação Orientada a Objetos

Prof. Luciano Rodrigo Ferretto

Uma breve revisão...

Abstração, Encapsulamento, Herança, Polimorfismo, Classe Object

Pilares da Orientação a objetos



Ciência da Computação

Abstração

- **Abstração** é o processo de identificar e definir características essenciais de objetos do mundo real, representando essas características em forma de **classes** e **objetos**.

- Uma **classe** é uma estrutura que define atributos e métodos representando um tipo de objeto, fornecendo um modelo para criar instâncias desse tipo.
- Um **objeto** é uma instância de uma classe, caracterizado por seus atributos e comportamentos definidos na classe.
  - Ele pode interagir com outros objetos por meio de métodos e troca de dados.

Objeto é uma **instância** concreta de uma classe na POO.

Encapsulamento

- **Encapsulamento** é a prática de ocultar os detalhes internos de um objeto, expondo apenas as operações relevantes para manipular esses detalhes. Isso promove a modularidade, a segurança e a manutenibilidade do código.

- Na maioria das linguagens de programação orientada a objetos, o encapsulamento é alcançado através da definição de **modificadores de acesso.**
  - **public, private**, **protected e default** (sem modificador)

Herança

- **Herança** é o princípio que permite que uma classe (subclasse) herde os atributos e métodos de outra classe (superclasse), facilitando a reutilização de código, a organização hierárquica e a promoção de relações entre objetos.

- Quando há duas ou mais classes com atributos e métodos em comum, a herança facilita a criação de uma estrutura hierárquica. Nessa estrutura, uma classe "pai" define os elementos comuns que serão herdados pelas classes "filhas".

Polimorfismo

- **Polimorfismo** é um princípio da programação orientada a objetos que permite que funções ou métodos assumam comportamentos diferentes, dependendo do contexto.

- Com o polimorfismo, é possível reutilizar e adaptar comportamentos em diferentes contextos, permitindo que classes e métodos sejam usados de maneira flexível, sem perder a consistência da interface ou lógica comum.

- Ele se divide em dois tipos principais:
  - Polimorfismo Estático
  - Polimorfismo Dinâmico

Polimorfismo

- **Polimorfismo dinâmico** (em tempo de execução): Permite que objetos de sub-classes de uma mesma classe base invoquem métodos que têm a mesma assinatura (nome e parâmetros), mas que se comportam de maneira diferente, de acordo com o tipo específico do objeto instanciado.
  - Sobrescrita de métodos

- **Polimorfismo estático** (em tempo de compilação): Refere-se à capacidade de métodos com o mesmo nome, mas com assinaturas diferentes (número ou tipos de parâmetros), serem utilizados de formas distintas.
  - Sobrecarga de métodos.

Polimorfismo - Estático x Dinâmico

Estático	Dinâmico
Também conhecido como **Sobrecarga de Método.**	Também conhecido como **Sobrescrita de Método.**
Ocorre quando há vários métodos com o mesmo nome, mas com diferentes assinaturas (diferentes tipos ou números de parâmetros) dentro da mesma classe.	Ocorre quando uma subclasse fornece uma implementação específica de um método que já é definido na sua superclasse.
A ligação (binding) das chamadas de método ao código correspondente é feita em tempo de compilação.	A ligação (binding) das chamadas de método ao código correspondente é feita em tempo de execução.
Permite que métodos com o mesmo nome (assinaturas diferentes) possam ser chamados, e a versão correta do método é invocada de acordo com os parâmetros fornecidos na chamada	Permite que o mesmo método (mesma assinatura) possa ser chamado em diferentes tipos de objetos, e a versão correta do método é invocada de acordo com o tipo real do objeto em tempo de execução.

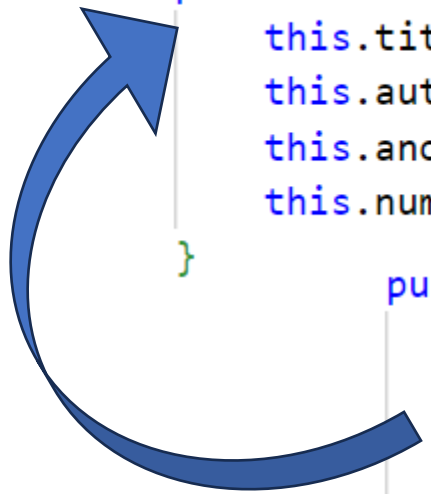
Palavra Chave “Super”

Acesso via “super” - constructors

Usando super para chamar construtores da superclasse

Uma subclasse pode chamar um construtor definido por sua superclasse usando a forma de **super** a seguir:

super(*lista-parâmetros*);



```
public Livro(String titulo, String autor, int anoPublicacao, int numeroPaginas) {
    this.titulo = titulo;
    this.autor = autor;
    this.anoPublicacao = anoPublicacao;
    this.numeroPaginas = numeroPaginas;
}
        public LivroDigital(String titulo, String autor, int anoPublicacao, int numeroPaginas,
                            double tamanhoArquivo, String formatoArquivo) {
            super(titulo, autor, anoPublicacao, numeroPaginas);
            this.tamanhoArquivo = tamanhoArquivo;
            this.formatoArquivo = formatoArquivo;
        }
```

Fonte: SCHILDT, H. Java para iniciantes: crie, compile e execute programas Java rapidamente. 6. ed. Porto Alegre: Bookman, 2015..

Ciência da Computação

Acesso via “super”

Usando super para acessar membros da superclasse

Há uma segunda forma de **super** que age um pouco como **this**, exceto por referenciar sempre a superclasse da subclasse em que é usada. Essa aplicação tem a forma geral a seguir:

super.*membro*

Aqui, *membro* pode ser um método ou uma variável de instância.

```java
@Override //annotations
public String toString() {
    String descricao =
        "Título: " + getTitulo() +
        " - Autor: " + getAutor() +
        " - Ano: " + getAnoPublicacao();
    return descricao;
}
```

```java
@Override //annotations
public String toString() {
    String descricao = super.toString();
    descricao += " - Formato:" + getFormatoArquivo();
    return descricao;
}
```

Fonte: SCHILDT, H. Java para iniciantes: crie, compile e execute programas Java rapidamente. 6. ed. Porto Alegre: Bookman, 2015..

Ciência da Computação

Palavra chave “**abstract**”

Classes e Métodos abstratos

Classes Abstratas

- Uma **classe abstrata** é um conceito fundamental na programação orientada a objetos (POO) e é amplamente utilizado na linguagem de programação Java. Uma classe abstrata é uma classe que não pode ser instanciada diretamente, ou seja, **você não pode criar objetos a partir dela**.
- Em vez disso, ela é projetada para ser uma classe base que fornece uma estrutura comum e recursos compartilhados para suas subclasses.
- Para declarar uma classe abstrata em Java, você utiliza a palavra-chave **abstract** antes da palavra-chave **class**

```
public abstract class Livro {
    private String titulo;
    private String autor;
    private int anoPublicacao;
    private int numeroPaginas;
```

```
Livro novoLivro = new Livro();
```

Cannot instantiate the type Livro Java(16777373)

Livro

View Problem (Alt+... Quick Fix... (Ctrl... Fix using Copilot (Ctrl...

Classes Abstratas

As classes que não são abstratas e herdam classes abstratas são denominadas classes concretas. Sobre as classes abstratas, podemos destacar os seguintes conceitos.

- Uma classe abstrata serve como modelo para uma classe concreta.
- Como são apenas modelos, não podem ser instanciadas diretamente.
- Por não ser instanciadas, devem ser herdadas por classes concretas.
- Uma classe abstrata pode, ou não, conter métodos abstratos, ou seja, pode ou não implementar um método.
- Contudo, métodos abstratos definidos em uma classe abstrata devem, obrigatoriamente, ser implementados em uma classe concreta.

Fonte: SILVA, F. M.; LEITE, M. C. D.; OLIVEIRA, D. B. **Paradigmas de programação**. Porto Alegre: Sagah, 2019.

Métodos Abstratos

- Uma **classe abstrata** pode conter **métodos abstratos**, que são declarados sem uma implementação.
- Os métodos abstratos servem como "contratos" que as subclasses devem implementar.
- A lógica é dizer o que precisa ser feito, e não como fazer.

```
public abstract class Livro {

    // Método abstrato para definir o formato do livro
    //     (ex.: "Físico" ou "Digital")
    public abstract String getFormato();
```

```
public class LivroDigital extends Livro {
```

The type LivroDigital must implement the inherited abstract method Livro.getFormato() Java(67109264)

LivroDigital

View Problem (Alt+F8) Quick Fix... (Ctrl+.) Fix using Copilot (Ctrl+I)

Palavra chave “**final**”

Criando constantes e impedindo a herança e a sobrescrita

Palavra chave “**final**”

- Mesmo com a sobreposição de métodos e a herança sendo tão poderosas e úteis, em alguns casos pode ser desejável **evitar que ocorram**.
- Por exemplo, no nosso Sistema de Gestão de Biblioteca, na classe “Livro” podemos ter um método para calcular a “idade” do livro, ou seja, a quantos anos já fazem desde a publicação.
- Esse é um exemplo prático de método que NÃO deve ser sobrescrito, pois o cálculo não muda independente do tipo de livro.
- Neste exemplo, para evitar que alguém da equipe sobrescreva o método e possa, “sem querer ou não”, comprometer o cálculo, podemos utilizar a palavra chave “**final**”.

Palavra chave “**final**”

- Nas imagens podemos constatar que ao tentar sobrescrever um método definido com “final” na sua assinatura, temos um erro em tempo de compilação.

```
public abstract class Livro {

    public final int calcularTempoPublicacao() {
        int anoAtual = LocalDate.now().getYear();
        return anoAtual - this.getAnoPublicacao();
    }
```

```
public class LivroFisico extends Livro {

    public int calcularTempoPublicacao() {
        int ano
        return
    }

Cannot override the final method from Livro Java(67109265)
Go to Super Implementation
int LivroFisico.calcularTempoPublicacao()
```

Palavra chave “**final**”

- Quando queremos que uma classe não seja instanciada e sirva apenas para ser herdada utilizamos a palavra chave “**abstract**”.
- Com a palavra chave “**final**” temos o contrário, podemos definir que uma classe não possa ser herdada.
- Como era de se esperar, é inválido declarar uma classe como “**abstract**” e “**final**”.

Palavra chave “**final**”

```
public final class LivroFisico extends Livro {
    private int nExemplares;
    private String dimensoes;
```

```
public class SubLivroFisico extends LivroFisico{

}
```

The type SubLivroFisico cannot subclass the final class LivroFisico

LivroFisico

Palavra chave “**final**”

- Você também pode usar a palavra-chave “**final**” para definir **constantes** dentro do seu sistema. Isso é útil para valores que **não devem ser modificados após sua inicialização**.

```
public class Biblioteca {
    // BD em memória
    private List<Livro> acervo = new ArrayList<>();

    private final int ANO_PUBLICACAO_MINIMO = 1400;
```

Sistema SysBiblio

- O Sistema deverá apresentar ao usuário o seguinte menu:

====== SYSBIBLIO ======

Escolha uma das opções abaixo:

1 - Adicionar novo livro

2 - Pesquisar livro por título

3 - Listar todos os livros

4 - Remover livro por título

0 - Sair

🧠 Entenda bem as camadas do nosso projeto

Antes de começar, é essencial ter clareza da **separação de responsabilidades** entre as camadas do sistema:

🔹 Camada View (classe `Main`)

- Responsável por **interagir com o usuário**.
- Exibe o menu e lê as opções escolhidas.
- Mostra mensagens e resultados no console.
- **Não contém regras de negócio** – apenas chama métodos da camada de negócio para executar as ações.

🔸 Camada de Negócio (classe `Biblioteca`)

- Responsável por **manipular os dados e aplicar as regras de negócio**.
- Aqui ficam os métodos como `adicionarLivro()`, `removerLivroPorTitulo()`, `buscarPorAutor()`, etc.
- Deve garantir que os dados sejam válidos e consistentes.
- A **View não deve implementar regras**, apenas repassar comandos para essa camada.

🎯 O que você deve fazer

✅ 1. Implementar a funcionalidade de **remover um livro por título**

- Essa funcionalidade já está no menu (opção 4), mas ainda não foi implementada.
- A lógica deve remover **todos os livros** cujo título contenha o texto informado (ignorar maiúsculas/minúsculas).
- Após a remoção, exibir uma mensagem indicando **quantos livros foram removidos**.

✅ 2. Implementar a opção de menu para **pesquisar livro por título**

- Essa será a **opção 2 do menu**.
- Deve buscar e exibir todos os livros cujo título contenha o texto digitado (ignorar maiúsculas/minúsculas).
- Ao exibir os livros encontrados, devem ser mostradas também as informações de:
    - **Formato** do livro (retornado pelo método `getFormato()`)
    - **Tempo de publicação** (calculado pelo método `calcularTempoPublicacao()`)

✅ 3. Mostrar o formato e o tempo de publicação dos livros listados

- Tanto na **opção 2 (buscar por título)** quanto na **opção 3 (listar todos os livros)**, deve-se:
    - Exibir o resultado com o formato do livro
    - E o tempo de publicação desde o ano de publicação até o ano atual

✅ 4. Implementar uma funcionalidade extra personalizada

Para deixar o seu sistema mais completo e diferenciado, você deverá **implementar uma nova funcionalidade** à sua escolha

Essa funcionalidade deverá:

- Ser inserida como uma **nova opção no menu principal**
- Ter **interação com o usuário**
- Usar os dados dos livros da biblioteca (ou adicionar novos)

📌 Exemplos de funcionalidades possíveis:

1. **Listar livros publicados em um determinado ano**
2. **Listar os livros com mais de X páginas (onde X é informado pelo usuário)**
3. **Ordenar os livros pelo ano de publicação (do mais novo ao mais antigo)**
4. **Buscar livros que tenham uma palavra-chave no título ou no autor**
5. **Listar todos os livros com menos de X anos de publicação (ex: livros recentes)**
6. **Exibir um "resumo" da biblioteca: total de livros, média de páginas, mais antigo, mais novo**
7. **Exportar os dados da biblioteca para um arquivo .txt**
8. **Listar apenas os livros físicos ou apenas os digitais**

🖥️ Na camada View (classe Main)

- Sempre que for exibir o **menu principal**, limpe a tela utilizando os seguintes comandos:

```
System.out.print("\033[H\033[2J");
System.out.flush();
```

- Adicionar no menu:
    - Opção 2 – Pesquisar por título
    - Opção nova – Pesquisar por autor
- Garantir que as opções de exibição mostrem as novas informações (formato e tempo de publicação)

🧠 Na camada de negócio (classe Biblioteca)

No método de **adicionar livro**, aplicar as seguintes validações:

1. ❌ **Título e autor não podem estar em branco**
2. ❌ **Ano de publicação não pode ser menor que `ANO_PUBLICACAO_MINIMO` (1400)**
3. ❌ **Número de páginas não pode ser zero ou negativo**
4. ❌ **Não permitir cadastrar dois livros com o mesmo título (ignorar maiúsculas/minúsculas)**

🚀 Quando terminar

- Teste todas as funcionalidades com diferentes tipos de entrada.
- Lembre-se de manter o código limpo, organizado e com mensagens amigáveis no console.
- Valorize boas práticas de programação: responsabilidade única, encapsulamento, reutilização de código, etc.