Estrutura do Projeto e Classe POJO Pessoa

1. Criando o Projeto Java

- **Ação:** Criar um novo projeto Java via `File > New > Other > Java Project`, nomeando-o como `aula01`.
- **Por que:** Em Java, um projeto organiza seus arquivos de código-fonte, bibliotecas e recursos. Ele define o ambiente de desenvolvimento para sua aplicação.

2. Criando a Classe POJO Pessoa

- **Ação:** Clicar com o botão direito na pasta `src` do projeto, escolher `New > Class`, e nomear a classe como `Pessoa` dentro do pacote `aula01.aula`.
- **Por que:**
    - **Classe POJO:** Um "Plain Old Java Object" é uma classe simples que encapsula dados e, idealmente, não possui dependências complexas de frameworks. Ele serve como um modelo para representar entidades do mundo real (neste caso, uma pessoa).
    - **Pacotes (`package aula01.aula;`):** Pacotes são usados para organizar classes e interfaces Java em grupos lógicos, evitando conflitos de nomes e melhorando a modularidade e a legibilidade do código.

Atributos da Classe Pessoa

- **Código:**

```
public class Pessoa{
    private String nome;
    private int idade;
}
```

- **Por que:**
    - `private String nome;`: Declara um atributo chamado `nome` do tipo `String` (para armazenar texto).
    - `private int idade;`: Declara um atributo chamado `idade` do tipo `int` (para armazenar números inteiros).
    - **`private` (Encapsulamento):** O modificador de acesso `private` é crucial para o conceito de **encapsulamento**, um dos pilares da Programação Orientada a Objetos (POO). Ele significa que esses

atributos só podem ser acessados diretamente de dentro da própria classe `Pessoa`. Isso protege os dados de acesso ou modificação indevida de fora da classe, promovendo a integridade e a manutenção do código.

Construtores da Classe `Pessoa`

- **Ação:** Utilizar o Eclipse (`Source > Generate Constructor using Fields`) para criar construtores.
- **Por que:** Construtores são métodos especiais usados para inicializar objetos de uma classe. Eles são chamados quando você cria uma nova instância da classe usando a palavra-chave `new`.
    - **Construtor Parametrizado (com `nome` e `idade`):**
        - **Código (gerado pela IDE):** Exemplo (não explicitamente no texto, mas implícito pelo processo):
```
public Pessoa(String nome, int idade) {
    this.nome = nome;
    this.idade = idade;
}
```
        - **O que faz:** Permite criar objetos `Pessoa` já fornecendo valores iniciais para `nome` e `idade`. O `this.nome` e `this.idade` se referem aos atributos da instância atual do objeto, distinguindo-os dos parâmetros do construtor.
        - **O que retorna:** Não retorna explicitamente um valor (não tem `return` statement), mas implicitamente retorna uma nova instância da classe `Pessoa` inicializada.
    - **Construtor Vazio (Padrão):**
        - **Código (gerado pela IDE):**

```
public Pessoa() {
    // Corpo vazio ou inicialização padrão, se houver
}
```

        - **O que faz:** Permite criar objetos `Pessoa` sem fornecer nenhum argumento. Os atributos serão inicializados com seus valores padrão (ex: `null` para `String`, `0` para `int`). Se você não definir nenhum construtor em sua classe, Java fornece um construtor padrão vazio automaticamente. No entanto, se você definir um construtor parametrizado, precisará definir explicitamente o construtor vazio se quiser usá-lo.

        - **O que retorna:** Uma nova instância da classe `Pessoa` com atributos inicializados para seus valores padrão.

Métodos Getters e Setters

- **Ação:** Utilizar o Eclipse (`Source > Generate Getter and Setters`) para criar esses métodos para `nome` e `idade`.
- **Por que:** Getters (métodos `get`) e Setters (métodos `set`) fornecem uma interface pública para acessar e modificar os atributos privados da classe, respectivamente. Isso adere ao princípio de **encapsulamento**, controlando como os dados são manipulados.
    - **Métodos get (Getters):**
        - **Exemplos:** `public String getNome()`, `public int getIdade()`
        - **O que fazem:** Permitem que outras classes "leiam" o valor de um atributo privado.
        - **O que retornam:** O valor do atributo correspondente (`String` para `getNome()`, `int` para `getIdade()`).
    - **Métodos set (Setters):**
        - **Exemplos:** `public void setNome(String nome)`, `public void setIdade(int idade)`
        - **O que fazem:** Permitem que outras classes "modem" o valor de um atributo privado. Eles frequentemente incluem lógica de validação para garantir que os dados sejam consistentes.
        - **O que retornam:** `void` (não retornam nenhum valor, apenas executam uma ação de modificação).

A Classe de Teste (`PessoaTeste`) e Sobrescrita de Métodos

3. Criando a Classe de Teste `PessoaTeste`

- **Ação:** Criar uma nova classe chamada `PessoaTeste.java`.
- **Por que:** Esta classe é usada para **testar** o comportamento da classe `Pessoa` e entender a necessidade de sobrescrever métodos padrão de Java, como `toString()`, `equals()` e `hashCode()`.
- **`public static void main(String[] args)`:** Este é o **método principal** de qualquer aplicação Java. Quando você executa uma classe, a Java Virtual Machine (JVM) (um componente chave do JDK, que também inclui ferramentas e bibliotecas) procura e executa este método como ponto de entrada do programa.
    - `public`: Acessível de qualquer lugar.

- `static`: Pode ser chamado sem a necessidade de criar uma instância da classe `PessoaTeste`.
- `void`: Não retorna nenhum valor.
- `String[] args`: Um array de strings que pode receber argumentos passados pela linha de comando.

4. O Método `toString()`

Comportamento Padrão (`toString()` não sobrescrito)

- **Código de Teste:**

```
public class PessoaTeste {
    public static void main(String[] args) {
        Pessoa pessoa1 = new Pessoa("Maria", 20);
        System.out.print(pessoa1);
    }
}
```

- **O que acontece:** Quando `System.out.print(pessoa1)` é chamado sem o `toString()` sobrescrito, Java invoca o método `toString()` padrão da classe `Object` (que é a superclasse de todas as classes em Java).
- **Saída:** A saída será uma representação padrão, como `aula01.aula.Pessoa@XXX`, onde XXX é o código hash do objeto (em hexadecimal).
- **Por que essa saída não é útil:** Essa representação **não é informativa** e não mostra o estado real dos atributos (`nome` e `idade`) do objeto `Pessoa`. Não é adequada para depuração ou exibição ao usuário.

Sobrescrevendo o Método `toString()`

- **Ação:** Utilizar o Eclipse (`Source > Generate toString()`), selecionando os campos `nome` e `idade`.
- **Código (gerado pela IDE):** Exemplo:

```
@Override
public String toString() {
    return "Pessoa [nome=" + nome + ", idade=" + idade + "]";
}
```

- **Por que sobrescrever:** Para fornecer uma **representação mais legível e significativa** do objeto, mostrando os valores de seus atributos. Isso é extremamente útil para depuração e logs.
- **`@Override`:** Esta anotação é um marcador que indica que o método abaixo está sobrescrevendo um método de uma superclasse (neste caso, `Object`). Embora não seja obrigatória para a funcionalidade, é uma boa prática porque ajuda o compilador a verificar se a assinatura do método está correta. Se houvesse um erro na assinatura (ex: `toStrng()` em vez de `toString()`), o compilador sinalizaria um erro.
- **O que faz:** Constrói uma `String` formatada que inclui o nome da classe e os valores dos atributos `nome` e `idade`.
- **O que retorna:** Uma `String` que representa o estado do objeto.
- **Saída Após Sobrescrita:** Ao executar o teste novamente, a saída será algo como `Pessoa [nome=Maria, idade=20]`, que é muito mais útil.

5. O Método `equals()`

Comportamento Padrão (`equals()` não sobrescrito)

- **Código de Teste:**

```
public class TestePessoa{
    public static void main(String[] args){
        Pessoa pessoa1 = new Pessoa("Maria", 20);
        Pessoa pessoa2 = new Pessoa("Maria", 20);
        System.out.println(pessoa1.equals(pessoa2));
    }
}
```

- **O que acontece:** A implementação padrão de `equals()` na classe `Object` verifica se **duas referências de objeto apontam para o mesmo endereço na memória**.
- **Saída:** `false`.
- **Por que:** Embora `pessoa1` e `pessoa2` tenham os mesmos valores para `nome` e `idade`, eles são **objetos distintos** criados em diferentes locais na memória (`new Pessoa(...)` sempre cria um novo objeto). Portanto, suas referências de memória são diferentes.

6. O Método hashCode()

Comportamento Padrão (hashCode() não sobrescrito)

- **Código de Teste:**

```
public class TestePessoa{
    public static void main(String[] args){
        Pessoa pessoa1 = new Pessoa("Maria", 20);
        Pessoa pessoa2 = new Pessoa("Maria", 20);
        System.out.print(pessoa1); // toString() impresso
        System.out.println(pessoa1.equals(pessoa2)); // false
        System.out.println(pessoa1.hashCode() + " | " +
pessoa2.hashCode());
    }
}
```

- **O que acontece:** O método `hashCode()` tem como objetivo fornecer um **valor numérico (hash)** para a instância de um objeto. A implementação padrão na classe `Object` geralmente retorna um valor único para cada objeto distinto em memória (geralmente derivado do endereço de memória).
- **Saída:** Os dois códigos hash retornados serão **diferentes**.
- **Por que `hashCode()` é importante:** Ele é fundamental para o funcionamento eficiente de **estruturas de dados baseadas em hash** em Java, como `HashMap`, `HashSet` e `Hashtable`. Essas coleções usam o valor hash para determinar rapidamente onde um objeto deve ser armazenado ou recuperado.
- **Regra crucial:** Se dois objetos são considerados "iguais" pelo método `equals()`, eles **DEVE**m ter o mesmo valor `hashCode()`. Se esta regra for violada, coleções baseadas em hash não funcionarão corretamente, pois você pode adicionar um objeto a um `Set` e não conseguir encontrá-lo depois, ou adicionar duplicatas a um `Set` que deveria armazenar apenas elementos únicos.

7. Sobrescrevendo `equals()` e `hashCode()` Juntos

- **Ação:** Utilizar o Eclipse (`Source > Generate hashCode() and equals()`), selecionando os campos `nome` e `idade`.
- **Por que sobrescrever ambos:** É uma **prática obrigatória** em Java: **se você sobrescreve `equals()`, você DEVE sobrescrever `hashCode()`**, e vice-versa. Isso garante o "contrato" entre esses dois métodos e o funcionamento correto

das coleções baseadas em hash. A lógica de igualdade deve ser consistente com a geração do código hash.

- **Código (gerado pela IDE):** O Eclipse gerará a lógica completa para ambos os métodos, verificando se os atributos `nome` e `idade` são iguais para a igualdade de objetos, e gerando um hash code combinado desses atributos.
    - **Exemplo (simplificado, IDE gera mais complexo):**
```
@Override
public boolean equals(Object o) {
    if (this == o) return true; // Se forem a mesma instância, são iguais
    if (o == null || getClass() != o.getClass()) return false; // Null ou classe diferente
    Pessoa pessoa = (Pessoa) o; // Cast para Pessoa
    return idade == pessoa.idade && nome.equals(pessoa.nome); // Compara atributos
}

@Override
public int hashCode() {
    return Objects.hash(nome, idade); // Gera hash baseado nos atributos
}
```

- **O que fazem:**
    - `equals()`: Agora, a lógica de `equals()` compara os **valores** dos atributos `nome` e `idade` dos dois objetos. Se ambos os atributos forem iguais, então os objetos `Pessoa` são considerados iguais.
    - `hashCode()`: Agora, a lógica de `hashCode()` gera um código hash que é consistente com a lógica de `equals()`. Se `nome` e `idade` forem iguais para dois objetos, seus `hashCode()`s serão idênticos.
- **Saída Após Sobrescrita:** Ao executar o teste novamente, `pessoa1.equals(pessoa2)` retornará `true`, e `pessoa1.hashCode()` e `pessoa2.hashCode()` retornarão **códigos hash iguais**. Isso demonstra que os objetos agora são considerados iguais com base em seu conteúdo.

Conceitos Adicionais de Java (Aprofundando)

- **Encapsulamento (Revisão):** Já mencionamos, mas é fundamental. Ele esconde os detalhes internos da implementação de um objeto e expõe apenas uma interface controlada (através de getters e setters). Isso torna o código mais robusto e fácil de manter, pois mudanças internas na classe não afetam o código externo que a utiliza, desde que a interface pública permaneça a mesma.

- **Programação Orientada a Objetos (POO):** O que vimos são pilares da POO:
    - **Encapsulamento:** Agrupamento de dados (atributos) e métodos que operam sobre esses dados em uma única unidade (classe), e a restrição de acesso direto aos dados.
    - **Abstração:** Foco nos aspectos essenciais de um objeto e ocultação dos detalhes complexos. Classes POJO são um bom exemplo de abstração.
    - **Herança:** Permite que uma classe herde características (atributos e métodos) de outra classe. A sobrescrita de métodos (`@Override`) é um conceito relacionado à herança e polimorfismo, pois você está modificando o comportamento de um método herdado da classe `Object`.
    - **Polimorfismo:** A capacidade de um objeto assumir várias formas. No contexto de `toString()`, `equals()` e `hashCode()`, o polimorfismo ocorre porque você está fornecendo implementações específicas para métodos que foram definidos em uma superclasse (`Object`), e a JVM saberá qual versão do método chamar em tempo de execução.
- **JDK (Java Development Kit):** A documentação mencionada é parte do JDK 21. O JDK é um conjunto de ferramentas de desenvolvimento Java que inclui a JRE (Java Runtime Environment), que permite executar aplicações Java, e ferramentas como o compilador Java (`javac`), o depurador (`jdb`), e o JShell (uma ferramenta interativa para explorar Java). É o ambiente completo para desenvolver e executar aplicações Java.
- **JVM (Java Virtual Machine):** É a máquina virtual que executa o bytecode Java (os arquivos `.class` compilados). A JVM é responsável por interpretar e executar o código, gerenciar a memória (incluindo a Coleta de Lixo - Garbage Collection), e garantir a portabilidade do Java ("escreva uma vez, execute em qualquer lugar"). O método `main` é o ponto de entrada para a JVM iniciar a execução do seu programa.
- **APIs e Documentação:** A documentação do JDK 21 é um recurso essencial para desenvolvedores Java. Ela detalha as APIs (Application Programming Interfaces) — as bibliotecas de classes e métodos pré-construídos que você pode usar — e fornece guias para ferramentas, segurança, gerenciamento de memória (como o `Garbage Collection Tuning`) e muito mais.