**Java: trabalhando com listas e coleções de dados**

**Declarando variáveis com var**

A partir da versão 10 do Java, foi adicionada uma nova funcionalidade para a declaração de variáveis chamada **var**. Essa nova palavra-chave permite que o compilador infira automaticamente o tipo da variável com base no valor atribuído a ela. Isso pode tornar o código mais limpo e legível, além de reduzir a digitação de código redundante.

**Sintaxe básica**

A sintaxe básica para declarar uma variável com var é a seguinte:

**var** nomeDaVariavel = valorInicial;

Onde **nomeDaVariavel** é o nome que você quer dar à variável e **valorInicial** é o valor que você quer atribuir a ela. O tipo da variável será inferido automaticamente pelo compilador com base no valor atribuído. Exemplo:

**var** numero = 10;

Nesse exemplo, a variável **numero** será inferida como sendo do tipo **int**, já que o valor atribuído é um número inteiro.

**Limitações**

A declaração de variáveis com var possui algumas limitações:

1. O tipo da variável deve ser inferido automaticamente pelo compilador. Isso significa que não é possível utilizar **var** em variáveis cujo tipo não possa ser inferido automaticamente.
2. Não é possível usar **var** em variáveis sem valor inicial. É necessário atribuir um valor à variável na mesma linha em que ela é declarada.

A declaração de variáveis com **var** é uma funcionalidade relativamente nova no Java, mas que pode trazer diversos benefícios para o código, como a redução de digitação e melhor legibilidade do código. No entanto, é importante lembrar que existem limitações e que nem sempre é possível utilizar esse recurso.

**Arrays no Java**

Em Java, **arrays** são estruturas de dados que permitem armazenar uma coleção de elementos do mesmo tipo. Eles são muito utilizados para manipulação de dados em projetos de programação.

Para declarar um **array** em Java, é preciso definir seu tipo e tamanho. Por exemplo, para criar um **array** de inteiros com tamanho 5, podemos escrever o seguinte código:

**int**[] numeros = **new** **int**[5];

Aqui, estamos declarando um **array** chamado "**numeros**" do tipo "**int**" e com tamanho 5. É importante lembrar que o índice dos elementos de um array começa em 0 e vai até o tamanho do ***array menos 1***.

Após declarar um **array**, podemos inicializá-lo com valores. Por exemplo, podemos preencher o **array** "**numeros**" com os números de 1 a 5 da seguinte forma:

**for** (**int** i = 0; i < numeros.length; i++) {

numeros[i] = i + 1;

}

Aqui, estamos percorrendo o **array** "**numeros**" utilizando um loop for e preenchendo cada posição com seu respectivo índice mais 1.

Também é possível criar **arrays** de objetos e não apenas de tipos primitivos. Por exemplo:

Filme[] filmes = **new** **Filme**[2];

**Filme** filme1 = **new** **Filme**("Avatar", 2009);

**Filme** filme2 = **new** **Filme**("Dogville", 2003);

filmes[0] = filme1;

filmes[1] = filme2;

Embora os **arrays** sejam úteis, eles possuem algumas limitações que podem causar problemas em projetos. Alguns desses problemas incluem:

1. **Tamanho fixo**: o tamanho de um **array** é fixo e não pode ser alterado após a sua criação. Isso pode ser problemático em situações em que o tamanho dos dados a serem armazenados é desconhecido ou variável.
2. **Ausência de métodos**: **arrays** não possuem métodos que permitam a inserção, remoção ou pesquisa de elementos de forma eficiente. Isso pode levar a soluções de código complicadas e ineficientes para tarefas simples.

Justamente por conta desses problemas e dificuldades é que não devemos utilizar **arrays** para representar uma coleção de elementos, mas sim alguma classe do Java, como a **ArrayList**, que encapsula e abstrai um **array**, facilitando a sua utilização via métodos e deixando o código do projeto mais simples de entender e evoluir.

**JavaDoc do ArrayList**

Para conhecer mais detalhes da classe, bem como conhecer todos os métodos que ela possui, você pode consultar o [JavaDoc de ArrayList](https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/ArrayList.html).

**Carrinho de compras**

Sua amiga Iasmin trabalha em uma aplicação Java de uma loja virtual e precisa criar uma classe que representa um carrinho de compras. Ela pensou em fazer da seguinte maneira:

**public** **class** **Carrinho** {

ArrayList<Produto> produtos = **new** ArrayList<>();

**public** **void** **adicionaProduto**(Produto produto) {

produtos.**add**(produto);

}

}

Porém, a loja tem uma restrição de permitir uma quantidade máxima de apenas 10 produtos no carrinho de compras. Como fazer a mudança correta no código anterior para atender a essa restrição?

* É preciso alterar o método adicionaProduto para:

**public** **void** **adicionaProduto**(Produto produto) {

**if** (produtos.size() < 10) {

produtos.**add**(produto);

} **else** {

System.**out**.prinln("Carrinho cheio!");

}

}

O método **size**, da classe **ArrayList**, serve justamente para indicar a quantidade atual de elementos da lista.

**ArrayList e toString**

Faça o mesmo procedimento feito na aula, sobrescrevendo o método toString() na classe Filme, e alterando a classe Principal para criar uma lista de filmes.

@Override

**public** String toString() {

**return** "Filme: " +**this**.getNome() + " (" +**this**.getAnoDeLancamento() + ")";

}

E no método main da classe Principal, crie um ArrayList, adicione os filmes e imprima a lista:

**public** **class** **Principal** {

**public** **static** **void** **main**(String[] args) {

//codigo anterior omitido

ArrayList<Filme> listaDeFilmes = **new** ArrayList<>();

listaDeFilmes.**add**(favorito);

listaDeFilmes.**add**(outro);

System.**out**.println(“Tamanho da lista: “ +listaDeFilmes.size());

System.**out**.println(“Primeiro Filme: “ +listaDeFilmes.**get**(0));

System.**out**.println(listaDeFilmes);

}

}

**Construtor padrão**

Em Java, um construtor é um método especial usado para criar e inicializar um objeto recém-criado. Quando uma classe é definida, ela pode ter um ou mais construtores, sendo que se nenhum construtor for definido explicitamente, o Java criará um construtor default (padrão) automaticamente.

Um construtor default é um construtor que não possui parâmetros e não executa nenhuma instrução. Ele é chamado sempre que um objeto da classe é criado sem argumentos. Por exemplo:

**public** **class** **Pessoa** {

**private** String nome;

**private** String email;

**public** **Pessoa**() {

}

//metodos getters/setters

}

No exemplo de código anterior, a classe **Pessoa** possui um construtor default, que será exatamente o mesmo construtor que o Java criará automaticamente, caso nenhum construtor tivesse sido definido na classe.

Se uma classe define explicitamente um ou mais construtores, mas não define um construtor sem parâmetros, então não há construtor default. Nesse caso, se um objeto é criado sem argumentos, um erro de compilação será gerado.

É importante ressaltar que mesmo que um construtor default possa ser útil em alguns casos, é sempre recomendável definir explicitamente os construtores da classe, especialmente se a classe tiver atributos que precisam ser inicializados com valores específicos ou obrigatórios. Isso também torna o código mais claro e fácil de entender.

**Problemas com construtores**

Você está analisando o código de uma aplicação Java e encontra a seguinte classe:

**public** **class** **Produto** {

**private** String nome;

**private** double preco;

**public** **Produto**() {

}

**public** **Produto**(String nome, double preco) {

nome = nome;

preco = preco;

}

**public** **void** **setNome**(String nome) {

**this**.nome = nome;

}

**public** **void** **setPreco**(double preco) {

**this**.preco = preco;

}

**public** String **getNome**() {

**return** nome;

}

**public** double **getPreco**() {

**return** preco;

}

}

Em relação ao uso de construtores, qual o problema no código acima?

Como os atributos nome e preço são do tipo private, o segundo construtor precisa trocar `nome = nome;`por `this.setNome(nome);` e `preco = preco;` por `this.setPreco(preco);`

**Construtor da classe herdada**

Observe as seguintes classes em seu projeto:

**public** **class** **Pessoa** {

**private** String nome;

**private** int idade;

**public** **Pessoa**(String nome, int idade) {

**this**.nome = nome;

**this**.idade = idade;

}

**public** String **getNome**() {

**return** nome;

}

**public** int **getIdade**() {

**return** idade;

}

}

**public** **class** **Aluno** **extends** **Pessoa** {

**private** int matricula;

**private** String curso;

**public** **Aluno**(String nome, int idade, int matricula, String curso) {

**this**.matricula = matricula;

**this**.curso = curso;

}

**public** int **getMatricula**() {

**return** matricula;

}

**public** String **getCurso**() {

**return** curso;

}

}

Entretanto, o código dessas classes está com um erro. Qual?

O construtor da classe Aluno não está chamando o construtor da classe Pessoa. Ao não chamar o construtor da superclasse Pessoa na subclasse Aluno, os atributos nome e idade não serão inicializados corretamente, gerando um erro de compilação.

**Outras formas de percorrer a lista**

A forma mais comum de percorrer uma lista no Java é utilizando o laço **foreach** tradicional, também conhecido como **for-each**. Esse laço permite que se percorra todos os elementos de uma lista, sem a necessidade de se preocupar com índices ou o tamanho dela, tornando o código mais simples e legível. Por exemplo, suponha que tenhamos uma lista de nomes de pessoas e que desejamos imprimi-los na tela:

ArrayList<String> nomes = **new** ArrayList<>();

nomes.**add**("Jacqueline");

nomes.**add**("Paulo");

nomes.**add**("Suellen");

nomes.**add**("Emily");

**for** (String nome : nomes) {

System.**out**.println(nome);

}

Esse loop for percorre todos os elementos da lista, atribuindo cada um deles à variável **nome**, que é usada para imprimir o valor na tela. Esse tipo de loop é muito útil em situações em que não precisamos realizar nenhuma operação complexa sobre os elementos da lista.

No entanto, a partir do Java 8, foi adicionado na interface **List**, a qual a classe **ArrayList** implementa, um novo método chamado forEach, que possibilita a iteração sobre os elementos da lista de forma mais concisa e elegante. Por exemplo, o exemplo anterior pode ser reescrito utilizando o método forEach da seguinte forma:

nomes.forEach(nome -> System.**out**.println(nome));

Nesse caso, o método **forEach** é chamado sobre a lista nomes e recebe como parâmetro uma expressão lambda que realiza a impressão do valor na tela. A expressão lambda **nome -> System.out.println(nome)** é uma forma compacta de definir uma função que recebe um parâmetro nome e realiza a operação de impressão.

É possível simplificar ainda mais o exemplo de código anterior, utilizando o recurso conhecido como **Method Reference**, que nada mais é do que uma forma reduzida de uma expressão lambda:

nomes.forEach(System.out::println);

No código anterior, o **símbolo ::** é a sintaxe do **Method Reference**, que no exemplo mostrado faz uma referência para o método **println**.

**Variáveis e referências**

Referências são **ponteiros** para objetos em memória, ou seja, elas apontam para um objeto e permitem que você trabalhe com ele. No Java, toda variável de objeto é na verdade uma referência a esse objeto que foi alocado na memória.

Quando você instancia um objeto, está, na realidade, criando um novo bloco de memória que armazena as informações desse objeto. A maneira de chegar a esse bloco de memória, para armazenar e ler informações dele, ocorre por meio de uma referência, que é representada por uma variável. Por exemplo:

**Filme** filme1 = **new** **Filme**("Avatar", 2009);

No exemplo de código anterior, criamos um novo objeto da classe **Filme** e armazenamos uma referência a ele na variável **filme1**.

É importante lembrar que as referências a objetos em Java não são o próprio objeto em si, pois elas apenas apontam para o objeto. Quando você passa uma referência a um método ou atribui uma referência a outra variável, está apenas copiando o valor da referência e não do objeto em si. Por exemplo:

**Filme** filme1 = **new** **Filme**("Avatar", 2009);

**Filme** filme2 = **new** **Filme**("The Matrix", 1999);

**Filme** filme3 = filme1;

No exemplo de código anterior, foram criados apenas dois objetos em memória. A variável **filme3** é apenas uma referência que aponta para o mesmo objeto sendo referenciado pela variável **filme1**.

Uma questão importante relacionada com referências a objetos em Java é a questão da igualdade e identidade de objetos. Quando você compara duas referências de objeto usando o operador de igualdade ==, está comparando as referências em si, não os objetos que elas apontam. Por exemplo:

Filme filme1 = **new** Filme("Avatar", 2009);

Filme filme2 = **new** Filme("Avatar", 2009);

**if** (filme1 == filme2) {

System.**out**.println("Iguais");

} **else** {

System.**out**.println("Diferentes");

}

No exemplo de código anterior, a saída no console será: "**Diferentes**". Embora os dois objetos tenham as mesmas informações na memória, a comparação com == verifica se as referências são iguais, ou seja, se apontam para o mesmo objeto na memória.

**Referências de objetos**

Considere o seguinte código em Java:

**public** **class** **Produto** {

**private** String nome;

**private** double preco;

**public** **Produto**(String nome, double preco) {

**this**.nome = nome;

**this**.preco = preco;

}

**public** String **getNome**() {

**return** nome;

}

**public** double **getPreco**() {

**return** preco;

}

**public** **void** **setPreco**(double preco) {

**this**.preco = preco;

}

}

**public** **class** **Teste** {

**public** **static** **void** **main**(String[] args) {

**Produto** p1 = **new** **Produto**("Caneta", 1.50);

**Produto** p2 = p1;

p2.setPreco(2.00);

System.out.println(p1.getPreco());

}

}

Qual o valor que será impresso ao rodar a classe **Teste**?

2.00

**Comparable e Comparator**

Além da interface **Comparable**, o Java possui uma outra interface chamada **Comparator**, que nos fornece outra alternativa para ordenação de coleções.

Você pode entender melhor a diferença entre elas, com exemplos em códigos, lendo o [artigo Ordenando coleções com Comparable e Comparator.](https://www.alura.com.br/artigos/ordenando-colecoes-com-comparable-e-comparator)

**Método compareTo**

Considere a seguinte classe Java:

public class Conta {

private String titular;

private double saldo;

public Conta(String titular, double saldo) {

this.titular = titular;

this.saldo = saldo;

}

//getters e setters omitidos

}

Implemente corretamente o método **compareTo**, para poder ordenar uma lista de objetos do tipo Conta pelo saldo, de maneira decrescente, via implementação da interface **Comparable**:

@Override

public int compareTo(Conta outra) {

if (this.getSaldo() < outra.getSaldo()) {

return 1;

} else if (this.getSaldo() > outra.getSaldo()) {

return -1;

} else {

return 0;

}

}

A implementação do método **compareTo** está correta e fazendo a ordenação pelo saldo de maneira **decrescente**.

**Interface List**

Considere o seguinte trecho de código. O que pode ser considerado?

**List**<**String**> nomes = **new** **ArrayList**<>();

* A declaração acima está correta, e cria um objeto do tipo **ArrayList** capaz de armazenar **Strings**.
  + A declaração cria corretamente um objeto **ArrayList** que implementa a interface List, capaz de armazenar objetos do tipo **String**.
* É possível trocar o objeto sendo instanciado de **ArrayList** para **LinkedList**, sem que nenhum erro de compilação ocorra no código.
  + A variável foi declarada como sendo do tipo **List**, que é a interface, então qualquer implementação dela que seja instanciada terá os mesmos métodos padronizados.

**Outras classes de listas no Java**

O Java oferece diferentes classes para representar uma lista de objetos. Essas classes são úteis em diferentes cenários, dependendo das necessidades de cada aplicação. As classes mais comuns para representar uma lista no Java são:

**ArrayList**

A principal característica do ArrayList é que ele é baseado em um array dinâmico. Ele armazena os elementos em uma matriz interna e, conforme novos elementos são adicionados, o tamanho da matriz é automaticamente ajustado para acomodar o novo elemento. Da mesma forma, quando um elemento é removido, o tamanho do array é ajustado para evitar o desperdício de espaço. O ArrayList é amplamente utilizado devido à sua facilidade de uso e eficiência em termos de desempenho.

**LinkedList**

A classe LinkedList fornece uma lista encadeada de elementos. Diferentemente do ArrayList, que é baseado em um array, o LinkedList é baseado em uma lista encadeada, o que significa que cada elemento da lista é um objeto que contém uma referência para o próximo elemento. Isso permite que os elementos sejam adicionados e removidos de maneira eficiente em qualquer posição da lista, mas pode tornar a pesquisa de um elemento específico menos eficiente.

O LinkedList é uma boa escolha quando a inserção e remoção de elementos em qualquer posição da lista é frequente e quando não é necessário acessar os elementos de forma aleatória.

**Vector**

A classe Vector é semelhante ao ArrayList, mas é sincronizada, o que significa que é segura para uso em threads concorrentes. No entanto, a sincronização adiciona uma sobrecarga de desempenho, então o Vector pode ser mais lento que o ArrayList em algumas situações.

**Stack**

A classe Stack implementa uma pilha, que é uma coleção ordenada de elementos onde a inserção e remoção de elementos ocorrem sempre no mesmo extremo da lista. Os elementos são adicionados e removidos em uma ordem conhecida como "last-in, first-out" (LIFO), ou seja, o último elemento adicionado é o primeiro a ser removido. A classe Stack é usada com frequência em algoritmos de processamento de texto, bem como em outras situações em que a LIFO é a maneira natural de organizar os dados.

Cada uma dessas classes tem seus próprios pontos fortes e fracos, e a escolha de qual usar dependerá das necessidades específicas da aplicação. Para um melhor entendimento sobre estruturas de dados, recomendamos a leitura dos seguintes artigos:

1. [Estruturas de dados: uma introdução](https://www.alura.com.br/artigos/estruturas-de-dados-introducao?_gl=1*zkdru9*_ga*NjUzNTI4NjEzLjE3MTIwODAxNDE.*_ga_1EPWSW3PCS*MTcxNjc2NTc3NC4xMDQuMS4xNzE2NzY2MDQ4LjAuMC4w*_fplc*OUZTMWZnQWNFUiUyRkRDdFRlT1VnbDhmREhRazRpSUI1a1lQOW8wNGZmYWIlMkZMN0x1bXVuVUpsSUVHa1clMkZHJTJGUWN1MGNRT2JvTEl0WmN0QWJqeWgwRXFHcDYzNkVsWGFZU1dVTk5xYUc3cW5WJTJCS3FGMlVoNldZN1ZmamIlMkZaRGJBJTNEJTNE)
2. [Estrutura de Dados: computação na prática com Java](https://www.alura.com.br/artigos/estrutura-dados-computacao-na-pratica-com-java?_gl=1*zkdru9*_ga*NjUzNTI4NjEzLjE3MTIwODAxNDE.*_ga_1EPWSW3PCS*MTcxNjc2NTc3NC4xMDQuMS4xNzE2NzY2MDQ4LjAuMC4w*_fplc*OUZTMWZnQWNFUiUyRkRDdFRlT1VnbDhmREhRazRpSUI1a1lQOW8wNGZmYWIlMkZMN0x1bXVuVUpsSUVHa1clMkZHJTJGUWN1MGNRT2JvTEl0WmN0QWJqeWgwRXFHcDYzNkVsWGFZU1dVTk5xYUc3cW5WJTJCS3FGMlVoNldZN1ZmamIlMkZaRGJBJTNEJTNE)

**Map e HashMap**

Uma das características mais **importantes** do Java é sua vasta biblioteca padrão, que oferece muitas classes e interfaces úteis para os desenvolvedores. Entre elas, estão o **Map** e o **HashMap**, que são ferramentas essenciais para associação de **chaves** e **valores** em muitas aplicações Java.

**Map**

O Map é uma **interface** que permite que os desenvolvedores associem chaves a valores. É uma estrutura de dados útil para muitas aplicações Java, especialmente aquelas que envolvem a manipulação de grandes quantidades de dados, portanto, é comum usá-lo para realizar buscas, atualização e recuperação de elementos por chaves. Ele é implementado por diversas classes, sendo a mais comum delas o **HashMap**.

**HashMap**

O **HashMap** é uma **classe** que implementa a interface **Map** usando uma tabela hash para armazenar os pares chave-valor. Ele é conhecido por sua eficiência em termos de tempo de execução. Essa classe tem uma complexidade de tempo O(1) - constante - para inserção, recuperação e remoção de elementos. Isso significa que o desempenho do HashMap não depende do tamanho da coleção de dados!

No entanto, é importante lembrar que o HashMap não mantém a ordem de inserção dos elementos e não garante a ordem dos elementos na saída. Isso ocorre porque a ordem dos elementos depende da função de hash usada para mapear as chaves para índices na tabela hash. Além disso, o desempenho do HashMap pode ser afetado se houver muitas colisões de hash entre as chaves. Por exemplo:

import java.util.HashMap;

import java.util.Map;

**public** **class** **ExemploHashMap** {

**public** **static** **void** **main**(String[] args) {

//Criando um objeto da classe HashMap que implementa a interface Map

Map<String, Integer> usandoHashMap = **new** HashMap<>();

// Adicionando pares chave-valor

usandoHashMap.put("Gatos", 1);

usandoHashMap.put("Cachorros", 2);

usandoHashMap.put("Roedores", 3);

// Acessando um valor através de uma chave

int valor = usandoHashMap.**get**("Cachorros");

System.**out**.println("Valor da chave Cachorros: " + valor);

// Removendo um par chave-valor

usandoHashMap.**remove**("Gatos");

// Iterando sobre as chaves

**for** (String chave : usandoHashMap.keySet()) {

System.**out**.println("Chave: " + chave);

System.**out**.println("Valor: " + usandoHashMap.**get**(chave));

}

}

}

O resultado será:

Valor da chave Cachorros: 2

Chave: Cachorros

Valor: 2

Chave: Roedores

Valor: 3