Programação em Java

OXETECH LAB

Módulo 3

Arrays e Coleções

- Introdução aos Arrays
- Tipos de Arrays
- Operações com Arrays
- Arrays como Parâmetros e Retorno de Funções
- Coleções em Java
- Principais Interfaces das Coleções
- Classes de Coleções e Métodos Comuns

Introdução aos Arrays

O que é um array ?

Um array é uma estrutura de dados que armazena um conjunto de elementos de mesmo tipo em posições contíguas de memória. Cada elemento é acessado através de um índice, que representa sua posição dentro do array. Arrays são amplamente utilizados em programação para organizar dados em listas, tabelas, matrizes e até mesmo para manipulações de dados mais complexas.

Imagine um array como uma "caixa de compartimentos": cada compartimento armazena um valor e possui um número identificador (índice), começando sempre do zero.

Exemplo de um array que armazena números inteiros:

Índice	Valor
0	5
1	10
2	15
3	20
4	25

Aqui, temos um array com cinco elementos, e cada número pode ser acessado usando seu índice. Por exemplo, para acessar o valor 15, utilizamos o índice 2.

Vantagens e Limitações dos Arrays

Vantagens:

- Acesso direto: Arrays permitem acessar qualquer elemento de forma rápida usando o índice.
- Organização: Útil para armazenar e organizar grandes quantidades de dados de forma ordenada.
- Simplicidade: São fáceis de entender e de implementar em situações onde o número de elementos é fixo ou conhecido.

Limitações:

- Tamanho fixo: O tamanho de um array é definido no momento da criação e não pode ser alterado.
- Tipo único: Arrays armazenam elementos de um único tipo, como inteiros ou strings.
- Inserção e remoção limitadas: Adicionar ou remover elementos exige deslocar outros valores, tornando essas operações mais lentas.

Declaração

Declarar um array em Java significa especificar o tipo de dados dos elementos que ele conterá e indicar que ele é um array usando colchetes **[]**.

Exemplo:

```
int[] numeros; // Declara um array de inteiros
String[] nomes; // Declara um array de strings
```

Inicialização

Inicializar um array é criar o espaço de memória que ele ocupará e definir o tamanho. Isso é feito usando a palavra-chave **new**.

Exemplo:

```
numeros = new int[5]; // Inicializa o array com espaço
para 5 inteiros
```

Atribuição

Após a inicialização, os elementos podem ser atribuídos aos índices. No Java, o índice do primeiro elemento é 0.

Exemplo:

```
numeros[0] = 5;
numeros[1] = 10;
numeros[2] = 15;
numeros[3] = 20;
numeros[4] = 25;
```

Declaração, Inicialização e Atribuição Simplificada

Há uma maneira simplificada de declarar, inicializar e atribuir valores ao mesmo tempo:

```
int[] numeros = {5, 10, 15, 20, 25}; // Array com
valores já definidos
```

Nesse caso, o compilador Java automaticamente determina o tamanho do array com base no número de elementos fornecidos.

Exemplo Completo

```java
public class ExemploArray {
    public static void main(String[] args) {
        // Declara e inicializa um array de inteiros
        int[] numeros = {2, 4, 6, 8, 10};

        // Acessa e imprime os elementos do array usando seu índice
        System.out.println("Primeiro elemento: " + numeros[0]);
        System.out.println("Terceiro elemento: " + numeros[2]);
        System.out.println("Quinto elemento: " + numeros[4]);
    }
}
```

Ao executar esse código, a saída será:

```
Primeiro elemento: 2
Terceiro elemento: 6
Quinto elemento: 10
```

Arrays são fundamentais para manipulação de dados em Java, oferecendo uma forma organizada de armazenar informações. É importante entender as vantagens e limitações dos arrays para decidir quando e como usá-los, especialmente em casos onde o tamanho e o tipo dos elementos são definidos e previsíveis.

Tipos de Arrays

Arrays são uma estrutura versátil em Java e podem ser usados de diversas maneiras. Além dos arrays básicos unidimensionais, como os que vimos anteriormente, existem também arrays multidimensionais e arrays de objetos. Cada tipo serve para propósitos específicos e permite organizar e manipular dados de maneiras variadas. Vamos explorar esses tipos em detalhes.

Arrays Unidimensionais

Um array unidimensional é o tipo mais comum de array, que armazena uma sequência de elementos em uma única linha. Esses elementos podem ser acessados individualmente por um índice.

Exemplo

```
int[] numeros = {1, 2, 3, 4, 5};
System.out.println(numeros[2]); // Saída: 3
```

Aqui, temos um array de inteiros com 5 elementos, todos dispostos em uma única linha.

Arrays Multidimensionais

Arrays multidimensionais permitem criar estruturas mais complexas, como tabelas, matrizes ou até mesmo cubos de dados. Em Java, o tipo mais comum de array multidimensional é o array bidimensional (também chamado de "matriz"), que armazena dados em linhas e colunas.

Array Bidimensional: Este tipo de array é composto por "arrays de arrays". Imagine uma tabela com linhas e colunas; cada linha é um array e todas as linhas formam um array bidimensional.

Exemplo:

```
int[][] matriz = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};

System.out.println(matriz[1][2]); // Saída: 6 (Elemento
da linha 1, coluna 2)
```

Neste caso, matriz é um array de 3 linhas e 3 colunas. Cada elemento é acessado por dois índices: o primeiro para a linha e o segundo para a coluna.

Arrays Multidimensionais com mais de 2 dimensões: Java permite a criação de arrays com mais de duas dimensões (como cubos 3D). Embora raramente usados em programação básica, esses arrays podem ser úteis em aplicações específicas, como gráficos tridimensionais ou certos tipos de algoritmos.

Exemplo de array tridimensional

```
int[][][] cubo = new int[3][3][3]; // Array
tridimensional de 3x3x3
```

Arrays Irregulares

Java permite a criação de arrays irregulares, também conhecidos como "jagged arrays". Em um array irregular, cada linha de um array bidimensional pode ter um número diferente de colunas.

Exemplo

```
int[][] jaggedArray = new int[3][];

jaggedArray[0] = new int[2]; // Primeira linha com 2 colunas

jaggedArray[1] = new int[4]; // Segunda linha com 4 colunas

jaggedArray[2] = new int[3]; // Terceira linha com 3 colunas

jaggedArray[1][2] = 10; // Atribuindo um valor na segunda

linha, terceira coluna
```

Esse tipo de array é útil quando diferentes conjuntos de dados precisam de armazenamento variado. Ele permite um uso mais eficiente de memória, já que o tamanho de cada linha é definido individualmente.

Operações com Arrays

Nos tópicos anteriores, já vimos operações que nos permitem declarar, inicializar, acessar e manipular elementos de um array. Vamos focar agora em outras operações.

Comprimento de um Array

Em Java, o comprimento de um array pode ser obtido usando a propriedade *.length*. Isso é útil para verificar o tamanho do array e evitar erros de índice fora dos limites (out-of-bounds).

Exemplo:

```
int[] valores = {10, 20, 30, 40, 50};
System.out.println(valores.length); // Saída 5
```

Iteração com Arrays

A iteração é a repetição de um processo para acessar ou modificar elementos. Em Java, as estruturas for, while, e for-each são comumente usadas para percorrer arrays.

Exemplo usando for

```
for (int i = 0; i < valores.length; i++) {
    System.out.println(valores[i]);
}
```

Usamos a variável de controle i para iterar sobre o array percorrendo todas as posições até o tamanho do array.

Exemplo usando for-each

```
for (int valor : valores) {
    System.out.println(valor);
}
```

O for-each é uma forma simplificada do for, a cada iteração a variável valor armazena o elemento do array valores. Por debaixo dos pontos vai ocorrer o incremento e a comparação para saber se o limite do array foi atingido.

Ordenação de Arrays

A classe Arrays fornece o método Arrays.sort() para ordenar elementos em ordem crescente.

Exemplo de ordenação:

```
Arrays.sort(valores);
for (int valor : valores) {
    System.out.println(valor); // Exibe em ordem crescente
}
```

Imprimindo os elementos

O método Arrays.toString() é muito útil para exibir todos os elementos de um array em uma única linha, sem a necessidade de um loop. Ele converte o array em uma representação de String, onde cada elemento é separado por vírgulas e cercado por colchetes.

Exemplo

```
import java.util.Arrays;

int[] valores = {10, 20, 30, 40, 50};

System.out.println(Arrays.toString(valores));

// Saída: [10, 20, 30, 40, 50]
```

Esse método é particularmente útil para depuração (debugging), pois permite ver rapidamente o conteúdo do array sem escrever loops adicionais.

Arrays como Parâmetros e Retorno de Funções

Quando trabalhamos com arrays em Java, é possível usá-los como parâmetros e também como valores de retorno em funções (ou métodos). Esse recurso amplia o potencial das funções, pois permite processar e modificar coleções inteiras de dados com facilidade.

Arrays com Parâmetros de Funções

Quando passamos um array como parâmetro, a função não recebe uma cópia do array, mas uma referência ao array original. Isso significa que qualquer alteração feita no array dentro da função afetará o array original.

Como Passar Arrays como Parâmetro

Para passar um array para uma função, basta declarar o parâmetro como um array do tipo desejado. Aqui está um exemplo simples:

```java
public class ArrayParametroExemplo {

    public static void imprimirArray(int[] numeros) {
        for (int numero : numeros) {
            System.out.print(numero + " ");
        }
    }

    public static void main(String[] args) {
        int[] meuArray = {1, 2, 3, 4, 5};
        imprimirArray(meuArray);
    }
}
```

No exemplo acima, o array meuArray é passado como argumento para o método imprimirArray. O método recebe esse array e imprime cada elemento. Como o método trabalha com uma referência ao array, ele pode modificar os dados, e essas modificações serão refletidas no array original.

Arrays como Valor de Retorno

Também podemos usar arrays como valor de retorno de uma função. Para isso, basta declarar o tipo de retorno do método como o tipo de array desejado.

Exemplo

```
public class ArrayRetornoExemplo {

    public static int[] gerarImpares(int n) {
        int[] impares = new int[n];
        for (int i = 0; i < n; i++) {
            impares[i] = 2 * i + 1;
        }
        return impares;
    }

    public static void main(String[] args) {
        int[] primeirosImpares = gerarImpares(5);
        System.out.println(Arrays.toString(primeirosImpares));
        // Saída: [1, 3, 5, 7, 9]
    }
}
```

Neste exemplo, o método gerarImpares cria um novo array impares contendo os primeiros n números ímpares e retorna esse array. No main, imprimimos o array retornado usando Arrays.toString().

Coleções em Java

As coleções em Java são uma parte fundamental da linguagem, permitindo o armazenamento, manipulação e acesso a grupos de objetos. O Collections Framework fornece uma arquitetura que permite trabalhar com diferentes tipos de coleções de forma unificada, facilitando o desenvolvimento de aplicativos eficientes.

Introdução ao Conceito de Coleções

As coleções são estruturas de dados que permitem agrupar e manipular objetos. Elas oferecem várias funcionalidades, como inserção, remoção, busca e iteração sobre os elementos. O uso de coleções facilita o gerenciamento de grandes volumes de dados, proporcionando flexibilidade e eficiência.

No Java, o Collections Framework define interfaces e classes que representam diferentes tipos de coleções, permitindo que os desenvolvedores escolham a estrutura mais adequada para suas necessidades.

Diferença entre Arrays e Coleções

Embora tanto arrays quanto coleções sejam usados para armazenar grupos de objetos, existem várias diferenças importantes entre eles:

Tamanho Fixo vs. Tamanho Variável:

- **Arrays:** Têm um tamanho fixo, definido no momento da criação. Uma vez que um array é criado, seu tamanho não pode ser alterado.
- **Coleções:** Permitem a adição e remoção dinâmica de elementos, o que significa que seu tamanho pode crescer ou diminuir conforme necessário.

Tipo de Elementos:

- **Arrays:** Podem armazenar elementos de um único tipo de dados (sejam primitivos ou objetos).
- **Coleções:** Podem armazenar objetos de diferentes tipos, facilitando a manipulação de dados heterogêneos.

Funcionalidade:

- **Arrays:** Oferecem operações básicas de acesso e modificação de elementos, mas não possuem métodos embutidos para manipulação de dados.
- **Coleções:** Fornecem uma ampla gama de métodos para manipulação de dados, como busca, ordenação e filtragem.

Estrutura de Dados:

- **Arrays:** São estruturas de dados simples, baseadas em índices.
- **Coleções:** Incluem uma variedade de estruturas de dados, como listas, conjuntos, filas e mapas, cada uma projetada para atender a diferentes requisitos de uso.

Estrutura Geral do Collections Framework

O Collections Framework em Java é dividido em várias interfaces e classes. Aqui estão as principais interfaces e suas características:

Interfaces Principais:

- **Collection:** A interface raiz para todas as coleções. Fornece métodos fundamentais como add(), remove(), clear(), e size().
- **List:** Uma coleção ordenada que permite elementos duplicados. Os elementos podem ser acessados por índice. Exemplos: ArrayList, LinkedList.
- **Set:** Uma coleção que não permite elementos duplicados. Exemplos: HashSet, TreeSet.
- **Queue:** Uma coleção que representa uma fila de elementos. Implementações comuns incluem LinkedList e PriorityQueue.
- **Map:** Uma coleção que associa chaves a valores. Cada chave deve ser única. Exemplos: HashMap, TreeMap.

Classes de Implementação

- **ArrayList:** Implementa a interface List e usa um array dinâmico para armazenar elementos. Permite acesso rápido e fácil modificação.
- **LinkedList:** Implementa a interface List e usa uma lista encadeada, permitindo inserções e remoções eficientes em qualquer lugar da lista.

- **HashSet:** Implementa a interface Set e usa uma tabela de hash para armazenar elementos. Não garante a ordem dos elementos.
- **TreeSet:** Implementa a interface Set e armazena elementos em uma árvore binária, garantindo que os elementos sejam ordenados.
- **HashMap:** Implementa a interface Map e armazena pares chave-valor usando uma tabela de hash.

Utilitários da Classe Collections

A classe Collections fornece métodos utilitários para manipular coleções, como sort(), shuffle(), reverse(), e copy(). Esses métodos podem ser usados em qualquer implementação de coleção.

Principais Interfaces das Coleções

O Collections Framework em Java é baseado em várias interfaces que fornecem uma estrutura unificada para manipulação de coleções. As principais interfaces incluem Collection, List, Set e Map. Cada uma delas tem características específicas e diferentes implementações que atendem a diversas necessidades.

Interface Collection

A interface Collection é a raiz da hierarquia das coleções. Ela define os métodos básicos que todas as coleções devem implementar, como:

- add(E e): Adiciona um elemento à coleção.
- remove(Object o): Remove um elemento da coleção.
- clear(): Remove todos os elementos da coleção.
- size(): Retorna o número de elementos na coleção.
- isEmpty(): Verifica se a coleção está vazia.

Interface List

A interface List é uma extensão da interface Collection que representa uma coleção ordenada de elementos, permitindo elementos duplicados e acesso por índice.

Implementação de List:

- **ArraysList:**
  - Armazena os elementos em um array dinâmico.

- Permite acesso rápido aos elementos (tempo constante) e inserções/remoções mais lentas (tempo linear) em comparação com LinkedList.
- É ideal para cenários onde o acesso a elementos é mais frequente do que inserções e remoções.

Exemplo

```
ArrayList<String> lista = new ArrayList<>();

lista.add("Java");

lista.add("Python");

System.out.println(lista.get(0)); // Saída: Java
```

- **LinkedList:**
  - Utiliza uma lista encadeada, onde cada elemento (nó) contém uma referência ao próximo e ao anterior.
  - Permite inserções e remoções eficientes em qualquer posição, mas o acesso aos elementos é mais lento (tempo linear).
  - É útil quando há muitas inserções e remoções.

Exemplo

```
LinkedList<String> lista = new LinkedList<>();

lista.add("Java");

lista.add("Python");

lista.addFirst("C#"); // Adiciona "C#" no início

System.out.println(lista.get(0)); // Saída: C#
```

Interface Set

A interface Set é uma coleção que não permite elementos duplicados. Ela é utilizada quando a unicidade dos elementos é importante.

Implementação de Set:

- **HashSet:**
  - Armazena elementos em uma tabela de hash.
  - Não garante a ordem dos elementos, mas fornece acesso rápido (tempo constante) para operações como add, remove e contains.
  - É ideal para armazenar elementos únicos.

Exemplo

```
HashSet<String> conjunto = new HashSet<>();

conjunto.add("Java");

conjunto.add("Python");

conjunto.add("Java"); // Não será adicionado

System.out.println(conjunto.size()); // Saída: 2
```

- **TreeSet:**
  - Armazena elementos em uma árvore binária, garantindo que os elementos sejam armazenados em ordem natural (ou de acordo com um comparador).

- Oferece operações de acesso, inserção e remoção em tempo logarítmico.
- É útil quando a ordem dos elementos é importante.

Exemplo

```
TreeSet<String> conjunto = new TreeSet<>();

conjunto.add("Java");

conjunto.add("Python");

conjunto.add("C#");

System.out.println(conjunto); // Saída: [C#, Java, Python]
```

Interface Map

A interface Map representa uma coleção de pares chave-valor, onde cada chave é única e está associada a um valor. É importante notar que o Map não é uma subinterface de Collection, mas faz parte do Collections Framework.

Implementação de Map:

- **HashMap:**
  - Armazena pares chave-valor usando uma tabela de hash.
  - Permite chaves únicas e não garante a ordem dos elementos.
  - É ideal para acesso rápido a elementos por chave.

Exemplo

```
HashMap<String, Integer> mapa = new HashMap<>();

mapa.put("Java", 1);

mapa.put("Python", 2);

System.out.println(mapa.get("Java")); // Saída: 1
```

- **TreeMap**
  - Armazena pares chave-valor em uma árvore binária, garantindo que as chaves estejam ordenadas.
  - Fornece acesso logarítmico para operações como inserção e busca.
  - É útil quando a ordem das chaves é importante.

Exemplo

```
TreeMap<String, Integer> mapa = new TreeMap<>();

mapa.put("Java", 1);

mapa.put("Python", 2);

mapa.put("C#", 3); System.out.println(mapa);

// Saída: {C#=3, Java=1, Python=2}
```

Essas interfaces e suas implementações são fundamentais para o desenvolvimento em Java, proporcionando estruturas eficientes para armazenar e manipular dados de forma organizada.

Classes de Coleções e Métodos Comuns

As classes de coleções em Java oferecem diversas operações que facilitam o gerenciamento de conjuntos de dados. Abaixo, discutiremos como adicionar, remover e acessar elementos, além de apresentar os principais métodos de manipulação e consulta, e formas de percorrer as coleções.

Adicionar, Remover e Acessar Elementos

A manipulação básica de coleções envolve três operações fundamentais: adicionar, remover e acessar elementos. Cada interface e implementação pode ter métodos específicos, mas algumas operações são comuns entre elas.

- **Adicionar Elementos**
  - O método add(E e) é usado para adicionar um elemento a uma coleção. Em listas, o elemento é adicionado na posição padrão (geralmente no final), mas em Set, a inserção pode falhar se o elemento já estiver presente.

Exemplo

```
List<String> lista = new ArrayList<>();

lista.add("Java"); // Adiciona "Java" à lista
```

- **Removendo Elementos**
  - O método remove(Object o) é utilizado para remover um elemento específico. Se o elemento estiver presente, ele é removido e o método retorna true. Para Map, o método remove(Object key) remove o par chave-valor associado à chave especificada.

Exemplo

```
lista.remove("Java"); // Remove "Java" da lista
```

- **Acessar Elementos**
  - O método get(int index) é usado para acessar elementos em listas. Para conjuntos, não há um método direto para acessar, pois eles não mantêm uma ordem. Para Map, o método get(Object key) recupera o valor associado a uma chave.

Exemplo

```
String elemento = lista.get(0);

// Acessa o primeiro elemento da lista
```

Métodos de Manipulação e Consulta

Além das operações básicas de adicionar, remover e acessar elementos, as classes de coleções oferecem uma variedade de métodos para manipulação e consulta. Aqui estão alguns dos métodos mais comuns:

- **contains(Object o)**
  - Verifica se um determinado elemento está presente na coleção. Retorna true se o elemento estiver presente e false caso contrário.

Exemplo

```
boolean existe = lista.contains("Java");

// Verifica se "Java" está na lista
```

- **size()**
  - Retorna o número de elementos presentes na coleção.

Exemplo

```
int tamanho = lista.size(); // Obtém o tamanho da lista
```

- **clear()**
  - Remove todos os elementos da coleção, deixando-a vazia.

Exemplo

```
lista.clear(); // Limpa todos os elementos da lista
```

Esses métodos são essenciais para gerenciar e consultar as coleções de maneira eficaz.

Exercícios

1. Crie um array de inteiros com 5 elementos. Atribua valores a esses elementos e imprima cada um deles utilizando um loop for.

2. Dado um array de double com 4 elementos, atribua um novo valor ao segundo elemento e imprima todo o array utilizando o método Arrays.toString().

3. Crie um array de inteiros com 10 elementos. Calcule e imprima a soma de todos os elementos do array.

4. Dado um array de float com 7 elementos, encontre e imprima o maior e o menor valor presente no array.

5. Crie um array de caracteres com 6 elementos. Inverta a ordem dos elementos e imprima o array resultante.

6. Crie uma ArrayList de String e adicione 3 nomes. Imprima todos os nomes na lista.

7. Dado uma ArrayList com 5 cores, remova uma cor específica e imprima a lista atualizada.

8. Crie um HashSet de inteiros e adicione 5 números inteiros. Em seguida, imprima todos os números no conjunto.

9. Dado um HashSet com 4 elementos, verifique se um número específico está presente e imprima uma mensagem indicando a presença ou ausência.

10. Crie um HashMap onde as chaves são String (nomes de alunos) e os valores são Integer (notas). Adicione 3 alunos e suas notas, e imprima todos os pares chave-valor.

11. Dado um HashMap com 3 pares chave-valor, busque a nota de um aluno específico e imprima o resultado.

12. Crie uma ArrayList de Integer com 5 números. Ordene a lista e imprima os números na ordem crescente.

13. Crie dois HashSet de inteiros. Adicione 3 números a cada um e, em seguida, crie um terceiro HashSet que contenha a união dos dois conjuntos. Imprima o resultado.

14. Dado um TreeMap com 3 pares chave-valor (nomes e idades), atualize a idade de uma pessoa específica e imprima todos os pares após a atualização.

15. Crie uma ArrayList de String contendo 6 palavras. Filtre e imprima apenas as palavras que começam com a letra "A".