

Arrays, ArrayLists e Listas de Objetos

1. Introdução

Durante o desenvolvimento de aplicações, é muito comum precisarmos armazenar vários valores relacionados. Por exemplo, uma aplicação pode precisar armazenar as notas de vários alunos, os nomes de clientes, os produtos de uma loja ou os lançamentos financeiros de uma empresa.

Uma variável comum armazena apenas um valor:

```
String nome = "João";
```

Quando precisamos armazenar vários valores, podemos utilizar estruturas de dados como **arrays** e **listas**.

Em Java, três conceitos importantes são:

- **Arrays** — estruturas de tamanho fixo;
- **ArrayList** — listas dinâmicas fornecidas pela API do Java;
- **Listas de objetos** — estruturas que permitem armazenar objetos de uma determinada classe.

Esses recursos são fundamentais para trabalhar com coleções de dados em aplicações Java.

2. Arrays

Um **array** é uma estrutura utilizada para armazenar vários valores do mesmo tipo.

Uma característica importante dos arrays em Java é que seu **tamanho é definido no momento de sua criação e não pode ser alterado posteriormente**.

Por exemplo:

```
int[] notas = new int[5];
```

Nesse caso, foi criado um array capaz de armazenar cinco valores do tipo `int`.

Podemos visualizar a estrutura da seguinte maneira:

Índice:	0	1	2	3	4
	↓	↓	↓	↓	↓
Notas:	[8]	[7]	[9]	[6]	[10]

Os índices começam sempre em **zero**.

2.1 Declarando e inicializando um array

Existem várias maneiras de criar um array.

Criando um array vazio

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

Os valores serão inicialmente iguais a `0`:

```
[0] [0] [0] [0] [0]
```

Podemos atribuir valores posteriormente:

```
numeros[0] = 10;  
numeros[1] = 20;  
numeros[2] = 30;
```

Inicializando diretamente

Também podemos criar o array já com seus valores:

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

Nesse caso, o Java determina automaticamente o tamanho do array.

2.2 Acessando elementos

Para acessar um elemento, utilizamos seu índice:

```
int[] numeros = {10, 20, 30, 40, 50};  
  
System.out.println(numeros[0]);  
System.out.println(numeros[2]);  
System.out.println(numeros[4]);
```

Resultado:

```
10  
30  
50
```

Também podemos alterar um elemento:

```
numeros[1] = 25;
```

O array passa a ser:

```
[10] [25] [30] [40] [50]
```

3. Percorrendo um array

Uma das operações mais comuns é percorrer todos os elementos de um array.

Podemos utilizar o `for` tradicional:

```
int[] notas = {8, 7, 9, 6, 10};  
  
for (int i = 0; i < notas.length; i++) {  
    System.out.println(notas[i]);  
}
```

A propriedade `length` informa a quantidade de posições existentes no array.

3.1 Enhanced for

Java também possui uma forma simplificada de percorrer arrays:

```
int[] notas = {8, 7, 9, 6, 10};  
  
for (int nota : notas) {  
    System.out.println(nota);  
}
```

Essa estrutura é conhecida como **enhanced for** ou **for-each**.

Ela é especialmente útil quando precisamos apenas acessar os elementos e não precisamos conhecer seus índices.

4. Exemplo prático com array

Imagine uma aplicação que precisa calcular a média de cinco notas:

```
public class MediaNotas {  
  
    public static void main(String[] args) {  
  
        double[] notas = {8.0, 7.5, 9.0, 6.5, 10.0};  
  
        double soma = 0;  
  
        for (double nota : notas) {  
            soma += nota;  
        }  
  
        double media = soma / notas.length;  
  
        System.out.println("Média: " + media);  
    }  
}
```

Nesse exemplo:

1. Um array armazena as notas;
2. O `for-each` percorre as notas;
3. Os valores são somados;
4. A soma é dividida pela quantidade de notas.

5. Limitação dos Arrays

Embora sejam muito úteis, arrays apresentam uma limitação importante: **seu tamanho é fixo**.

Por exemplo:

```
String[] nomes = new String[3];
```

Esse array possui três posições.

Não podemos simplesmente adicionar um quarto elemento:

```
nomes[3] = "Carlos";
```

Isso provocará:

```
ArrayIndexOutOfBoundsException
```

Quando a aplicação precisa adicionar e remover elementos dinamicamente, uma estrutura como `ArrayList` normalmente é mais apropriada.

6. ArrayList

O `ArrayList` é uma implementação da interface `List`, pertencente ao pacote:

```
java.util
```

Para utilizá-lo:

```
import java.util.ArrayList;
```

Exemplo:

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

Diferentemente de um array, o `ArrayList` pode **crescer e diminuir dinamicamente**.

7. Adicionando elementos ao ArrayList

Utilizamos o método `add()`:

```
ArrayList<String> nomes = new ArrayList<>();  
  
nomes.add("João");  
nomes.add("Maria");  
nomes.add("Pedro");
```

Agora temos:

```
João  
Maria  
Pedro
```

Podemos adicionar quantos elementos forem necessários, respeitando os recursos disponíveis da aplicação.

8. Acessando elementos

Para acessar um elemento utilizamos o método `get()` :

```
System.out.println(nomes.get(0));
```

Resultado:

```
João
```

Assim como nos arrays, os índices começam em `0`.

9. Alterando elementos

Para alterar um elemento, utilizamos `set()` :

```
nomes.set(1, "Ana");
```

Antes:

```
João  
Maria  
Pedro
```

Depois:

```
João  
Ana  
Pedro
```

10. Removendo elementos

Para remover um elemento, podemos utilizar `remove()` :

```
nomes.remove(1);
```

O elemento da posição `1` será removido.

Também podemos remover pelo próprio objeto:

```
nomes.remove("Pedro");
```

11. Obtendo a quantidade de elementos

Para descobrir quantos elementos existem em um `ArrayList`, utilizamos:

```
nomes.size();
```

Exemplo:

```
System.out.println("Quantidade: " + nomes.size());
```

É importante observar a diferença:

```
array.length
```

para arrays e:

```
lista.size()
```

para listas.

12. Percorrendo um ArrayList

Podemos utilizar o `for` tradicional:

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

nomes.add("João");
nomes.add("Maria");
nomes.add("Pedro");

for (int i = 0; i < nomes.size(); i++) {
    System.out.println(nomes.get(i));
}
```

Também podemos utilizar o `for-each`:

```
for (String nome : nomes) {  
    System.out.println(nome);  
}
```

13. ArrayList versus Array

As duas estruturas possuem características diferentes.

Característica	Array	ArrayList
Tamanho	Fixo	Dinâmico
Índice	Sim	Sim
<code>length</code>	Sim	Não
<code>size()</code>	Não	Sim
<code>add()</code>	Não	Sim
<code>remove()</code>	Não	Sim
<code>get()</code>	Não	Sim
Tipos primitivos	Sim	Não diretamente
Objetos	Sim	Sim

Uma diferença importante é que `ArrayList` trabalha com **objetos**, não diretamente com tipos primitivos.

Por exemplo, não podemos fazer:

```
ArrayList<int> numeros;
```

Devemos utilizar a classe wrapper:

```
ArrayList<Integer> numeros = new ArrayList<>();
```

O Java realiza automaticamente o processo de **autoboxing** entre `int` e `Integer` quando necessário.

14. Interface List

Embora possamos declarar:

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

é uma boa prática utilizar a interface `List` como tipo da variável:

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

Nesse caso:

```
import java.util.ArrayList;
import java.util.List;
```

A vantagem é programar utilizando a abstração da interface.

Por exemplo:

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

Posteriormente, podemos utilizar outra implementação de `List` sem alterar necessariamente o restante do código.

Essa abordagem é muito comum em aplicações profissionais Java.

15. Listas de Objetos

Até agora trabalhamos principalmente com `String`, `Integer` e outros tipos.

Entretanto, uma das aplicações mais importantes de `List` é armazenar **objetos de classes criadas pelo próprio desenvolvedor**.

Imagine uma classe `Produto`:

```
public class Produto {

    private Long id;
    private String nome;
    private double preco;

    public Produto(Long id, String nome, double preco) {
        this.id = id;
        this.nome = nome;
        this.preco = preco;
    }
}
```

```

    public Long getId() {
        return id;
    }

    public String getNome() {
        return nome;
    }

    public double getPreco() {
        return preco;
    }
}

```

Agora podemos criar uma lista de produtos:

```
List<Produto> produtos = new ArrayList<>();
```

Essa lista poderá armazenar vários objetos `Produto`.

16. Adicionando objetos à lista

Podemos criar produtos e adicioná-los:

```

Produto p1 = new Produto(1L, "Notebook", 3500.00);
Produto p2 = new Produto(2L, "Mouse", 80.00);
Produto p3 = new Produto(3L, "Teclado", 150.00);

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

produtos.add(p1);
produtos.add(p2);
produtos.add(p3);

```

A estrutura pode ser representada como:

```

produtos
|
├── Produto
|   id = 1
|   nome = Notebook
|   preco = 3500
|
├── Produto
|   id = 2
|   nome = Mouse

```

```
|      preco = 80
|      |
|      └─ Produto
|          id = 3
|          nome = Teclado
|          preco = 150
```

17. Percorrendo uma lista de objetos

Podemos utilizar o `for-each`:

```
for (Produto produto : produtos) {

    System.out.println("ID: " + produto.getId());
    System.out.println("Nome: " + produto.getNome());
    System.out.println("Preço: " + produto.getPreco());

    System.out.println("-----");
}
```

Esse padrão é extremamente comum em aplicações Java.

18. Exemplo completo

Podemos criar uma classe responsável por demonstrar o funcionamento:

```
import java.util.ArrayList;
import java.util.List;

public class Principal {

    public static void main(String[] args) {

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

        produtos.add(new Produto(1L, "Notebook", 3500.00));
        produtos.add(new Produto(2L, "Mouse", 80.00));
        produtos.add(new Produto(3L, "Teclado", 150.00));

        for (Produto produto : produtos) {

            System.out.println(
                produto.getId() + " - " +
                produto.getNome() + " - R$ " +
```

```
        produto.getPreco()  
    );  
}  
}  
}
```

Resultado:

```
1 - Notebook - R$ 3500.0  
2 - Mouse - R$ 80.0  
3 - Teclado - R$ 150.0
```

19. Lista de objetos e aplicações reais

Listas de objetos aparecem constantemente em aplicações reais.

Por exemplo, uma aplicação de vendas pode possuir:

```
List<Produto> produtos;
```

Uma aplicação acadêmica:

```
List<Aluno> alunos;
```

Uma aplicação financeira:

```
List<Lancamento> lancamentos;
```

Uma aplicação de recursos humanos:

```
List<Funcionario> funcionarios;
```

Uma API REST:

```
List<Cliente> clientes;
```

Essa estrutura permite representar no código uma coleção de entidades do domínio da aplicação.

20. Filtrando objetos

Também podemos percorrer uma lista e aplicar condições.

Por exemplo, para encontrar produtos com preço superior a R\$ 100:

```
for (Produto produto : produtos) {  
    if (produto.getPreco() > 100) {  
        System.out.println(produto.getNome());  
    }  
}
```

Resultado:

```
Notebook  
Teclado
```

Esse tipo de processamento é muito comum em sistemas de informação.

21. Ordenando uma lista

Podemos ordenar listas utilizando recursos da API Java.

Por exemplo, suponha que desejamos ordenar produtos pelo preço:

```
produtos.sort(  
    (p1, p2) -> Double.compare(  
        p1.getPreco(),  
        p2.getPreco()  
    )  
);
```

Depois podemos percorrer a lista:

```
for (Produto produto : produtos) {  
    System.out.println(  
        produto.getNome() + " - " +  
        produto.getPreco()  
    );  
}
```

Esse recurso utiliza uma **expressão lambda**, um conceito importante introduzido no Java 8.

22. ArrayList e Generics

Observe novamente:

```
List<Produto> produtos = new ArrayList<>();
```

O trecho:

```
<Produto>
```

é chamado de **generic**.

Ele informa que a lista trabalhará com objetos do tipo `Produto`.

Assim, isto é permitido:

```
produtos.add(new Produto(1L, "Notebook", 3500));
```

Mas isto não é permitido:

```
produtos.add("Notebook");
```

O compilador detectará o erro de tipo.

Os generics aumentam a **segurança de tipos** e evitam muitos erros durante a execução do programa.

23. Array de objetos

Também podemos utilizar arrays para armazenar objetos.

Por exemplo:

```
Produto[] produtos = new Produto[3];

produtos[0] = new Produto(1L, "Notebook", 3500);
produtos[1] = new Produto(2L, "Mouse", 80);
produtos[2] = new Produto(3L, "Teclado", 150);
```

Percorrendo:

```
for (Produto produto : produtos) {  
    System.out.println(produto.getNome());  
}
```

Portanto, tanto arrays quanto listas podem armazenar objetos.

A diferença principal continua sendo o **tamanho fixo do array** em comparação com o **tamanho dinâmico da lista**.

24. ArrayList em aplicações orientadas a objetos

Em uma aplicação orientada a objetos, podemos ter várias classes relacionadas.

Por exemplo:

```
Cliente  
├── List<Pedido>  
  
Pedido  
├── List<ItemPedido>  
  
ItemPedido  
├── Produto
```

Isso permite representar relacionamentos complexos entre os objetos.

Um cliente pode possuir vários pedidos:

```
List<Pedido> pedidos;
```

Um pedido pode possuir vários itens:

```
List<ItemPedido> itens;
```

E cada item pode estar relacionado a um produto:

```
Produto produto;
```

Essa estrutura aparece frequentemente em sistemas comerciais.

25. Listas em APIs REST

As listas de objetos também são muito utilizadas no desenvolvimento de APIs REST com Spring Boot.

Por exemplo:

```
@GetMapping
public List<Produto> listar() {

    return produtoService.listarTodos();
}
```

O método retorna:

```
List<Produto>
```

O Spring Boot transforma automaticamente essa lista de objetos em JSON.

Por exemplo:

```
[
  {
    "id": 1,
    "nome": "Notebook",
    "preco": 3500.0
  },
  {
    "id": 2,
    "nome": "Mouse",
    "preco": 80.0
  }
]
```

Assim, o conceito de listas de objetos está diretamente relacionado ao desenvolvimento de APIs modernas.

26. ArrayList e banco de dados

Em aplicações que utilizam JPA/Hibernate, também encontramos frequentemente listas de objetos.

Por exemplo:

```
@OneToMany
private List<ItemPedido> itens;
```

Nesse caso, um `Pedido` pode possuir vários `ItemPedido`.

Outro exemplo:

```
@ManyToOne
private Cliente cliente;
```

Enquanto o relacionamento de um pedido com vários itens pode ser representado por:

```
List<ItemPedido>
```

Esse conceito demonstra como estruturas de dados da linguagem Java podem representar relacionamentos existentes no modelo de negócio.

27. Comparando as principais estruturas

Podemos resumir:

Array

```
Produto[] produtos = new Produto[10];
```

Indicado quando:

- o tamanho é conhecido;
- o tamanho não precisa mudar;
- precisamos de uma estrutura simples e eficiente.

ArrayList

```
List<Produto> produtos = new ArrayList<>();
```

Indicado quando:

- a quantidade de elementos pode mudar;
- precisamos adicionar ou remover elementos;
- queremos utilizar os recursos da API `List`.

Lista de objetos

```
List<Produto> produtos = new ArrayList<>();
```

É especialmente importante quando precisamos trabalhar com **coleções de entidades ou objetos do domínio da aplicação**.

28. Conclusão

Arrays, `ArrayList` e listas de objetos são conceitos fundamentais para o desenvolvimento de aplicações Java.

O **array** oferece uma estrutura simples e eficiente, porém possui tamanho fixo.

O **ArrayList** oferece uma estrutura dinâmica, permitindo adicionar e remover elementos conforme a necessidade da aplicação.

Já as **listas de objetos** permitem representar coleções de entidades do mundo real dentro do programa:

```
List<Aluno> alunos;  
List<Produto> produtos;  
List<Cliente> clientes;  
List<Pedido> pedidos;  
List<Funcionario> funcionarios;
```

Em aplicações Java modernas, especialmente aquelas desenvolvidas com **Spring Boot, JPA e APIs REST**, o uso de `List<T>` é extremamente frequente.

Por isso, compreender a diferença entre:

```
Produto[] produtos;
```

e:

```
List<Produto> produtos = new ArrayList<>();
```

é fundamental para evoluir no desenvolvimento orientado a objetos e na construção de aplicações profissionais.