🔁 Arrays de Objetos em Java

Como visto, podemos criar **vetores (arrays)** de objetos da mesma forma que fazemos com tipos primitivos. Um exemplo:

```
Cliente[] clientes = new Cliente[5];

clientes[0] = new Cliente("Marília", "SP");
clientes[1] = new Cliente("Campinas", "SP");
```

Esse tipo de estrutura é **fixa**. Ou seja:

- O tamanho é definido na criação (`new Cliente[5]`);
- Se precisar armazenar mais de 5 clientes, será necessário criar um novo array maior;
- É indicado quando a quantidade de elementos é conhecida e não muda.

📦 Coleções em Java – `ArrayList`

Quando precisamos de algo **mais flexível**, entramos no mundo das **coleções**. A mais comum e versátil é o `ArrayList`.

✅ **Vantagens:**

- **Tamanho dinâmico** (cresce automaticamente conforme necessário);
- Métodos utilitários como `add()`, `remove()`, `contains()`, etc.;
- É parte do pacote `java.util`.

📌 **Exemplo:**

```
import java.util.ArrayList;

ArrayList<Cliente> clientes = new ArrayList<>();

clientes.add(new Cliente("Marília", "SP"));
clientes.add(new Cliente("Campinas", "SP"));

// Exibindo os dados dos clientes
for (Cliente cliente : clientes) {
```

```
    System.out.println(cliente.listarDados());
}
```

Aqui, podemos adicionar quantos clientes quisermos, sem nos preocupar com o tamanho inicial da lista.

⚠️ Comparando `Array` e `ArrayList`

Característica	`Array`	`ArrayList`
Tamanho	Fixo	Dinâmico
Performance	Levemente mais rápido	Um pouco mais lento (geralmente irrelevante)
Funcionalidades	Básico	Rico em métodos
Tipagem	Pode armazenar tipos primitivos	Só armazena objetos (wrapper se precisar de primitivos)
Conversão para String	`Arrays.toString(array)`	`clientes.toString()` ou iteração manual

🔄 Conversão de Array para `ArrayList`

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

Cliente[] arrayClientes = new Cliente[3];
// preencher...

List<Cliente> listaClientes = new ArrayList<>(Arrays.asList(arrayClientes));
```

Excelente! Esse trecho aprofunda o uso de **`ArrayList` em Java** com um exemplo bem didático dentro de um contexto de sistema bancário. A introdução da classe `Agencia`, com sua relação de **composição** com a classe `Conta`, é uma ótima forma de ilustrar como usar coleções de objetos na prática.

📘 Resumo Teórico

📌 Composição

- **Agencia** é composta por várias **Contas**;
- As contas só existem dentro de uma agência (em termos conceituais);
- Em Java, modelamos isso com um `ArrayList<Conta>` dentro da classe `Agencia`.

🧰 Métodos úteis do `ArrayList`:

Método	O que faz
`add(objeto)`	Adiciona um objeto na lista
`remove(objeto)`	Remove um objeto da lista
`size()`	Retorna a quantidade de objetos na lista
`for-each`	Permite percorrer a lista facilmente

✅ Exemplo prático: Agência com contas

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

// Classe Conta
public class Conta {
    private String numero;
    private String titular;

    public Conta(String numero, String titular) {
        this.numero = numero;
        this.titular = titular;
    }

    public void exibirInfo() {
        System.out.println("Conta: " + numero + " | Titular: " + titular);
    }

    // Getters e setters podem ser adicionados conforme necessário
}

// Classe Agencia
```

```
public class Agencia {
    private String nome;
    private List<Conta> contas;

    public Agencia(String nome) {
        this.nome = nome;
        this.contas = new ArrayList<>();
    }

    public void adicionarConta(Conta conta) {
        contas.add(conta);
    }

    public void removerConta(Conta conta) {
        contas.remove(conta);
    }

    public void listarContas() {
        System.out.println("Contas da agência " + nome + ":");
        for (Conta c : contas) {
            c.exibirInfo();
        }
    }

    public int quantidadeDeContas() {
        return contas.size();
    }
}

// Classe principal para testar
public class BancoApp {
    public static void main(String[] args) {
        Agencia agencia = new Agencia("Centro");

        Conta conta1 = new Conta("001", "Patrick");
        Conta conta2 = new Conta("002", "Eduardo");

        agencia.adicionarConta(conta1);
        agencia.adicionarConta(conta2);

        agencia.listarContas();

        System.out.println("Total de contas: " + agencia.quantidadeDeContas());

        agencia.removerConta(conta1);

        agencia.listarContas();
```

```
        System.out.println("Total de contas: " + agencia.quantidadeDeContas());
    }
}
```

🎓 Dicas adicionais

- No mundo real, a composição muitas vezes exigiria validações: por exemplo, **não permitir uma `Agencia` sem contas**, mas como o texto disse, essa regra foi relaxada aqui para simplificar.

- Você pode tornar o `ArrayList<Conta>` **imutável fora da classe**, para evitar manipulação direta.

✅ 1. Classe `Agencia` (com correção do nome do parâmetro em `excluirConta`)

```
package banco.modelo;

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

public class Agencia {

    private int numero;
    private String nome;
    private List<Conta> contas;

    public Agencia(int numero, String nome) {
        this.numero = numero;
        this.nome = nome;
        this.contas = new ArrayList<>();
    }

    public int getNumero() {
        return numero;
    }

    public String getNome() {
        return nome;
    }
```

```
    public List<Conta> getContas() {
        return contas;
    }

    public void incluirConta(Conta conta) {
        contas.add(conta);
    }

    public void excluirConta(Conta conta) { // Corrigido o nome do parâmetro aqui
        contas.remove(conta);
    }
}
```

✅ 2. Exemplo da `Classe Principal` com menu simples no terminal

Aqui está um exemplo simples do que seria esse menu para incluir contas, listar e sair:

```
package banco;

import banco.modelo.*;

import java.util.Scanner;

public class Principal {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        Agencia agencia = new Agencia(1, "Agência Central");

        while (true) {
            System.out.println("\n=== MENU ===");
            System.out.println("1. Incluir nova conta");
            System.out.println("2. Listar contas");
            System.out.println("3. Sair");
            System.out.print("Escolha uma opção: ");
            int opcao = scanner.nextInt();
            scanner.nextLine(); // limpar buffer

            switch (opcao) {
                case 1:
                    System.out.print("Nome do titular: ");
                    String titular = scanner.nextLine();

                    System.out.print("Número da conta: ");
```

```
                String numeroConta = scanner.nextLine();

                System.out.print("Tipo de conta (1 = Corrente, 2 = Poupança): ");
                int tipoConta = scanner.nextInt();

                Conta novaConta;
                if (tipoConta == 1) {
                    novaConta = new ContaCorrente(numeroConta, titular);
                } else {
                    novaConta = new ContaPoupanca(numeroConta, titular);
                }

                agencia.incluirConta(novaConta); // <-- LINHA 63 citada no material
                System.out.println("Conta adicionada com sucesso!");
                break;

            case 2:
                if (agencia.getContas().size() == 0) {
                    System.out.println("Nenhuma conta cadastrada ainda.");
                } else {
                    System.out.println("Agência: " + agencia.getNome());
                    System.out.println("Total de contas: " + agencia.getContas().size());

                    for (Conta conta : agencia.getContas()) {
                        conta.listarDados();
                    }
                }
                break;

            case 3:
                System.out.println("Encerrando...");
                scanner.close();
                return;

            default:
                System.out.println("Opção inválida!");
        }
    }
  }
}
```

✅ 3. Suporte para Conta, ContaCorrente, ContaPoupanca

Classe `Conta` (superclasse):

```
package banco.modelo;

public abstract class Conta {
    protected String numero;
    protected String titular;

    public Conta(String numero, String titular) {
        this.numero = numero;
        this.titular = titular;
    }

    public abstract void listarDados();
}
```

Subclasse `ContaCorrente`:

```
package banco.modelo;

public class ContaCorrente extends Conta {

    public ContaCorrente(String numero, String titular) {
        super(numero, titular);
    }

    @Override
    public void listarDados() {
        System.out.println("Conta Corrente - Número: " + numero + ", Titular: " + titular);
    }
}
```

Subclasse `ContaPoupanca`:

```
package banco.modelo;

public class ContaPoupanca extends Conta {

    public ContaPoupanca(String numero, String titular) {
        super(numero, titular);
    }

    @Override
    public void listarDados() {
        System.out.println("Conta Poupança - Número: " + numero + ", Titular: " + titular);
    }
}
```

🧠 O que você praticou aqui?

✔️ Pilares de POO

- **Encapsulamento** – uso de atributos privados e métodos públicos;
- **Herança e Generalização** – `ContaCorrente` e `ContaPoupanca` estendem `Conta`;
- **Polimorfismo** – `listarDados()` é sobrescrito nas subclasses;
- **Composição** – `Agencia` contém uma lista de `Conta`;
- **Coleções (ArrayList)** – estrutura dinâmica de armazenamento de objetos.

Se quiser, posso te ajudar a:

- Expandir para incluir `Cliente`, `PessoaFisica`, `PessoaJuridica`;
- Persistir os dados (com arquivos ou banco de dados);
- Criar uma versão gráfica com JavaFX ou interface web com Spring Boot.