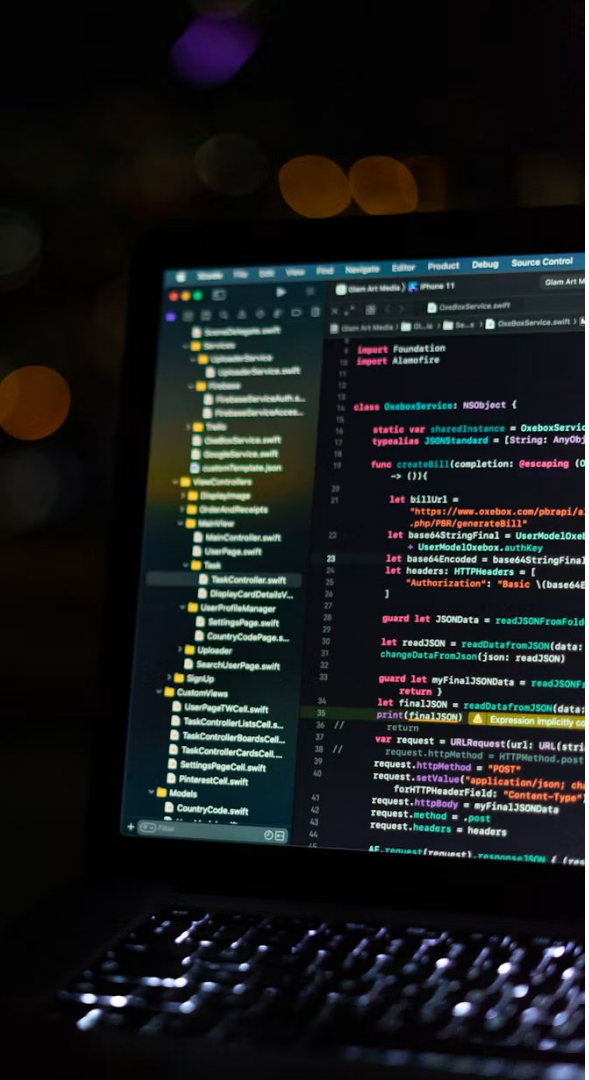


5954018 - Programação Orientada a Objetos

Aula 8 - Associação e Composição de Objetos

Prof. Dr. Denis Martins

DCM | FFCLRP | USP

Objetivos de Aprendizagem

01

Identificar & Implementar

Associações entre objetos em Java.

02

Compreender a Composição

Como uma relação fundamental de todo-parte.

03

Relacionar Conceitos

Associação e composição integradas ao princípio do encapsulamento.

04

Proteger o Estado Interno

Garantir segurança dos objetos contra modificações indevidas.

05

Interpretar Diagramas UML

Leitura e mapeamento de relações básicas na modelagem visual.

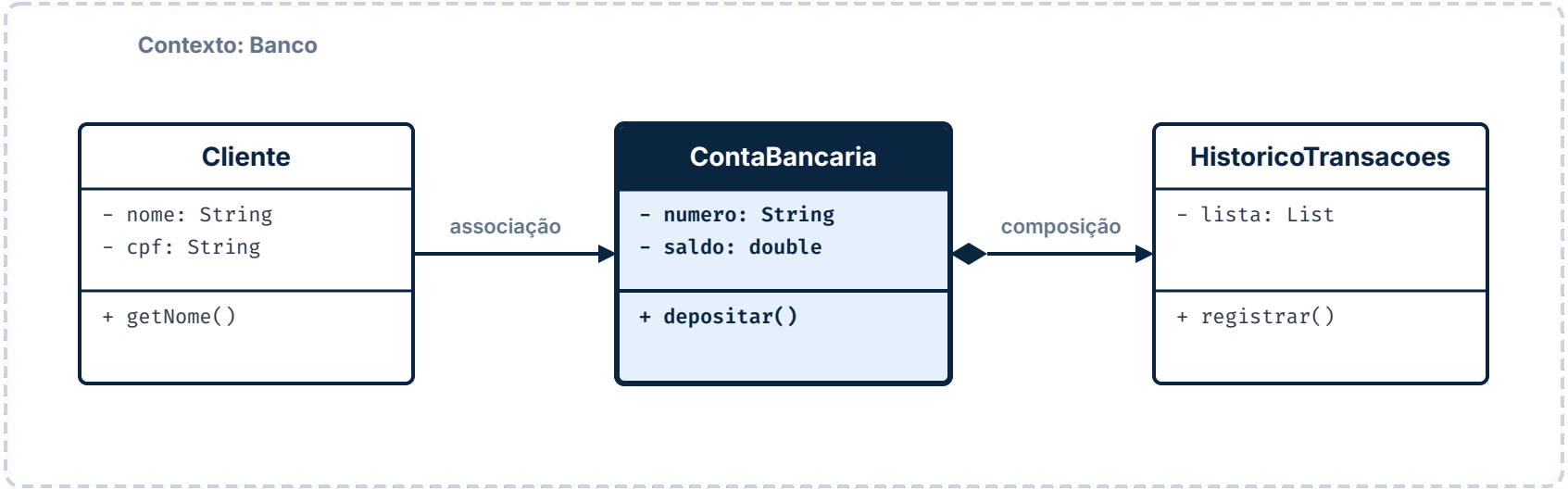
06

Implementar Relacionamentos

Uso prático de referências e arrays de objetos para estruturar o código.

Associação, Composição e Encapsulamento

Como objetos colaboram sem perder o controle sobre seu próprio estado.



Revisando: o estado de uma conta

Por que o atributo `saldo` deve ser declarado como `private`?

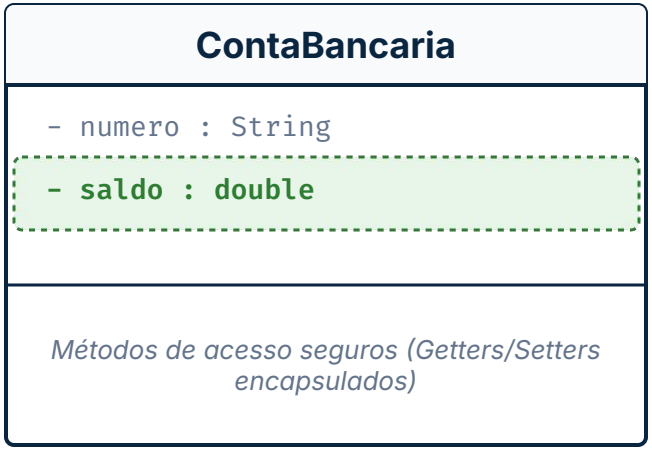
ContaBancaria.java

```java
public class ContaBancaria {

private String numero;

private double saldo;

}
```



✔ Proteção e segurança do estado interno

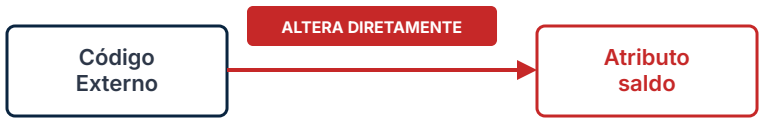
Quem pode alterar o saldo?

O objeto deve controlar as operações que modificam seu próprio estado.

Inadequado (Evite)

Permitir que agentes externos manipulem atributos diretamente quebra o encapsulamento e expõe o estado a inconsistências graves.

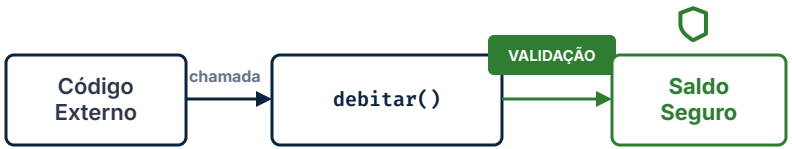
```
conta.saldo = -5000;
```



Adequado (Boa Prática)

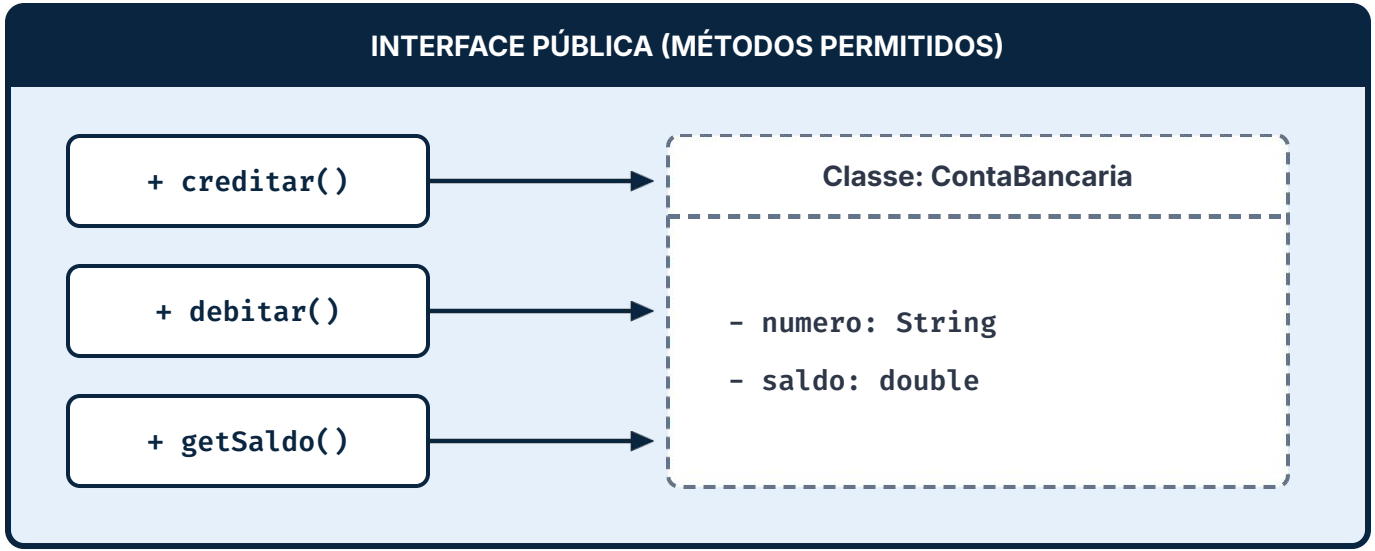
A mudança de estado ocorre estritamente via métodos. A própria classe valida se a operação é segura e permitida.

```
conta.debitar(500);
```



O Princípio do Encapsulamento

O encapsulamento protege o estado interno de um objeto e define estritamente como outros objetos podem interagir com ele. É a união de dados privados com operações bem definidas.



Nem todo atributo precisa de um setter

Considere o código: `public void setSaldo(double saldo) { this.saldo = saldo; }`. Isso realmente protege a conta bancária?

Inadequado (Modificação Direta)

Permite alteração irrestrita do estado interno do objeto a partir de qualquer ponto do sistema, violando regras de integridade.

```
// Estado exposto de forma perigosa
conta.setSaldo(-10000);
```

Adequado (Chamadas Semânticas)

Modificações de estado ocorrem apenas através de operações de negócio explícitas que validam as regras internas da classe.

```
// Comportamento válido e regras de negócio
conta.creditar(500);
conta.debitar(100);
```

Importante: Prefira métodos que representem operações reais do domínio.

Objetos não vivem isolados

Em um sistema bancário real, uma conta existe de forma completamente isolada?

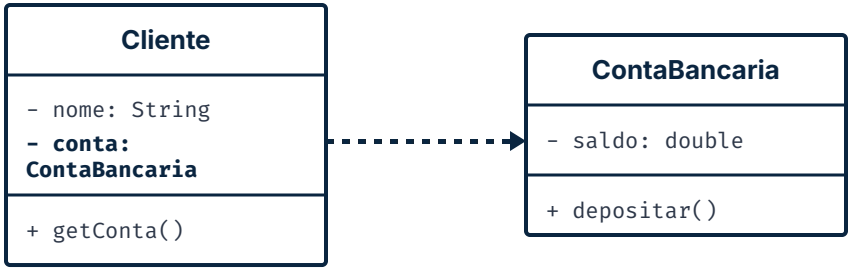
Sistemas Orientados a Objetos são formados por **múltiplos objetos** que colaboram continuamente entre si para realizar tarefas.

Como representar relacionamentos entre Objetos?

Um atributo pode ser um objeto complexo?

Sim, o **tipo** de um atributo pode ser uma classe criada pelo próprio programador, e não apenas tipos primitivos.

```
// Atributo como tipo de classe
public class Cliente {
    private String nome;
    private ContaBancaria conta;
}
```



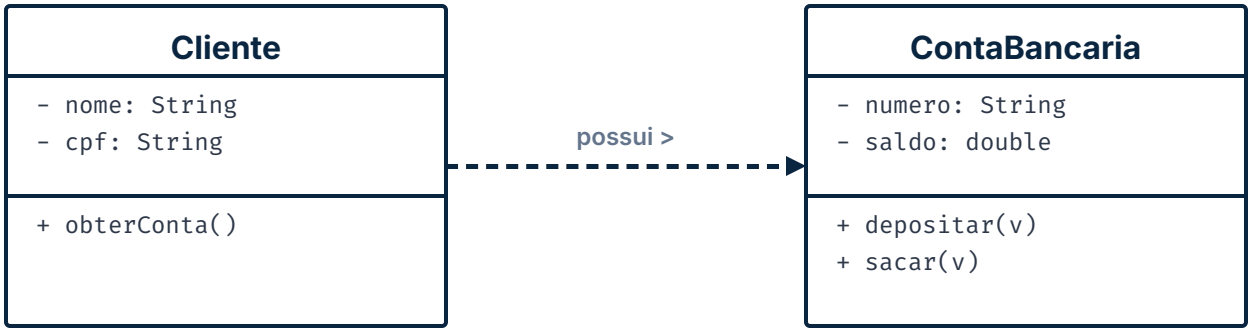
Associação Direta em UML:

A seta tracejada indica uma associação unidirecional do atributo complexo conta apontando diretamente para a classe que define seu comportamento estrutural (ContaBancaria).

Conceito: Associação

Definição: Associação é uma relação estrutural em que um objeto conhece, utiliza ou mantém uma referência direta para outro objeto.

Exemplo: A relação onde um Cliente possui uma ContaBancaria.



Exemplo de Associação

```
public class Cliente {
  private String nome;
  private ContaBancaria conta;
}
```

Atributo
conta

Referência na Memória
Ponteiro/endereço lógico que aponta para o objeto real

Objeto Independente
ContaBancaria (Instanciada no sistema)

Objetos independentes

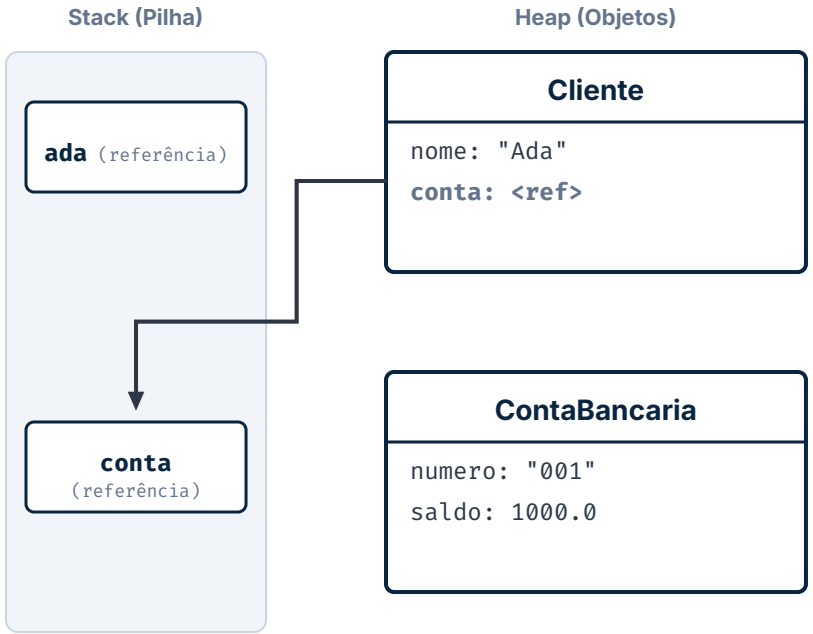
Não existe uma cópia da conta dentro da classe Cliente. Eles mantêm ciclos de vida próprios.

Código de Instanciação

```
Cliente ada = new Cliente("Ada");

ContaBancaria conta = new
    ContaBancaria("001", 1000);

ada.associarConta(conta);
```



Duas referências, um objeto

Código em execução:

```
ada.getConta().creditar(500);
```

PERGUNTA

Qual será o saldo acessado pela variável local `conta`?

Duas referências, um objeto

Código em execução:

```
ada.getConta().creditar(500);
```

PERGUNTA

Qual será o saldo acessado pela variável local `conta`?

Resposta (Saldo)

R$ 1.500

Ambas as referências apontam para o mesmo objeto na memória.

Associação não elimina encapsulamento

Exposição direta (Inadequado)

Permite que qualquer classe externa altere a referência de associação diretamente, ignorando regras de negócio e validações necessárias.

```
public class Cliente {
    public ContaBancaria conta;
}
```

Relacionamento controlado (Melhor)

O uso de atributos privados associados a métodos de controle garante que novas associações passem por validações de segurança.

```
public class Cliente {
   private ContaBancaria conta;
   public void associarConta(ContaBancaria conta) {
        if (conta != null)
       this.conta = conta;
   }
}
```

O método permite validar a associação antes de concretizá-la.

Quem é responsável por quê?

Cenário de execução

ada.getConta().creditar(500);

Quem deve validar se `500` é um valor válido para a transação?

- ○ **Cliente**
- ○ **Banco**
- ○ **ContaBancaria**

Quem é responsável por quê?

Cenário de execução

`ada.getConta().creditar(500);`

Quem deve validar se `500` é um valor válido para a transação?

○ Cliente

Apenas consome o serviço de crédito chamando o método.

Não deve duplicar a lógica ou deter o conhecimento das regras de validação internas.

○ Banco

Gerencia a orquestração e fluxo das contas de forma ampla.

Controlar e validar o estado interno de cada conta diretamente no banco viola o encapsulamento.

✓ ContaBancaria

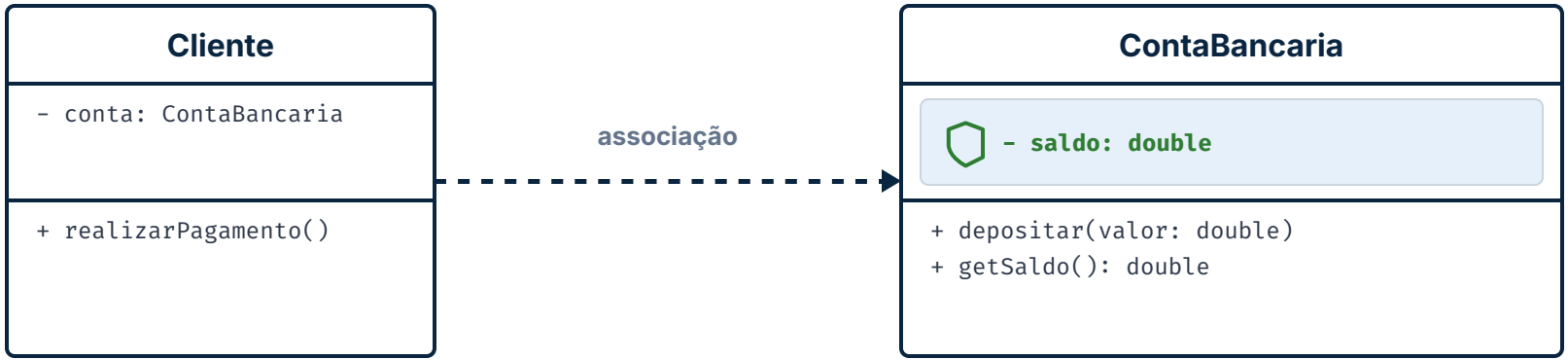
Resposta Correta

Dona direta do estado (**saldo**).

Cada objeto deve ser o único responsável por proteger e controlar as regras de negócio associadas ao seu próprio estado interno.

Associação + Encapsulamento

A Associação define quem conhece quem no sistema. O Encapsulamento define como essa interação acontece de forma segura.



Organização com Pacotes em Java

Como agrupar classes relacionadas, evitar conflitos e controlar a visibilidade

Para que servem os Pacotes?

Organização lógica

- Agrupam classes que compartilham do mesmo domínio de negócio ou utilidade.

Evitam conflitos de nomes

- Permitem a existência de classes homônimas em pacotes distintos (ex: **banco.Cliente** vs **loja.Cliente**).

Controle de visibilidade

- Ajudam a restringir o acesso a membros e classes que não devem ser expostos fora do escopo.

Estrutura de Diretórios & Código

```
src/
├── banco/
│   ├── ContaBancaria.java
│   ├── ContaPoupanca.java
│   └── Banco.java
└── cliente/
    └── Cliente.java
```

```java
package banco;
public class ContaBancaria { /* ... */ }
```

```java
import banco.ContaBancaria;
public class Cliente { /* usa ContaBancaria */ }
```

ATIVIDADE PRÁTICA

Atividade Prática 1

Objetivo: Implementar o relacionamento **Cliente** —— **ContaBancaria** em Java.

Tarefas a Desenvolver:

1. Completar a implementação da classe **ContaBancaria**.

2. Criar a classe **Cliente**.

3. Associar a conta ao cliente de forma segura (garantir encapsulamento).

4. Realizar operações financeiras através do cliente no método principal.

ATIVIDADE PRÁTICA

Atividade Prática 1

```
public class ContaBancaria {
    private String numero;
    private double saldo;

    public void creditar(double valor) {
        // TODO: implementar
    }

    public void debitar(double valor) {
        // TODO: implementar
    }

    public double getSaldo() {
        // TODO: implementar
    }
}
```

Regra de Negócio 1

O valor da transação deve ser sempre **> 0**.

Regra de Negócio 2

O saldo nunca pode se tornar negativo após uma operação (deve ser sempre **>= 0**).

ATIVIDADE PRÁTICA

Atividade Prática 1

```
public class Cliente {
private String nome;
private ContaBancaria conta;
public void associarConta(ContaBancaria
conta) {}
public ContaBancaria getConta() {}
}
```

ASSOCIAÇÃO

ANÁLISE ESTRUTURAL

Onde está a associação?

A associação estrutural ocorre quando uma classe declara um atributo de referência que aponta para outra classe do domínio.

ATIVIDADE PRÁTICA

Atividade Prática 1: Desafio

Cenário de Teste (Código)

```
Cliente ada = new Cliente("Ada");
ContaBancaria conta = new
        ContaBancaria("001", 1000);
ada.associarConta(conta);
```

Desafio Técnico

Sem usar a variável local conta diretamente, escreva o código para realizar as seguintes operações:

- **Crédito de R$ 500 (+500)**
- **Débito de R$ 200 (-200)**

Dica: Navegue exclusivamente a partir do objeto ada utilizando ada.getConta().

O que aprendemos na Atividade 1?

01 . INDEPENDÊNCIA

Objetos mantêm-se independentes na memória.

02 . COMUNICAÇÃO

O referenciamento permite a comunicação entre as instâncias.

03 . EXCLUSIVIDADE

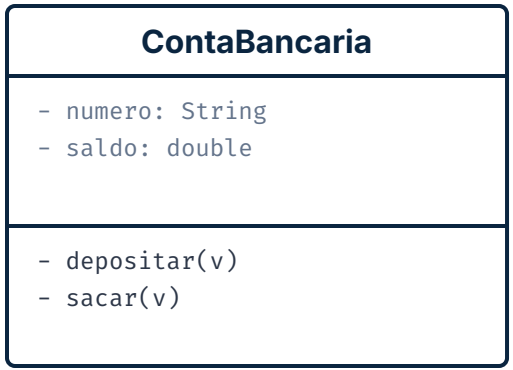
A responsabilidade de validar o saldo continua sendo exclusiva da conta.

04 . ENCAPSULAMENTO

A associação permite colaboração, mas não quebra o encapsulamento.

Um novo problema

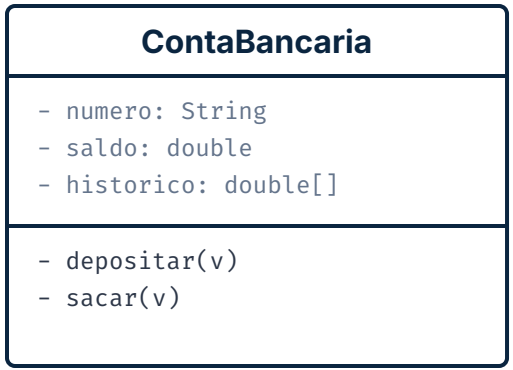
Cenário: O sistema evoluiu e agora precisamos manter um histórico detalhado de transações da conta (ex: `+500, -200, +100`).



Pergunta: Onde essas operações deveriam ser armazenadas estruturalmente?

Um novo problema

Cenário: O sistema evoluiu e agora precisamos manter um histórico detalhado de transações da conta (ex: `+500, -200, +100`).



Pergunta: Onde essas operações deveriam ser armazenadas estruturalmente?

Um novo problema

Cenário: O sistema evoluiu e agora precisamos manter um histórico detalhado de transações da conta (ex: `+500, -200, +100`).

Pergunta: Onde essas operações deveriam ser armazenadas estruturalmente?

Histórico de Transações

HistoricoTransacoes.java

```
public class HistoricoTransacoes {

    private String[] operacoes;
    private double quantidade;

    public HistoricoTransacoes() {
        operacoes = new String[10];
        quantidade = 0;
    }

    public void registrar(String operacao) {
        // TODO
    }

    public void exibir() {
        // TODO
    }
}
```

Reflexão Arquitetural

Este objeto é associado ou faz parte de ContaBancaria?

Esse histórico recém-criado é apenas mais um objeto associado de forma independente, ou ele faz parte estrutural da própria conta bancária?

Esta reflexão introduz o conceito de Composição (relação todo-parte).

Conceito: Composição

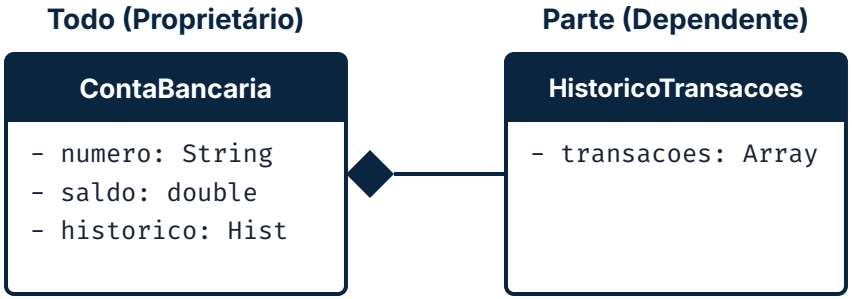
Relação de Composição

A composição é uma relação forte do tipo todo-parte (“tem um”), onde um objeto contém e controla exclusivamente uma parte.

Se o objeto principal for destruído, suas partes também serão eliminadas.

No diagrama UML, essa relação é representada por um losango preenchido apontando do "Todo" (ContaBancaria) para a "Parte" (HistoricoTransacoes).

Visualmente, a ContaBancaria encapsula atributos como número e saldo, além de conter a lista de transações gerenciada exclusivamente pelo HistoricoTransacoes.



Regra de Ciclo de Vida:

- O **losango preenchido** aponta para o Todo (ContaBancaria).
- A parte não tem existência independente fora de seu proprietário.
- Se o objeto "Todo" for excluído, todas as suas "Partes" gerenciadas de forma exclusiva também serão eliminadas.

Exemplo: Composição

```java
public class ContaBancaria {

    // Atributo: Parte interna
    private HistoricoTransacoes historico;
    private double saldo;

    public ContaBancaria() {
        // Inicializa atributos básicos
        this.saldo = 0.0;

        // Instanciação e controle
        this.historico = new
              HistoricoTransacoes();
    }
}
```

Na implementação em Java, a própria classe **"Todo" (ContaBancaria)** é responsável por instanciar e gerenciar o ciclo de vida da sua **"Parte" interna (HistoricoTransacoes)**.

No construtor da ContaBancaria, inicializamos os atributos básicos e instanciamos o histórico diretamente usando a instrução new HistoricoTransacoes().

Garantia de Composição

Isso garante que, sem a existência da conta, o histórico correspondente não exista, estabelecendo a relação de dependência estrita e o controle absoluto do ciclo de vida.

Associação × Composição

Associação (Ciclos Independentes)

Os objetos têm ciclos de vida independentes. A classe apenas recebe e usa um vínculo externo.

Exemplo: `Cliente` *recebe referência para* `ContaBancaria`.

Composição (Relação Todo-Parte)

Relação forte onde a parte é criada, rigidamente controlada e destruída diretamente pelo Todo.

Exemplo: `ContaBancaria` *instancia seu* `HistoricoTransacoes`.

Critério	Associação	Composição
Independência	Alta. Os objetos possuem ciclos de vida completamente independentes.	Nenhuma. A existência da parte depende estritamente do ciclo de vida do Todo.
Gerenciamento	Usa vínculos externos. Recebe referências criadas fora de seu escopo.	Gerencia partes internamente. Instancia e controla suas partes de forma privada.
Esquema UML	`Cliente ——— ContaBancaria`	`Conta ◆—— HistoricoTransacoes`

Devemos expor o histórico?

```java
// Quebra de Encapsulamento
public class ContaBancaria {
  private String[] historico;

  // RETORNA A REFERÊNCIA DIRETA!
  public String[] getHistorico() {
    return this.historico;
  }
}
```

ALERTA DE SEGURANÇA

Risco de Segurança

Exposição de Referência Interna

Retornar a referência direta de uma parte interna através de um método de acesso representa um sério risco de segurança e uma grave quebra de encapsulamento.

Se expusermos essa referência, o código externo poderá contornar as regras da conta bancária e **alterar o histórico de forma arbitrária**, como por exemplo, apagar um histórico de fraudes.

O sistema externo não deve, sob nenhuma circunstância, manipular diretamente as estruturas internas da conta.

Encapsulamento: Escondendo detalhes internos

```
// Solução Segura (Delegação)
public class ContaBancaria {
  private HistoricoTransacoes historico;
  public void exibirExtrato() {
    this.historico.imprimir();
  }
}
```

Boa Prática

Encapsulamento Rigoroso

A classe **ContaBancaria** fornece métodos de interface que ocultam os detalhes de implementação. O código externo faz chamadas simples e seguras, reduzindo o acoplamento do sistema.

Fluxo Visual de Execução Seguro

Código Externo ➔ **exibirExtrato()** ➔ **ContaBancaria** ➔ **HistoricoTransacoes**

Objetos colaborando internamente

```java
// Validação, encapsulamento e delegação
public void creditar(double valor) {
    if (valor > 0) {
        this.saldo += valor; // Alteração
        this.historico.registrar(valor); // Delegação
    }
}
```

Fluxo de Execução & Conceitos

- **Validação:** O método valida se o valor é positivo para garantir a integridade antes da operação.
- **Encapsulamento:** O saldo e o histórico permanecem estritamente privados dentro do objeto.
- **Composição:** O histórico existe apenas dentro da conta, definindo a relação de dependência existencial.
- **Colaboração:** A conta delega o registro ao histórico, coordenando a ação conjunta sem expor detalhes.

ENCAPSULAMENTO • COMPOSIÇÃO • COLABORAÇÃO

Pilares essenciais do design orientado a objetos agindo em perfeita harmonia.

ATIVIDADE PRÁTICA

Atividade Prática 2

Objetivo: Desenvolver um mini sistema bancário integrando as estruturas de dados e regras de negócio da orientação a objetos.

```
// Arquitetura do Sistema
class Banco {
  private ContaBancaria[] contas;
}
class ContaBancaria {
  private HistoricoTransacoes historico;
}
```

Você deve construir as seguintes relações:

- **Associação:** O **Banco** gerencia múltiplas instâncias de **ContaBancaria**.
- **Composição:** Cada conta controla exclusivamente seu **HistoricoTransacoes**.
- **Encapsulamento:** Proteção rigorosa de saldos e listas internas.

CONCLUSÃO

Design Orientado a Objetos na Prática

Combinando responsabilidades claras, estado protegido e relacionamentos bem definidos para uma arquitetura robusta.

1. Encapsulamento

Protege o estado interno dos objetos e controla rigorosamente como ele pode ser alterado.

```
// Estado Protegido
```

2. Relações & Estrutura

Associação permite a colaboração de objetos independentes. **Composição** define um forte vínculo todo-parte.

```
Cliente → Contas[]
```

3. Colaboração Segura

Objetos colaboram de forma simples por **métodos públicos bem definidos**, ocultando seus detalhes de implementação.

```
public void creditar()
```

Em POO, não basta criar classes. É necessário definir quem é responsável por quê e como os objetos colaboram entre si. Assim, o código fica mais fácil de manter ao longo do tempo.

Associação e Composição: Exemplo Pokémon

Diferenciando o forte vínculo de posse (Composição) da colaboração independente de ciclo de vida (Associação)


```java
// Composição: cria seu próprio Time
public class Treinador {
  private String nome;
  private Time time;

  public Treinador(String nome) {
    this.nome = nome;
    this.time = new Time();
  }
}
```

```java
// Associação: recebe instâncias externas
public class Time {
  private Pokemon[] pokemon = new Pokemon[6];
  private int qnt = 0;

  public void adicionarPokemon(Pokemon pokemon) {
    if (qnt < pokemon.length) {
      pokemon[qnt++] = pokemon;
    }
  }
}
```

COMPOSIÇÃO
Forte relação **todo-parte**. O Time não existe sem o Treinador.

ASSOCIAÇÃO
Objetos **independentes**. Pokémon existem sem um Time.

Dúvidas e Discussão