

Composição, Agregação e Variáveis de Instância Final

1. Introdução

Na Programação Orientada a Objetos (POO), os objetos frequentemente precisam se relacionar uns com os outros. Uma classe pode utilizar objetos de outra classe para representar conceitos mais complexos.

Por exemplo:

- Um **Pedido** possui vários **ItensPedido**.
- Uma **Universidade** possui vários **Professores**.
- Um **Carro** possui um **Motor**.
- Um **Cliente** possui um **Endereço**.

Esses relacionamentos podem ser representados principalmente por meio de **agregação** e **composição**.

Outro conceito importante é o uso de **variáveis de instância final**, que permite definir atributos que, depois de inicializados, não podem receber outra referência ou outro valor.

2. Agregação

A **agregação** representa uma relação entre objetos em que um objeto possui ou utiliza outro objeto, mas os objetos possuem **ciclos de vida independentes**.

Podemos pensar na agregação como uma relação:

"tem um" ou "possui um", mas o objeto relacionado pode existir independentemente.

Exemplo

Imagine uma universidade que possui professores.

Um professor pode existir mesmo que a universidade deixe de existir. Da mesma forma, um professor pode ser associado a outra universidade.

Podemos representar essa situação em Java:

```
public class Professor {  
  
    private String nome;  
  
    public Professor(String nome) {
```

```

        this.nome = nome;
    }

    public String getNome() {
        return nome;
    }
}

```

A classe `Universidade` pode manter uma lista de professores:

```

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

public class Universidade {

    private String nome;
    private List<Professor> professores;

    public Universidade(String nome) {
        this.nome = nome;
        this.professores = new ArrayList<>();
    }

    public void adicionarProfessor(Professor professor) {
        professores.add(professor);
    }

    public void listarProfessores() {
        for (Professor professor : professores) {
            System.out.println(professor.getNome());
        }
    }
}

```

A utilização poderia ser:

```

public class Main {

    public static void main(String[] args) {

        Professor p1 = new Professor("João");
        Professor p2 = new Professor("Maria");

        Universidade universidade =
            new Universidade("Universidade Federal");

        universidade.adicionarProfessor(p1);
        universidade.adicionarProfessor(p2);
    }
}

```

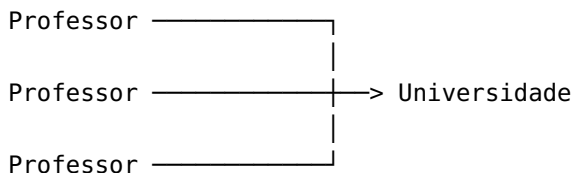
```

        universidade.listarProfessores();
    }
}

```

Observe que os objetos `Professor` foram criados **fora** da classe `Universidade`.

Isso é uma característica importante da agregação:



Os professores não dependem da existência da universidade para existirem.

3. Composição

A **composição** é uma relação mais forte entre objetos.

Na composição, o objeto principal é responsável pela existência dos objetos que fazem parte dele.

Podemos interpretar como:

"é parte de".

Um exemplo clássico é um **Pedido** e seus **ItensPedido**.

Um item de pedido faz sentido dentro do contexto de um pedido. Se o pedido for eliminado, seus itens também deixam de fazer sentido.

Exemplo

Primeiramente, criamos a classe `ItemPedido`:

```

public class ItemPedido {

    private String produto;
    private int quantidade;

    public ItemPedido(String produto, int quantidade) {
        this.produto = produto;
        this.quantidade = quantidade;
    }

    public void exibir() {

```

```

        System.out.println(
            "Produto: " + produto +
            " | Quantidade: " + quantidade
        );
    }
}

```

Agora podemos criar a classe `Pedido`:

```

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

public class Pedido {

    private List<ItemPedido> itens;

    public Pedido() {
        itens = new ArrayList<>();
    }

    public void adicionarItem(String produto, int quantidade) {

        ItemPedido item =
            new ItemPedido(produto, quantidade);

        itens.add(item);
    }

    public void exibirItens() {

        for (ItemPedido item : itens) {
            item.exibir();
        }
    }
}

```

Observe que o objeto `ItemPedido` é criado **dentro da classe** `Pedido`:

```

ItemPedido item =
    new ItemPedido(produto, quantidade);

```

Isso caracteriza uma relação de composição.

Exemplo de utilização:

```

public class Main {

```

```

public static void main(String[] args) {

    Pedido pedido = new Pedido();

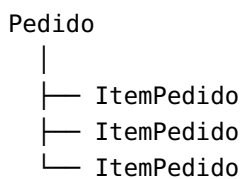
    pedido.adicionarItem("Notebook", 1);
    pedido.adicionarItem("Mouse", 2);
    pedido.adicionarItem("Teclado", 1);

    pedido.exibirItens();

}
}

```

Podemos representar essa relação da seguinte maneira:



Os `ItemPedido` pertencem ao `Pedido`.

4. Diferença entre Agregação e Composição

A principal diferença está no **ciclo de vida dos objetos**.

Característica	Agregação	Composição
Relação	"Tem um"	"É parte de"
Dependência	Menor	Maior
Ciclo de vida	Independente	Dependente
Objeto pode existir sozinho?	Sim	Normalmente não
Exemplo	Universidade → Professor	Pedido → ItemPedido

Uma forma simples de memorizar:

Agregação:

O objeto faz parte de outro, mas consegue existir sozinho.

Composição:

O objeto faz parte de outro e depende dele para seu significado ou ciclo de vida.

5. Variáveis de Instância `final`

Em Java, podemos declarar uma variável de instância utilizando o modificador `final`.

Por exemplo:

```
public class Produto {  
  
    private final int codigo;  
    private String descricao;  
}
```

Nesse caso, `codigo` precisa ser inicializado e, depois disso, não poderá receber outro valor.

Uma variável `final` pode ser entendida como uma variável que recebe uma atribuição definitiva.

```
public class Produto {  
  
    private final int codigo;  
    private String descricao;  
  
    public Produto(int codigo, String descricao) {  
        this.codigo = codigo;  
        this.descricao = descricao;  
    }  
}
```

Depois que o construtor executar:

```
Produto produto = new Produto(10, "Notebook");
```

o atributo:

```
codigo
```

não poderá ser alterado.

Por exemplo, isto causaria erro de compilação:

```
produto.codigo = 20;
```

Além disso, como `codigo` é `private`, seu acesso direto também não é permitido fora da classe.

6. Por que utilizar `final` em atributos?

O modificador `final` é bastante utilizado quando determinado atributo deve ser definido **uma única vez**.

Alguns exemplos:

- código de identificação;
- número de matrícula;
- CPF associado a um cadastro;
- data de criação de um objeto;
- identificador de uma entidade;
- dependências obrigatórias de uma classe.

Por exemplo:

```
import java.time.LocalDate;

public class Cliente {

    private final String cpf;
    private final LocalDate dataCadastro;

    private String nome;

    public Cliente(String cpf, String nome) {
        this.cpf = cpf;
        this.nome = nome;
        this.dataCadastro = LocalDate.now();
    }
}
```

Nesse exemplo, depois que o cliente foi criado:

```
Cliente cliente =
    new Cliente("12345678900", "Carlos");
```

não podemos alterar:

```
cpf
dataCadastro
```

Enquanto o atributo:

```
nome
```

pode ser alterado posteriormente.

7. `final` em referências para objetos

Um ponto importante é que `final` em uma referência de objeto **não torna o objeto imutável**.

Considere:

```
public class Pessoa {  
  
    private final Endereco endereco;  
  
    public Pessoa(Endereco endereco) {  
        this.endereco = endereco;  
    }  
}
```

A referência `endereco` não poderá apontar para outro objeto:

```
this.endereco = outroEndereco; // Erro
```

Porém, se a classe `Endereco` permitir alterações, seus dados ainda poderão ser modificados:

```
endereco.setCidade("Goiânia");
```

Portanto:

`final` impede a alteração da referência, mas não necessariamente a alteração do objeto referenciado.

8. `final` e composição

O uso de `final` é muito comum em composição.

Imagine uma classe `Pedido` que possui um `Cliente`.

```
public class Pedido {  
  
    private final Cliente cliente;  
  
    public Pedido(Cliente cliente) {  
        this.cliente = cliente;  
    }  
}
```

```
}  
}
```

Nesse caso, o pedido recebe um cliente no momento de sua criação.

Depois disso, não podemos fazer:

```
pedido.cliente = outroCliente;
```

Isso ajuda a preservar uma regra importante do domínio:

Um pedido, depois de criado, possui um cliente definido.

Outro exemplo:

```
public class Carro {  
  
    private final Motor motor;  
  
    public Carro(Motor motor) {  
        this.motor = motor;  
    }  
}
```

O atributo `motor` é inicializado no construtor:

```
public Carro(Motor motor) {  
    this.motor = motor;  
}
```

Depois disso, a referência não pode ser substituída.

9. Inicialização de atributos `final`

Um atributo de instância declarado como `final` precisa ser inicializado uma vez.

Uma forma comum é utilizar o construtor:

```
public class Conta {  
  
    private final int numero;  
    private double saldo;  
  
    public Conta(int numero) {
```

```
        this.numero = numero;  
        this.saldo = 0;  
    }  
}
```

Essa abordagem é muito utilizada porque garante que o objeto seja criado já em um estado válido.

Por exemplo:

```
Conta conta = new Conta(1001);
```

A conta obrigatoriamente terá um número.

10. `final` e imutabilidade

É importante não confundir os conceitos.

Uma classe pode ter atributos `final` e ainda assim ser mutável.

Exemplo:

```
public class Cliente {  
  
    private final String cpf;  
    private String nome;  
  
    public Cliente(String cpf, String nome) {  
        this.cpf = cpf;  
        this.nome = nome;  
    }  
  
    public void setNome(String nome) {  
        this.nome = nome;  
    }  
}
```

O `cpf` não pode ser alterado, mas o `nome` pode:

```
Cliente cliente =  
    new Cliente("12345678900", "João");  
  
cliente.setNome("Carlos");
```

Portanto:

`final` ≠ objeto imutável

O `final` garante que determinada variável não receba uma nova atribuição depois de inicializada.

11. Exemplo completo: Pedido, Cliente e Itens

Podemos combinar os conceitos apresentados em um pequeno modelo de domínio.

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

public class Pedido {

    private final int numero;
    private final Cliente cliente;

    private final List<ItemPedido> itens;

    public Pedido(int numero, Cliente cliente) {

        this.numero = numero;
        this.cliente = cliente;
        this.itens = new ArrayList<>();
    }

    public void adicionarItem(String produto, int quantidade) {

        ItemPedido item =
            new ItemPedido(produto, quantidade);

        itens.add(item);
    }
}
```

Nesse exemplo temos:

Variáveis `final`

```
private final int numero;
private final Cliente cliente;
private final List<ItemPedido> itens;
```

Esses atributos não podem receber uma nova atribuição depois de inicializados.

Composição

Os objetos `ItemPedido` são criados pelo próprio `Pedido`:

```
ItemPedido item =  
    new ItemPedido(produto, quantidade);
```

Existe uma relação forte entre:

```
Pedido  
|  
├── ItemPedido  
├── ItemPedido  
└── ItemPedido
```

Agregação

O `Cliente` pode existir independentemente do `Pedido`.

Por exemplo:

```
Cliente cliente =  
    new Cliente("12345678900", "João");  
  
Pedido pedido =  
    new Pedido(1001, cliente);
```

O cliente foi criado fora do pedido e pode ser utilizado em outros contextos.

12. Exemplo aplicado a um sistema empresarial

Imagine um sistema de vendas.

Temos:

```
Cliente  
|  
└── Pedido  
    |  
    ├── ItemPedido  
    ├── ItemPedido  
    └── ItemPedido
```

Podemos interpretar as relações da seguinte maneira:

Cliente → Pedido

Agregação.

O cliente existe independentemente dos pedidos.

Pedido → ItemPedido

Composição.

Os itens pertencem ao pedido.

Pedido.numero

Variável de instância `final`.

O número do pedido é definido na criação e não deve ser substituído.

Essa combinação é bastante comum em aplicações Java, especialmente em sistemas desenvolvidos utilizando conceitos de POO, MVC, Spring Boot e JPA.

13. Boas práticas

Ao utilizar composição, agregação e atributos `final`, algumas boas práticas devem ser observadas:

1. Utilize composição quando houver dependência de ciclo de vida

Exemplo:

```
Pedido → ItemPedido
```

2. Utilize agregação quando os objetos puderem existir independentemente

Exemplo:

```
Universidade → Professor
```

3. Utilize `final` para atributos que não devem ser reatribuídos

Exemplo:

```
private final int codigo;
```

4. Prefira inicializar atributos obrigatórios no construtor

```
public Produto(int codigo) {  
    this.codigo = codigo;  
}
```

5. Não confunda `final` com imutabilidade

Um objeto referenciado por uma variável `final` ainda pode ser mutável.

14. Conclusão

Agregação, composição e variáveis de instância `final` são conceitos importantes para desenvolver classes mais organizadas e representar corretamente relacionamentos entre objetos.

A **agregação** representa uma relação em que os objetos possuem ciclos de vida independentes.

A **composição** representa uma relação mais forte, na qual um objeto é parte integrante de outro e normalmente depende dele.

Já o modificador `final` permite definir atributos que, depois de inicializados, não podem receber uma nova atribuição.

Em conjunto, esses conceitos ajudam a construir modelos de objetos mais seguros e coerentes:

```
Agregação  
  ↓  
Objetos independentes  
  
Composição  
  ↓  
Objetos dependentes  
  
final  
  ↓  
Referência/valor não pode ser reatribuído
```

Dominar esses conceitos é fundamental para avançar em Java e compreender posteriormente temas como **encapsulamento, herança, polimorfismo, interfaces, classes imutáveis, padrões de projeto e modelagem de entidades em aplicações Spring Boot**.