

Hierarquia: Agregação, Composição e Herança

As três formas fundamentais de relacionar classes em Java

1. Introdução

Ao projetar um sistema orientado a objetos, praticamente toda relação entre duas classes pode ser descrita por uma destas três formas: **herança** (relação "é um"), **composição** (relação "tem um" forte) e **agregação** (relação "tem um" fraca). Juntas, elas formam a base para construir **hierarquias** de classes bem organizadas. Este texto explica cada uma dessas relações, compara suas diferenças e apresenta um exemplo integrado em Java que combina as três.

2. Herança — Relação "É um" (is-a)

A **herança** representa uma relação de **especialização**: uma subclasse é um tipo mais específico de sua superclasse, herdando seus atributos e métodos por meio da palavra-chave `extends`. A pergunta que caracteriza essa relação é sempre "X é um Y?" — por exemplo, "um Carro é um Veículo".

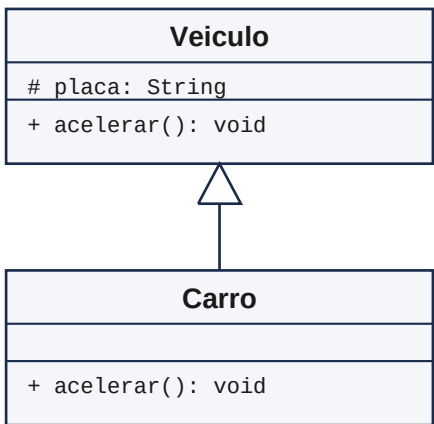


Figura 1 — Herança: Carro é um Veiculo (seta vazada apontando para a superclasse).

```

public class Veiculo {
    protected String placa;

    public Veiculo(String placa) {
        this.placa = placa;
    }

    public void acelerar() {
        System.out.println(placa + " acelerando.");
    }
}

public class Carro extends Veiculo { // Carro É UM Veiculo
    public Carro(String placa) {
        super(placa);
    }

    @Override
    public void acelerar() {
        System.out.println("Carro " + placa + " acelerando suavemente.");
    }
}

```

3. Composição — Relação "Tem um" Forte

A **composição** representa uma relação de **posse exclusiva**: o objeto "parte" não existe fora do objeto "todo" e é destruído junto com ele. A pergunta característica é "X **tem um** Y, e Y não existe sem X?" — por exemplo, "um Carro **tem um** Motor, que não existe fora dele". No diagrama UML, é representada por um losango **preenchido**.

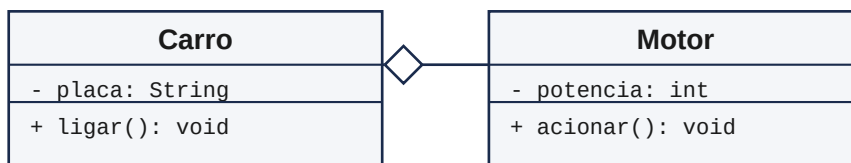


Figura 2 — Composição: o Motor é criado e destruído junto com o Carro (losango preenchido).

```

public class Motor {
    private int potencia;

    public Motor(int potencia) {
        this.potencia = potencia;
    }

    public void acionar() {
        System.out.println("Motor de " + potencia + " cv acionado.");
    }
}

public class Carro extends Veiculo {
    private final Motor motor; // Carro TEM UM Motor (composição)

    public Carro(String placa, int potenciaMotor) {
        super(placa);
        this.motor = new Motor(potenciaMotor); // criado junto com o Carro
    }

    public void ligar() {
        motor.acionar();
    }
}

```

4. Agregação — Relação "Tem um" Fraca

A **agregação** também é uma relação "tem um", mas com posse **fraca**: o objeto "parte" tem vida própria e pode existir e ser reaproveitado independentemente do "todo". A pergunta característica é "X **tem um** Y, mas Y continua existindo mesmo sem X?" — por exemplo, "uma Frota **tem** Veículos, mas os veículos existem mesmo se a frota deixar de existir". No diagrama UML, é representada por um losango **vazado**.

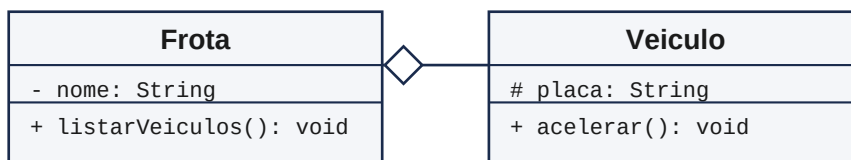


Figura 3 — Agregação: um Veiculo pode existir e ser reaproveitado fora de uma Frota específica.

```

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

public class Frota {
    private String nome;
    private List<Veiculo> veiculos = new ArrayList<>(); // agregação de Veiculos

    public Frota(String nome) {
        this.nome = nome;
    }

    public void adicionarVeiculo(Veiculo v) { // recebe um objeto já existente
        veiculos.add(v);
    }

    public void listarVeiculos() {
        for (Veiculo v : veiculos) {
            v.acelerar();
        }
    }
}

// Uso:
Carro carro = new Carro("ABC-1234", 120); // o Carro existe independentemente
Frota frota = new Frota("Frota Regional");
frota.adicionarVeiculo(carro); // apenas passa a ser referenciado pela frota

```

5. Comparando as Três Relações

Aspecto	Herança	Composição	Agregação
Pergunta-chave	X é um Y?	X tem um Y (exclusivo)?	X tem um Y (compartilhável)?
Palavra-chave em Java	extends	Atributo criado com new no construtor	Atributo recebido de fora
Ciclo de vida	Subclasse incorpora a superclasse	Parte morre com o todo	Parte é independente do todo
Símbolo UML	Seta vazada	Losango preenchido	Losango vazado
Exemplo	Carro extends Veiculo	Carro tem um Motor	Frota tem Veiculos

6. Exemplo Integrado: as Três Relações Juntas

O exemplo a seguir reúne as três relações em um único domínio de frota de veículos: Carro e Caminhao **herdam** de Veiculo; cada Carro é **composto** por um Motor (criado junto com ele); e a Frota **agrega** vários objetos Veiculo, que continuam existindo independentemente dela.

```

public class Veiculo {
    protected String placa;

    public Veiculo(String placa) {
        this.placa = placa;
    }

    public void acelerar() {
        System.out.println(placa + " acelerando.");
    }
}

public class Motor {
    private int potencia;
    public Motor(int potencia) { this.potencia = potencia; }
    public void acionar() {
        System.out.println("Motor de " + potencia + " cv acionado.");
    }
}

public class Carro extends Veiculo {
    private final Motor motor;

    public Carro(String placa, int potenciaMotor) {
        super(placa);
        this.motor = new Motor(potenciaMotor);
    }

    public void ligar() { motor.acionar(); }
}

public class Caminhao extends Veiculo {
    private double capacidadeCarga;

    public Caminhao(String placa, double capacidadeCarga) {
        super(placa);
        this.capacidadeCarga = capacidadeCarga;
    }
}

```

```

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

public class Frota {
    private List<Veiculo> veiculos = new ArrayList<>(); // agregação

    public void adicionarVeiculo(Veiculo v) {
        veiculos.add(v);
    }

    public void acelerarTodos() {
        for (Veiculo v : veiculos) {
            v.acelerar(); // polimorfismo: cada Veiculo se comporta à sua maneira
        }
    }
}

public class Main {
    public static void main(String[] args) {
        Carro carro = new Carro("ABC-1234", 120);
        Caminhao caminhao = new Caminhao("XYZ-9876", 8000);

        Frota frota = new Frota();
        frota.adicionarVeiculo(carro);
        frota.adicionarVeiculo(caminhao);

        frota.acelerarTodos();
        carro.ligar();
        // Se "frota" for destruída, carro e caminhao continuam existindo (agregação).
        // Se "carro" for destruído, seu Motor também deixa de existir (composição).
    }
}

```

7. Como Escolher a Relação Certa

Uma boa regra prática: use **herança** quando existir uma relação clara de especialização ("é um tipo de"), evitando herança apenas para reaproveitar código sem essa relação fazer sentido conceitualmente. Use **composição** quando a parte não tiver significado fora do todo e seu ciclo de vida depender totalmente dele. Use **agregação** quando a parte tiver existência própria e puder, inclusive, ser compartilhada entre diferentes "todos". Em caso de dúvida entre herança e composição, um princípio de design conhecido resume bem a escolha: *"prefira composição a herança"*, pois relações "tem um" tendem a gerar sistemas mais flexíveis do que hierarquias profundas de herança.

8. Conclusão

Herança, composição e agregação são as três ferramentas fundamentais para construir hierarquias e relacionamentos entre classes em Java. Enquanto a herança modela especializações ("é um"), composição e agregação modelam posse ("tem um"), variando apenas na força do vínculo entre o todo e suas partes. Combinadas, como demonstrado no exemplo da frota de veículos, essas três relações permitem representar praticamente qualquer estrutura de domínio de forma clara, organizada e alinhada aos princípios da Programação Orientada a Objetos.