

Herança: Simples, Múltipla e Operador Super

Reaproveitando estrutura e comportamento entre classes em Java

1. Introdução

A **herança** é um dos pilares da Programação Orientada a Objetos e permite que uma classe reaproveite atributos e métodos de outra. Este texto aborda a **herança simples**, forma padrão suportada por Java; a **herança múltipla**, que Java não permite diretamente entre classes, mas simula por meio de interfaces; e o operador **super**, usado para acessar construtores, métodos e atributos da superclasse.

2. Herança Simples

Na **herança simples**, uma classe estende exatamente **uma** superclasse, usando a palavra-chave **extends**. É a única forma de herança de classes permitida em Java: uma classe nunca pode ter mais de uma superclasse direta.

```
public class Animal {
    protected String nome;

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

    public void comer() {
        System.out.println(nome + " está comendo.");
    }
}

public class Cachorro extends Animal { // herança simples: uma única superclasse
    public Cachorro(String nome) {
        super(nome);
    }

    public void latir() {
        System.out.println(nome + " está latindo.");
    }
}

// Cachorro rex = new Cachorro("Rex");
// rex.comer(); // herdado de Animal
// rex.latir(); // definido na própria classe
```

*Mesmo que não seja escrito explicitamente, toda classe Java que não declara **extends** estende implicitamente a classe **Object**, que é a raiz de toda a hierarquia de classes em Java.*

3. O Operador super

A palavra-chave **super** permite que uma subclasse acesse membros (construtor, métodos e atributos) definidos em sua superclasse direta, sendo especialmente útil quando esses membros foram sobrescritos ou ocultados na subclasse.

3.1. super() para Chamar o Construtor da Superclasse

Quando usado como `super ( . . . )`, deve ser a **primeira instrução** do construtor da subclasse e é responsável por inicializar a parte herdada do objeto, chamando o construtor correspondente da superclasse.

```
public class Funcionario {
    protected String nome;
    protected double salario;

    public Funcionario(String nome, double salario) {
        this.nome = nome;
        this.salario = salario;
    }
}

public class Gerente extends Funcionario {
    private double bonus;

    public Gerente(String nome, double salario, double bonus) {
        super(nome, salario); // inicializa nome e salario na superclasse
        this.bonus = bonus;
    }
}
```

3.2. super.metodo() para Chamar um Método Sobrescrito

Quando a subclasse sobrescreve (*overrides*) um método da superclasse, ainda é possível chamar a versão original por meio de `super.metodo()`, útil para reaproveitar parte do comportamento herdado antes de complementá-lo.

```
public class Funcionario {
    public String descricao() {
        return "Funcionário genérico";
    }
}

public class Gerente extends Funcionario {
    @Override
    public String descricao() {
        return super.descricao() + " (atua como Gerente)"; // reaproveita + complementa
    }
}

// new Gerente().descricao(); // "Funcionário genérico (atua como Gerente)"
```

3.3. super.atributo para Acessar um Atributo Ocultado

Se a subclasse declarar um atributo com o **mesmo nome** de um atributo da superclasse (ocultando-o), `super.atributo` permite acessar especificamente a versão herdada.

```
public class A {  
    protected String valor = "valor da superclasse";  
}  
  
public class B extends A {  
    protected String valor = "valor da subclasse"; // oculta o valor de A  
  
    public void exibir() {  
        System.out.println(valor);           // valor da subclasse  
        System.out.println(super.valor);      // valor da superclasse  
    }  
}
```

4. Herança Múltipla

Herança múltipla ocorre quando uma classe herda de **mais de uma** superclasse ao mesmo tempo. Linguagens como C++ permitem esse recurso diretamente entre classes, mas **Java não permite herança múltipla de classes** — cada classe só pode ter uma única superclasse direta.

4.1. O Problema do Diamante

A principal razão para essa restrição é o chamado **problema do diamante**: se duas superclasses distintas definirem um método com o mesmo nome e uma subclasse herdar de ambas, torna-se ambíguo qual das duas versões deveria ser executada.

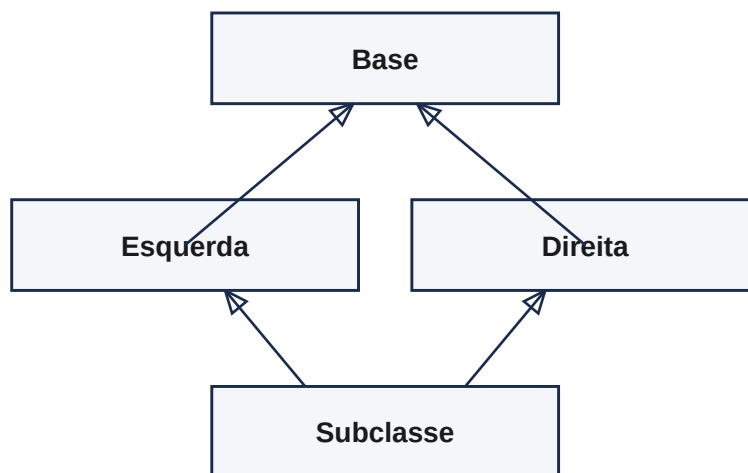


Figura 1 — Problema do diamante: a Subclasse herdaria de Esquerda e Direita, ambas derivadas de Base, gerando ambiguidade caso Esquerda e Direita sobrescrevam o mesmo método.

4.2. Herança Múltipla de Interfaces

Embora não permita herança múltipla de **classes**, Java permite que uma classe implemente **várias interfaces** ao mesmo tempo, o que é uma forma de herança múltipla de **tipo** (e, com métodos default, também de comportamento).

```

public interface Nadador {
    default void nadar() {
        System.out.println("Nadando...");
    }
}

public interface Corredor {
    default void correr() {
        System.out.println("Correndo...");
    }
}

public class Triatleta implements Nadador, Corredor { // "herda" de duas interfaces
    // Triatleta ganha nadar() e correr() sem precisar reimplementá-los
}

// Triatleta t = new Triatleta();
// t.nadar();
// t.correr();

```

4.3. Resolvendo Conflitos entre Métodos default

Se duas interfaces implementadas pela mesma classe definirem um método default com a **mesma assinatura**, o compilador Java exige que a classe **sobrescreva** explicitamente esse método, resolvendo a ambiguidade manualmente — e evitando, assim, o problema do diamante.

```

public interface Voador {
    default void mover() {
        System.out.println("Voando...");
    }
}

public interface Nadador {
    default void mover() {
        System.out.println("Nadando...");
    }
}

public class Pato implements Voador, Nadador {
    @Override
    public void mover() { // obrigatório: resolve o conflito explicitamente
        Voador.super.mover(); // escolhe (ou combina) o comportamento desejado
        Nadador.super.mover();
    }
}

```

5. Exemplo Integrado

O exemplo a seguir combina herança simples (Funcionario → Gerente), o uso de super no construtor e no método sobrescrito, e herança múltipla de interfaces por meio de `Avaliavel`.

```

public interface Avaliavel {
    default String avaliar(double nota) {
        return nota >= 7 ? "Aprovado" : "Reprovado";
    }
}

public class Funcionario implements Avaliavel {
    protected String nome;
    protected double salario;

    public Funcionario(String nome, double salario) {
        this.nome = nome;
        this.salario = salario;
    }

    public String descricao() {
        return nome + " (Funcionário)";
    }
}

public class Gerente extends Funcionario { // herança simples
    private double bonus;

    public Gerente(String nome, double salario, double bonus) {
        super(nome, salario); // super() no construtor
        this.bonus = bonus;
    }

    @Override
    public String descricao() {
        return super.descricao() + " atuando como Gerente"; // super no método
    }
}

// Gerente g = new Gerente("Ana", 5000, 800);
// System.out.println(g.descricao()); // herança simples + super
// System.out.println(g.avaliar(8.5)); // herdado da interface Avaliavel

```

6. Conclusão

Java adota a herança simples entre classes justamente para evitar as ambiguidades do problema do diamante, mas compensa essa limitação permitindo que uma classe implemente múltiplas interfaces, inclusive com métodos default prontos. O operador `super` complementa esse modelo, dando à subclasse controle preciso sobre quando reaproveitar e quando complementar o que foi herdado — seja no construtor, em métodos sobrescritos ou em atributos ocultos.