DANIEL ABELLA
www.daniel-abella.com

Daniel Abella

Profissional PMP, PMI-ACP e CSM, Gerente de Projetos no Virtus & UniFacisa e Professor do Curso de Sistemas de Informação da UniFacisa. Autor dos livros Gestão A3 (Ágil) e Scrum Arretado.

Conteúdo:

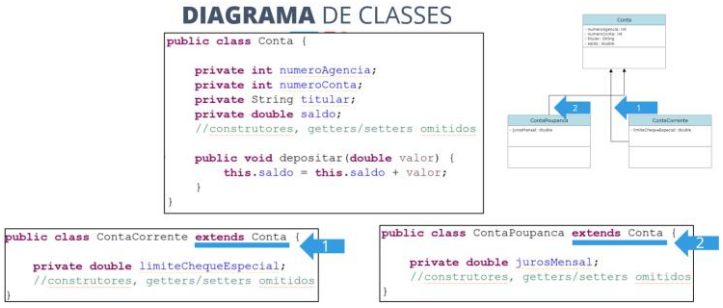
Capítulo 6

Orientação à Objetos na Veia! Herança!

Livro Java do Zero (Uma Viagem ao Mundo Java)

01 Herança

- A herança é um recurso de linguagens orientadas à objetos como Java que permitem o reuso de código e a organização de classes de maneira hierarquizada. A seguir, temos um exemplo da organização do cenário de Conta, que é a classe pai/mãe, que chamamos de superclasse e, duas classes, ContaCorrente e ContaPoupanca, as subclasses.



- Inicialmente a classe Conta, a qual chamamos de superclasse (também ouço chamar de classe pai, classe mãe, entre outros termos) possui apenas as variáveis e métodos genéricos, ou seja, que todo tipo de conta (como corrente ou poupança) possui. Neste caso, temos as 4 variáveis básicas de uma conta e o método depositar (poderiam ter mais outros métodos, concordo).
- Na sequência, temos as classes ContaCorrente e ContaPoupanca com suas variáveis específicas. Para que estas duas classes herdam de Conta, precisamos, logo após a definição da classe, colocarmos a palavra-chave extends seguida pela superclasse, neste exemplo,

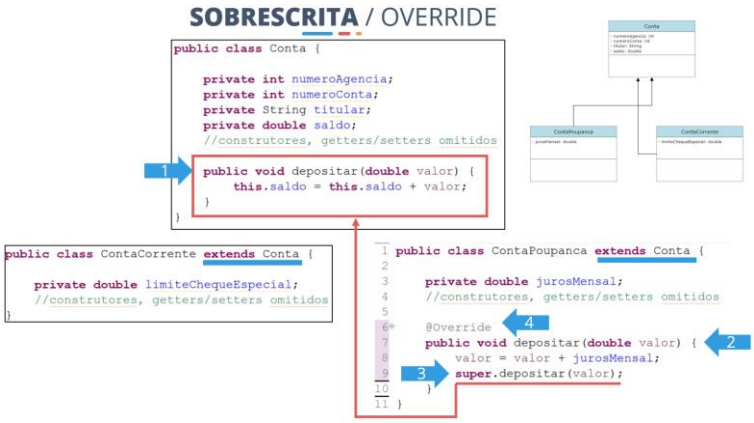
```
extends Conta
```

Herança Múltipla

Em Java, diferente de outras linguagens como C++, só podemos estender apenas uma classe. Ou seja, não permite herança múltipla. Entretanto, em contrapartida, está permitido a uma classe implementar quantas interfaces forem necessárias.

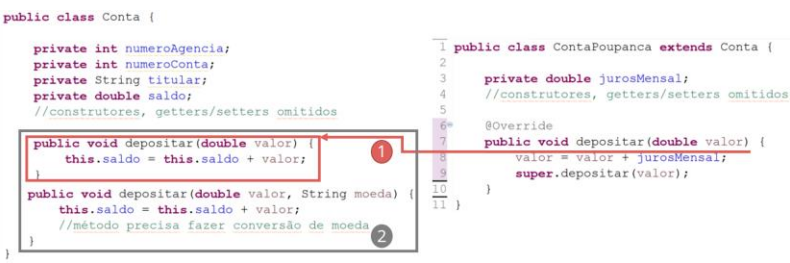
02 Sobrescrita (Override)

- Quando você tem um método herdado e você quer manter a mesma assinatura*, mas ter um comportamento específico na subclasse, precisamos aplicar o conceito de sobrescrita (em inglês, override).
- *A assinatura do método é formada pelo seu nome e pelo tipo, quantidade e ordem de seus parâmetros.
- Para fazer a sobrescrita de um método, basta criarmos na subclasse um método com a mesma assinatura do método existente na superclasse. Neste exemplo, criamos o método depositar (seta 1) na superclasse e criamos na subclasse com a mesma assinatura (seta 2). Caso queira evidenciar que este método é sobrescrito, podemos, opcionalmente, adicionar a anotação @Override (seta 4), que não requer nenhum import, pois pertence ao pacote java.lang.



Não confunda! Primeiramente, sobrecarga são métodos com mesmo nome, na mesma classe, porém com assinaturas diferentes (por exemplo, número ou tipos de parâmetros diferentes). Por outro lado, sobrescrita são métodos com a mesma assinatura, tendo um método na superclasse e outro(s) com a mesma assinatura na(s) subclasse(s).

No exemplo abaixo, evidenciado no quadrado indicado pela seta 2, temos na classe Conta dois métodos depositar, porém com número de parâmetros diferentes, o que evidencia uma sobrecarga. Enquanto que, na seta 1, temos uma sobrescrita, onde temos um método depositar na superclasse e um método com a mesma assinatura na subclasse. Antes de seguir, garanta que este conceito esteja sedimentado na sua alma.



03 Métodos Final

- Na sessão anterior, apresentamos como fazer uma sobrescrita de um método. Entretanto, existem situações em que, queremos que as subclasses não alterem o comportamento do método da superclasse, em outras palavras, que não façam sobrescrita. Para que um método não possa ser sobrescrito (obviamente, nas subclasses), usamos o modificador final, conforme exemplo abaixo.

```java
public class Conta {

    private int numeroAgencia;
    private int numeroConta;
    private String titular;
    private double saldo;
    //construtores, getters/setters omitidos

    public final void depositar(double valor) {
        this.saldo = this.saldo + valor;
    }
}
```

```java
public class ContaPoupanca extends Conta {

    private double jurosMensal;
    //construtores, getters/setters omitidos

    @Override
    public void depositar(double valor) {
        valor = valor + jurosMensal;
        super.depositar(valor);
    }
}
```

2 → 1

Cannot override the final method from Conta
1 quick fix available:
Remove 'final' modifier of 'Conta.depositar(..)'

- Usamos a palavra-chave final no método depositar da superclasse (seta 1) para indicar que ele não está elegível para sobrescrita. Como resultado, por exemplo, a ContaPoupanca não pode sobrescrever, conforme indicado na seta 2. Na parte inferior, relacionamos a mensagem de erro apresentada pelo Eclipse (Cannot override the final method from Conta - Em português, não podemos sobrescrever o método final de Conta).

04 Classes Final

- Como vimos anteriormente, métodos final fazem com que eles não sejam sobrescritos. E, vimos lá em capítulos anteriores que, variáveis final fazem com que, elas se tornem constantes (isto é, ao receber um valor não pode ter outro valor atribuído). Por outro lado, Classes final indica que ela não poder ter filhos, ou seja, não pode ser superclasse, isto é, ninguém pode "dar um extends nela".
- Exemplos de classes elegíveis a usar o Final, temos a ContaPoupanca e ContaCorrente que, provavelmente não incluiremos subclasses. Como visto, o funcionamento do final tem um significado em cada lugar (classe, método e variável), conforme evidenciado na tabela a seguir.

Tipo	Descrição
Variável Final	Uma vez atribuído o valor, não pode receber outro valor. `final int variavelFinal = 2;` `variavelFinal = 3;` ⛔
Método Final	Não pode ser sobrescrito. `public final void depositar(double valor) { this.saldo = this.saldo + valor; }`
Classe Final	Não pode ser estendida. `public final class Conta { //atributos, métodos, construtores, etc }` `public class ContaPoupanca extends Conta { }` ⛔

05 Classes e Métodos Abstract

- Precisamos encontrar uma maneira para definir a classe Conta como genérica (como base para as subclasses ContaCorrente e ContaPoupanca) e que não passa ser instanciada. Esta maneira é usando o modificador abstract na superclasse (neste caso, Conta)
- No exemplo a seguir, apenas adicionamos a palavra-chave abstract na classe Conta. Como consequência, a classe Conta não pode ser instanciada, conforme destacado na imagem. É importante compreender que, não é possível instanciar a classe Conta, mas é possível declarar uma variável.

ABSTRACT EM CLASSES

```java
public abstract class Conta {
    //atributos, métodos, construtores, etc
}
```

```java
public class ContaCorrente extends Conta {
    //construtores, getters/setters omitidos
}
```

```java
public class Teste {

    public static void main(String[] args) {

        Conta novaConta = new Conta();
        ContaCorrente contaCorrente1 = new ContaCorrente();
        ContaPoupanca contaPoupanca1 = new ContaPoupanca();
    }
}
```

```java
public class ContaPoupanca extends Conta {
    //construtores, getters/setters omitidos
}
```

- Em classes Abstract podemos ter métodos concretos (que tem corpo, como os que fizemos até agora), bem como métodos abstract, que são métodos que não tem corpo e precisam ser implementados pela primeira subclasse concreta (não abstract).

ABSTRACT E FINAL BRIGAM!

Uma classe pode ser Abstract ou Final, mas não ambas ao mesmo tempo, o que faz total sentido. Abstract cria um arcabouço para a classe, esperando que tenhamos subclasses para implementar, por exemplo, seus métodos abstract. Por outro lado, classes Final não podem ter subclasses. Ou seja, os dois são conceitos distintos e que não podem conviver juntos 😉

- No exemplo abaixo, na classe Abstract Conta, temos um método abstract intitulado calcularSaldo, indicados na seta 1. Note que, este método tem a palavra-chave abstract e não tem corpo, ou seja, não tem chaves de abrir e fechar, tendo ao invés disto, um ponto e vírgula.

ABSTRACT EM MÉTODOS

```java
public abstract class Conta {
    //atributos, métodos, construtores, omitidos
    private double saldo;

    public abstract void calcularSaldo();

    public void depositar(double valor) {
        this.saldo += valor;
    }
}
```

1 →

```java
public class ContaCorrente extends Conta {

    //construtores, getters/setters omitidos

    @Override
    public void calcularSaldo() {
        // Aqui seria a implementação de
        // calcular saldo em ContaCorrente
    }
}
```

2 →

- Neste exemplo, a subclasse concreta (isto é, não abstract) ContaCorrente precisou implementar o método calcularSaldo (seta 2) que foi definido na superclasse (seta 1). Diferente do método calcularSaldo da superclasse que não possui corpo (por ser abstract), o método calcularSaldo da subclasse possui corpo e não possui o modificador abstract. Desta maneira, concluímos que, quando temos um método abstract na superclasse, este precisa ser implementado na subclasse concreta.

06 Polimorfismo

- Polimorfismo é uma palavra de origem grego que significa muitas formas. Em orientação à objetos, refere-se à capacidade de um objeto em se comportar de diferentes maneiras (formas), dependendo do contexto em que é utilizado, permitindo a execução de métodos com a mesma assinatura, mas com comportamentos distintos em classes diferentes.

POLIMORFISMO

```
public class TesteUm {

    public static void main(String[] args) {

        Conta conta = null;          <- 2

        //Usuario escolhe se quer criar
        //Conta Corrente (opção 1) ou Conta Poupança (opção 2)
        int opcao = 1; //exemplo: usuario informou opção 1

        if (opcao == 1) {
            conta = new ContaCorrente();     <- 3
        } else if (opcao == 2) {
            conta = new ContaPoupanca();     <- 4
        } else {
            System.out.println("Opção inválida!");
        }
    }
}
```

- Com base na imagem acima, note que, temos apenas uma variável da superclasse (neste caso, Conta, indicado na seta 2). Lembre-se que, é possível declarar uma variável do tipo de uma classe Abstract, o que não podemos é instanciar uma classe Abstract.
- Com uso desta variável da superclasse, podemos atribuir qualquer instância de uma subclasse, como acontece nas setas 3 e 4. Toda esta informação é consolidada na imagem a seguir. Nas 3 setas, declaramos uma variável do tipo Conta. Na primeira linha (da var1), instanciamos a classe Conta, fato possível desde que esta classe não seja abstract. Na segunda e terceira setas instanciamos as subclasses de Conta, corroborando para o polimorfismo

07 Modificadores de Acesso

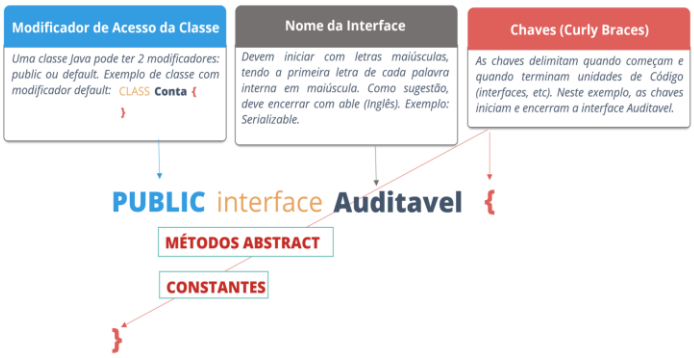
- Modificadores de acesso são palavras-chave utilizadas para controlar a visibilidade e o acesso aos elementos (classes, enumerations, interfaces, métodos, variáveis de classe e variáveis de instância) em Java, sendo eles:
    - **public:** Não possui restrições, ou seja, o elemento é acessível de qualquer lugar;
    - **protected:** Elemento acessível dentro da mesma classe, mesmo pacote e também por subclasses (mesmo que estejam em pacotes diferentes);
    - **default (sem modificador explícito):** Elemento é acessível dentro da mesma classe e por outros elementos do mesmo pacote; e
    - **private:** Elemento acessível apenas dentro da própria classe em que está definido.
- Para uma classe ou Enum Java, temos apenas dois modificadores possíveis: public e default. Por outro lado, variáveis de instância e de classe podem assumir qualquer um dos 4 modificadores. Como Java tem muitos itens, abaixo resumo os modificadores possíveis em cada um destes itens.

Item / Modificador de Acesso	Default	Public	Protected	Private
Classe	Sim	Sim	Não	Não
Classe Interna	Sim	Sim	Sim	Sim
Interface	Sim	Sim	Não	Não
Interface (dentro de Classe)	Sim	Sim	Sim	Sim
Enum	Sim	Sim	Não	Não
Enum (dentro de Classe)	Sim	Sim	Sim	Sim
Enum (dentro de Interface)	Sim	Não	Não	Não
Construtor	Sim	Sim	Sim	Sim
Métodos (dentro de Classe)	Sim	Sim	Sim	Sim
Métodos (dentro de Interface)	Sim	Não	Não	Não

08 Interfaces

- Como vimos anteriormente, Classes abstract podem ter métodos abstract, bem como métodos concretos (não abstract). Nesta seção apresentaremos o conceito de Interfaces que, a primeiro momento, imaginem que seja uma classe abstract que possui apenas métodos abstract e constantes.

INTERFACE JAVA



- Interface é um contrato de modo que, uma classe concreta que a implementa, tem que seguir o contrato, ou seja, implementar todos os métodos abstract declaradas na Interface. Diferente de Classes, Interfaces NÃO podem ser instanciadas!

INTERFACES EXEMPLO 1

```
interface Drawable {

    public static final String RED = "FF0000";
    public static final String GREEN = "00FF00";
    public static final String BLUE = "0000FF";
    public static final String BLACK = "000000";
    public static final String WHITE = "FFFFFF";

    void draw(String color);

    String getColor();
}
```

```
public class Circle implements Drawable {

    private double x;
    private double y;
    private double radius;

    //construtores, getters e setters omitidos

    @Override
    public void draw(String color) {
        System.out.println("Drawing circle");
    }

    @Override
    public String getColor() {
        return Drawable.BLUE;
    }
```

IMPLÍCITOS DA INTERFACE

Como vimos, uma classe possui métodos e constantes. Implicitamente, ou seja, "por baixo dos panos", todo método em uma interface é **public abstract** e uma interface é implicitamente **public static final**.

Desta maneira as duas versões da interface Clonavel a seguir são a mesma coisa.

```
public interface Clonavel {

    int MIN_SIZE = 1;

    Object clone();
}
```

```
public interface Clonavel {

    public static final int MIN_SIZE = 1;

    public abstract Object clone();
}
```

09 Java 8 - Default Methods em Interfaces

- Um método default, em interfaces, fornece uma implementação padrão para os casos em que uma classe que implementa essa interface não fornece uma implementação específica para o método.

INTERFACE COM DEFAULT METHODS

```
interface Drawable {

    void draw();

    default void msg() {
        System.out.println("default method");
    }
}
```

CLASSE IMPLEMENTANDO A INTERFACE

```
class Rectangle implements Drawable {

    public void draw() {
        System.out.println("drawing rectangle");
    }
}
```

USO DO CÓDIGO ACIMA


```
class TesteInterfaceMetodoDefault {

    public static void main(String args[]) {

        Drawable d = new Rectangle();
        d.draw();
        d.msg();
    }
}
```

10 Interface com Static Methods

- Abaixo temos um exemplo de métodos static em interfaces, sendo estes úteis para prover métodos utilitários, como métodos para ordenação, checagem de um padrão de uma dada String, entre outros.

INTERFACE COM STATIC METHODS

```
interface Drawable {

    void draw();

    static int cube(int x) {
        return x * x * x;
    }
}
```

CLASSE IMPLEMENTANDO A INTERFACE

```
class Rectangle implements Drawable {

    public void draw() {
        System.out.println("drawing rectangle");
    }
}
```

USO DO CÓDIGO ACIMA

```
class TesteInterfaceMetodoDefault {

    public static void main(String args[]) {

        Drawable d=new Rectangle();
        d.draw();

        System.out.println(Drawable.cube(3));
    }
}
```

11 Java 9

- No Java 8, tivemos duas novidades relacionadas a interfaces: métodos static e métodos default. Na versão posterior (Java 9), torna-se possível termos métodos private em interfaces.
- Um método private em interface privada é um tipo especial de método que é acessível apenas dentro da interface, tendo desta maneira como benefício o encapsulamento.
- Métodos privados podem ser de dois tipos: métodos privados static e métodos privados não static.

12 Juntando tudo de Interfaces

- Agora que, vimos que dentro de Interfaces podemos ter um montão de coisas, abaixo listo tudo o que pode estar relacionado em uma interface:
    - Constantes (implicitamente public static final);
    - Métodos abstract (implicitamente public abstract);
    - Métodos default;
    - Métodos static;
    - Métodos private;
    - Métodos private static.
- Tudo o que é possível dentro de uma interface, a partir do Java 9, está relacionado no exemplo a seguir.

```
public interface InterfaceComTudo {

    public static final int DEFAULT_NUMBER = 1;

    public abstract void mul(int a, int b);

    public default void add(int a, int b) {

        //método private dentro de método default
        sub(a, b);

        // método static dentro de outro método não static
        div(a, b);
        System.out.println(a + b);
    }

    public static void mod(int a, int b) {
        div(a, b);
        System.out.println(a % b);
    }

    private void sub(int a, int b) {
        System.out.println(a - b);
    }

    private static void div(int a, int b) {
        System.out.println(a / b);
    }
}
```

Constantes

Método abstract

Método default

Método static

Método private

Método static private