

Classes Abstratas, Polimorfismo, Sobreposição e Interfaces



Conteudista: Prof. Esp. Alexander Gobbato Albuquerque

Revisão Textual: Prof.^a M.^a Rosemary Toffoli

Objetivos da Unidade:

- Apresentar os conceitos de Classes Abstratas, Polimorfismo, Sobreposição e Interfaces;
- Compreender a importância dos conceitos em Orientação a Objetos que permitem a extensibilidade de componentes e seu reuso.

 Contextualização

 Material Teórico

 Material Complementar

 Referências



Contextualização

Na Unidade anterior estudamos os conceitos de Herança em POO, e o conceito de herança múltipla. Porém, alguns mecanismos adicionais devem ser implementados para garantir a estruturação das classes mantendo um certo padrão de nomenclatura.

Classes abstratas servem como molde para subclasses concretas que devem implementar seus métodos abstratos e herdar os métodos concretos. Além disso, temos interfaces que obrigam as classes que implementam a ter os métodos da interface. Isso faz com que tenhamos certeza que todas as subclasses dessa estrutura irão possuir os mesmos métodos (a mesma assinatura) através de sobreposição de métodos, criando um padrão.

Cada subclasse que implementa os métodos pode ter uma implementação de método diferente, apesar de ter a mesma assinatura. Isso caracteriza o polimorfismo, pois o nome será o mesmo, mas a forma diferente.

Projetos de sistemas OO possuem dezenas ou centenas de classes. A inserção de uma nova classe na hierarquia pode fazer com que o sistema perca o padrão, porém se ele herdar de classes abstratas e implementar interfaces, isso fará com que ele se “encaixe” no padrão.

Uma subclasse pode herdar apenas de uma superclasse, mas pode implementar várias interfaces.

Material Teórico

Classes Abstratas

À medida que você sobe na hierarquia de herança, as classes tornam-se mais gerais. Em algum momento, a classe torna-se tão geral que ela seria apenas uma base para as outras classes e não mais uma classe com instâncias específicas que você queira utilizar.

Uma classe abstrata é desenvolvida para representar entidades e conceitos abstratos. Usamos uma classe abstrata para criar um padrão de subclasses com comportamentos iguais.

Vamos analisar a Figura 1 a seguir:

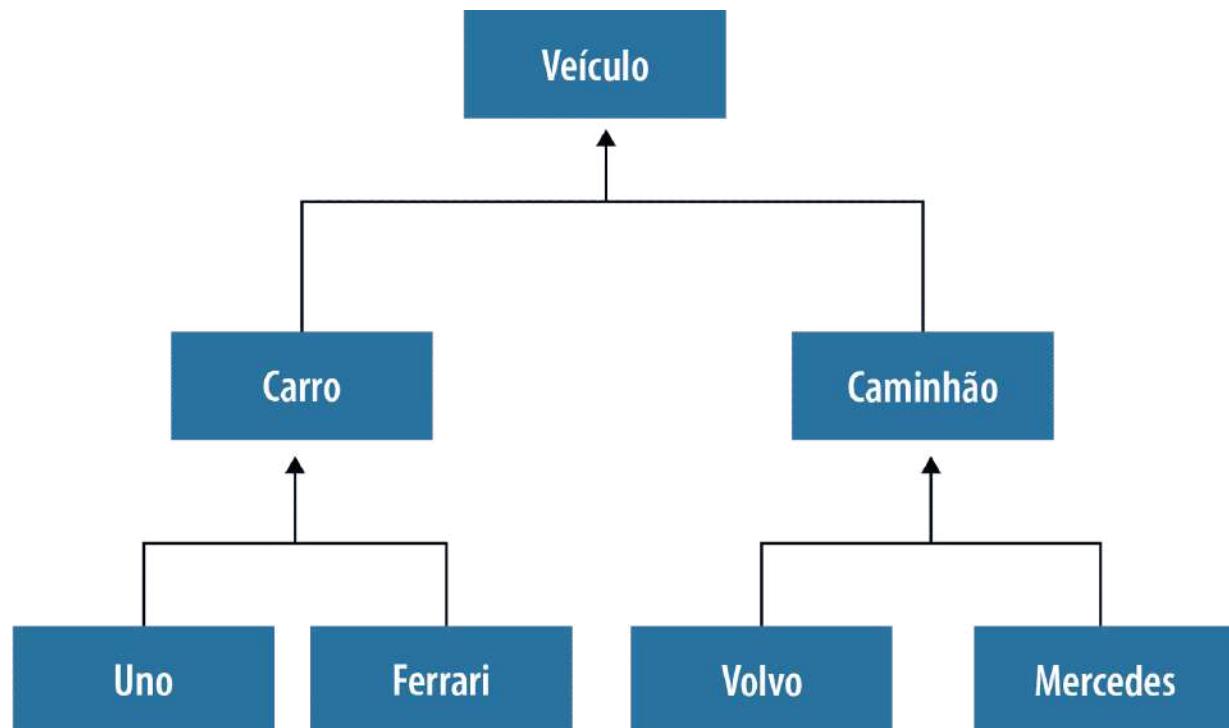


Figura 1 – Diagrama de Herança para a classe Veiculo e suas subclasses

No Diagrama de classes representado na Figura 1 podemos criar classes comuns (classes concretas) para todas as classes representadas. Isso fará com que possamos instanciar objetos de qualquer uma das sete classes. Porém, a classe Veículo só serviu de molde para atributos e métodos comuns às subclasses Carro e Caminhão. Podemos, por exemplo, ter como atributos na classe Veículo “velocidade” e “passageiros”, e alguns métodos como “acelera”, “freia” e seus métodos de acesso “set” e “get”.

Desse modo, não há motivos para instanciar objetos da classe Veículo, pois ela serviu apenas como abstração das subclasses da mesma. Nesse caso, dizemos que Veículo pode ser uma classe abstrata.

Para a criação de uma classe abstrata usamos a palavra chave “*abstract*”. Veja na Figura 2 a implementação da classe Veículo.

```

public abstract class Veiculo {

    //Atributos
    private float velocidade;
    private int passageiros;

    //Construtores
    public Veiculo(float v, int p) {
        velocidade = v;
        passageiros = p;
    }
    public Veiculo() {
        this(0f, 0);
    }

    //Métodos de Acesso
    public void setVelocidade(float v) {
        velocidade = v;
    }

    public float getVelocidade() {
        return velocidade;
    }

    public void setPassageiros(int p) {
        passageiros = p;
    }

    public int getPassageiros() {
        return passageiros;
    }

    //Métodos abstratos
    public abstract void acelera();
    public abstract void freia();
}

```

Figura 2 – Implementação de classe abstrata Veiculo

Fonte: Reprodução

Uma classe abstrata, além de seus métodos concretos, pode conter também métodos abstratos. Note na Figura 2 que temos dois métodos abstratos, “acelera” e “freia”. Métodos abstratos não possuem corpo, apenas a sua assinatura e eles devem ser implementados obrigatoriamente em suas subclasses, a não ser que suas subclasses também sejam abstratas e nesse caso, devem ser implementados na primeira classe concreta do diagrama.

Portanto, uma subclass de uma classe abstrata herda os métodos concretos e se obriga a implementar os métodos abstratos.

```

public class Carro extends Veiculo{

    //Atributos
    private float combustivel;

    //Construtores
    public Carro(float vel, int pas, float comb) {
        super(vel, pas);
        combustivel = comb;
    }
    public Carro() {
        this(0f, 0, 0f);
    }

    //Métodos de acesso
    public void setCombustivel(float comb) {
        combustivel = comb;
    }
    public float getCombustivel() {
        return combustivel;
    }

    //Métodos obrigatórios
    public void acelera() {
        setVelocidade(getVelocidade() + 1);
    }
    public void freia() {
        setVelocidade(getVelocidade() - 1);
    }
}

```

Figura 3 – Implementação da subclasse concreta Carro

Fonte: Reprodução

Na Figura 3 temos a implementação da classe Carro, que é subclasse de Veiculo. Note que, por ser uma classe concreta, temos obrigatoriamente que implementar os métodos “acelera” e “freia”.

Nesse caso, não podemos instanciar objetos da classe abstrata Veiculo, ou seja, um comando do tipo “Veiculo v1 = new Veiculo();” estaria incorreto.

Polimorfismo e Sobreposição

O termo polimorfismo é originário do grego e significa “muitas formas” (*poli* = muitas, *morphos* = formas). A propriedade segundo o qual uma operação pode comportar-se diferentemente em classes diferentes.

O polimorfismo é o responsável pela extensibilidade em programação orientada a objetos, promovendo o reuso. Em termos de programação, é o princípio pelo qual duas ou mais subclasses de uma mesma superclasse podem invocar métodos quem tem a mesma identificação (assinatura), mas com comportamentos distintos, especializados para cada classe derivada.

Para exemplificar, pense no termo “abrir”. Você pode abrir uma porta, uma caixa, uma janela ou conta no banco. A palavra abrir pode ser aplicada a muitos objetos diferentes no mundo real e cada objeto interpreta a ação de “abrir” à sua maneira, entretanto podemos simplesmente dizer “abrir” para expressar a ação.

Uma linguagem polimórfica é a que suporta polimorfismo (*Actionscript, Java*), já a linguagem monomórfica não suporta polimorfismo (*Pascal, ASP*).

Uma maneira de utilizar o polimorfismo, utilização o mecanismo de **Sobreposição de Métodos** (do inglês *override*), que consiste em reescrever o método herdado da superclasse. Não confundir Sobrecarga de Métodos (*overload*), que consiste em escrever métodos com o mesmo nome, porém com parâmetros diferentes, o que faz com que eles tenham assinaturas diferentes. Na Sobreposição de Métodos, a assinatura é idêntica ao método herdado, porém o corpo do método é alterado para o propósito específico da subclasse.

Na Figura 2, a classe abstrata Veiculo possui métodos abstratos “acelera” e “freia”. Isso fez com que fôssemos obrigados a implementar esses métodos na classe Carro conforme Figura 3. Se analisarmos novamente a Figura 1, notamos que as classes Uno e Ferrari herdam todos os atributos e métodos de Carro, incluindo os métodos acelera e freia.

Porém, sabemos que os objetos da classe Uno aceleram e freiam mais lentamente que objetos da classe Ferrari, portanto, devemos sobrepor esses métodos para que ajam de acordo com a realidade de cada objeto.

```
public class Uno extends Carro{  
  
    //Sem atributos  
  
    //Construtores  
    public Uno(float vel, int pas, float comb) {  
        super(vel, pas, comb);  
    }  
    public Uno() {  
        this(0f, 0, 0f);  
    }  
  
    //Sobreposição  
    public void acelera() {  
        setVelocidade(getVelocidade() + 0.5f);  
    }  
    public void freia() {  
        setVelocidade(getVelocidade() - 0.5f);  
    }  
}
```

```
public class Ferrari extends Carro{  
  
    //Sem atributos  
  
    //Construtores  
    public Ferrari(float vel, int pas, float comb) {  
        super(vel, pas, comb);  
    }  
    public Ferrari() {  
        this(0f, 0, 0f);  
    }  
  
    //Sobreposição  
    public void acelera() {  
        setVelocidade(getVelocidade() + 3.5f);  
    }  
    public void freia() {  
        setVelocidade(getVelocidade() - 3.5f);  
    }  
}
```

Figura 4 – Implementação das classes Uno e Ferrari

Fonte: Reprodução

Note na Figura 4 que nós reescrevemos os métodos acelera e freia das subclasses Uno e Ferrari, adaptando para as próprias necessidades. Podemos notar que a assinatura do método é idêntica à superclasse carro usando então o mecanismo de Sobreposição.

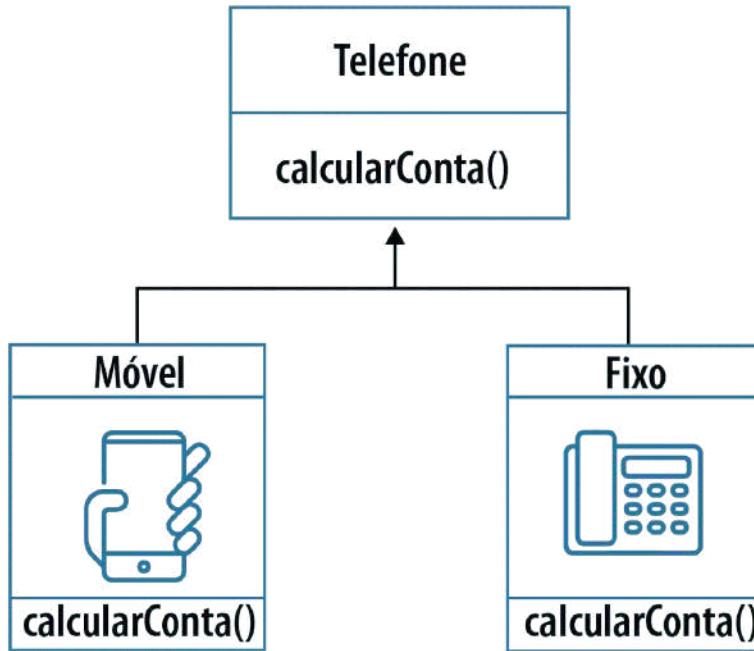


Figura 5 – Exemplo de Polimorfismo e Sobreposição

Fonte: Adaptada de Freepik

Veja mais um exemplo na Figura 5. Temos uma classe abstrata “Telefone” que possui um método **calcularConta**. Por herança, as classes “Móvel” e “Fixo” recebem o método **calcularConta**, porém a maneira de calcular a conta de um Telefone Móvel e de um Telefone Fixo é diferente.

Nesse caso, através da Sobreposição de Métodos, reescrevemos os métodos **calcularConta** em cada subclasse para que se adapte à sua necessidade, criando então o conceito de Polimorfismo.

```
Exemplo:  
public class Animal {  
    public void andar(){  
        System.out.println("pessoa andando");  
    }  
}  
public class Cavalo extends Animal{  
    public void andar(){  
        System.out.println("Cavalo andando");  
    }  
}
```

Figura 6 – Exemplo de Polimorfismo e Sobreposição

Fonte: Reprodução

Veja o exemplo da Figura 6, onde temos uma classe “Animal” que possui um único método “andar”. Criamos uma subclasse de Animal chamada Cavalo, que por herança recebe o método andar. Nesse exemplo, estamos sobrescrevendo o método andar para que faça outra coisa.

Tanto a classe Animal como a classe Cavalo são concretas e aceitam instanciar objetos, podemos ter o seguinte código Java:

```
Animal a = new Animal();  
Cavalo b = new Cavalo();  
a.andar();  
b.andar();
```

Para o objeto “a”, aparecerá a mensagem de “pessoa andando” e para o objeto “b” a mensagem será “Cavalo Andando”. Porém, podemos ter também o seguinte código:

```
Animal a = new Animal();  
Animal b = new Cavalo();
```

```
a.andar();  
b.andar();
```

No exemplo anterior, o resultado será o mesmo, para o objeto “a” aparecerá “pessoa andando” e para o “b” será “Cavalo andando”. Porém temos agora “b” como uma **variável polimórfica**. Um objeto do tipo Animal pode referir-se a um objeto do tipo Animal ou a um objeto de qualquer subclasse da classe Animal.

Interfaces

Muitas linguagens de programação orientada a objetos não possuem o conceito de Herança Múltipla. Isso ocorre porque alguns mecanismos da POO são incompatíveis com processos de herança múltipla. Linguagens como C++ e Eiffel possuem mecanismos de herança múltipla, porém são pouco utilizados por serem complexos e por vezes ineficientes. Vamos analisar a Figura 7 a seguir:

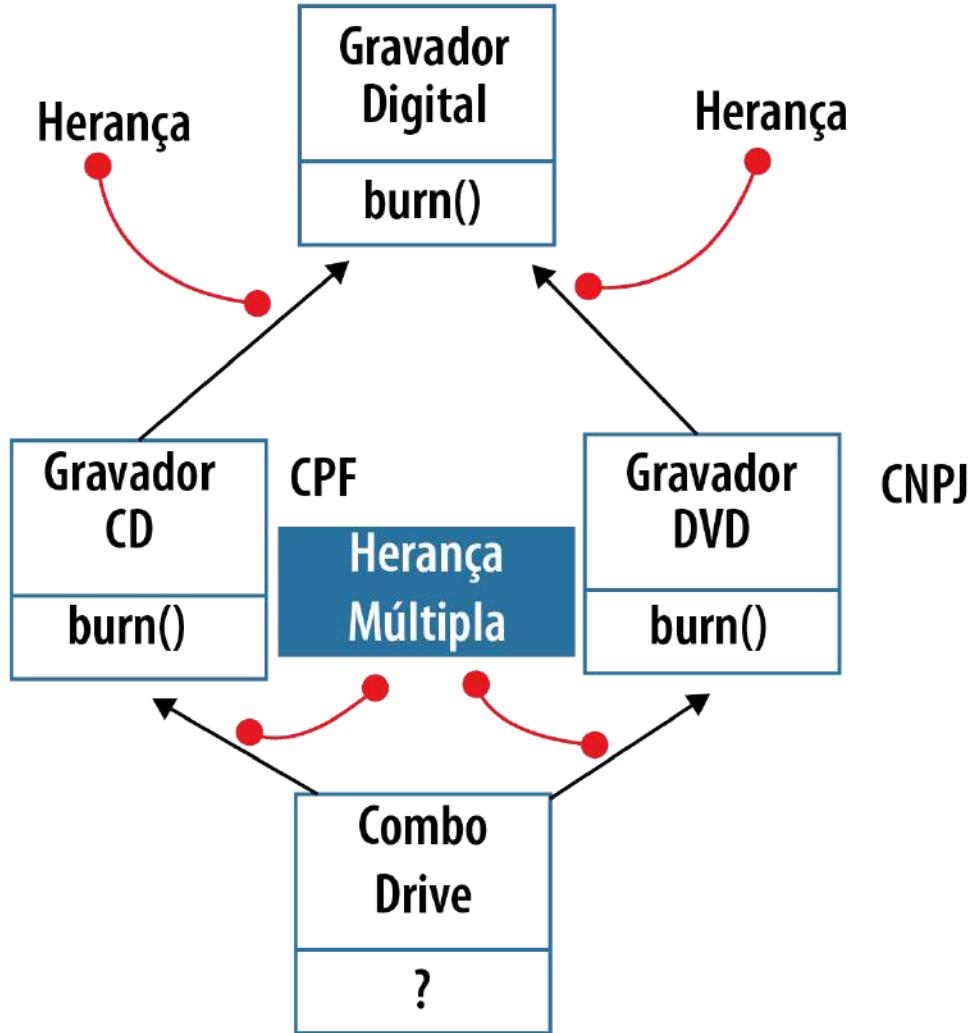


Figura 7 – Cenário de Herança Múltipla

Analisando a Figura 7 podemos notar que a classe **ComboDrive** herda características das superclasses **GravadorDVD** e **GravadorCD**, caracterizando uma herança múltipla. Porém, sabemos que por herança as subclasses herdam todos os atributos e métodos da classe pai.

No nosso exemplo, o método “*burn*” foi sobreescrito nas duas classes **GravadorCD** e **GravadorDVD**. Qual método será chamado quando invocarmos o método “*burn*” da classe **ComboDrive**?

Esse cenário não é possível em Java e em muitas outras linguagens OO. Nesse caso, usamos outro mecanismo chamado Interfaces.

Uma interface especifica um conjunto de operações públicas de uma classe e/ou um componente. Uma interface não é uma classe, mas um conjunto de requisitos para classes que precisam se adequar a ela, ou seja, você nunca poderá utilizar o operador *new* para instanciar uma interface.

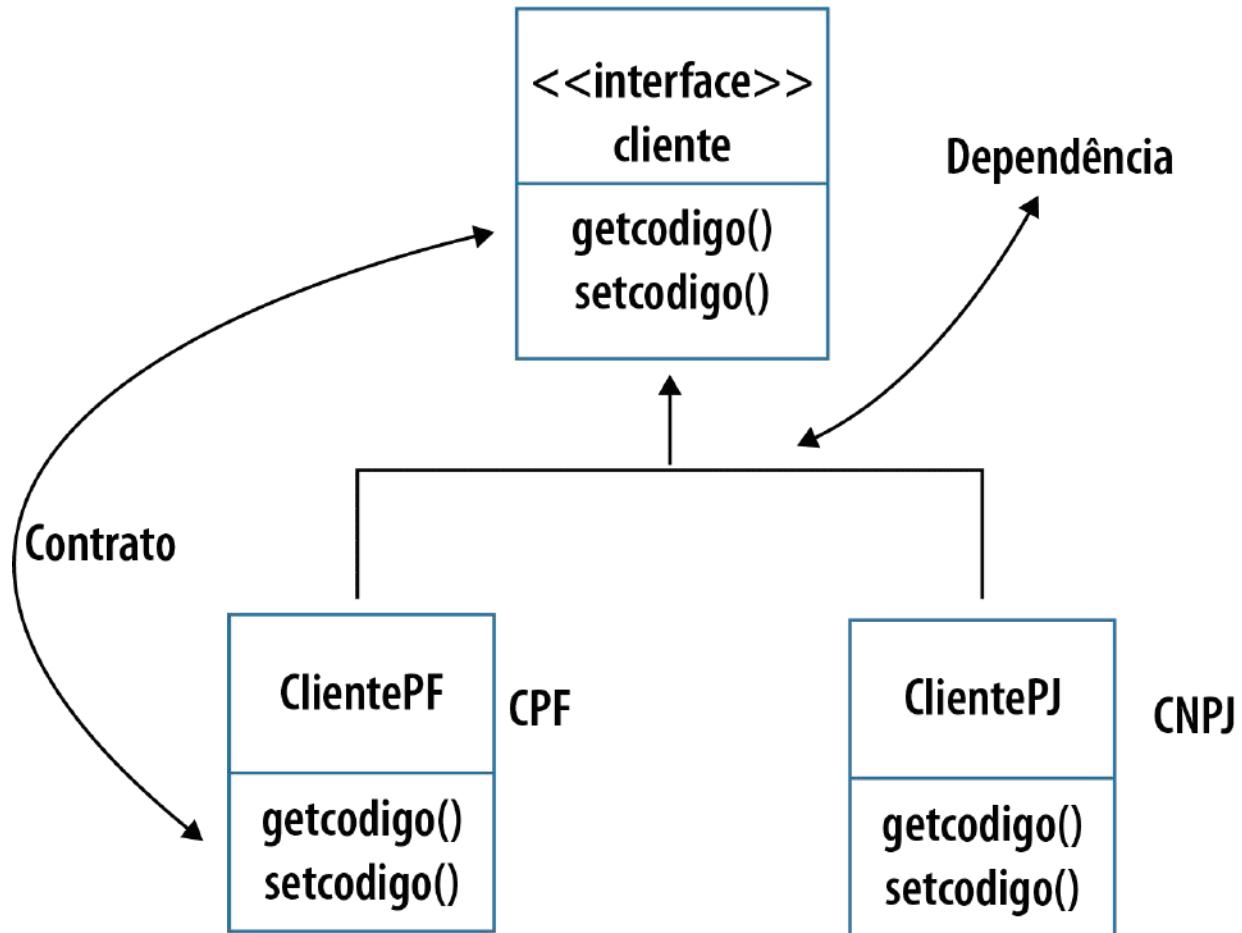


Figura 8 – Interface

Note na Figura 8 que Cliente não é uma classe e sim uma Interface. Ela define quais os métodos que deverão ser criados nas classes que a implementam. Uma interface não é herdada e sim implementada. Quando dizemos que uma classe implementa uma interface, estamos firmando um contrato de compromisso que essa classe, obrigatoriamente, implementará os métodos da interface.

Os métodos na interface são métodos “ocos”, ou seja, só possuem as assinaturas, mas não possuem corpo. Além dos métodos, as interfaces podem conter constantes, cujas classes que implementam a interface recebem as constantes como um mecanismo de herança.

Uma classe pode herdar apenas de uma superclasse, porém podem implementar diversas interfaces, podendo assim simular o mecanismo de herança múltipla.

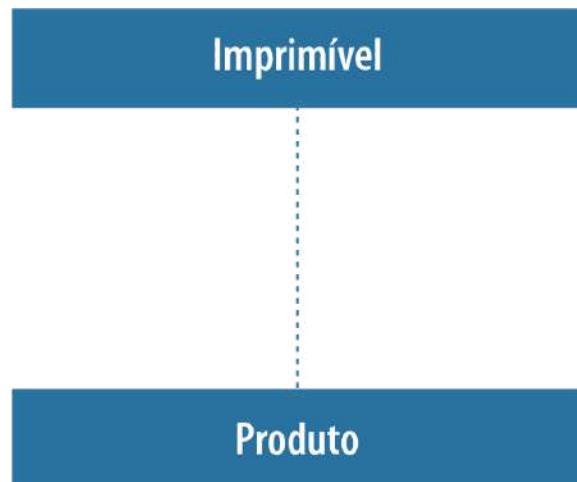


Figura 9 – Interface Imprimível e classe Produto

Na Figura 9 temos uma Interface Imprimivel e uma classe Produto. A implementação da Interface pode ser algo como:

```
public interface Imprimivel {  
    final char nlin = '\n';  
    public String toString();  
    public void toSystemOut();  
}
```

Podemos notar que, ao invés de usar “*public class*”, usamos “*public interface*” para criação de interfaces e não classes. Nessa interface temos uma constante de nome “*nlin*”, definida com o modificador “*final*” e a assinatura de dois métodos “*toString*” e “*toSystemOut*”. Notem que esses métodos são ocos e não possuem implementação, semelhante aos métodos abstratos vistos anteriormente, mas sem o “*abstract*”. Todos os métodos de interface serão obrigatoriamente públicos.

Todas as classes que implementam a interface “Imprimivel” terão a obrigação de implementar os métodos “*toString*” e “*toSystemOut*”. Isso faz com que tenhamos um padrão de programação, assim não importa quais os métodos que as subclasses terão, saberemos que pelo menos esses 2 métodos existirão em todas as classes que implementam a interface.

Todos os métodos em uma interface são implicitamente abstratos, porém o modificador “*abstract*” não é utilizado com métodos de interface, apesar de poder ser, ele é desnecessário.

Veja a seguir como uma classe implementa uma interface, utilizando o operador “*implements*”.

```
public class Produto implements Imprimivel {  
    private String descricao;  
    private int quantidade;  
    public Produto(String d,int q) {  
        descricao = d;  
        quantidade = q;  
    }  
  
    public String toString() {  
        String resp = "Descrição:" + descricao;  
        resp += nlin + "Qtde:" + quantidade;  
        return resp;  
    }  
  
    public void toSystemOut() {  
        String resp = "Descrição:" + descricao;  
        resp += nlin + "Qtde:" + quantidade;  
        System.out.print(resp);  
    }  
}
```

Veja no código da classe Produto que usamos a palavra chave “*implements*” para dizer que a classe implementa a interface. Veja que tivemos que criar os métodos “*toString*” e “*toSystemOut*” dentro da classe.

```
public class TesteProduto {  
    public static void main(String args[]) {  
        Produto prod = new Produto("Macarrão",10);  
        prod.toSystemOut();  
        System.out.println("\n" + prod.toString());  
    }  
}
```

Apesar de não podermos instanciar interfaces, podemos criar variáveis de interface e instanciar objetos de classes que a implementam. O código abaixo teria o mesmo efeito que o anterior:

```
public class TesteProduto {  
    public static void main(String args[]) {  
        Imprimivel prod = new Produto("Macarrão",10);  
        prod.toSystemOut();  
        System.out.println("\n" + prod.toString());  
    }  
}
```

Podemos então perceber que o conceito de classes abstratas é semelhante ao de interfaces. Quando devemos usar uma ou outra?

Use classes abstratas quando você quer definir um “*template*” para subclasses e você possui alguma implementação (métodos concretos) que todas as subclasses podem utilizar. Use interfaces quando você quer definir uma regra que todas as classes que implementem a interface devem seguir, independentemente se pertencem a alguma hierarquia de classes ou não. Claro que podemos também usar ambas no caso de uma simulação de herança múltipla.

Material Complementar

Indicações para saber mais sobre os assuntos abordados nesta Unidade:

Livros

Aprenda Programação Orientada a Objetos em 21 dias

SINTES A. Aprenda programação orientada a objetos em 21 dias. Ed. Pearson; 1^a edição 2002.

Java Como Programar

DEITEL P.; DEITEL H. Java Como Programar. Ed: Pearson Universidades; 8^a edição 2009.

Core Java

HORSTMAN C.; CORNELL H. Core Java. Ed: Pearson Universidades; 8^a edição 2009.

Leitura

Lesson: Object-Oriented Programming Concepts

Clique no botão para conferir o conteúdo.

ACESSE

Referências

SINTES, T. **Aprenda Programação Orientada a Objetos em 21 dias.** 1 ed. São Paulo: Pearson Education do Brasil, 2002, v. 1.

DEITEL, P.; DEITEL, H. **Java Como Programar**, 8. ed. São Paulo: Pearson Education do Brasil, 2010.

FURGERI, S. **Java 2 – Ensino Didático**, EDITORA ÉRICA – 3. ed., 2003.

HORSTMANN, C. S.; CORNELL, G. **Core Java**. 8 ed. São Paulo: Pearson Education do Brasil, 2010, v. 1.