Unidade 5: Orientação a Objetos

Polimorfismo e Abstração: Interfaces

Prof. Daniel Caetano

Objetivo: Revisar e Aplicar os Conceitos de Polimorfismo e Abstração com uso de Interfaces.

Bibliografia: DEITEL, 2005; HOFF, 1996

INTRODUÇÃO

Na aula anterior vimos como o uso de herança pode simplificar o desenvolvimento, seja pela facilidade de expansão de uma classe, seja pela capacidade de substituir uma classe por outra que seja sua descendente.

O uso de classes, entretanto, nem sempre é adequado; por esta razão, veremos uma maneira alternativa para implementar o polimorfismo.

1. QUANDO EXTENDER UMA CLASSE NÃO SERVE

- Quando uma classe não é um subtipo da outra

Quando fazemos uma extensão de uma classe, isto é, usamos o conceito de herança, vimos que sempre temos uma relação direta entre a idéia das duas classes, isto é, se a classe B extende a classe A, dizemos que B é uma versão mais específica de A.

Assim, quando falamos na classe Produto e posteriormente que a classe Livro extende a classe produto, estamos dizendo que qualquer objeto do tipo Livro também é um objeto do tipo Produto e, assim, podemos fazer uso do polimorfismo e, de quebra, isso nos permitia aproveitar muito código da classe Produto.

Bem, algumas vezes temos duas classes diferentes que compartilham um grande número de atributos ou métodos mas, **em termos de idéia, uma não tem nada a ver com outra**. Por exemplo:

Classe: Retangulo
- Largura
- Comprimento
- Preencher()

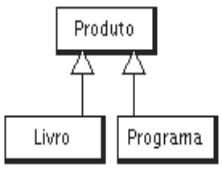
Classe: Piscina
- Largura
- Comprimento
- Profundidade
- Preencher()

Apesar de estas classes compartilharem atributos e métodos - o que ocorre porque, neste caso, a piscina tem a forma plana retangular (um paralelepípedo na verdade), **NÃO** dá para imaginar alguém dizendo que a **Piscina É um Retangulo**.

Assim, dizer que Piscina "extends" Retangulo é inadequado, filosoficamente. Mas a razão para isso não é apenas "filosófica". É prática também: um sistema construído por Classes que se extendem **apenas** para aproveitar código, sem que isso faça um sentido lógico, destrói uma boa parte das vantagens do uso de orientação a objetos, como a facilidade de compreensão de um projeto pronto ou a manutenção do mesmo.

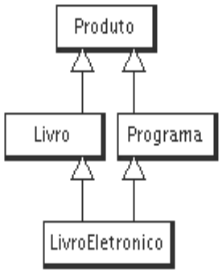
- Quando uma classe não é um subtipo da outra

Uma outra razão que leva a não ser sempre adequado o uso de extensão é quando precisamos herdar características de duas classes diferentes, simultaneamente. Por exemplo, originalmente temos as seguintes classes:



Nossa loja vende Produtos dos tipos Livro e Programa. Cada um tem características bem específicas. Livro não é um Programa e Programa não é um Livro, mas ambos são um Produto.

Quando a loja passa a querer vender Livros Eletrônicos, o projetista/programador conclui que as características deste novo produto levam à idéia de que LivroEletronico deve ser uma classe que extenda, simultaneamente, um Livro **E** um Programa, como indicado a seguir.

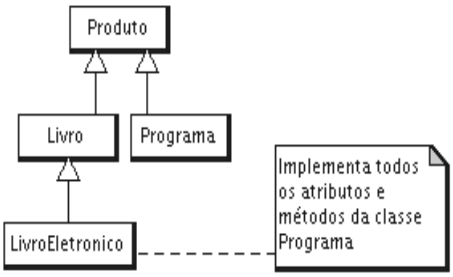


Porém, o programador vai ler a documentação e descobre que o Java não permite que uma classe herde características de várias outras, o que é chamado de **herança múltipla**. As razões detalhadas pelas quais o java não permite isso estão além do escopo deste curso, mas, simplificadamente, a razão para esta limitação é que o uso de herança múltipla, em geral, traz mais dores de cabeça do que resolve problemas.

O programador pensa, então, em duas situações disintas:

A) Implementar LivroEletronico como extensão de Livro
B) Implementar LivroEletronico como extensão de Programa

Nenhuma destas duas soluções é boa; vamos analisar o primeiro caso, que o programador extende a classe Livro e implementa, na classe LivroEletronico, todos os atributos e métodos da classe Programa:



Apesar de o programador ter inserido no LivroEletronico **todas** as características (atributos e métodos) de um Programa (sem reaproveitamento de código), o Java **não tem como saber disso**. Como consequência, as partes do software que souberem manipular um objeto da classe Programa **não poderão usar polimorfismo** com objetos da classe LivroEletronico.

Como na situação B o problema é análogo e, em nenhum dos casos é possível reaproveitar diretamente totalmente o código, chegamos a um beco sem saída? Na verdade, não.

2. AS SALVADORAS INTERFACES

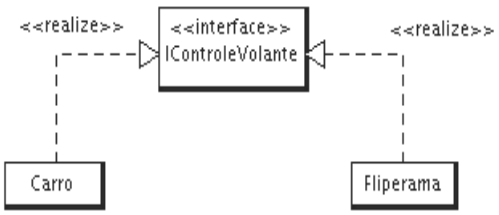
Bem, é claro que os desenvolvedores da Sun (agora Oracle) não são obtusos e, assim, eles criaram um jeito de avisar ao Java quando um objeto de uma classe é capaz de fazer as mesmas coisas que um objeto de outra classe, permitindo o polimorfismo mesmo quando uma classe não é extensão de outra. O nome mais natural que descobriram para esse recurso foi o nome **interface**.

Ora, esse nome faz total sentido, se pensarmos que, no mundo real, só precisamos que conhecer a interface de um objeto, para que saibamos operá-lo. Precisamos saber **qual é o tipo de interface** que ele tem.

Mas, qual é o intuito das interfaces? Em poucas palavras, o objetivo das interface é permitir o polimorfismo sem o uso de herança, exatamente para os casos onde a herança não se justifica ou para os casos em que seria necessária a herança múltipla.

- Quando a herança não faz sentido

O exemplo a seguir permite visualizar o uso de interfaces em situações em que a herança não faz o menor sentido.

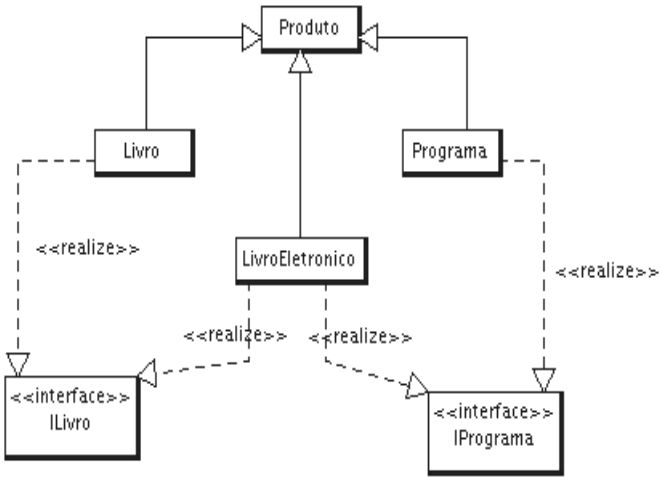


Todos já viram fliperamas de corrida que simulam a cabine de um carro. Ora, embora Carro e Fliperama sejam classes completamente distintas, isto é, não faria o menor sentido dizer "Carro é um Fliperama" ou vice-versa, é possível dizer que ambos **implementam** (*realize*) uma mesma interface, à qual foi dado o nome de **IControleVolante**.

Assim, se em algum ponto do meu sistema eu tiver algum código que saiba usar a interface IControleVolante, ele saberá usar, ao mesmo tempo, qualquer classe que implemente a interface IControleVolante; no exemplo, são as classes Carro e Fliperama. E, observe, não foi necessário o uso de herança!

- Quando a herança faz sentido, mas é necessária uma herança múltipla

O exemplo a seguir permite que tiremos todo o proveido necessário de polimorfismo, sem a necessidade do uso de herança múltipla:



Agora, para que se possa usar polimorfismo genérico, usa-se a classe Produto; para usar um polimorfismo entre classes que possuem a Interface ILivro (como Livro e LivroEletronico) usa-se a **Interface ILivro**. O mesmo vale para a Interface IPrograma.

Observe duas coisas importantes: está explícito que LivroEletronico implementa (*realize*) tanto a interface ILivro quanto a interface IPrograma, o que leva à segunda observação: **uma classe pode implementar várias interfaces**.

Mas como implementar uma interface?

3. AS INTERFACES NO CÓDIGO

Uma interface é declarada diretamente da seguinte forma:

```
[escopo] interface <nome da interface> [extends <outra interface>] {
        // Métodos da interface
        }
```

Exemplo:

IControleVolante.java

```
public interface IControleVolante {
      public void virarDireita();
      public void virarEsquerda();
      }
```

Essa declaração diz **O QUE** um objeto que implemente esta interface deve fazer; observe, entretanto, que interfaces **NÃO DEFINEM ATRIBUTOS** e também **NÃO IMPLEMENTAM CÓDIGO DOS MÉTODOS**. As interfaces servem apenas para que o Java saiba que "toda classe que implementar aquela interface, tem essa lista de métodos implementados".

Quando quisermos implementar uma interface, devemos fazê-lo na declaração da classe, de maneira muito parecida com a herança:

```
[escopo] class <nome da classe> [extends <outra classe>] [implements <interface>]  {
        // definições internas da classe
        // tudo  que estiver aqui *faz parte* da classe
        }
```

Observe atentamente os códigos a seguir, dados como exemplo.

Carro.java

```
public class Carro implements IControleVolante {
	// Atributos do carro
	private String modelo;

	// Métodos do carro
	public String getModelo() { return modelo; }
	public void setModelo(String desc) { modelo = desc; }

	// Métodos da IControleVolante
	public void virarEsquerda() { System.out.println("Esquerda!"); }
	public void virarDireita() { System.out.println("Direita!"); }
	}
```

Outro exemplo:

Fliperama.java

```
public class Fliperama implements IControleVolante {
	// Atributos do fliperama
	private String nome;

	// Métodos do fliperama
	public String getNomeDoJogo() { return nome; }
	public void setNomeDoJogo(String desc) { nome = desc; }

	// Métodos da IControleVolante
	public void virarEsquerda() { System.out.println("Esquerda!"); }
	public void virarDireita() { System.out.println("Direita!"); }
	}
```

E como isso é usado pelo programa? Simples!

3.1. Usando as Interfaces

Suponhamos que temos a nossa classe pessoa, e que ela tenha um método chamado "dirigir", definido da seguinte forma:

Pessoa.java

```
public class Pessoa {
	public void dirigir(Carro umCarro) {
		umCarro.virarDireita();
		umCarro.virarEsquerda();
		}
	}
```

Com essa implementação, um objeto pessoa só pode dirigir um objeto do tipo Carro. Observe o código principal a seguir.

Main.java

```
public class Main {
	public static void main(String[] args) {
		Carro meuCarro;
		Pessoa eu;
		eu = new Pessoa();
		meuCarro = new Carro();

		eu.dirigir(meuCarro);
		}
	}
```

Essse código deve funcionar perfeitamente e imprimir as mensagens "Esquerda" e "Direita". Entretanto, o proximo código, onde indicamos para dirigir um Fliperama, não irá funcionar!

Main.java

```
public class Main {
	public static void main(String[] args) {
		Carro meuCarro;
		Fliperama meuFliperama;
		Pessoa eu;
		eu = new Pessoa();
		meuCarro = new Carro();
		meuFliperama = new Fliperama();

		eu.dirigir(meuCarro);
		eu.dirigir(meuFliperama);
		}
	}
```

Observe que um erro é indicado na última linha deste código do método main da classe Main. Isso ocorre porque o método Pessoa.dirigir está declarado assim:

Pessoa.java

```
public class Pessoa {
	public void dirigir(Carro umCarro) {
		umCarro.virarDireita();
		umCarro.virarEsquerda();
		}
	}
```

Ou seja: ele só sabe usar objetos do tipo Carro, ainda que só sejam usados métodos da IControleVolante (virarDireita e virarEsquerda). Para tornar nosso código mais genérico, podemos mudar a classe Pessoa da seguinte forma:

Pessoa.java

```
public class Pessoa {
	public void dirigir(IControleVolante equipamento) {
		equipamento.virarDireita();
		equipamento.virarEsquerda();
		}
	}
```

Agora não temos mais o erro no código principal, porque tanto Carro quanto Fliperama implementam a interface IControleVolante. Observe, porém, que agora NÃO PODEMOS acessar métodos específicos das classes Carro e/ou Fliperama dentro do método dirigir. Assim, por exemplo, a classe Carro possui o método setModelo() mas, se eu usar o código abaixo, vou ter um erro:

Pessoa.java

```
public class Pessoa {
	public void dirigir(IControleVolante equipamento) {
		equipamento.virarDireita();
		equipamento.virarEsquerda();
		equipamento.setModelo("Vectra");
		}
	}
```

Isso ocorre porque quando dissemos que este método poe usar QUALQUER objeto que implemente a interface IControleVolante, nos comprometemos a usar métodos APENAS desta interface. Como o método setModelo **não está** na interface IControleVolante, eu não posso usá-lo. Isso não me impede, entretanto, de usá-lo na classe principal, onde eu sei exatamente quem é Carro e quem é Fliperama:

Main.java

```
public class Main {
	public static void main(String[] args) {
		Carro meuCarro;
		Fliperama meuFliperama;
		Pessoa eu;
		eu = new Pessoa();
		meuCarro = new Carro();
		meuCarro.setModelo("Omega");
		meuFliperama = new Fliperama();
		meuFliperama.setNomeDoJogo("Super Mário 57");

		System.out.println(meuCarro.getModelo());
		eu.dirigir(meuCarro);
		System.out.println(meuFliperama.getNomeDoJogo());
		eu.dirigir(meuFliperama);
		}
	}
```

NOTA: SEMPRE que for indicado que uma classe implementa uma interface, TODOS os métodos daquela interface precisam ser implementados. A falha em fazê-lo irá causar erros na hora de gerar o programa.

Existe uma forma de contornar esse problema parcialmente: declarar a classe em questão como abstrata:

```
public abstract class MinhaClasse implementes IMinhaInterface {
        }
```

O uso da palavra "abstract" na declaração da classe impede que o java reclame de que algum método não tenha sido implementado; por outro lado, NÃO É POSSÍVEL CRIAR OBJETOS de classe abstrata.

Para que ela serve? Para ser extendida por outra, onde os métodos da Interface faltantes deverão ser implementados.

3.2. Atributos em Interfaces (OPCIONAL)

Em princípio, não se deve indicar atributos nas interfaces; caso eles sejam indicados, só poderão ser do tipo "**static final**", isto é, valores que serão os mesmos em qualquer uma das classes que implementem a interface e que não podem ser mudados.

Em geral esse tipo de recurso é usado para indicar a versão da classe, como mostrado no código a seguir:

IControleVolante.java

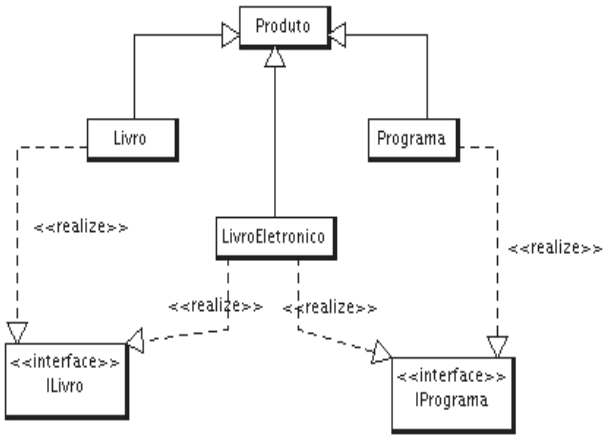
```
public interface IControleVolante {
      public static final int version = 17;

      public void virarDireita();
      public void virarEsquerda();
      }
```

Observe, porém, que nem todo tipo de dado pode ser usado; em especial, os tipos comumente definidos desta forma são os inteiros (int) e os booleanos (boolean).

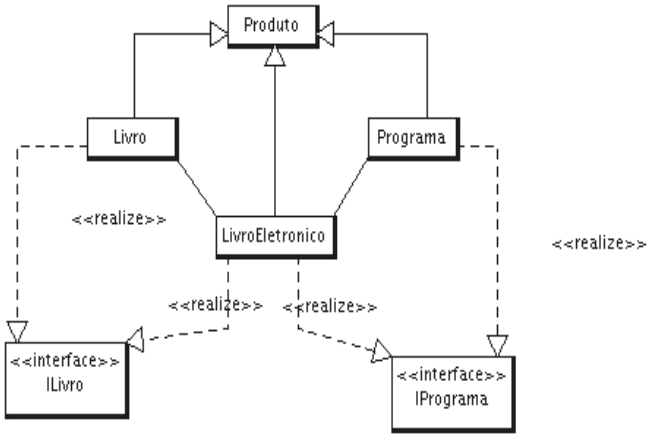
3.3. Usando Interfaces e Reaproveitando Código (OPCIONAL)

Como deve estar claro, até pelo exemplo citado, o uso de interfaces não permite reaproveitamento direto de código... mas isso não significa que não possamos "trapacear".

A idéia é usar uma "classe casca" e, para mostrar como isso é feito, usaremos o nosso exemplo anterior, da livraria. Para refrescar a memória, o exemplo está reproduzido a seguir.



Neste exemplo, embora seja possível o uso do polimorfismo entre objetos do tipo Livro e LivroEletronico (através da interface ILivro) e entre objetos do tipo Programa e LivroEletronico (através da interface IPrograma), os códigos referentes aos métodos destas interfaces precisam estar repetidos em todas as classes que as implementam. Em alguns casos é possível "trapacear" da seguinte forma:



Isso significa que LivroEletronico agora "tem" um Livro e "tem" um Programa, e eu vou usá-los para executar algumas tarefas, ao invés de re-implementá-las. Por exemplo: imagine que a ILivro seja declarada assim:

ILivro.java

```
public interface ILivro {
      public int getNumPaginas();
      }
```

E, a interface IPrograma, seja definida assim:

IPrograma.java

```
public interface ILivro {
      public int getNumBytes();
      }
```

Bem, as classes Livro e Programa poderiam ser definidas assim:

Livro.java

```
public class Livro implements ILivro {
      // Atributos do livro
      private int pags;

      // Construtor
      public Livro(int numPags) { pags = numPags; }

      // Métodos da ILivro
      public int getNumPaginas() { return pags; }
      }
```

Programa.java

```
public class Programa implements IPrograma {
      // Atributos do programa
      private int bytes;

      // Construtor
      public Programa(int numBytes) { bytes = numBytes; }

      // Métodos da IPrograma
      public int getNumBytes() { return bytes; }
      }
```

Agora, para podermos criar a classe LivroEletronico **sem** ter que reprogramar os métodos getNumPaginas() e getNumBytes() - que num programa real podem ser bastante complexos, eu posso construir a minha classe LivroEletronico assim:

LivroEletronico.java

```
public class LivroEletronico implements ILivro, IPrograma {
      // Atributos do LivroEletronico
      private Livro umLivro;
      private Programa umPrograma;

      // Construtor
      public LivroEletronico(int numPags, int numBytes) {
            umLivro = new Livro(numPags);
            umProgroama = new Programa(numBytes);
            }

      // Métodos da ILivro e IPrograma
      public int getNumPaginas() { return umLivro.getNumPaginas; }
      public int getNumBytes() { return umPrograma.getNumBytes; }
      }
```

Observe que ao invés de reimplementar os métodos getNumPaginas() e getNumBytes(), eu implementei métodos que apenas "redirecionam" a responsabilidade para outros objetos.

4. ATIVIDADES DE FIXAÇÃO (NÃO VALEM NOTA!)

Relembrando, uma interface pode ser declarada da seguinte forma:

public inferface nome {
    // definição dos métodos da interface
}

Por exmeplo:

InterfacePessoa.java

```
/**
 * Interface Pessoa
 *
 * @author (seu nome)
 * @version 20100306_001  <<== Versão é muito importante!
 */

public interface InterfacePessoa {
   public String toString();
   public String getNome();
   public int getIdade();
   public void setNome(String novoNome);
   public void setIdade(int novaIdade);
   }
```

A classe Pessoa, por exemplo, pode ser definida como "implementando" a classe InterfacePessoa:

Pessoa.java

```
/**
 * A classe Pessoa define características importantes de uma pessoa.
 *
 * @author (seu nome)
 * @version 20100306_001  <<== Versão é muito importante!
 */

public class Pessoa implements InterfacePessoa {
   //...
```

1. Modifique as classes Homem e Mulher, apresentadas na aula passada, para que implementem a interface InterfacePessoa SEM extender a classe Pessoa.

2. Essa abordagem foi sábia? Por quê?

BIBLIOGRAFIA

DEITEL, H.M; DEITEL, P.J. **Java: como programar** - Sexta edição. São Paulo: Pearson-Prentice Hall, 2005.

HOFF, A; SHAIO, S; STARBUCK, O. **Ligado em Java**. São Paulo: Makron Books, 1996.

SUN MICROSYSTEMS: http://java.sun.com/j2se/5.0/docs/api/index.html