Exercício – Interfaces.

O uso de Interfaces em Java por muitas vezes é feito de forma errada, ou mesmo nem utilizado. Este artigo tem como principal objetivo demonstrar os usos práticos de Interfaces em Java.

Antes de tudo é importante entender qual o conceito principal de Interface: Esta tem objetivo criar um "contrato" onde a Classe que a implementa deve obrigatoriamente obedecer. Na listagem 1 vemos como criar uma simples Interface em Java.

Listagem 1: Minha primeira Interface

```
public interface MinhaPrimeiraInterface {

	/* Métodos que obrigatoriamente
	 * devem ser implementados pela
	 * Classe que implementar esta Interface */
	public void metodo1();
	public int metodo2();
	public String metodo3(String parametro1);

}
```

Perceba que os métodos na interface não têm corpo, apenas assinatura. Agora temos um "contrato" que deve ser seguido caso alguém a implemente. Veja na listagem 2, uma classe que implementa a nossa Interface acima.

Listagem 2: Implementando a Interface

```
public class MinhaClasse implements MinhaPrimeiraInterface {

	@Override
	public void metodo1() {
		// TODO Auto-generated method stub

	}
```

```
	@Override
	public int metodo2() {
		// TODO Auto-generated method stub
		return 0;
	}

	@Override
	public String metodo3(String parametro1) {
		// TODO Auto-generated method stub
		return null;
	}

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		// TODO Auto-generated method stub

	}

}
```

Ao usar a palavra reservada “implements” na MinhaClasse, você verá que a IDE (Eclipse, Netbeans e etc) obriga você a implementar os métodos descritos na Interface.

Usos Práticos da Interface

Tendo conhecimento do uso básico de uma Interface, podemos entender qual a verdadeira funcionalidade dela em um caso real.

Seguindo o Padrão

A Interface é muito utilizada em grandes projetos para obrigar o programador a seguir o padrão do projeto, por esta tratar-se de um contrato onde o mesmo é obrigado a implementar seus métodos, ele deverá sempre seguir o padrão de implementação da Interface.

Vamos supor o seguinte caso: Temos uma Interface BasicoDAO que dirá aos programadores do nosso projeto o que suas classes DAO devem ter (para efeito de conhecimento, o DAO é onde ficará nosso CRUD), qualquer método diferente do que tem na nossa Interface DAO será ignorado e não utilizado.

Listagem 3: Nossa Interface DAO

```
import java.util.List;

public interface BasicoDAO {

	public void salvar(Object bean);
	public void atualizar(Object bean);
	public void deletar(int id);
	public Object getById(int id);
	public List<Object> getAll();



}
```

Agora um dos programadores que está trabalhando no módulo de RH quer criar um DAO para realizar o CRUD de Funcionários, ele implementa a Interface acima e ainda adiciona métodos a parte (que serão ignorados mais a frente).

Listagem 4: Implementado a Interface DAO

```
import java.util.List;

public class FuncionarioDAO implements BasicoDAO {

	@Override
```

```
	public void salvar(Object bean) {
		// TODO Auto-generated method stub

	}

	@Override
	public void atualizar(Object bean) {
		// TODO Auto-generated method stub

	}

	@Override
	public void deletar(int id) {
		// TODO Auto-generated method stub

	}

	@Override
	public Object getById(int id) {
		// TODO Auto-generated method stub
		return null;
	}

	@Override
	public List<Object> getAll() {
		// TODO Auto-generated method stub
		return null;
	}

	//Método a parte criado e implementado pelo próprio
programador
	public void calcularSalario(){

	}
```

```
}
```

Temos agora todos os itens da “receita”, vamos agora utilizá-lo em nossa aplicação. Suponha que um novo programador (que não criou a classe FuncionarioDAO), precise inserir um novo funcionário.

Este novo programador não tem idéia de como foi implementada a classe FuncionarioDAO, ele nem mesmo tem acesso a esta classe, porém ele sabe de algo muito mais importante: A Definição da Interface. Sendo assim ele irá usar todo o poder do polimorfismo e criar um novo objeto FuncionarioDAO do tipo BasicoDAO. Veja a listagem 5.

Listagem 5: Usando o polimorfismo

```
public class MeuApp {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		BasicoDAO funcionarioDAO = new FuncionarioDAO();

		funcionarioDAO.salvar(Funcionario001);

	}

}
```

Perceba que criamos o objeto FuncionarioDAO do tipo BasicoDAO, sendo assim só conseguimos chamar os métodos da Interface BasicoDAO. Mas o mais importante é que o novo programador que utilizará a classe FuncionarioDAO, poderá chamar os métodos descritos na Interface.

Mas o que obriga que o programador que criou a classe FuncionarioDAO implemente BasicoDAO ? Na verdade nada, ele pode

criar FuncionarioDAO sem implementar a interface, porém o novo programador que utilizará essa classe não conseguirá realizar o polimorfismo acima e verá que a classe está errada, foi criada de forma errada, fora do padrão. Há ainda outra forma de sabermos se a classe que foi criada implementou a interface BasicoDAO, veja na listagem 6.

Listagem 6: Uso do instanceof

```
public class MeuApp {

	/**
	 * @param args
	 */
	public static void main(String[] args) {
		FuncionarioDAO funcionarioDAO = new
FuncionarioDAO();

		if (funcionarioDAO instanceof BasicoDAO)
		funcionarioDAO.salvar(Funcionario001);
		else
			System.err.println("A Classe FuncionarioDAO
não implementa BasicoDAO, nenhum procedimento foi realizado");

	}

}
```

Agora conseguimos ver os métodos implementados a parte pelo programador (fora da implementação da interface), porém testamos antes se a classe é uma instancia (instanceof) de BasicoDAO.

Interface de Marcação

Existe ainda um conceito que chamamos de: Interface de Marcação. São interfaces que servem apenas para marcar classes, de forma que ao realizar os "instanceof" podemos testar um conjunto de classe.

Vamos a outro exemplo prático: Temos uma Interface Funcionario sem nenhum método ou atributo, isso porque será apenas uma interface de marcação. Veja na listagem 7.

Listagem 7: Interface de Marcação Funcionario

```
public interface Funcionario {

}
```

Agora criamos 3 Beans, que correspondem a 3 tipos distintos de funcionários: Gerente, Coordenador e Operador. Todos implementando Funcionario.

Listagem 8: Criação de Gerente, Coordenador e Operador

```
public class Gerente implements Funcionario {

	private int id;
	private String nome;

}

public class Coordenador implements Funcionario {

	private int id;
	private String nome;


}

public class Operador implements Funcionario {

	private int id;
```

```
        private String nome;


}
```

Agora em nossa aplicação temos um método que realiza um procedimento de calculo de salário diferente para cada tipo de funcionário. Poderiamos não utilizar o poder da Interface e fazer a implementação abaixo.

Listagem 9: Uso indevido da Interface de Marcação

```
public class MeuApp {


        public void calcularSalarioParaGerente(Gerente gerente){


        }


public void calcularSalarioParaCoordenador(Coordenador
coordenador){


        }


public void calcularSalarioParaOperador(Operador operador){


}


}
```

Muito trabalho pode ser reduzido a apenas 1 método, mostrado na listagem 9.

Listagem 10: Usando a interface de marcação

```
public class MeuApp {


        public void calculaSalarioDeFuncionario(Funcionario
funcionario){
```

```
            if (funcionario instanceof Gerente){
                  //calculo para gerente
            }else if (funcionario instanceof Coordenador){
                  //calculo para coordenador
            }else if (funcionario instanceof Operador){
                  //calculo para operador
            }

      }



}
```

Em vez de ficar criando um método para cada tipo de funcionário, juntamos tudo em apenas 1 utilizando a interface de marcação.

Fonte: Java Interface:(devmedia.com.br)