# Polimorfismo

Revisando o conceito de uma classe que é uma abstração de um conjunto de elementos comuns (atributos e ações) e objeto é uma instancia, ou seja, uma materialização de um desses elementos que representa tal classe. Uma classe concreta é aquela que pode ser instanciada, ou seja, que imediatamente representa o conjunto de elementos.

A interface, para muitos profissionais da área, é considerada um conceito até mais importante do que o entendimento de classe, pois atualmente quase todos os padrões de projeto as utilizam. No contexto da análise de desenvolvimento orientado a objetos, entender o que é e como utilizá-la possibilita o desenvolvimento de soluções mais sofisticadas a nível de flexibilidade e escalabilidade dessas soluções. Uma interface é como um contrato de acordo entre partes, uma definição de obrigatoriedade, podendo essas partes serem entre classe e interface e entre interfaces.

As 3 principais características que uma classe deve ter são: identidade, atributos e métodos. Uma interface esta diretamente associada à execução das ações (métodos). Servem para criar um protocolo ou contrato entre as classes para padronizar as ações. Ex: um aparelho de celular, um outro equipamento tablete e um notebook, que possuem um botão de ligar/desligar. Em todos estes aparelhos possuem obrigatoriamente uma chamada de método para que seja ligado, mas com implementação não obrigatoriamente igual. Quando tratamos de orientação a objetos, estamos criando um modelo ou template e um contrato que defina regras para que uma classe seja implementada. A interface serve como um modelo de implementação sem o conhecimento prévio de quem desenvolve, ou seja, neste exemplo todo aparelho eletrônico deve ter um dispositivo de ligar e desligar mas sem importar como.

Vamos trabalhar com outros exemplos de classes do tipo aparelho eletrônico usando o diagrama de classe, como TV e Cafeteira, ambos implementam a interface IapEletrotronico. Essa interface possui os métodos *ligar()* e *desligar()*. As classes que implementarem essa interface deverão ter também estes métodos, caso contrário ocorrerá erro de exceção em tempo de compilação.

Vamos agora supor que temos uma classe de Usuário e queremos que uma instância dessa classe ligue um dos aparelhos:

*Usuario u = new Usuário(Aparelho ap);*

*{ if(!ap.isLigado())*

*u.ligar(ap)*

*}*

Perceba que se não houvesse o objeto "*ap*" dentro dos parâmetros e a condição deste estar desligado, isso daria um erro do tipo *null pointer exception*, e que não pode ser tratado com o *try-catch*. Existe uma hierarquia para tratamento de erro de exceção.

A interface serve para evitar repetição de métodos de mesmo nome para, neste exemplo, qualquer aparelho. Todos os aparelhos que implementarem esta interface, obrigatoriamente deverão ter esses métodos da interface. A interface pode funcionar como uma classe. Por exemplo: ao invés de criar uma classe abstrata de automóvel e considerar suas classes filhas seja carro, moto, etc eu posso usar uma interface de automóvel e fazer com que as classes carros e motos implementem essa interface:

***IAutomóvel*** *carro = new Carro();*

*carro.setFlex(true);*

Dessa maneira, além do carro ser obrigado a executar os métodos da interface, poderá executar seus próprios métodos. Diferentemente de uma classe abstrata:

***Automóvel*** *carro = new Carro();*

*carro.setFlex(true); (ERRO, isso pode ser resolvido fazendo um cast)*

Em um método eu defino o tipo do objeto, que pode ser um tipo de uma classe ou interface. Nas partes em negrito nos exemplos anteriores são os tipos, respectivamente, interface e classe (abstrata), onde generalizo o tipo de parâmetro que vou receber. Na hora em que envio o parâmetro na chamada do método

estou mandando uma instancia, que pode ser de qualquer tipo que seja ou uma instância do tipo definido como parâmetro ou uma especialização.

Uma classe mãe que implemente uma interface, é suficiente para a implementação para suas classes filhas, ou seja, não é necessário que as filhas também tenham relação direta com uma interface. Mas uma classe pode implementar quantas interfaces forem necessárias, desde que tenham semântica. Isso na representação em um diagrama de classes os métodos devem ser escritos nas classes mais específicas (filhas) e não nas classes mais genéricas (mãe). Uma classe abstrata, embora nunca poderá ser instanciada, tem a vantagem de não ter a obrigatoriedade de implementar uma interface, nem mesmo de herança pois teoricamente método herdado nunca precisa ser implementado.

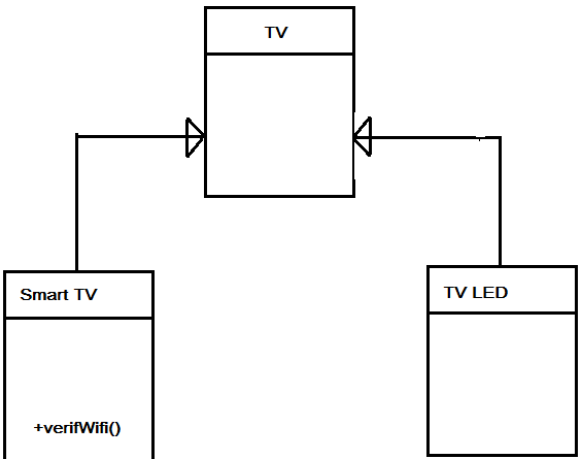
Uma classe abstrata é uma representação de um conjunto de outras classes, que possui um conjunto de características que serão herdadas por outras classes, porém estabelece métodos que precisam ser reimplementados nas classes especializadas (subclasses). Tais métodos podem ser especificados/definidos por uma interface implementada por ela ou novos métodos abstratos (pois **todos os métodos de uma interface por default são abstratos**, ou seja, não possuem implementação). Esses novos métodos abstratos obrigam as subclasses **concretas** implementarem.

Outra vantagem de se utilizar interfaces ao invés de classes, seja concretas ou abstratas, em polimorfismo, é que não existe restrição para herança múltipla entre interfaces. Diferentemente, para as linguagens de programação mais modernas, não é possível realizar herança múltipla entre classes. Isso ocorre no fato de haver um alto acoplamento e baixa coesão, fato que para manutenção de software, caso apareça uma nova necessidade, é uma péssima modelagem, pois na mudança de uma classe poderá gerar mudança em várias outras classes. Essa proibição de herança múltipla foi imposta justamente para evitar mais repetição ou correção de códigos.

Atualmente, na dúvida entre haver herança ou associação entre duas classes, é recomendável optar por associação. O primeiro passo para fazer a modelagem entre as classes é entender os requisitos e regras de negócio.

*ArrayList* é uma classe de manipulação de conjunto de dados, uma implementação de uma estrutura de dados de lista em Java e que pertence a interface *Collection*, que possui um método *add()*, pelo qual é possível adicionar um objeto. Existem duas interfaces herdeira da interface *Collection* chamadas List e Set. A primeira possui um método onde é possível fazer um Get passando o índice, mas o segundo não. A interface List possui uma classe que a implementa chamada *ArrayList* e que, portanto, terá o método *add()* e Get. A interface *Set* por sua vez possui uma classe que a implementa chamada *HashSet* e que terá apenas o método *add()*. Existe uma outra classe que implementa a interface *List* chamada *LinkedList* (lista ligada).

Vamos agora entrar para o assunto de polimorfismo, que é a capacidade de **executar ações diferentes na chamada de um mesmo método**. É consequência de uma herança e da reimplementação que é a sobrescrita de um método. Agora vejamos o seguinte diagrama de classes:




Se quisermos instanciar um objeto SmartTv da seguinte maneira:

```
TV stv = new SmartTV();
    stv.verifWifi();    /*(ERRO! Mas isso pode ser resolvido através de um cast). Ou instanciando um tipo Interface que está sendo implementado por uma classe*/
```

A utilização de interfaces no lugar de classes é muito útil quando quero definir um padrão de ação de classes e que semanticamente possuam simularidades. A classe abstrata tem sentido quando preciso, além das ações, definir atributos. As classes abstratas, mesmo quando indicamos que implementa uma interface, não tem obrigatoriedade de implementar nenhum método mas passa essa obrigatoriedade para suas subclasses concretas.

Vejamos uma implementação utilizando essas interfaces e classes de tipo Collection:

```
package Collection;

import java.util.ArrayList;
import java.util.Collection;
import java.util.List;
import java.util.Scanner;

public class TesteCollection
{     public static void main(String[] args)
          //Classe ArrayList que implementa interface collection
      {     ArrayList lista = new ArrayList();
            //add objetos de tipo string
            lista.add("Eduardo");
            lista.add("Rodrigo");
            lista.add("Marie");
            lista.add("Osamu");
            //looping para varrer todos os objetos da lista
            for(Object o:lista)
            {System.out.print(o+" ");}
      }
}
```

```
Eduardo Rodrigo Marie Osamu
```

No Java existe uma classe chamada *Object*, que por padrão já possui um método chamado *toString()*. Vejamos agora se criarmos uma classe cliente e mandarmos imprimir cada objeto dessa classe com os mesmos nomes:

```java
package Collection;
//classe cliente
public class Cliente
{	private String nome;		//atributo único
	public Cliente(String nome)	//construtor
	{this.nome = nome;}
	//GETTERS e SETTERS
	public String getNome() {return nome;}
	public void setNome(String nome)
	{this.nome = nome;}

}
```

```java
public class TesteCollection
{	public static void main(String[] args)
	{//Classe ArrayList que implementa interface collection
		ArrayList lista = new ArrayList();
		Cliente c1 = new Cliente("Eduardo");
		Cliente c2 = new Cliente("Rodrigo");
		Cliente c3 = new Cliente("Marie");
		Cliente c4 = new Cliente("Osamu");
		//add objetos de tipo string
		lista.add(c1);
		lista.add(c2);
		lista.add(c3);
		lista.add(c4);
		//looping para varrer todos os objetos da lista
		for(Object o:lista) {System.out.print(o.toString()+" ");}
	}
}
```

```
<terminated> TesteCollection [Java Application] C:\Program Files\Java\jre1.8.0_161\bin\javaw.exe (19/08/2018 10:13:24)
Collection.Cliente@6d06d69c Collection.Cliente@7852e922 Collection.Cliente@4e25154f Collection.Cliente@70dea4
```

Perceba que o que foi impresso não foi o nome de cada instancia da classe cliente criada mas sim o código dos objetos. Portanto podemos usar o polimorfismo para implementar um novo método *toString()*.

```java
package Collection;
//classe cliente
public class Cliente
{	private String nome;		//atributo único
	public Cliente(String nome)	//construtor
	{this.nome = nome;}
	//GETTERS e SETTERS
	public String getNome() {return nome;}
	public void setNome(String nome)
	{this.nome = nome;}
	@Override//sobrecarga (polimorfismo)
	public String toString()
	{return nome;}
}
```

```java
//Usando a mesma codificação anterior da Classe TestCollection
```

```
<terminated> TesteCollection [Java Application] (
Eduardo  Rodrigo  Marie  Osamu
```

Vamos agora criar uma nova classe chamada produto:

```java
package Collection;
//classe produto
```

```java
public class Produto
{	private String nome;
	public Produto(String nome)
	{this.nome = nome;}
	public String getNome() {return nome;}
	public void setNome(String nome) {this.nome = nome;}
	@Override//sobreescrita (polimorfismo)
	public String toString()
	{return nome;}
}
```

```java
public class TesteCollection
{	public static void main(String[] args)
	{	ArrayList lista = new ArrayList(); //Classe ArrayList que
implementa interface collection
		//instanciand objetos da classe cliente
		Cliente c1 = new Cliente("Eduardo");
		Cliente c2 = new Cliente("Rodrigo");
		Cliente c3 = new Cliente("Marie");
		Cliente c4 = new Cliente("Osamu");
		//instanciando objetos de tipo produto
		Produto p1 = new Produto("chocolate");
		//add objetos de tipo cliente na lista
		lista.add(c1);
		lista.add(c2);
		lista.add(c3);
		lista.add(c4);
		//add objeto de tipo produto na lista
		lista.add(p1);
		//looping para varrer todos os objetos da lista
		for(Object c:lista) {System.out.print(c.toString()+" ");}
	}
}
```

```
<terminated> TesteCollection [Java Application] C:\Progra
Eduardo Rodrigo Marie Osamu chocolate
```

Agora, para dificultar, queremos que quando um objeto seja add na lista, um log seja criado. Vamos criar uma classe qualquer:

```java
package Collection;
import java.util.ArrayList;
public class ListOf22018 extends ArrayList
{	@Override//sobreescrita (polimorfismo)
	public boolean add(Object arg0)
	//imprime uma mensagem de tipo erro indicando um hashcode
	{	System.err.println("Objeto "+arg0.hashCode()+"
				adicionado");
		return super.add(arg0);
	}
}
```

```java
public class TesteCollection
{	public static void main(String[] args)
	//Classe ArrayList que implementa interface collection
	{	//ArrayList lista = new ArrayList();
		//lista de log
		ListOf22018 lista = new ListOf22018();
		//instanciand objetos da classe cliente
		Cliente c1 = new Cliente("Eduardo");
```

```
            Cliente c2 = new Cliente("Rodrigo");
            Cliente c3 = new Cliente("Marie");
            Cliente c4 = new Cliente("Osamu");
            //instanciando objetos de tipo produto
            Produto p1 = new Produto("chocolate");
            //add objetos de tipo cliente na lista
            lista.add(c1);
            lista.add(c2);
            lista.add(c3);
            lista.add(c4);
            //add objeto de tipo produto na lista
            lista.add(p1);
            //looping para varrer todos os objetos da lista
            for(Object c:lista) {System.out.print(c.toString()+" ");}
      }
}
```

```
<terminated> TesteCollection [Java Application] C:\Progr
Objeto 1829164700 adicionado
Objeto 2018699554 adicionado
Objeto 1311053135 adicionado
Objeto 118352462 adicionado
Objeto 1550089733 adicionado
Eduardo Rodrigo Marie Osamu chocolate
```

Vamos agora fazer com que o usuário decida se quer log ou não:

```
import java.util.Scanner;
public class TesteCollection
{     public static void main(String[] args)
      //Classe ArrayList que implementa interface collection
      {     //ArrayList lista = new ArrayList();
            //Aguarda usuário optar por log ou não
            System.out.println("Quer log?");
            Scanner sc = new Scanner(System.in);
            char log = sc.nextLine().charAt(0);
            //lista de log, da interface List que implementa
            //Collection
            List lista;
            //se usuário optar por sim
            if(log == 's')
                  //Cria uma lista que herda de ArrayList com
                  //polimorfismo
                  lista = new ListOf22018();
            //senão
            else
                  //cria somente um tipo ArrayList
                  lista = new ArrayList();
            //instanciando objetos da classe cliente
            Cliente c1 = new Cliente("Eduardo");
            Cliente c2 = new Cliente("Rodrigo");
            Cliente c3 = new Cliente("Marie");
            Cliente c4 = new Cliente("Osamu");
            //instanciando objetos de tipo produto
            Produto p1 = new Produto("chocolate");
            //add objetos de tipo cliente na lista
            lista.add(c1);
            lista.add(c2);
            lista.add(c3);
            lista.add(c4);
```

```
            //add objeto de tipo produto na lista
            lista.add(p1);
            //looping para varrer todos os objetos da lista
            for(Object c:lista) {System.out.print(c.toString()+" ");}
      }
}
Quer log?
s
Objeto 460141958 adicionado
Objeto 1163157884 adicionado
Objeto 1956725890 adicionado
Objeto 356573597 adicionado
Objeto 1735600054 adicionado
Eduardo Rodrigo Marie Osamu chocolate
Quer log?
n
Eduardo Rodrigo Marie Osamu chocolate
```

O *hashset* é um pouco mais rápido devido ao seu modo de organizar as coisas, portanto vamos implementar esse mesmo programa usando esta classe:

```
package Collection;
import java.util.ArrayList;
import java.util.Collection;
import java.util.HashSet;
import java.util.List;
import java.util.Scanner;
public class TesteCollection
{     public static void main(String[] args)
      //Classe ArrayList que implementa interface collection
      {     //ArrayList lista = new ArrayList();
            //Aguarda usuário optar por log ou não
            System.out.println("Quer log?");
            Scanner sc = new Scanner(System.in);
            char log = sc.nextLine().charAt(0);
            //lista de log, da interface List que implementa
            //Collection
            //List lista;

            //troca por tipo Collection Para poder chamar o tipo
            //Hashset
            Collection lista;
            //se usuário optar por sim
            if(log == 's')
                  //Cria uma lista que herda de ArrayList com
                  //polimorfismo
                  lista = new ListOf22018();
            //senão
            else
            {     System.out.println("Quer ordená-la?");
                  char ordenar = sc.nextLine().charAt(0);
                  //se optar por sim
                  if(ordenar == 's') //chama apenas ArrayList
                        lista = new ArrayList();
                  //senão
                  else//a ordenação fica embaralhada
                        lista = new HashSet();
            }
            /*else
                  //cria somente um tipo ArrayList
                  lista = new ArrayList();*/
```

```
            //instanciando objetos da classe cliente
            Cliente c1 = new Cliente("Eduardo");
            Cliente c2 = new Cliente("Rodrigo");
            Cliente c3 = new Cliente("Marie");
            Cliente c4 = new Cliente("Osamu");
            //instanciando objetos de tipo produto
            Produto p1 = new Produto("chocolate");
            //add objetos de tipo cliente na lista
            lista.add(c1);
            lista.add(c2);
            lista.add(c3);
            lista.add(c4);
            //add objeto de tipo produto na lista
            lista.add(p1);
            //looping para varrer todos os objetos da lista
            for(Object c:lista) {System.out.print(c.toString()+" ");}
      }
}
```

```
Quer log?
n
Quer ordená-la?
n
Marie chocolate Rodrigo Eduardo Osamu
```

```
Quer log?
n
Quer ordená-la?
s
Eduardo Rodrigo Marie Osamu chocolate
```

**Obs: Perceba que a ordenação que estamos tratando é de acordo com a ordem que foi adicionado na lista e não em ordem alfabética ou algo do tipo.**

O *HashSet* é uma estrutura (pilha) de *hash* que não garante a ordenação dos objetos que sejam inseridos numa lista, mas o objetivo dela é a performance. Portanto é importante sabermos quando criamos atributos de tipo interface, pois isso nos possibilita de, depois no meio do caminho, mudar o tipo da instancia. Se já definirmos o tipo da instância, jamais poderíamos, neste caso, usar o *HashSet* por exemplo.

Vamos criar um método tipo *static*, fora da *main* chamado imprimir, e que receba como parâmetro um tipo *Collection*. Perceba que o resultado será o mesmo.

```
package Collection;
import java.util.ArrayList;
import java.util.Collection;
import java.util.HashSet;
import java.util.List;
import java.util.Scanner;
public class TesteCollection
{     public static void main(String[] args)
      //Classe ArrayList que implementa interface collection
      {     //ArrayList lista = new ArrayList();
            //Aguarda usuário optar por log ou não
            System.out.println("Quer log?");
            Scanner sc = new Scanner(System.in);
```

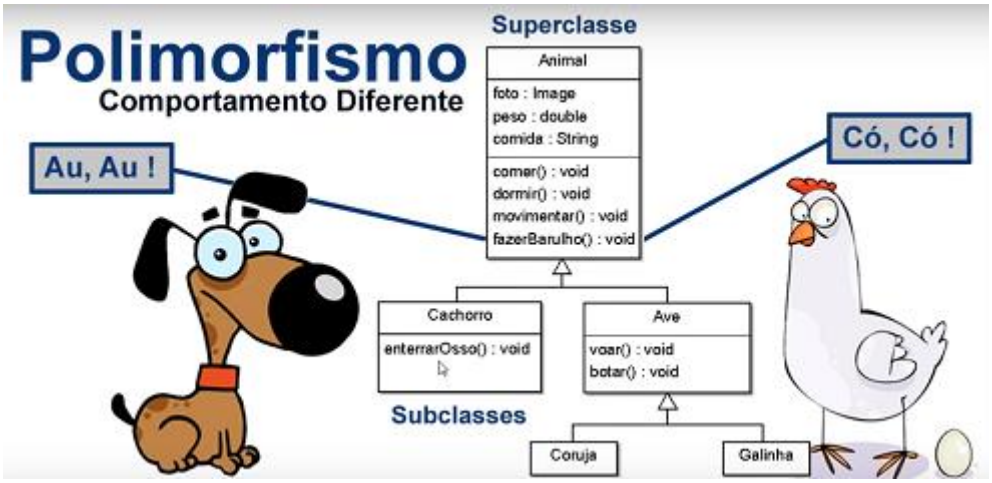
```
            char log = sc.nextLine().charAt(0);
            //lista de log, da interface List que implementa
            //Collection
            //List lista;

            //troca por tipo Collection (dá na mesma)
            Collection lista;
            //se usuário optar por sim
            if(log == 's')
                  //Cria uma lista que herda de ArrayList com
                  //polimorfismo
                  lista = new ListOf22018();
            //senão
            else
            {     System.out.println("Quer ordená-la?");
                  char ordenar = sc.nextLine().charAt(0);
                  //se optar por sim
                  if(ordenar == 's')
                        lista = new ArrayList();
                  //senão
                  else
                        lista = new HashSet();
            }
            /*else
                  //cria somente um tipo ArrayList
                  lista = new ArrayList();*/
            //instanciando objetos da classe cliente
            Cliente c1 = new Cliente("Eduardo");
            Cliente c2 = new Cliente("Rodrigo");
            Cliente c3 = new Cliente("Marie");
            Cliente c4 = new Cliente("Osamu");
            //instanciando objetos de tipo produto
            Produto p1 = new Produto("chocolate");
            //add objetos de tipo cliente na lista
            lista.add(c1);
            lista.add(c2);
            lista.add(c3);
            lista.add(c4);
            //add objeto de tipo produto na lista
            lista.add(p1);
            //looping para varrer todos os objetos da lista
            //for(Object c:lista) {System.out.print(c.toString()+"
            //");}
            Imprimir(lista);
      }

      //método estatico da classe
      public static void Imprimir(Collection coll)
      {     for(Object c:coll)
            {System.out.print(c.toString()+" ");}
      }
}
```

Portanto, quanto mais generalizado a interface ou classe abstrata for inserido, quando houver polimorfismo, abrirá maiores possibilidades de aproveitamento de suas subinterfaces ou subclasses e seus respectivos métodos no seu programa.

Exemplos de Polimorfismo:




Neste exemplo temos dois tipos de animais (subclasses da classe animal). A segunda hierarquia de herança que podemos ver da classe abstrata de Ave não seria possível no Java por exemplo (herança múltipla). No conceito de polimorfismo, indica que a super classe pode ter vários tipos, afinal nós temos neste caso vários tipos de animais ou várias formas diferentes com comportamentos diferentes, conforme podemos ver na imagem acima.

Infelizmente a herança nem sempre é eficiente. Por exemplo, nem todo animal faz o mesmo tipo de barulho e que, portanto, dependendo da forma que tal animal receber essa ação poderá ser executada de forma diferente. Por exemplo: o tipo cachorro deverá fazer um barulho semelhante a “Auau” e o mesmo podemos dizer com relação a galinha. Sendo assim, podemos modificar o comportamento de cada animal de diferentes tipos e é disso que trata o conceito de polimorfismo.

**Sobreescritas de métodos (override):**

Supondo o método citado anteriormente de *fazerBarulho()* e reescreve-la sempre que for necessário. Neste exemplo, tanto em cachorro quanto na galinha será necessário sobre escrever esse método, pois ambos as executam na realidade, mas de formas diferentes.

```
public static void main(String[] args) {

    Animal generico = new Animal(0,null);
    Animal toto = new Cachorro();
    Animal carijo = new Galinha();

    generico.fazerBarulho();
    toto.fazerBarulho();
    carijo.fazerBarulho();
```

**Obs: repare que nesses casos e em seus semelhantes, só poderiam ser executados os métodos que estejam na classe genérica, ou seja, qualquer outros métodos que pertencem às classes mais específicas, não poderiam ser executados.**

Caso optássemos em não utilizar o conceito de polimorfismo, é interessante fazer pesquisas na internet maneiras diferentes de substituir condicionais ou fluxos condicionais por polimorfismo, que é considerada a maneira de maior alto nível de se programar. Se não estivéssemos utilizando polimorfismo, precisaríamos fazer o seguinte: definir o tipo do animal e dependendo deste tipo (inserir uma condição) deverá escrever diretamente o tipo de barulho que este deverá fazer.

```
public static void barulho(String animal) {
    if(animal.equals("Cachorro")) {
        System.out.println("Au, Au !");
    } else if (animal.equals("Galinha")) {
        System.out.println("Có, Có !");
    }
```

Perceba que o código ficou já bem mais complexo do que o anterior. Portanto, imagine se tivéssemos centenas de tipos de animais diferentes para executar o mesmo método.

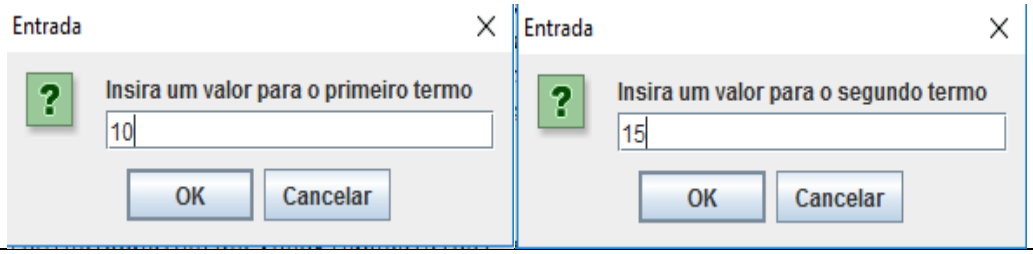
O polimorfismo te possibilita, modificar o comportamento natural da herança, ou seja, do método de origem de uma superclasse ser modificiada na(s) sua(s) subclasse(s). Vamos criar um outro exemplo que faça vários tipos de operações matemáticas.

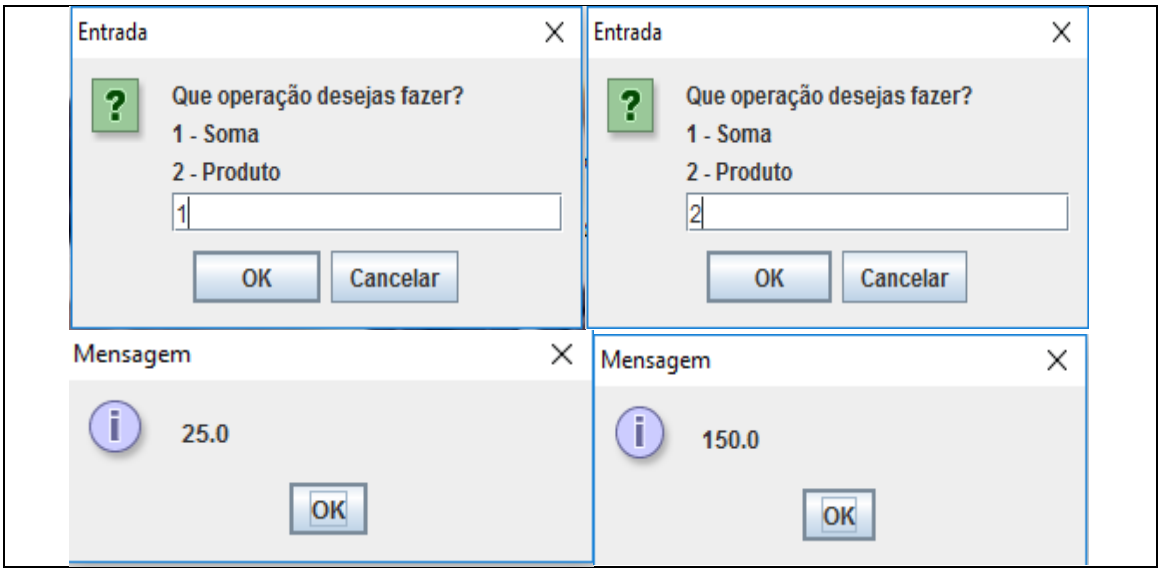
```java
public abstract class OperacoeMatematica
{	public double Calcular(double x, double y)
	{return 0.;}
}
```

```java
public class Soma extends OperacoeMatematica
{	@Override
	public double Calcular(double x, double y)
	{return x + y;}
}
```

```java
public class Multiplicacao extends OperacoeMatematica
{	@Override
	public double Calcular(double x, double y)
	{return x * y;}
}
```

```java
//classe swing que permite chamar o input de dados
import javax.swing.JOptionPane;
//classe que implementa os cálculos
public class CalcularValor
{	public static void main(String[] args)
	//primeiro parâmetro
	{	double a = Double.parseDouble(
			JOptionPane.showInputDialog(
				"Insira um valor para o primeiro termo"));
		//segundo parâmetro
		double b = Double.parseDouble(
				JOptionPane.showInputDialog(
				"Insira um valor para o segundo termo"));
		//instancia uma classe genérica com valor de objeto
		//inicial nulo
		OperacoeMatematica op = null;
		//pergunta ao usuário que tipo de operação deseja realizar
		int opcao = Integer.parseInt(JOptionPane.showInputDialog(
		"Que operação desejas fazer?\n1 - Soma\n2 - Produto"));
		switch(opcao)
		{	case 1://opção de soma
				op = new Soma();
				break;
			case 2://opção de multiplicação
				op = new Multiplicacao();
				break;
			default://opção inválida
				break;
		}
		//se o objeto não for nulo
		if(op != null) //retorna o valor da operação desejada
		   JOptionPane.showMessageDialog(null, op.Calcular(a, b));
		//se não
		else JOptionPane.showMessageDialog(null,"Opção inválida");
	}
}
```






**Classe Abstrata:**

Na aula anterior, definimos que a classe animal é um supertipo de classe e que a classe Cachorro e Galinha são classes específicas, que possuem formas e comportamentos completamente diferentes (específicas). Uma classe abstrata, neste exemplo, é a forma de um animal bem genérico e que em nenhum momento poderá ser instanciada. Essas classes abstratas possuem métodos (comportamentos) indefinidos, exatamente para obrigar o programador implementa-las nas suas subclasses. Os métodos podem tanto ter como não corpo de código na classe abstrata. Os métodos sem corpo devem sempre vir com a palavra *abstract* e finalizada com ";".

Voltando ao exemplo das operações matemáticas:

```
public abstract class OperacoeMatematica
{public abstract double Calcular(double x, double y);}
```

```
public class Soma extends OperacoeMatematica
{     @Override
      public double Calcular(double x, double y)
      {return x + y;}
}
```

```
public class Multiplicacao extends OperacoeMatematica
{     @Override
      public double Calcular(double x, double y)
      {return x * y;}
}
```

```
//classe swing que permite chamar o input de dados
import javax.swing.JOptionPane;
//classe que implementa os cálculos
public class CalcularValor
```

```
{     public static void main(String[] args)
      //primeiro parâmetro
      {     double a = Double.parseDouble(
                  JOptionPane.showInputDialog(
                        "Insira um valor para o primeiro termo"));
            //segundo parâmetro
            double b = Double.parseDouble(
                        JOptionPane.showInputDialog(
                        "Insira um valor para o segundo termo"));
            //instancia uma classe genérica com valor de objeto
            //inicial nulo
            OperacoeMatematica op = null;
            //pergunta ao usuário que tipo de operação deseja realizar
            int opcao = Integer.parseInt(JOptionPane.showInputDialog(
            "Que operação desejas fazer?\n1 - Soma\n2 -Produto"));
            switch(opcao)
            {     case 1://se a opção for soma
                        op = new Soma();
                        break;
                  case 2: //se a opção for produto
                        op = new Multiplicacao();
                        break;
                  default: //se a opção digitada for invalida
                        JOptionPane.showMessageDialog(null,
                                                "Opção inválida!");
                        break;
            }
            if(op != null) calcule(op, a, b);
      }
      //método pois todo arquivo de java é uma classe, mesmo as de
      //execução.
      public static void calcule(OperacoeMatematica op, double x,
                                    double y)
      {     op.Calcular(x, y);
            JOptionPane.showMessageDialog(null, op.Calcular(x, y));
      }
}
```

Os métodos abstratos só podem existir em classes que não podem ser instanciadas, ou seja, em classes abstratas ou interfaces, que será assunto mais adiante. Os métodos abstratos só não serão obrigados a serem implementados nas subclasses se estas também forem classes abstratas. Os métodos abstratos só devem ser implementados pela primeira classe concreta da herança.

## Interface

Uma forma de padronizar as interações da sua aplicação usando interfaces. A vantagem de se utilizar interfaces ao invés de classes é o poder da herança múltipla entre interfaces. Relacionar tipos de objetos díspares e a interface é muito importante dentro de

um projeto orientado a objetos e também saber que as variáveis dentro de uma interface são sempre constantes (*static final*).

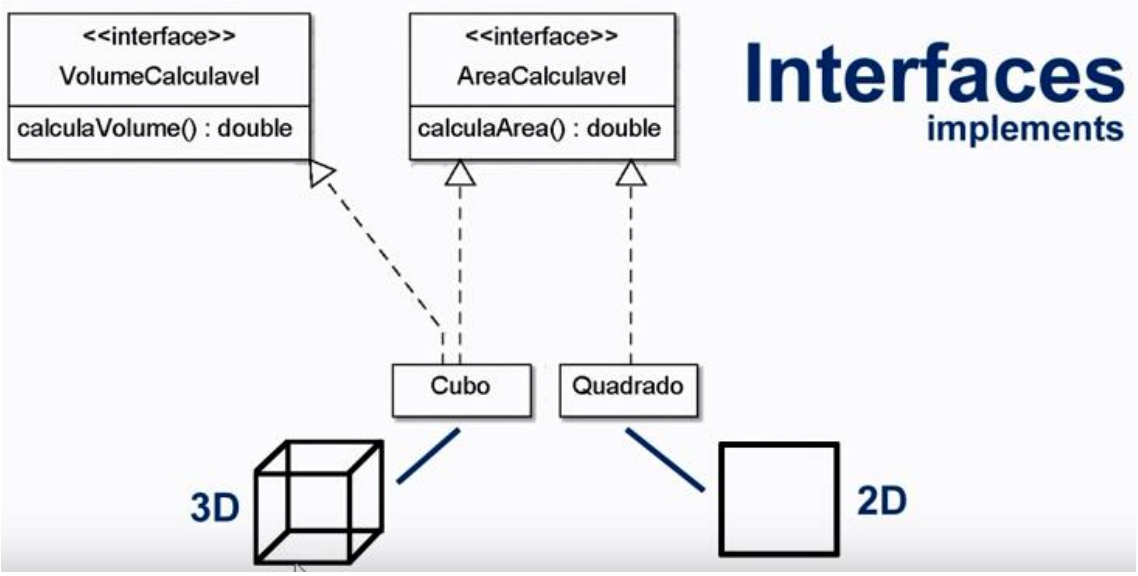
A criação de uma interface é muito semelhante a uma classe, pois ela também possui um nome e pode receber um modificador *public*. Mas diferente das classes utiliza a palavra identificadora chave *interface*. Dentro da interface **só pode declarar métodos abstratos**, embora não seja necessário declará-los (no java) como abstratos e nem como públicos pois os modificadores são redundantes, uma vez que todas as classes que as implementarem serão obrigados a implementar todos os seus métodos.

Diferentemente das classes abstratas, que obrigatoriamente necessitam ter os modificadores, descritos anteriormente, declarados em seus métodos. Além disso podem também implementar alguns métodos. Métodos abstratos são todos que não possuem corpo de código.

Como já dito anteriormente, as interfaces tem como papel principal padronizar as interações das aplicações. Ex: os controles em um rádio servem como interface entre os usuários do radio e os componentes internos do radio. Os controles permitem que os usuários realizem somente uma série limitada de operações, como mudança de estação, ajuste de volume, troca de FM para AM e vice e versa, e os diferentes rádios podem implementar os controles de formas diferentes como uso de botões ou sintonização automática, comando de voz. A interface especifica quais operações que um rádio deve permitir que os usuários realizem, mas não especifica como essas operações são realizadas. Diferente de uma classe abstrata, por exemplo, que algumas operações ela mesma implementa ou define como será a execução de tal método. Dentro de uma interface todos os métodos serão implementados pelas primeiras classes concretas que a implementarem.

Uma outra característica de uma interface é que aceita a herança múltipla entre interfaces, o que não é possível na maioria das linguagens orientados a objetos mais recentes. Uma classe qualquer só pode herdar de uma única classe, mas pode implementar quantas interfaces forem necessárias e não há limites.

Outra características das interfaces, é que se pode relacionar tipos díspares, ou seja, uma interface é geralmente utilizada quando classes díspares, ou seja, que não sejam relacionadas entre si, precisam compartilhar métodos e constantes comuns. Isso permite que objetos de classes que não estejam relacionadas, sejam processadas polimórficamente. Portanto, os objetos de classes que implementam a mesma interface podem responder as mesmas chamadas de métodos, ou seja, temos dois objetos que de alguma forma tem uma origem em comum pois ambos os objetos são formas geométricas, sendo um 2D e outro 3D.

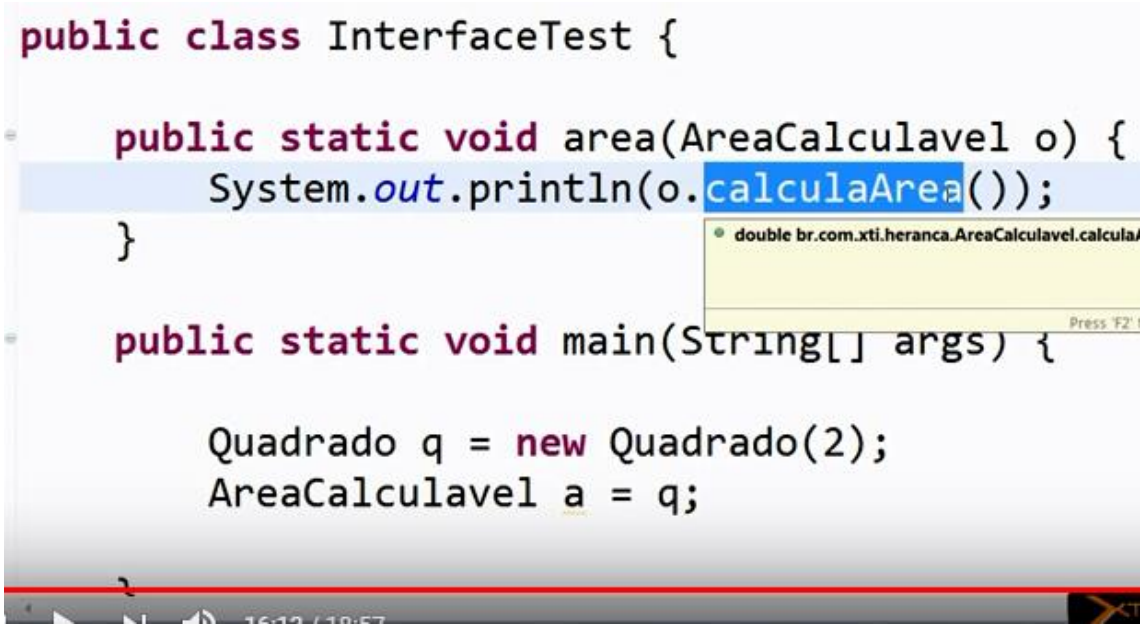


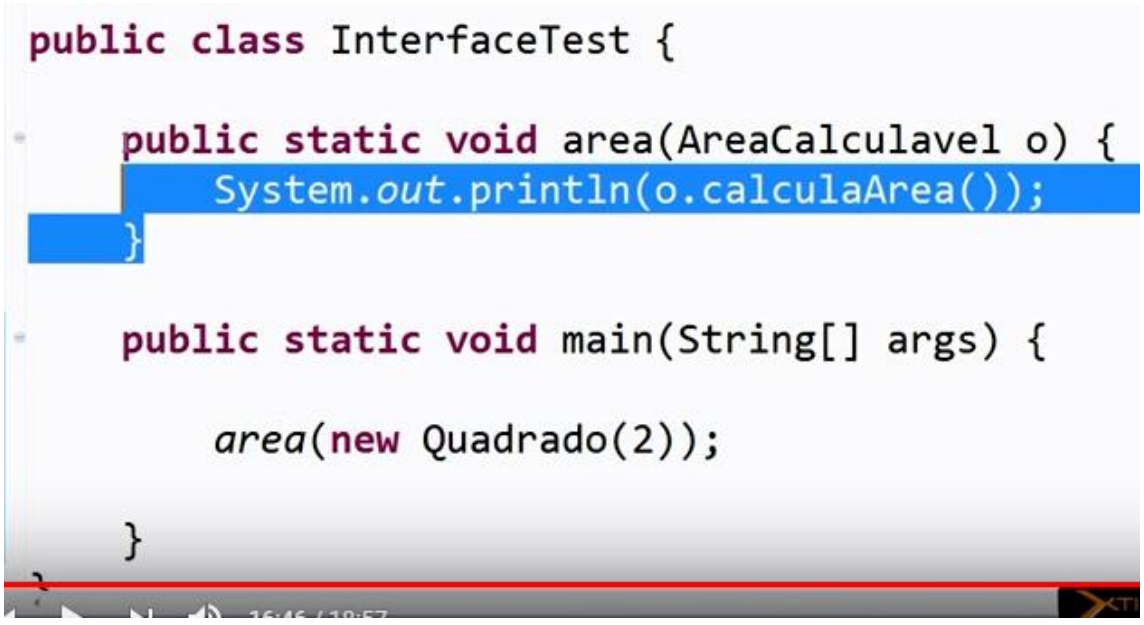

Ambos possuem uma relação com a interface AreaCalculavel. Mesmo não tendo nenhuma relação direta na vida real com uma interface que é implementada por, por exemplo, aparelhos eletrônicos, poderá se aproveitar de alguns métodos. Resumindo, a interface possui essa flexibilidade. Já uma classe abstrata não possui essa vantagem, onde todos os objetos que a herdarem, estão seguindo uma herança simples e não disponibilizam seus métodos a tipos diferentes.

Quando estamos definindo um projeto, o que estivermos declarando ou priorizando neste são os tipos, que é a unidade fundamental de programação. Na linguagem Java é a classe, ou seja, é na classe que serão implementados os algoritmos e códigos. O que é considerado dentro de um projeto é o tipo de dado. Enquanto classes definem tipos, é bastante útil e poderoso que

possamos sempre definir um tipo sem necessariamente definir uma classe e implementá-la.

Portanto uma interface é uma expressão de projeto puro, enquanto que uma classe é uma mistura de um projeto e de implementação. Uma classe pode implementar mais de uma interface quantas forem necessárias e seus métodos de qualquer maneira que o projetista escolher.




Por fim, uma outra característica é o atributo das interfaces sempre deverão ser constantes. Por default, todos os atributos de uma interface já são *public static final*.