==========

Programação Orientada a Objetos
Atividade 4
Classes Abstratas e Interfaces
Fonte: Java e Orientação a Objetos – Caelum

==========

1. Repare que a nossa classe Conta é uma excelente candidata para uma classe abstrata. Por quê? Que métodos seriam interessantes candidatos a serem abstratos? Transforme a classe Conta em abstrata, repare o que acontece no seu main já existente do TestaContas.

2. Para que o código do main volte a compilar, troque o new Conta() por new ContaCorrente(). Se agora não podemos dar new em Conta, qual é a utilidade de ter um método que recebe uma referência a Conta como argumento? Aliás, posso ter isso?

3. A sintaxe do uso de interfaces pode parecer muito estranha, à primeira vista. Vamos começar com um exercício para praticar a sintaxe:

```
interface AreaCalculavel {
    double calculaArea();
}
```

Queremos, agora, criar algumas classes que são `AreaCalculavel`:

```
class Quadrado implements AreaCalculavel {
    private int lado;

    public Quadrado(int lado) {
        this.lado = lado;
    }

    public double calculaArea() {

         return this.lado * this.lado;
     }
 }

 class Retangulo implements AreaCalculavel {
     private int largura;
     private int altura;

     public Retangulo(int largura, int altura) {
         this.largura = largura;
         this.altura = altura;
     }

     public double calculaArea() {
         return this.largura * this.altura;
     }
 }
```

Repare que, aqui, se você tivesse usado herança, não ia ganhar muita coisa, já que cada implementação é totalmente diferente da outra: um Quadrado, um Retangulo e um Circulo têm atributos e métodos bem diferentes.
Crie uma classe de Teste. Repare no polimorfismo. Poderíamos estar passando esses objetos como argumento para alguém que aceitasse AreaCalculavel como argumento:

```
class Teste {
    public static void main(String[] args) {
        AreaCalculavel a = new Retangulo(3,2);
        System.out.println(a.calculaArea());
    }
}
```

Opcionamente, crie a classe `Circulo`:

```
class Circulo implements AreaCalculavel {
    // ... atributos (raio) e métodos (calculaArea)
}
```

Utilize `Math.PI * raio * raio` para calcular a área.

4. Nosso banco precisa tributar dinheiro de alguns bens que nossos clientes possuem. Para isso, vamos criar uma interface:

```
interface Tributavel {
    double calculaTributos();
}
```

Lemos essa interface da seguinte maneira: "todos que quiserem ser *tributável* precisam saber *calcular tributos*, devolvendo um double".

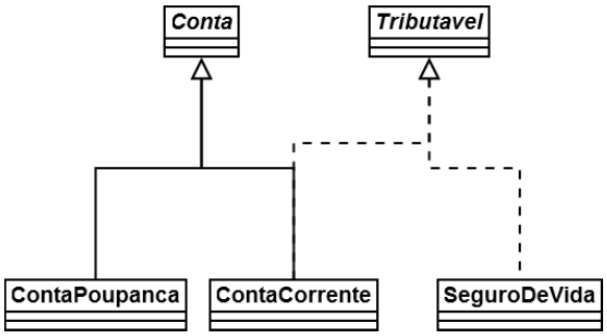
Alguns bens são tributáveis e outros não, `ContaPoupanca` não é tributável, já para `ContaCorrente` você precisa pagar 1% da conta e o `SeguroDeVida` tem uma taxa fixa de 42 reais. (faça a mudança em `ContaCorrente` e crie a classe `SeguroDeVida`):

```
class ContaCorrente extends Conta implements Tributavel {

    // outros atributos e metodos

    public double calculaTributos() {
        return this.saldo * 0.01;
    }
}

class SeguroDeVida implements Tributavel {
    public double calculaTributos() {
        return 42;
    }
}
```



Vamos criar uma classe `TestaTributavel` com um método `main` para testar o nosso exemplo:

```
class TestaTributavel {

    public static void main(String[] args) {
        ContaCorrente cc = new ContaCorrente();
        cc.deposita(100);
        Tributavel t = cc;
        System.out.println(t.calculaTributos());
    }
}
```

Tente, agora, chamar o método `getSaldo` através da referência `t`, o que ocorre? Por quê?

A linha em que atribuímos `cc` a um `Tributavel` é apenas para você enxergar que é possível fazê-lo. Nesse nosso caso, isso não tem uma utilidade. Essa possibilidade será útil para o próximo exercício.

5. Crie um GerenciadorDeImpostoDeRenda, que recebe todos os tributáveis de uma pessoa e soma seus valores, e inclua nele um método para devolver seu total:

```
class GerenciadorDeImpostoDeRenda {
    private double total;

    void adiciona(Tributavel t) {
        System.out.println("Adicionando tributavel: " + t);

        this.total = this.total + t.calculaTributos();
    }

    public double getTotal() {
        return total;
    }
}
```

Crie um `main` para instanciar diversas classes que implementam `Tributavel` e passar como argumento para um `GerenciadorDeImpostoDeRenda`. Repare que você não pode passar qualquer tipo de conta para o método adiciona, apenas a que implementa `Tributavel`. Além disso, pode passar o `SeguroDeVida`.

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

        GerenciadorDeImpostoDeRenda gerenciador = new GerenciadorDeImpostoDeRenda();

        SeguroDeVida sv = new SeguroDeVida();
        gerenciador.adiciona(sv);

        ContaCorrente cc = new ContaCorrente();
        cc.deposita(1000);
        gerenciador.adiciona(cc);

        System.out.println(gerenciador.getTotal());
    }
}
```

Repare que, de dentro do `GerenciadorDeImpostoDeRenda`, você não pode acessar o método `getSaldo`, por exemplo, pois você não tem a garantia de que o `Tributavel` que vai ser passado como argumento tem esse método. A única certeza que você tem é de que esse objeto tem os métodos declarados na interface `Tributavel`.

É interessante enxergar que as interfaces (como aqui, no caso, `Tributavel`) costumam ligar classes muito distintas, unindo-as por uma característica que elas tem em comum. No nosso exemplo, `SeguroDeVida` e `ContaCorrente` são entidades completamente distintas, porém ambas possuem a característica de serem tributáveis.

Se amanhã o governo começar a tributar até mesmo `PlanoDeCapitalizacao`, basta que essa classe implemente a interface `Tributavel`! Repare no grau de desacoplamento que temos: a classe `GerenciadorDeImpostoDeRenda` nem imagina que vai trabalhar como `PlanoDeCapitalizacao`. Para ela, a única coisa que importa é que o objeto respeite o contrato de um tributável, isso é, a interface `Tributavel`. Novamente: programe voltado a interface, não a implementação.

6. Transforme a classe Conta em uma interface. Atenção: faça isso num projeto a parte pois usaremos a Conta como classe em exercícios futuros.