53 45 52 45 49 20 46 49 45 4c 20
41 4f 53 20 50 52 45 43 45 49 54
4f 53 20 44 41 20 48 4f 4e 52 41
20 45 20 44 41 20 43 49 c3 8a 4e
43 49 41 2c 20 50 52 4f 4d 4f 56
45 4e 44 4f 20 4f 20 55 53 4f 20
45 20 4f 20 44 45 53 45 4e 56 4f
4c 56 49 4d 45 4e 54 4f 20 44 41
20 49 4e 46 4f 52 4d c3 81 54 49
43 41 20 45 4d 20 42 45 4e 45 46
c3 8d 43 49 4f 20 44 4f 20 43 49
44 41 44 c3 83 4f 20 45 20 44 41
20 53 4f 43 49 45 44 41 44 45 2e

RESIDÊNCIA DE SOFTWARE

CAPACITAR
TREINAR
EMPREGAR
TRANSFORMAR

Firjan SENAI

serratec
Parque Tecnológico Região Serrana

JAVA I

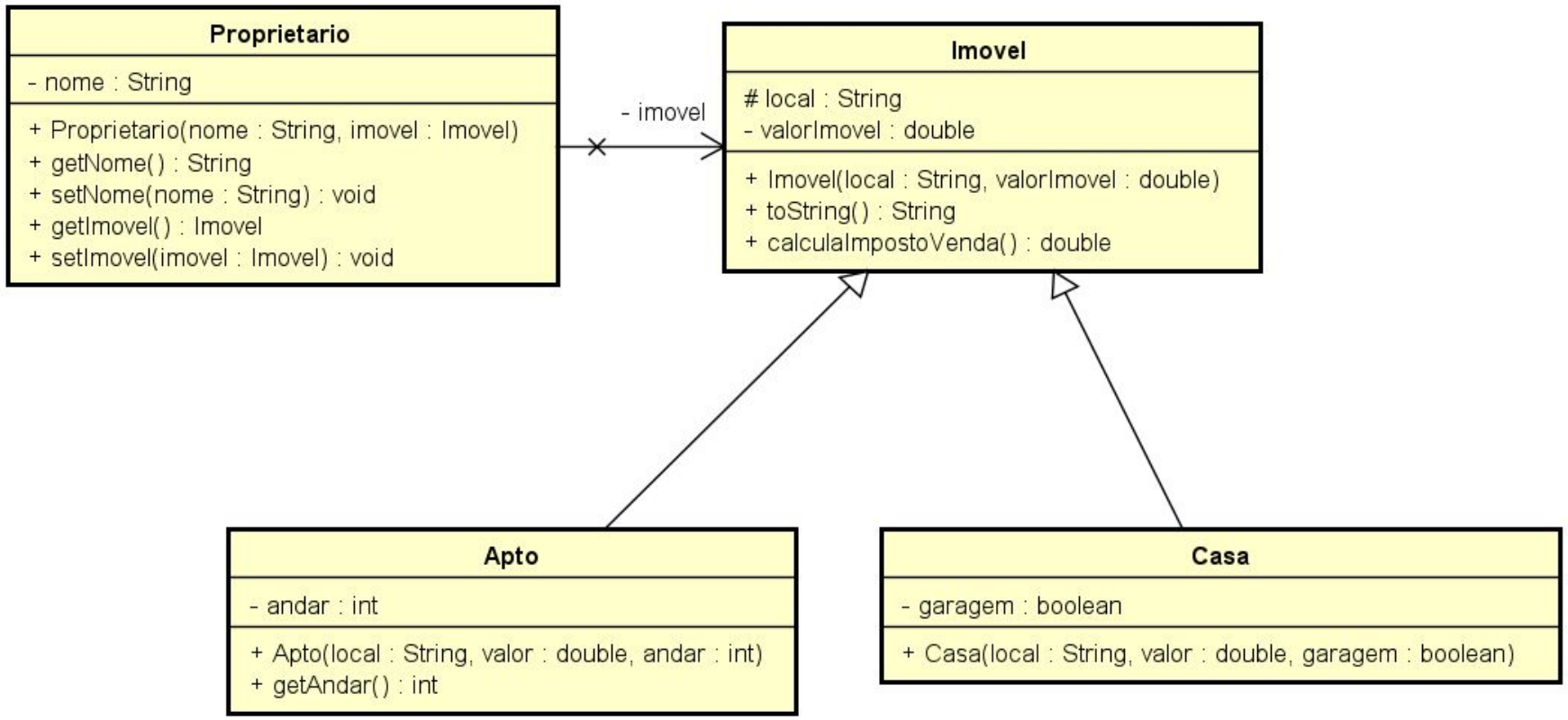
Classes Abstratas e Interfaces

04/08/2020

RECAPITULANDO

- O que já vimos?
  - O que é Java
  - Objetos e Classes
  - Encapsulamento
  - Getters e Setters
  - Modificadores de acesso
  - Herança e Polimorfismo
  - Relacionamento entre classes

RECAPITULANDO - EXERCÍCIOS



Construir as classes acima respeitando a herança e os relacionamentos.

O método calcularImpostoVenda na classe imóvel deverá ser descontado 3.5 de valor de itbi do valor do imóvel

EXERCÍCIOS – RESOLUÇÃO

```java
public class Imovel {
    protected String local;
    private double valorImovel;

    public Imovel(String local, double valorImovel) {
        super();
        this.local = local;
        this.valorImovel = valorImovel;
    }

    @Override
    public String toString() {
        return "Imovel [local=" + local + ", valor=" + valorImovel + "]";
    }

    public double calculaImpostoVenda() {
        double valorItbi;
        return valorItbi = valorImovel * 3.5 / 100;

    }
}
```

```java
public class Casa extends Imovel {
    private boolean garagem;

    public Casa(String local, double valor, boolean garagem) {
        super(local, valor);
        this.garagem = garagem;
    }

}
```

```java
public class Apto extends Imovel {
    private int andar;

    public Apto(String local, double valor, int andar) {
        super(local, valor);
        this.andar = andar;
    }

    public int getAndar() {
        return andar;
    }

}
```

EXERCÍCIOS – RESOLUÇÃO

```java
public class Proprietario {
    private String nome;
    private Imovel imovel;

    public Proprietario(String nome, Imovel imovel) {
        super();
        this.nome = nome;
        this.imovel = imovel;
    }

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public Imovel getImovel() {
        return imovel;
    }

    public void setImovel(Imovel imovel) {
        this.imovel = imovel;
    }

}
```

```java
public class TesteImovel {
    public static void main(String[] args) {
        Imovel imovel1 = new Casa("Centro", 575000, false);
        Imovel imovel2 = new Apto("Nogueira", 280500, 3);

        Proprietario proprietario1 = new Proprietario("Marcos", imovel1);
        Proprietario proprietario2 = new Proprietario("Sandra", imovel2);



        System.out.println("Proprietario:" + proprietario1.getNome());
        System.out.println(proprietario1.getImovel().toString());
        System.out.println("Valor a pagar imposto Itbi:" + proprietario1.getImovel().calculaImpostoVenda());

        System.out.println("Proprietario:" + proprietario2.getNome());
        System.out.println(proprietario2.getImovel().toString());
        System.out.println("Valor a pagar imposto Itbi:" + proprietario2.getImovel().calculaImpostoVenda());

    }

}
```

- Quando temos um Funcionario e ele não tem função?
  - Quando temos um funcionário, ele vai ser um Gerente, Diretor, Assessor, Faxineiro...
  - Faz sentido termos um funcionário para podermos usar o polimorfismo de referência, mas, por outro lado, não faz sentido instanciarmos um Funcionário, certo?

```
ControleDeBonificacoes cdb = new ControleDeBonificacoes();
Funcionario f = new Funcionario();
cdb.adiciona(f);  // faz sentido?
```

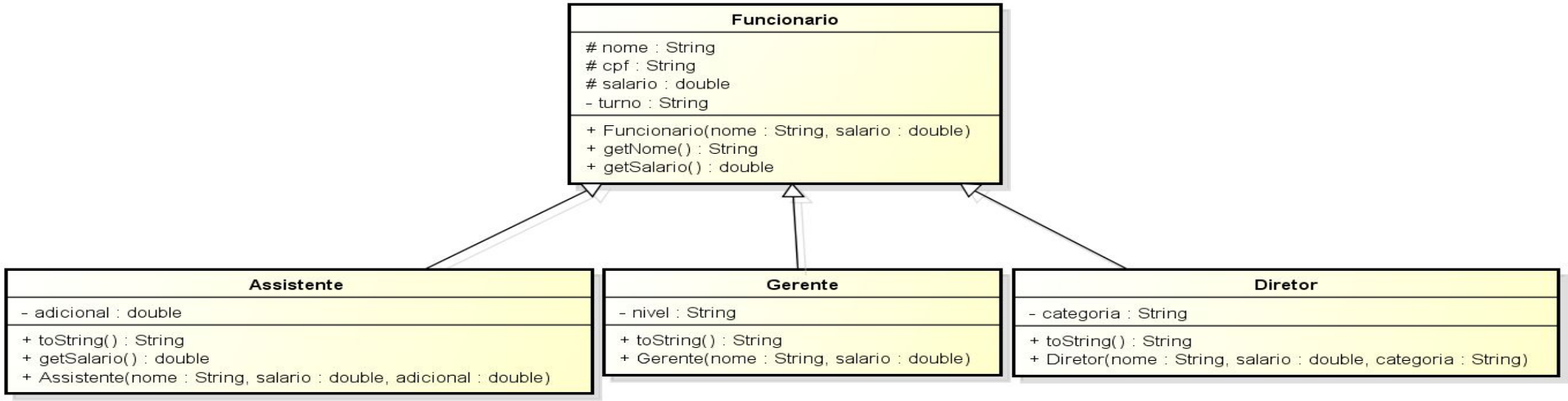
CLASSE ABSTRATA

- Quando não fizer sentido instanciar uma classe, ou seja, quando desejarmos apenas que as classes filhas possam ser instanciada, utilizamos a palavra chave **abstract** ao declarar a classe.
- Uma classe abstrata diferentemente de uma classe concreta não pode ser instanciada. Elas são usadas normalmente para se definir uma assinatura e uma implementação padrão para suas classes filhas.

```
public abstract class Funcionario {
        protected double salario;

        public double getBonificacao(){
                return this.salario * 0.10;
        }
// outros atributos e métodos comuns a todos Funcionarios
}
```

O que acontece se tentarmos executar:
Funcionario f = new Funcionario();

CLASSE ABSTRATA



Classe concreta

TestaFuncionario.java

```
package aula;

public class TestaFuncionario {
    Funcionario f = new Funcionario("Manoel", 900.);
}
```

Posso instanciar um Funcionario?
Sim.

Faz sentido?
Não.

Classe abstrata

```
public abstract class Funcionario {
```

Inserir **abstract** antes do class na classe **Funcionario**. A classe **TestaFuncionario** não compilará, pois não posso mais instanciar um funcionário. A classe **Funcionario** não pode ser instanciada, mas pode ser usada para polimorfismo e herança.

MÉTODOS ABSTRATOS

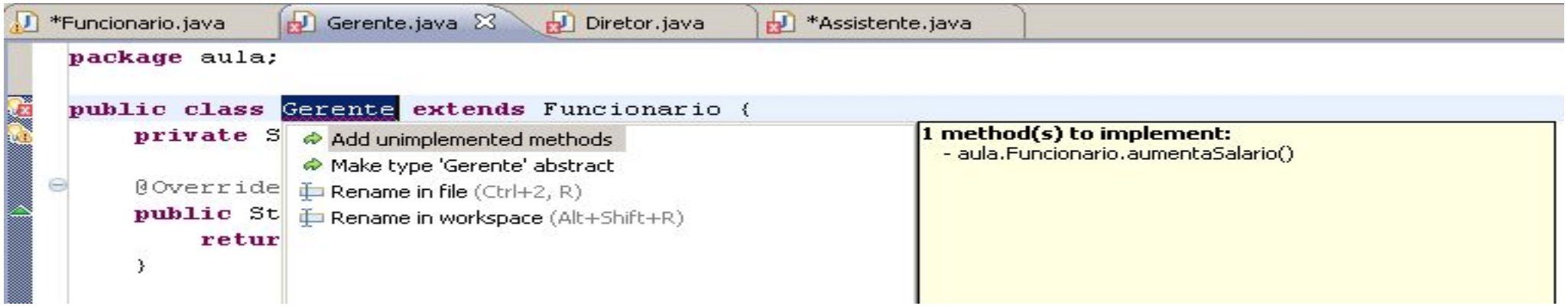
Os métodos abstratos definidos em uma classe abstrata devem obrigatoriamente ser implementados em uma classe concreta ou em suas classes filhas. Classes abstratas podem conter ou não métodos abstratos, ou seja, uma classe abstrata pode implementar ou não um método.
Para criarmos uma classe ou método abstrato usamos a palavra-chave **abstract.**

Insira o código abaixo na classe **Funcionario**

```
public abstract void aumentaSalario();
```

No nosso sistema precisamos aumentar o salário de todos funcionários, sendo que cada tipo de funcionário tem um tipo diferente de cálculo e queremos obrigar que o método seja implementado em suas classes filhas, para isto inserimos a assinatura do método na super classe.

Temos que implementar os métodos nas classes filhas. Utilize o **quick fix** do eclipse.



IMPLEMENTANDO MÉTODOS ABSTRATOS

Assistente

```
@Override
public void aumentaSalario() {
    this.salario *= 1.1;

}
```

Diretor

```
@Override
public void aumentaSalario() {
    this.salario = this.salario * 1.1 + 1000;

}
```

Gerente

```
@Override
public void aumentaSalario() {
    this.salario = this.salario * 1.15 + 500;

}
```

TestaFuncionario.java

```
package aula;

public class TestaFuncionario {
    public static void main(String[] args) {
        Assistente a = new Assistente("Manoel", 950., 100.);
        Gerente g = new Gerente("Maria Cristina", 5000.);
        Diretor d = new Diretor("Ana Luiza", 10000., "Administrativo");

        a.aumentaSalario();
        g.aumentaSalario();
        d.aumentaSalario();

        System.out.println(a.toString());
        System.out.println(g.toString());
        System.out.println(d.toString());
    }

}
```

ESTENDENDO CLASSES ABSTRATAS

Se inserirmos uma nova classe herdando da classe **Assistente**
não precisaremos aplicar o método abstrato da classe **Funcionario**
a classe **AssistenteProducao** vai herdar de **Assistente**.

AssistenteProducao.java

```
package aula;

public class AssistenteProducao extends Assistente {

    public AssistenteProducao(String nome, double salario, double adicional) {
        super(nome, salario, adicional);

    }

}
```

IMPLEMENTANDO MÉTODOS DA CLASSES ABSTRATAS

Vamos inserir uma classe chamada **Auxiliar** sendo abstrata e herdando de **Funcionario**. A classe **Auxiliar** como é abstrata não é obrigada a implementar o método abstrato da classe **Funcionario**

Auxiliar.java

```
package aula;

public abstract class Auxiliar extends Funcionario {
    protected String categoria;

    public Auxiliar(String nome, double salario, String categoria) {
        super(nome, salario);
        this.categoria = categoria;
    }

}
```

Inserimos a classe **AuxiliarAdm** herdando de **Auxiliar.** Dessa forma, temos que implementar o método **aumentaSalario** da classe **Funcionario**

AuxiliarAdm.java

```
package aula;

public class AuxiliarAdm extends Auxiliar{

    public AuxiliarAdm(String nome, double salario, String categoria) {
        super(nome, salario, categoria);
        // TODO Auto-generated constructor stub
    }

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

    }

}
```

INTRODUÇÃO AS INTEFACES

- Vamos considerar que apenas Gerente e Diretor possa autenticar no sistema interno do banco. Uma solução seria criar uma classe no meio da árvore herança para que apenas funcionário autenticáveis acessem o sistema.

```
public class FuncionarioAutenticavel extends Funcionario {
    public boolean autentica(int senha) {
        // faz autenticacao padrao
    }
}
```

Agora as classes Gerente e Diretor poderiam estender de FuncionarioAutenticavel

Mas e se um cliente puder acessar o sistema?
E se um atendente passar a acessar o sistema?

- Faria sentido um cliente herdar de funcionário para acessar o sistema? Como resolvemos essa situação?

INTERFACES

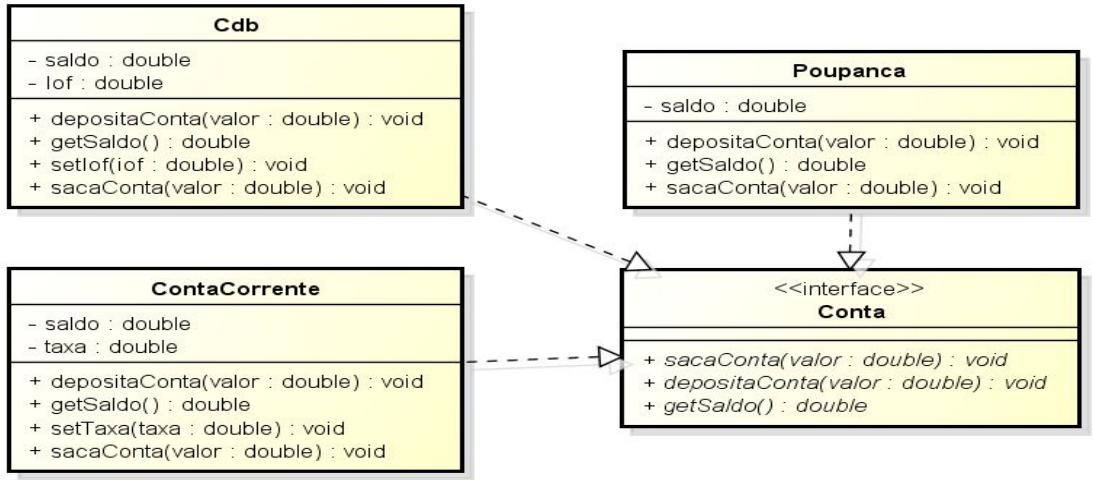
- A interface cria um “Contrato” que define o que a classe terá que fazer. Dessa forma, será necessário implementar o que a interface propõe.

INTERFACES

Interface é um recurso que define ações que devem ser obrigatoriamente executadas, mas que cada classe pode executar de forma diferente. Interfaces contém valores constantes, variáveis ou assinaturas de métodos que devem ser implementados dentro de uma classe. Uma interface é criada da mesma forma que uma classe, mas utilizando a palavra-chave **interface** no lugar de class.

Exemplo

Notação UML



Os métodos de uma interface não possuem corpo (implementação) pois serão implementados nas classes vinculadas a essa interface.

```
public interface Conta {
    public void sacaConta(double valor);
    public void depositaConta(double valor);
}
```

INTERFACES

Para obter as assinaturas dos métodos da interface em uma classe usamos a palavra chave **implements**

Uma interface é implementada e não herdada.

ContaCorrente.java

```
package aulainterface;

public class ContaCorrente implements Conta {
    private double saldo;
    private double taxa;

    @Override
    public void depositaConta(double valor) {
        this.saldo += valor;
    }

    @Override
    public double getSaldo() {
        return this.saldo;
    }

    public void setTaxa(double taxa) {
        this.taxa = taxa;
    }

    @Override
    public void sacaConta(double valor) {
        this.saldo -= valor + taxa;
    }

}
```

Cdb.java

```
package aulainterface;

public class Cdb implements Conta {
    private double saldo;
    private double Iof;
    @Override
    public void depositaConta(double valor) {
        this.saldo += valor;

    }
    @Override
    public double getSaldo() {
        return this.saldo;
    }

    public void setIof(double iof) {
        Iof = iof;
    }
    @Override
    public void sacaConta(double valor) {
        this.saldo -= valor + this.Iof;
    }
}
```

INTERFACES

Poupanca.java

```java
package aulainterface;

public class Poupanca implements Conta {
    private double saldo;

    @Override
    public void depositaConta(double valor) {
        this.saldo += valor;
    }

    @Override
    public double getSaldo() {
        return this.saldo;
    }

    @Override
    public void sacaConta(double valor) {
        this.saldo -= valor;
    }
}
```

TestaConta.java

```java
package aulainterface;

public class TestaConta {
    public static void main(String[] args) {
        Poupanca p = new Poupanca();
        Cdb c = new Cdb();
        ContaCorrente cc = new ContaCorrente();
        p.depositaConta(1000);
        c.depositaConta(2000);
        cc.depositaConta(2500);

        p.sacaConta(100);
        c.setIof(0.99);
        c.sacaConta(200);
        cc.setTaxa(1.3);
        cc.sacaConta(300);
        System.out.println(p.getSaldo());
        System.out.println(c.getSaldo());
        System.out.println(cc.getSaldo());
    }
}
```

HERANÇA ENTRE INTERFACES

Diferentemente das classes, uma interface pode herdar de mais de uma interface. É como um contrato que depende que outros contratos sejam fechados antes deste valer.

Irenda.java

```
package aula;

public interface Irenda {
    public double calculaIrenda();
}
```

Os atributos são implicitamente public ,static e final. Todo atributo em uma interface tem que ser inicializado.

Taxas.java

```
package aula;

public interface Taxas {
    double icms = 0.05;
public static final double irPessoa = 0.15;
private double irEmpresa = 0.18;
}
```

Errado. Métodos em interface tem que ser públicos. Retire o **private**

Tributos.java

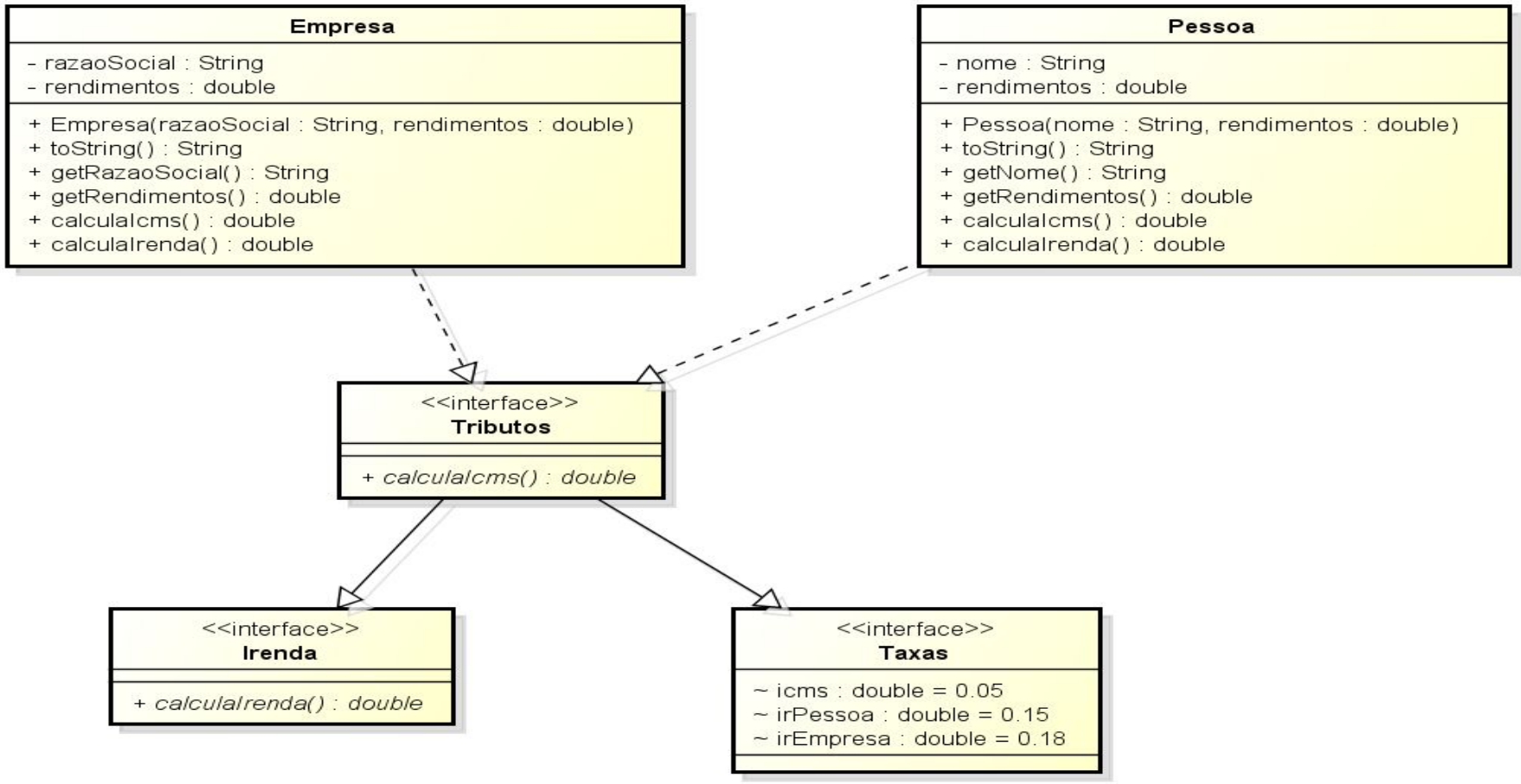
```
package aula;

public interface Tributos extends Taxas,Irenda {
    public abstract double calculaIcms();
}
```

A Interface **Tributos** herda de **Taxas** e **Irenda**.
Quem for implementar a interface **Tributos** precisa implementar todos os métodos herdados das suas super interfaces e os métodos declarados em **Tributos**.

Opcional. Todos os métodos das interfaces são implicitamente **public** e **abstract**

No diagrama UML abaixo temos a classe **Empresa** e a classe **Pessoa** implementando a interfaces **Tributos**.

interfaces costumam ligar classes distintas, unindo-as por característica que elas tem em comum.



*Pessoa.java

```java
package aula;

public class Pessoa implements Tributos {
    private String nome;
    private double rendimentos;


    public Pessoa(String nome, double rendimentos) {
        this.nome = nome;
        this.rendimentos = rendimentos;
    }

    @Override
    public String toString() {
        return nome + " " + rendimentos + " ";
    }
    public String getNome() {
        return nome;
    }

    public double getRendimentos() {
        return rendimentos;
    }

    @Override
    public double calculaIcms() {
        return 0;
    }

    @Override
    public double calculaIrenda() {
        return this.rendimentos * irPessoa;
    }

}
```

Pessoa vai implementar todos os métodos da interface Tributos e das outras que ela herda.

Pessoa não paga ICMS por isto retornamos 0.

*Empresa.java

```java
package aula;
public class Empresa implements Tributos {
    private String razaoSocial;
    private double rendimentos;

    public Empresa(String razaoSocial, double rendimentos) {
        this.razaoSocial = razaoSocial;
        this.rendimentos = rendimentos;
    }

    @Override
    public String toString() {
        return razaoSocial + " " + rendimentos + " ";
    }

    public String getRazaoSocial() {
        return razaoSocial;
    }

    public double getRendimentos() {
        return rendimentos;
    }

    @Override
    public double calculaIcms() {
        return this.rendimentos * icms;
    }

    @Override
    public double calculaIrenda() {
        return this.rendimentos * irEmpresa;
    }
}
```

POLIMORFISMO

O atributo **total** é para calcular todos os tributos pagos por **Pessoa** ou **Empresa**

Se uma classe implementa uma interface, podemos aplicar o polimorfismo assim como na herança.

Neste exemplo, a classe **Empresa** e **Pessoa** implementam a interface **Tributos**. Guardamos a referência do objeto **Empresa** ou **Pessoa** em uma variável do tipo **Tributos**.

*TotalTributos.java

```
package aula;

public class TotalTributos {
    private double total;

    public double getTotal() {
        return total;
    }

    public void totalizaTributos(Tributos t) {
        this.total += t.calculaIcms();
        this.total += t.calculaIrenda();

        System.out.println(t.toString() + "\n"+ "ICMS " + t.calculaIcms());
        System.out.println("IR " + t.calculaIrenda()+ "\n");
    }

}
```

TestaInterface.java

```java
package aula;

public class TestaInterface {
    public static void main(String[] args) {
        Pessoa p = new Pessoa("Carla", 12000.);
        Empresa e = new Empresa("XPTO LTDA", 880900.);
        TotalTributos t = new TotalTributos();

        t.totalizaTributos(p);
        t.totalizaTributos(e);
        System.out.println("Total Geral de Tributos:" + t.getTotal());

    }

}
```

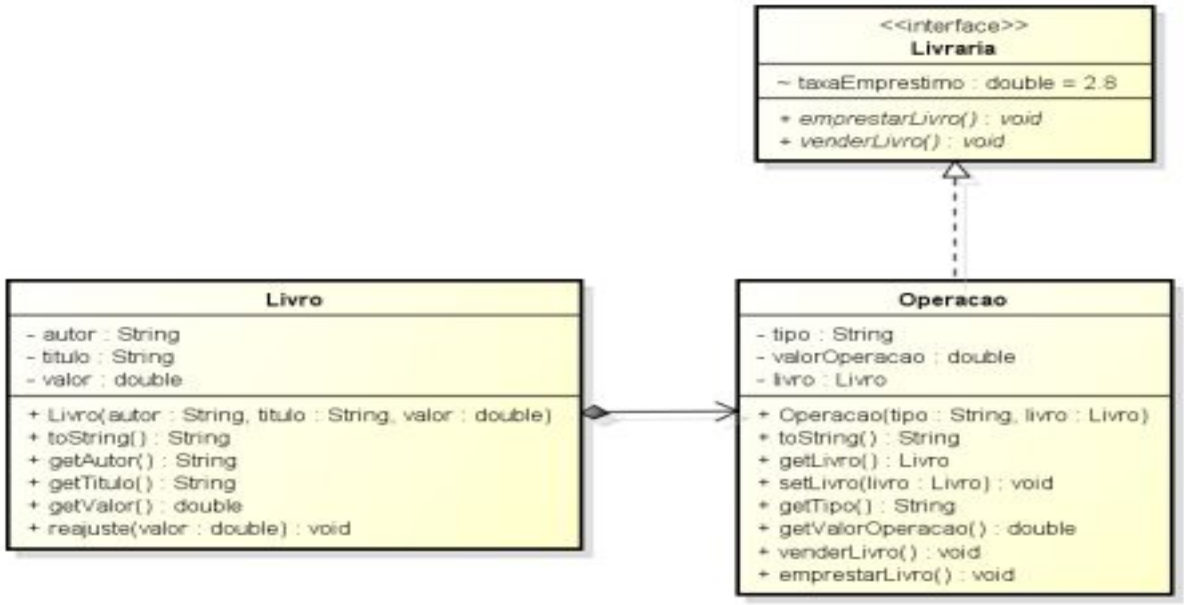
No exemplo abaixo mostramos a classe **TestaInterface** utilizando o polimorfismo. Quando crio uma variável do tipo **Tributos**, estou criando uma referência para qualquer objeto de uma classe que implemente **Tributos**.

TestaInterface.java

```java
package aula;

public class TestaInterface {
    public static void main(String[] args) {
        Tributos t1 = new Pessoa("Carla", 12000.);
        Tributos t2 = new Empresa("XPTO LTDA", 880900.);
        TotalTributos t = new TotalTributos();

        t.totalizaTributos(t1);
        t.totalizaTributos(t2);
        System.out.println("Total Geral de Tributos:" + t.getTotal());

    }

}
```

EXERCÍCIOS

1- Crie as classes, seus atributos, construtores, toString, getters e setters conforme o diagrama de classe abaixo



- No método **venderLivro** da classe **Operação** o atributo **valorOperacao** receberá o **valor** do **Livro** com acréscimo de 9% de impostos.
- No método **emprestarLivro** da classe **Operação** quando for feito um empréstimo o **valor da operacao** será 2% do valor do livro mais a **taxa de empréstimo.**
- Crie o método reajuste na classe Livro. Este método é para reajustar um valor de um livro.

- Crie um classe de teste com o main com o nome de **TestaLivro.** Crie duas instâncias para **Livro**. Crie duas instâncias para **Operação** e associe um livro a **Operacao**

Livro		
Autor	Título	Valor
Marco Antônio	Cisco CCNA	85,00
Kathy Sierra	Use a Cabeça Java	98,00

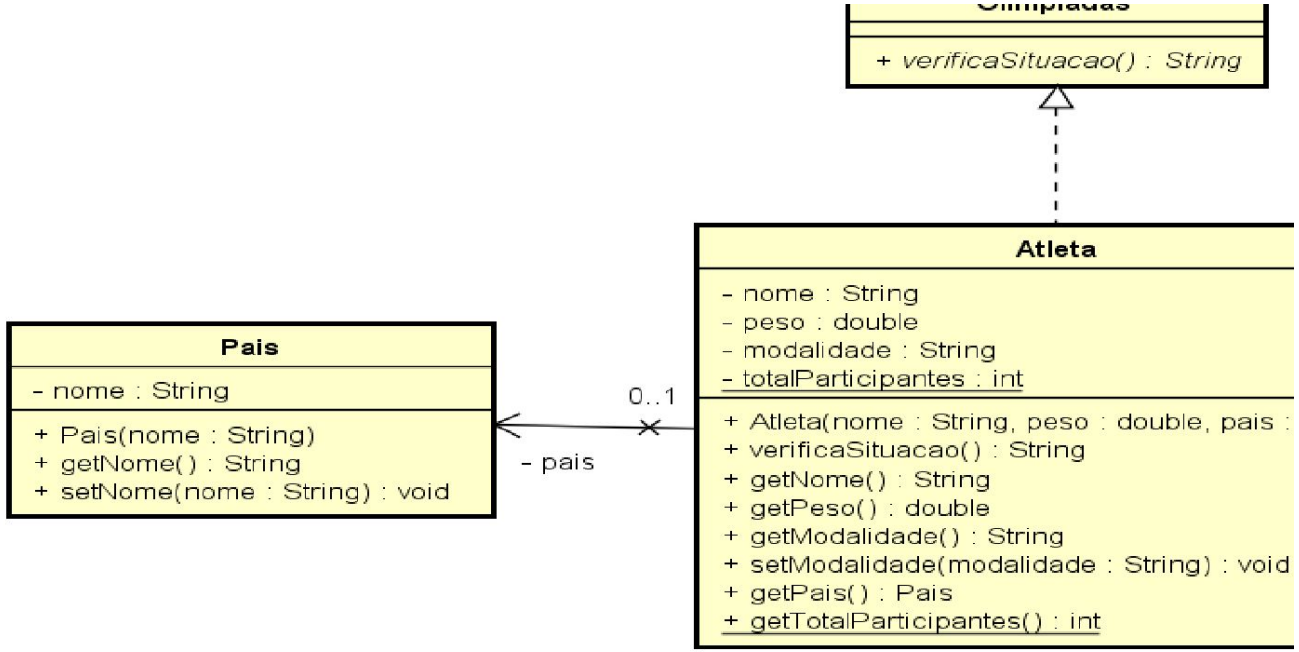
Operação	
Tipo	Livro
venda	Marco Antônio
empréstimo	Kathy Sierra

- Faça o reajuste de 6% no livro do Marco Antônio
- Faça a venda do livro do autor Marco Antônio
- Faça o empréstimo do livro da autora Kathy Sierra
- Exiba os dados conforme abaixo:

```
Tipo:venda Valor:98.209 Autor:Marco Antonio Titulo:Cisco CCNA
Tipo:empréstimo Valor:4.76 Autor:Kathy Sierra Titulo:Use a cabeça Java
```

EXERCÍCIOS

2- Criar as classes, seus atributos, construtores default e outro com os atributos, getters e setters conforme o diagrama de classe abaixo



- A classe Atleta não poderá ser instanciada.
- No método verificaSituacao da classe **Atleta** deverá ser feito o teste abaixo:
  Se o peso do atleta for maior que 90 a sua modalidade será “peso pesado” retornando a String “participará”
  Se o peso do atleta for entre 60 e 90 a sua modalidade será “peso médio” String “participará”
  Caso contrário o atleta não participará
- Criar uma classe de teste com o main com o nome de TestaAtleta. Criar dois objetos do tipo Pais e outros 3 objetos do tipo Atleta com os dados abaixo