

INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SÃO PAULO
Campus São Carlos

Análise e Desenvolvimento de Sistemas

Programação Orientada a Objetos (POO3)

Herança, Classe Abstrata e Polimorfismo



Pablo Dalbem

dalbem@ifsp.edu.br

Conteúdo

Herança

Sobrescrita de métodos

Classe Object

Classes abstratas

Polimorfismo

Exercício

Recapitulando...

Pilares da Programação Orientada a Objetos:

Abstração

Encapsulamento

Herança

Polimorfismo

Herança

Pilares da Programação Orientada a Objetos:

Abstração

Encapsulamento

Herança

Polimorfismo

Herança

Herança é um mecanismo que **permite que atributos e métodos comuns a diversas classes sejam colocadas em uma classe base.**

A partir da classe base, outras classes mais específicas poderão ser implementadas.

Cada classe mais específica poderá possuir também seus próprios métodos e atributos.

Herança

Em Java, é possível herdar apenas uma superclasse, mas uma superclasse pode ser herdada por várias subclasses.

A herança é uma relação **transitiva**. Se uma classe C herda uma classe B e a classe B herda uma classe A, então C “é uma” classe A.

A herança é uma relação **assimétrica**. Se uma classe B herda A, A não herda B. Logo, A não possuirá os métodos e atributos de B.

Exemplo I

Considere um sistema que precisa representar pessoas físicas e pessoas jurídicas.

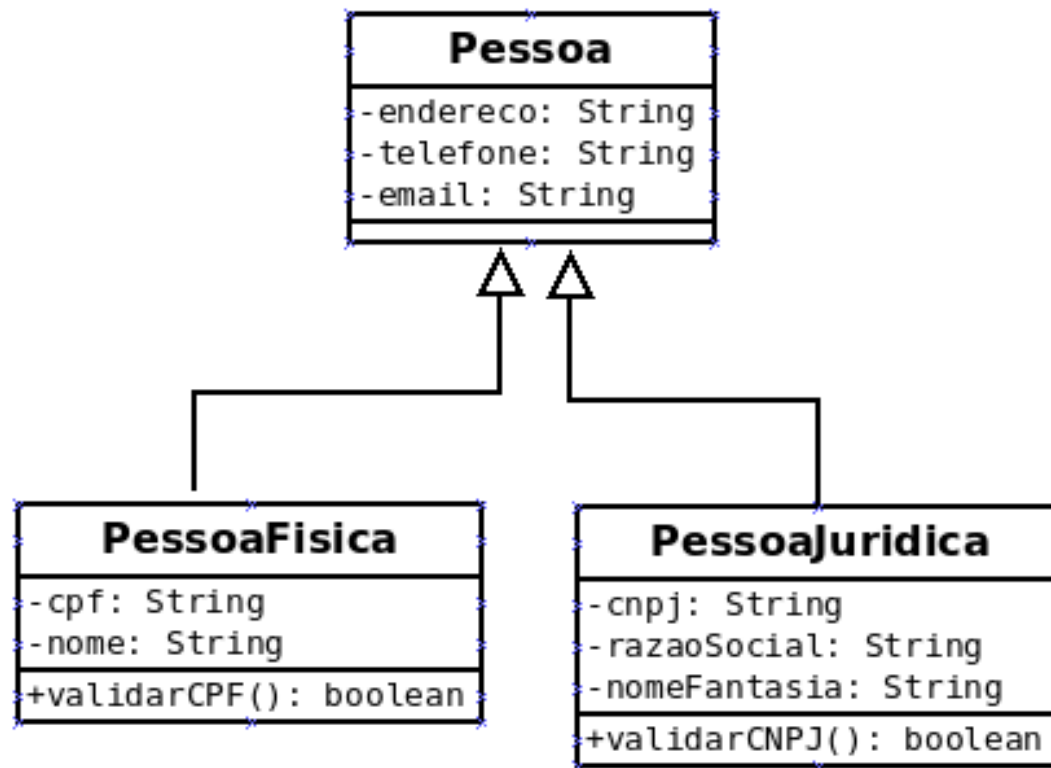
PessoaFisica
-cpf: String
-nome: String
-endereco: String
-telefone: String
-email: String
+validarCPF(): boolean

PessoaJuridica
-cnpj: String
-razaoSocial: String
-nomeFantasia: String
-endereco: String
-telefone: String
-email: String
+validarCNPJ(): boolean

Neste exemplo, temos 2 classes com atributos em comum (endereco, telefone, email).

Exemplo I

Mesmo sistema usando Herança



Classe Pessoa

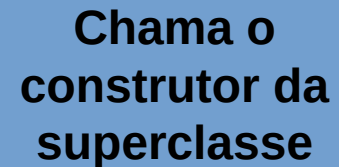
```
public class Pessoa {  
    private String endereco;  
    private String telefone;  
    private String email;  
  
    public Pessoa(String endereco, String telefone, String email) {  
        this.endereco = endereco;  
        this.telefone = telefone;  
        this.email = email;  
    }  
  
    // getters e setters..  
}
```

Classe PessoaFisica - 1/2

```
public class PessoaFisica extends Pessoa{  
    private String cpf;  
    private String nome;
```

```
    public PessoaFisica(String cpf, String nome, String endereco, String telefone,  
String email) {  
        super(endereco, telefone, email);  
        this.nome = nome;  
        this.cpf = cpf;  
    }
```

**Chama o
construtor da
superclasse**



```
    public String getNome() {  
        return nome;  
    }
```

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

Classe PessoaFisica - 2/2

CONTINUAÇÃO...

```
public String getCpf() {  
    return cpf;  
}  
  
public void setCpf(String cpf) {  
    this.cpf = cpf;  
}  
  
public boolean validarCPF(){  
    //código para validar CPF  
}  
}
```

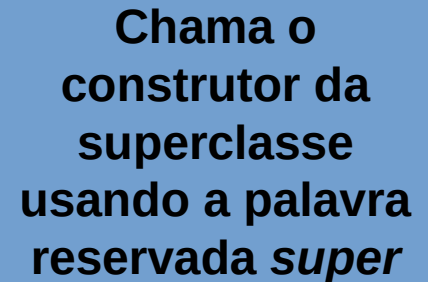
Classe PessoaJuridica - 1/2

```
public class PessoaJuridica extends Pessoa{
    private String cnpj;
    private String razaoSocial;
    private String nomeFantasia;

    public PessoaJuridica(String cnpj,String razaoSocial,String nomeFantasia,
        String endereco,String telefone,String email) {
        super(endereco, telefone, email);
        this.cnpj = cnpj;
        this.razaoSocial = razaoSocial;
        this.nomeFantasia = nomeFantasia;
    }

    public String getCnpj() {
        return cnpj;
    }

    public void setCnpj(String cnpj) {
        this.cnpj = cnpj;
    }
}
```



Chama o construtor da superclasse usando a palavra reservada *super*

Classe PessoaJuridica - 2/2

CONTINUAÇÃO...

```
public String getRazaoSocial() {  
    return razaoSocial;  
}
```

```
public void setRazaoSocial(String razaoSocial) {  
    this.razaoSocial = razaoSocial;  
}
```

```
public String getNomeFantasia() {  
    return nomeFantasia;  
}
```

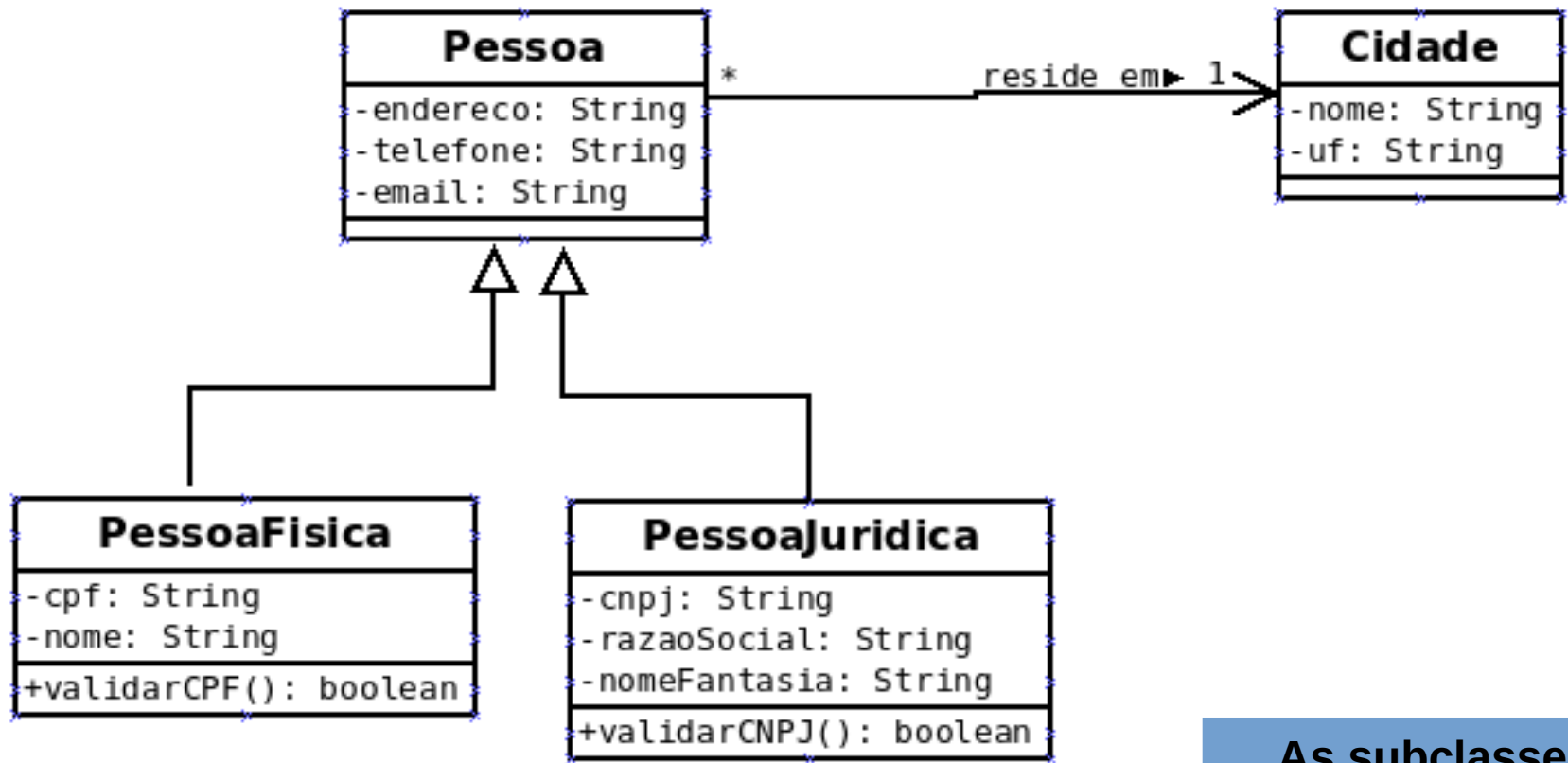
```
public void setNomeFantasia(String nomeFantasia) {  
    this.nomeFantasia = nomeFantasia;  
}
```

```
public boolean validarCNPJ(){  
    //código para validar cnpj  
}
```

Classe Principal

```
public class Principal {  
    public static void main(String[] args) {  
  
        PessoaFisica pessoaFisica = new PessoaFisica("1234567810",  
            "Maria", "Rua A, n10", "3351-1234", "maria@poo.com.br");  
  
        PessoaJuridica pessoaJuridica = new PessoaJuridica("111222333000190",  
            "Empresa S/A", "Empresa XPTO", "Rua B, n 20", "3351-9090",  
            "empresa@empresa.com.br");  
  
        System.out.println(pessoaFisica.getNome());  
  
        System.out.println(pessoaFisica.getEmail());  
    }  
}
```

Exemplo II



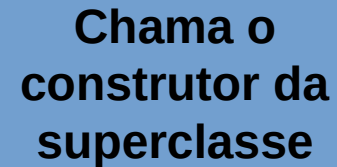
As subclasses
também herdam a
associação

Classe Pessoa – Exemplo II

```
public class Pessoa {  
    private String endereco;  
    private String telefone;  
    private String email;  
    private Cidade cidade;  
  
    public Pessoa(String endereco, String telefone, String e-mail, Cidade cidade) {  
        this.endereco = endereco;  
        this.telefone = telefone;  
        this.email = email;  
        this.cidade = cidade;  
    }  
  
    // getters e setters..  
}
```


PessoaFisica – Exemplo II

```
public class PessoaFisica extends Pessoa{  
    private String cpf;  
    private String nome;  
  
    public PessoaFisica(String cpf, String nome, String endereco, String telefone,  
String e-mail, Cidade cidade) {  
        super(endereco, telefone, e-mail, cidade);  
        this.nome = nome;  
        this.cpf = cpf;  
    }  
  
    ...  
}
```

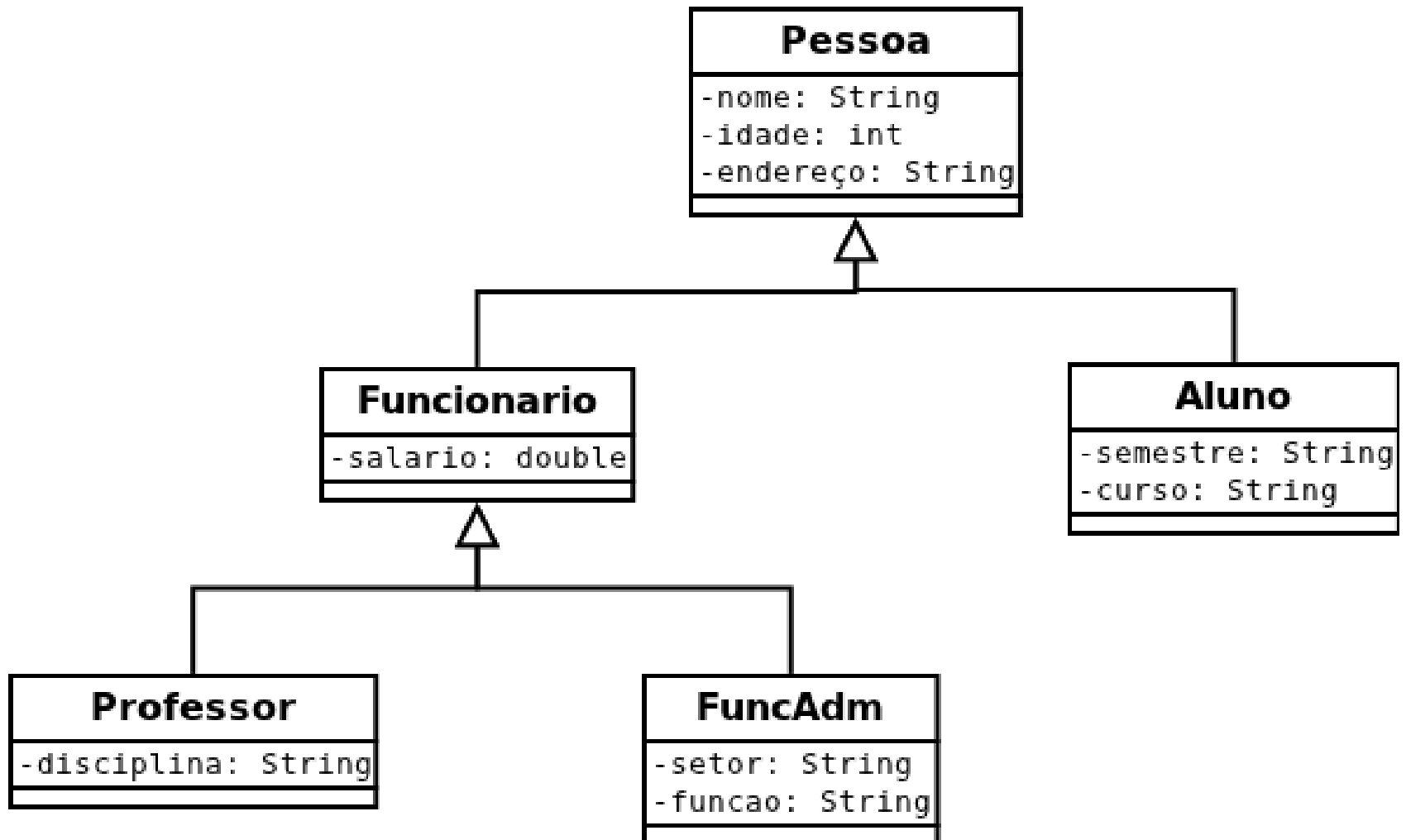


**Chama o
construtor da
superclasse**

Classe Principal – Exemplo II

```
public class Principal {  
    public static void main(String[] args) {  
  
        Cidade cidade = new Cidade("São Carlos", "SP");  
  
        PessoaFisica pessoaFisica = new PessoaFisica("1234567810",  
            "Maria", "Rua A, n10", "3351-1234", "maria@poo.com.br", cidade);  
  
        System.out.println(pessoaFisica.getNome());  
  
        System.out.println(pessoaFisica.getCidade().getNome());  
    }  
}
```

Mais Exemplos



Métodos na subclasse

Ao definir métodos na subclasse:

- I) Os métodos da superclasse são herdados automaticamente;
- II) Podemos definir novos métodos;
- III) Podemos sobrescrever métodos da superclasse.

Sobrescrita de Métodos

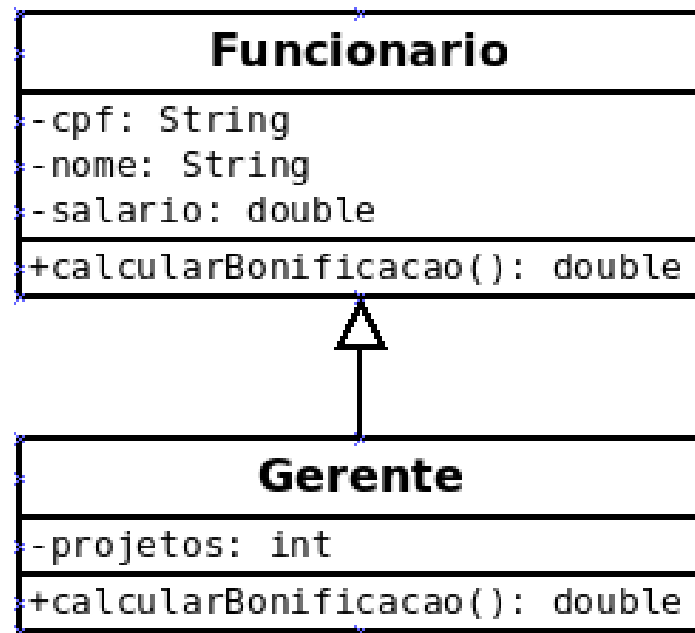
Uma subclasse pode sobrescrever (***override***) um método da superclasse para prover uma implementação mais adequada às suas necessidades.

Para sobrescrever um método, a subclasse deve prover uma implementação que possua a mesma assinatura do método da superclasse.

Para garantir que um método esteja realmente sobrescrevendo outro da superclasse, pode-se utilizar uma anotação **@Override** logo acima da assinatura do método da subclasse.

Sobrescrita de Métodos

Considere o seguinte exemplo: Ao final do ano, os funcionários recebem uma bonificação de 10% sobre o salário. Já os gerentes, recebem 5% sobre o salário e mais mil reais por projeto.



Classe Funcionario

```
public class Funcionario {  
    private String cpf;  
    private String nome;  
    private double salario;  
  
    public double calcularBonificacao(){  
        return this.salario * 0.10;  
    }  
  
    public Funcionario(String cpf, String nome, double salario) {  
        this.cpf = cpf;  
        this.nome = nome;  
        this.salario = salario;  
    }  
  
    // getters e setters  
}
```

Classe Gerente

```
public class Gerente extends Funcionario{  
    private int projetos;
```

```
@Override
```

```
public double calcularBonificacao(){  
    return (this.getSalario()*0.05) + (projetos*1000);  
}
```

```
public Gerente(String cpf, String nome, double salario, int projetos) {  
    super(cpf, nome, salario);  
    this.projetos = projetos;  
}
```

```
// getters e setters
```

```
}
```

Sobrescrita do
método



Principal

```
public class Principal {  
    public static void main(String[] args) {  
  
        Funcionario f1 = new Funcionario("123", "João", 2000);  
        System.out.println(f1.calcularBonificacao());  
  
        Gerente g1 = new Gerente("456", "Maria", 3000, 3);  
        System.out.println(g1.calcularBonificacao());  
    }  
}
```

Classe Object

Toda classe em Java é, direta ou indiretamente, subclasse de `java.lang.Object`.

Se uma classe não define explicitamente uma herança para outra classe, a linguagem assume que implicitamente há uma herança para a classe `Object`.

Em outras palavras, **Object é a raiz de todas as hierarquias de classe em Java.**

Apenas tipos primitivos não herdam de `Object` em Java.

Classe Object

Object **define alguns comportamentos comuns** que todos objetos devem ter, como a habilidade de serem comparados uns com os outros, poderem ser representados como texto e possuírem um número que identifica suas posições em coleções baseadas em hash.

Principais métodos:

getClass(): retorna a classe a qual o objeto pertence no momento da execução

clone(): cria e retorna uma cópia do objeto

hashCode(): retorna um código de espalhamento (hash);

equals(): indica se um objeto é “igual” a outro

toString(): retorna uma String representando o estado do objeto.

Sobrescrita de toString()

```
public class Funcionario {  
    private String cpf;  
    private String nome;  
    private double salario;  
  
    @Override  
    public String toString(){  
        return "CPF: " + this.getCPF()  
            + " Nome: " + this.getNome()  
            + "Salário: " + this.getSalario();  
    }  
  
    //construtores  
    // getters e setters  
}
```

**Sobrescrita do
método toString()**

Sobrescrita de toString()

```
public class Principal {  
    public static void main(String[] args) {  
  
        Funcionario f1 = new Funcionario("123","João",2000);  
  
        System.out.println(f1.toString());  
  
        // OU simplesmente:  
  
        System.out.println(f1);  
  
    }  
}
```

Comparar Objetos

```
public class Principal {  
    public static void main(String[] args) {  
  
        Funcionario f1 = new Funcionario("123","João",2000);  
        Funcionario f2 = new Funcionario("123","João",2000);  
  
        if (f1 == f2) //QUAL O RETORNO?  
            ...  
  
        Funcionario f3 = f1;  
        if (f1 == f3) //QUAL O RETORNO?  
            ....  
    }  
}
```

Comparar Objetos

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

```
        Funcionario f1 = new Funcionario("123","João",2000);
```

```
        Funcionario f2 = new Funcionario("123","João",2000);
```

```
        if (f1 == f2)
```

```
            ...
```

```
        Funcionario f3 = f1;
```

```
        if (f1 == f3)
```

```
            ....
```

```
    }
```

```
}
```

Retorna **FALSE**, pois f1 e f2 são referências para dois objetos distintos. Ou seja, f1 e f2 apontam para endereços de memória diferentes!

Retorna **TRUE**, pois f1 e f3 são referências para o mesmo objeto. Ou seja, f1 e f3 apontam para o mesmo endereço de memória.

Sobrescrita de equals()

```
public class Funcionario {  
    private String cpf;  
    private String nome;  
    private double salario;
```

@Override

```
public boolean equals(Object o) {  
    if (this == o) return true;  
    if (!(o instanceof Funcionario)) return false;  
    Funcionario f = (Funcionario) o;  
    return Double.compare(f.salario, this.salario) == 0  
        && Objects.equals(f.nome, this.nome) && Objects.equals(f.cpf, this.cpf);  
}
```

**Sobrescrita do
método equals()**

@Override

```
public int hashCode() {  
    return Objects.hash(nome, salario, cpf);  
}  
}
```

**Sobrescrita do método
hashCode() é obrigatória,
caso equals seja sobrescrito**

Comparar Objetos - equals

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

```
        Funcionario f1 = new Funcionario("123","João",2000);
```

```
        Funcionario f2 = new Funcionario("123","João",2000);
```

```
        if (f1.equals(f2))
```

```
            ...
```

```
    }
```

```
}
```

Comparação correta usando o método equals, o qual retorna **TRUE** para indicar que f1 e f2 são “iguais”,

Modificadores de Acesso

Recapitulando...

Modificadores de acesso viabilizam o encapsulamento de métodos e atributos de uma classe:

public

Acessados de qualquer lugar do código.

private

Acessados somente dentro do corpo da mesma classe.

protected

Acessados pelas classes do mesmo pacote ou classes herdadas.

Sem modificador

Acessados pelas classes do mesmo pacote

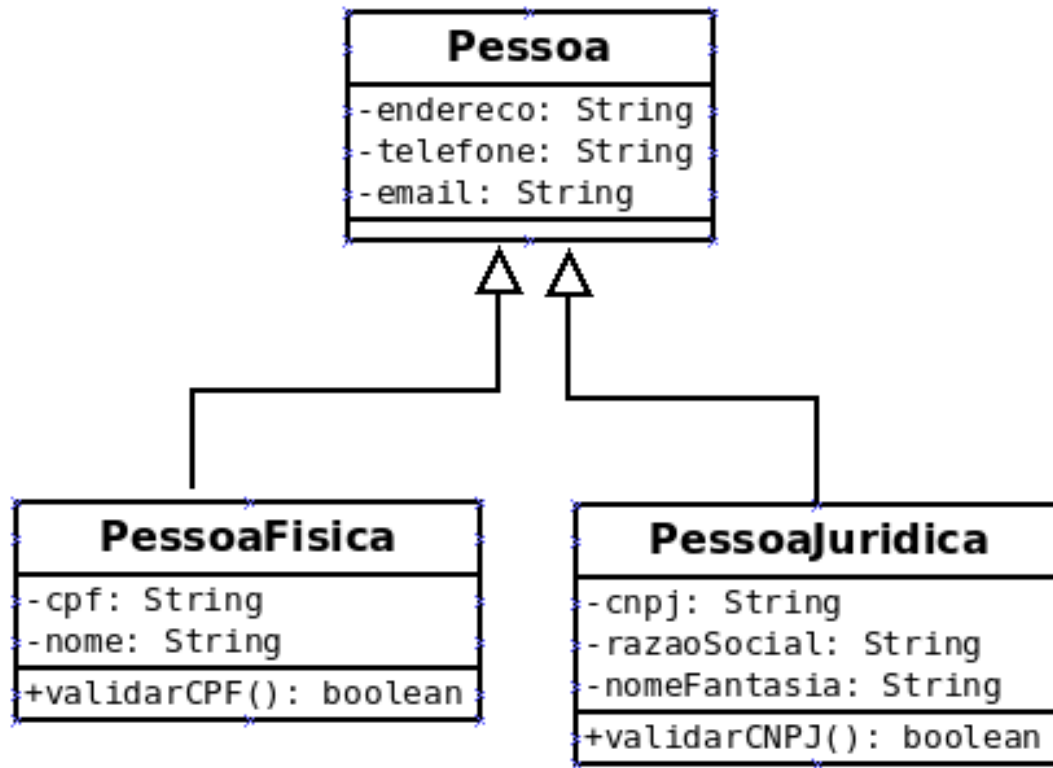
Modificadores de Acesso

```
public class Superclasse {  
    public int atributoPublico;  
    private int atributoPrivado;  
    protected int atributoProtegido;  
    int atributoSemModificador;  
}
```

```
public class Subclasse extends Superclasse{  
    public Subclasse() {  
        atributoPublico = 10; //OK, é público  
        atributoProtegido = 20; // OK, classe derivada  
        atributoSemModificador = 30; //OK, mesmo pacote  
        atributoPrivado = 40; //ERRO, só Superclasse acessa  
    }  
}
```

Classe Abstrata

Em algumas situações, não faz sentido instanciar um objeto de uma determinada classe. Em nosso primeiro exemplo, não faz sentido instanciar a classe Pessoa:



Classe Abstrata

Classe abstrata é aquela que não pode ser instanciada, apenas herdada. Ela serve de modelo para outras classes derivadas.

Classes abstratas podem conter métodos que não possuem implementação. Estes são os **métodos abstratos**, os quais devem ser sobrescritos, obrigatoriamente, por todas suas subclasses concretas (não abstratas).

Se a subclasse também for abstrata, ela não precisará implementar os métodos abstratos, deixando esta tarefa para uma classe concreta (não abstrata).

Classe Abstrata

Uma classe que contém um método abstrato deve ser, obrigatoriamente, abstrata. Uma classe abstrata não precisa ter, obrigatoriamente, métodos abstratos.

Classe abstrata precisa de método construtor?

Sim, mesmo não sendo possível instanciar um objeto de uma classe abstrata, é usual ela possuir métodos construtores. Isto porque as subclasses invocam estes métodos de forma implícita ou explícita.

Classe Abstrata

Para declarar uma **classe abstrata**, basta usar o modificador **abstract**:

```
public abstract class Pessoa {  
    private String endereco;  
    private String telefone;  
    private String email;  
  
    public Pessoa(String endereco, String telefone, String email) {  
        this.endereco = endereco;  
        this.telefone = telefone;  
        this.email = email;  
    }  
  
    // getters e setters..  
}
```

Método Abstrato

Para declarar uma **método abstrato**, também utilizamos o modificador **abstract**:

```
public abstract class Pessoa {  
    private String endereco;  
    private String telefone;  
    private String email;  
  
    public abstract void mostrarDados();  
  
    //construtores  
    //getters e setters  
}
```

Método abstrato



Implementação do Método Abstrato

A classe PessoaFisica precisa implementar mostrarDados()

```
public class PessoaFisica extends Pessoa{
    private String cpf;
    private String nome;

    @Override
    public void mostrarDados() {
        System.out.println("CPF: " + cpf
                           + "Nome: " + nome
                           + "E-mail: " + getEmail());
    }

    //construtores
    // getters e setters
}
```

Implementação do Método Abstrato

A classe PessoaJuridica precisa implementar mostrarDados()

```
public class PessoaJuridica extends Pessoa{
    private String cnpj;
    private String razaoSocial;
    private String nomeFantasia;

    @Override
    public void mostrarDados() {
        System.out.println("Nome Fantasia: " + nomeFantasia
            + "Endereço: " + getEndereco()
            + "Telefone: " + getTelefone());
    }

    //construtores, getters e setters
}
```

Classe Principal

```
public class Principal {  
    public static void main(String[] args) {  
  
        // Pessoa p1 = new Pessoa(); ERRO, classe Pessoa é abstrata  
        PessoaFisica pessoaFisica = new PessoaFisica("1234567810",  
            "Maria", "Rua A, n10", "3351-1234", "maria@poo.com.br");  
  
        PessoaJuridica pessoaJuridica = new PessoaJuridica("111222333000190",  
            "Empresa S/A",  
            "Empresa XPTO", "Rua B, n 20", "3351-9090",  
            "empresa@empresa.com.br");  
  
        pessoaFisica.mostrarDados();  
        pessoaJuridica.mostrarDados();  
    }  
}
```

Polimorfismo

Pilares da Programação Orientada a Objetos:

Abstração

Encapsulamento

Herança

Polimorfismo

Polimorfismo

É a habilidade de um objeto assumir **múltiplas formas**

Objetos de uma subclasse são tratados de forma genérica, como objetos da superclasse.

Porém, como eles são objetos diferentes, comportamentos diferentes são exibidos.

A decisão sobre qual método deve ser executado, de acordo com o tipo da classe derivada, é tomada em tempo de execução.

Polimorfismo - Exemplo

Acesse no **GitHub**:

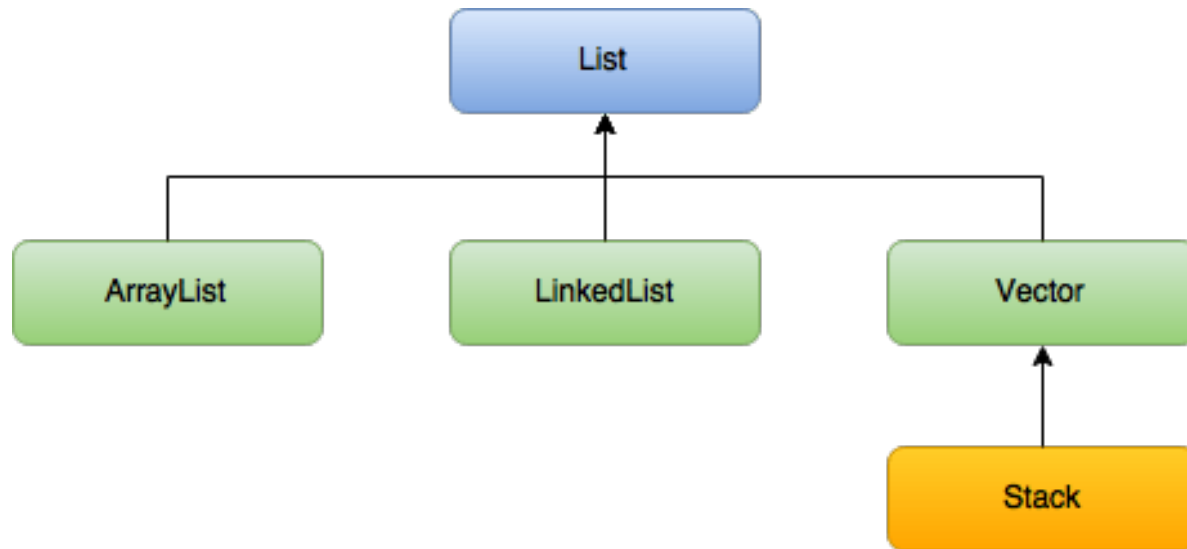
https://github.com/pdalbem/POO_2023_1/tree/main/Heran%C3%A7a%2C%20Classe%20Abstrata%20e%20Polimorfismo/Exemplo

```
public class Principal {  
    public static void main(String[] args) {  
  
        Pessoa pessoaFisica = new PessoaFisica("1234567810",  
            "Maria", "Rua A, n10", "3351-1234", "maria@poo.com.br");  
        Pessoa pessoaJuridica = new PessoaJuridica("111222333000190", "Empresa S/A",  
            "Empresa XPTO", "Rua B, n 20", "3351-9090", "empresa@empresa.com.br");  
  
        List<Pessoa> lista = new ArrayList<>();  
        lista.add(pessoaFisica);  
        lista.add(pessoaJuridica);  
  
        for (Pessoa p : lista)  
            p.mostrarDados();  
    }  
}
```

Permite percorrer coleções de objetos de uma superclasse, executando funcionalidades especializadas por diferentes subclasses, de forma homogênea.

Polimorfismo

Outro exemplo de polimorfismo muito utilizado por desenvolvedores Java é a interface List da API de Collections



```
List lista;  
lista = new ArrayList<>();  
lista = new LinkedList<>();  
lista = new Vector<>();
```

lista pode referenciar objeto do subtipo ArrayList, LinkedList ou Vector. Pode, portanto, acessar todos os métodos do supertipo List

Polimorfismo - Exemplo

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

```
        List lista1 = new ArrayList<>();  
        lista1.add("A");  
        lista1.add("B");
```

```
        List lista2 = new LinkedList<>();  
        lista2.add("C");
```

```
        List lista3 = new Vector<>();  
        lista3.add("D");  
        lista3.add("E");  
        lista3.add("F");
```

```
        System.out.println("Qtd de itens na lista1: "+ lista1.size());  
        System.out.println("Qtd de itens na lista2: "+lista2.size());  
        System.out.println("Qtd de itens na lista3: "+lista3.size());
```

O método **add** é abstrato na supertipo List. Cada subclasse (ArrayList, LinkedList e Vector) o implementa de maneira diferente. A JVM verifica em tempo de execução qual implementação será executada.

O mesmo ocorre para o método **size**.

```
    }
```

```
}
```


Exercício I

Implemente em Java

