

Mas a vereda dos justos é como a luz da aurora,  
que vai brilhando mais e mais até ser dia  
perfeito. Provérbios 4:18

# Curso de Especialização em Tecnologia Java

## LINGUAGEM DE PROGRAMAÇÃO JAVA I

- ▶ Prof: José Antonio Gonçalves
- ▶ [zag655@gmail.com](mailto:zag655@gmail.com)
- ▶ Ao me enviar um e-Mail coloque o “Assunto” começando: “pós2013\_2+seu nome”

# Ementa da disciplina:

- **Orientação a Objetos em Java:** Classes, Objetos, Herança, Polimorfismo, Classes Abstratas, Interface;
- **Exceções;**
- **Manipulação de Texto e Strings;**
- **Componentes básicos de interface gráfica;**
- **Tratamento de Eventos.**

## Bibliografia:

DEITEL, H.; DEITEL, P. JAVA – Como Programar. 3.ed. Porto Alegre: Bookman, 2001.

ECKEL, B. Thinking in Java , 2nd edition, EUA: Prentice Hall, 2000.

HORSTMANN, C. Core Java – Advanced Features. EUA: Prentice Hall, 2000. Volume II.

HORSTMANN, C. Core Java – Fundamentals. EUA: Prentice Hall, 2000. Volume I.

# Nestes Slides:

- **Orientação a Objetos em Java:**
- Definições e aplicação do conceito de Herança;
- Classe concreta vs. classe abstrata;
- Polimorfismo com sobrescrita;
- Método concreto vs. método concreto;

# Herança

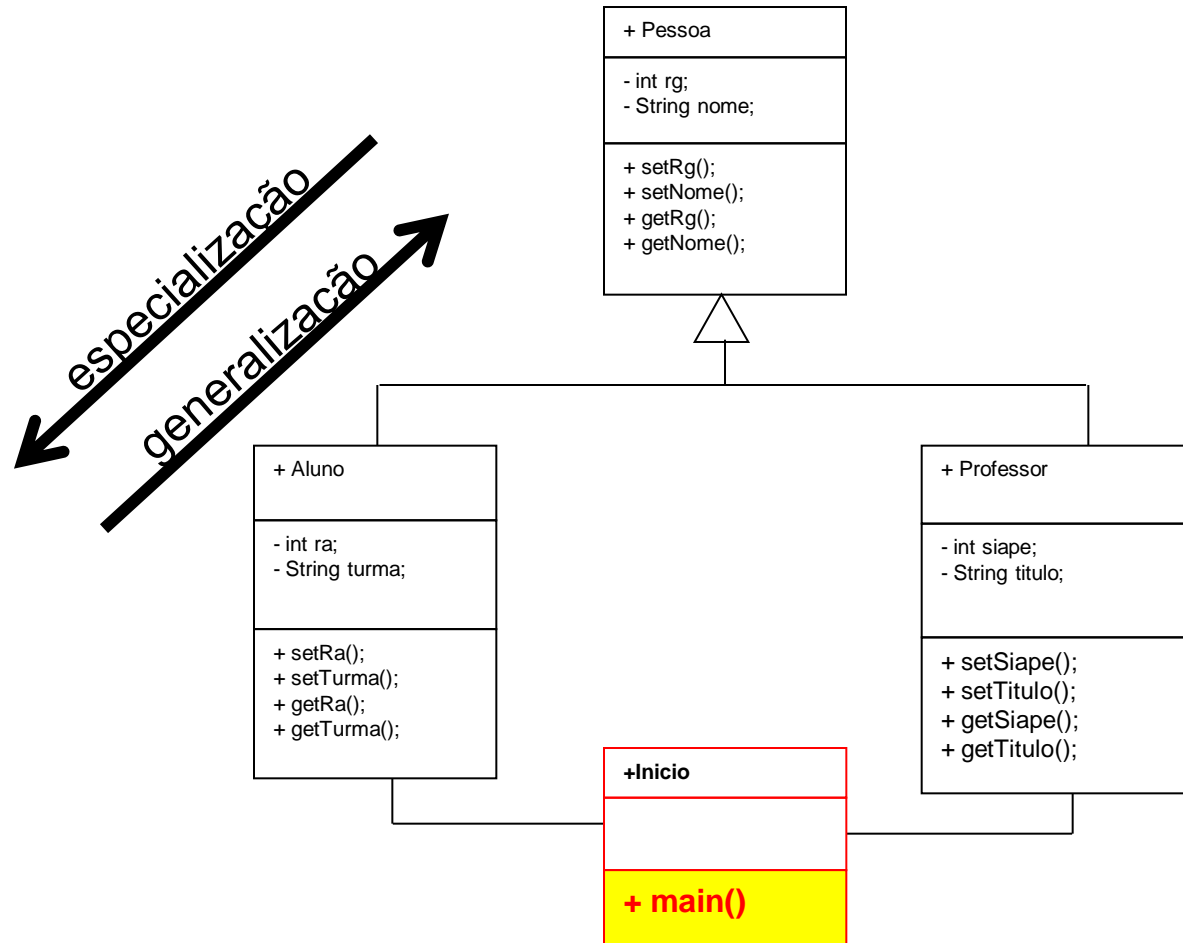
## Herança (*definição*)

**Herança (definição):** mecanismo que possibilita a construção de uma classe (classe filha) com base numa classe já existente (classe mãe).

**Importante:** embora exista o conceito de **Herança Múltipla**, que pode ser implementado em outras linguagens que suportam Orientação a Objetos, como C++ por exemplo, Java **não** oferece esta possibilidade. A maneira de se lidar com estas necessidade de Projeto é através de uma estrutura de dados chamada **Interface** (que será abordada mais a frente)

## Herança (*classes envolvidas*)

Para a construção da Aplicação imagine o seguinte Diagrama de Classes:

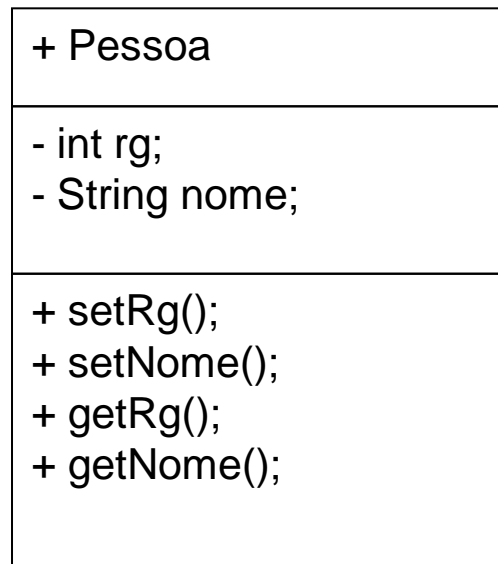


## Herança (*classes envolvidas – Pessoa*)

Podemos observar neste diagrama uma **classe chamada “Pessoa”**, composta por:

2 atributos (privados): **rg** (interio) e **nome** (String)

4 métodos (públicos): **setRg()**, **setNome()**, **getRg()**, **getNome()** e **Calc1()**;





## Herança codificação de “Pessoa”

**O código para a criação desta classe Pessoa será:**

```
public class Pessoa{
    private int rg;
    private String nome;

    public Pessoa( ) { }

    public Pessoa(int rg, String nome){
        this.rg = rg;
        this.nome = nome;
    }

    public void setRg(int rg){
        this.rg = rg;
    }
    public void setNome(String nome){
        this.nome = nome;
    }
    public int getRg(){
        return rg;
    }
    public int getNome(){
        return nome;
    }
}
```

|  |
|--|
| + Pessoa   |
| - int rg;<br>- String nome;                              |
| + setRg();<br>+ setNome();<br>+ getRg();<br>+ getNome(); |

## Herança (*classes envolvidas - Aluno*)

A **classe** chamada “**Aluno**” é composta por:

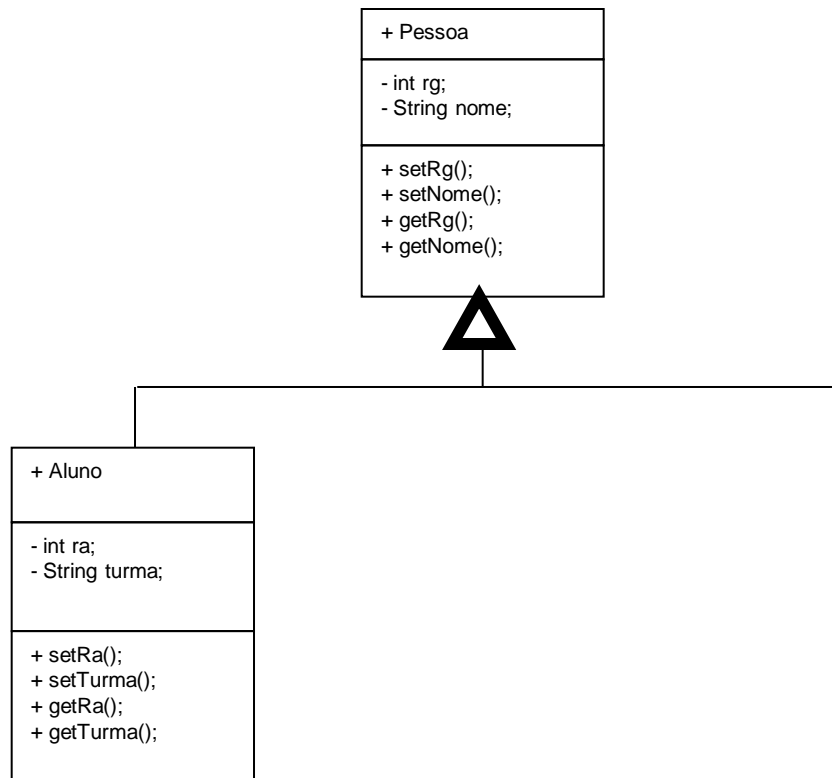
2 atributos (privados): **ra** (interio) e **turma** (String)

4 métodos (públicos): **setRa()**, **setTurma()**, **getRa()**, **getTurma()** e **Calc1()**;

|  |
|--|
| + Aluno  |
| - int ra;<br>- String turma;                               |
| + setRa();<br>+ setTurma();<br>+ getRa();<br>+ getTurma(); |

## Herança: como identificar *(no projeto e na implementação)*

Porém, observe no diagrama que a classe **Aluno** “herda” a classe **Pessoa**. Isso pode ser observado pelo **triângulo** presente na “**ligação**” entre Aluno e Pessoa. Ele indica a existência do mecanismo de **Herança**. No código fonte esta herança será definida pela palavra “**extends**”.



## Herança codificação de “Aluno”

```
public class Aluno extends Pessoa{  
    private int ra;  
    private String turma;  
  
    public Aluno( ) { }  
  
    public Aluno(int ra, String turma){  
        this.ra = ra;  
        this.turma = turma;  
    }  
  
    public void setRa(int ra){  
        this.ra = ra;  
    }  
  
    public void setTurma(String turma){  
        this.turma= turma;  
    }  
  
    public int getRa(){  
        return ra;  
    }  
  
    public int getTurma(){  
        return turma;  
    }  
}
```

|  |
|--|
| + Aluno  |
| - int ra;<br>- String turma;                               |
| + setRa();<br>+ setTurma();<br>+ getRa();<br>+ getTurma(); |

**Observe o  
Polimorfismo com  
sobrecarga também  
na classe Aluno!**

## Herança (*classes envolvidas – Professor*)

A classe chamada “**Professor**” é composta por:

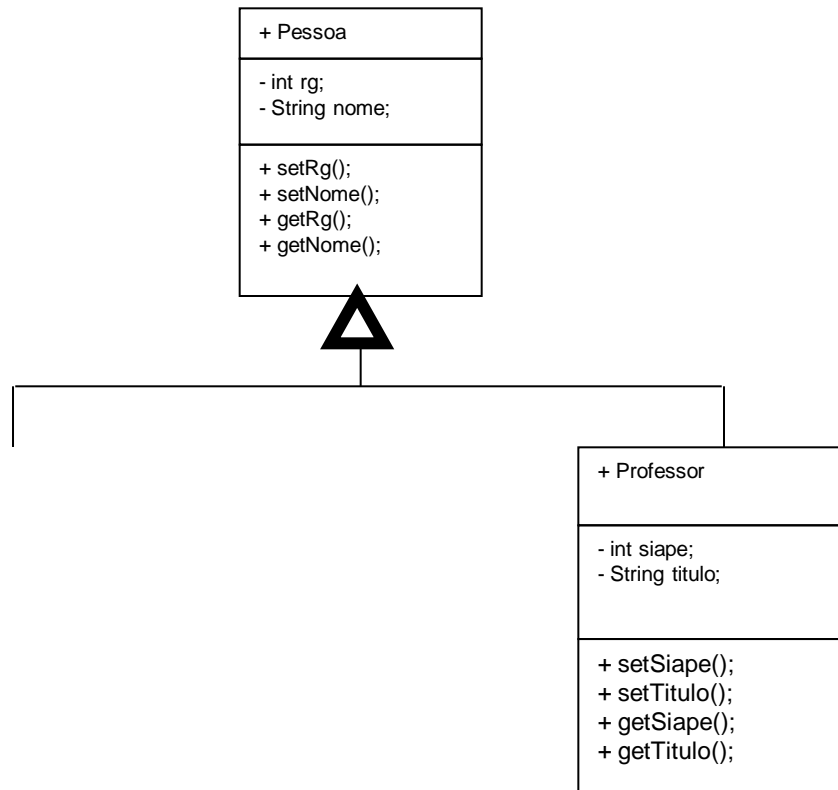
2 atributos (privados): **siape** (interio) e **titulo** (String)

4 métodos (públicos): **setSiape()**, **setTitulo()**, **getSiape()**, **getTitulo()**

|  |
|--|
| + Professor  |
| - int siape;<br>- String titulo;                                   |
| + setSiape();<br>+ setTitulo();<br>+ getSiape();<br>+ getTitulo(); |

## Herança: como identificar *(no projeto e na implementação)*

Da mesma forma como a classe Aluno **herda** a classe Pessoa, a classe **Professor** também herda a classe Pessoa:



# Herança codificação de “Professor”

```
public class Professor extends Pessoa{  
    private int siape;  
    private String titulo;
```

```
    public Professor( ) { }
```

```
    public Professor(int siape, String titulo){  
        this.siape = siape;  
        this.titulo = titulo;  
    }
```

```
    public void setSiape(int siape){  
        this.siape = siape;  
    }
```

```
    public void setTitulo(String titulo){  
        this.titulo= titulo;  
    }
```

```
    public int getSiape(){  
        return siape;  
    }
```

```
    public int getTitulo(){  
        return titulo;  
    }
```

```
}
```

+ Professor

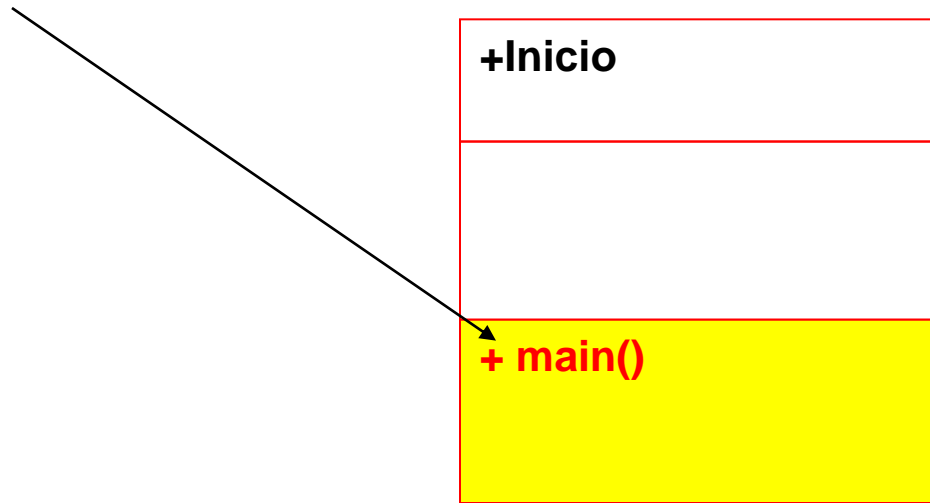
- int siape;  
- String titulo;

+ setSiape();  
+ setTitulo();  
+ getSiape();  
+ getTitulo();

**Observe o  
Polimorfismo com  
sobrecarga também  
na classe Professor!**

# Testando a Herança (codificação de “Início”)

E por fim a **classe “Início”**, que, neste caso, **é composta apenas pelo método `main()`**. Não tem nenhum outro método ou atributo.



- Nesta modelagem, a classe Início é a **única** que faz “**Troca de Mensagens**” com as outras classes (Aluno e Professor – que não se comunicam diretamente entre si).
- Esta **troca de mensagens** se dá pela declaração de objetos dos tipos de classes e pela invocação dos métodos através dos objetos declarados. Veja parte do código:



# Testando a Herança (codificação de “Início”)

```
public class Inicio{  
    public static void main(String args[]){  
        Aluno a = new Aluno();  
        Professor p = new Professor();  
  
        a.setRg(1);  
        a.setNome("Jesus");  
        a.setRa(10);  
        a.setTurma("A");  
        System.out.println("\n Rg: "+a.getRg());  
        System.out.println("\n Rg: "+a.getNome());  
        System.out.println("\n Rg: "+a.getRa());  
        System.out.println("\n Rg: "+a.getTurma());  
    }  
}
```

Declarando 2 objetos:

“a” do tipo Aluno

e

“p” do tipo Professor

Utilização do **Mecanismo de Herança** pelo objeto “a” (que é do tipo Aluno) através dos métodos get’s e set’s (herdados de Pessoa) que por sua vez instanciam os atributos também herdados de Pessoa: rg e nome

A classe Inicio **Trocando Mensagens** com a classe Aluno através do objeto “a”, que é do tipo Aluno.

**Exercícios:** Até aqui temos somente de parte do código (Aluno). Faltou a parte que trata do Professor. Termine o código (parte do “Professor”) na classe Inicio.

# Classe Abstrata

## *Classe concreta é possível...*

Até agora trabalhamos com **classes concretas**, isto é, classes das quais podemos declarar e **instanciar (new...)**, de forma direta, **objetos de seu tipo**.

Por exemplo: dentro da classe **Início**, utilizada anteriormente, além dos objetos dos tipos **Aluno** e **Professor**, que fazem parte daquele exemplo, também poderia declarar e **instanciar** um objeto do tipo **Pessoa** e através deste utilizar os membros públicos de Pessoa, como mostra o código a seguir:

## *Classe concreta é possível...*

```
public class Inicio{  
    public static void main(String args[]){  
        Aluno a = new Aluno();  
        Professor p = new Professor();  
        Pessoa pes = new Pessoa();  
        pes.setRg(1);  
        pes.setNome("Jesus");  
        System.out.println("\n Rg: "+pes.getRg());  
        System.out.println("\n Rg: "+pes.getNome());  
    }  
}
```

Declarando e  
**instanciando 1**  
objeto:  
"pes" do tipo Pessoa

# *Classe abstrata*

Observe que conseguiu declarar e **instanciar** (e utilizar) um objeto do tipo Pessoa, pois ela, até então é uma classe “Concreta”. Assim como Aluno e Professor.

Mas poderia haver uma situação na qual não devesse ser permitida a declaração direta de uma “Pessoa” que não fosse Aluno ou Professor.

Para resolver este problema deveríamos declarar a classe Pessoa como “abstrata” pois:

## **Classe abstrata:**

- Embora seja possível declarar um objeto de seu tipo, **não é possível instanciar este objeto**;
- Quase sempre é usada com classe como classe base (classe mãe) num mecanismo de herança.

**Faça o teste:** altere a classe Pessoa, na linha de definição da classe coloque a especificação de acesso “abstract” (vide próximo slide), depois teste novamente a classe Início.

# Classe abstrata

Alteração da classe Pessoa:

```
abstract class Pessoa{
    private int rg;
    private String nome;

    public Pessoa( ) { }

    public Pessoa(int rg, String nome){
        this.rg = rg;
        this.nome = nome;
    }

    public void setRg(int rg){
        this.rg = rg;
    }
    public void setNome(String nome){
        this.nome = nome;
    }
    public int getRg(){
        return rg;
    }
    public int getNome(){
        return nome;
    }
}
```

**Nota:** Após fazer esta alteração teste novamente a classe Inicio que deve ter o código parecido com o apresentado no próximo slide

## Classe Abstrata *(testando a possibilidade de declarar objetos)*

```
public class Inicio{  
    public static void main(String args[]){  
        Aluno a = new Aluno();  
        Professor p = new Professor();  
        Pessoa pes = new Pessoa();  
        pes.setRg(1);  
        pes.setNome("Jesus");  
        System.out.println("\n Rg: "+pes.getRg());  
        System.out.println("\n Rg: "+pes.getNome());  
    }  
}
```

**Nota:** Observe o erro logo na compilação do código.

# **Polimorfismo com Sobrescrita**



## Polimorfismo Sobrescrita (*definição*)

Métodos **com a mesma assinatura em classes distintas mas envolvidas num mecanismo de herança**

- Implementação através de dois tipos de métodos:

**-Método abstrato:**

- Contido numa classe abstrata (classe-mãe);
- **Não apresenta “corpo” na classe-mãe.** Somente sua assinatura;
- Torna obrigatório sua implementação (sobrescrição) na classe-filha.

**-Método concreto:**

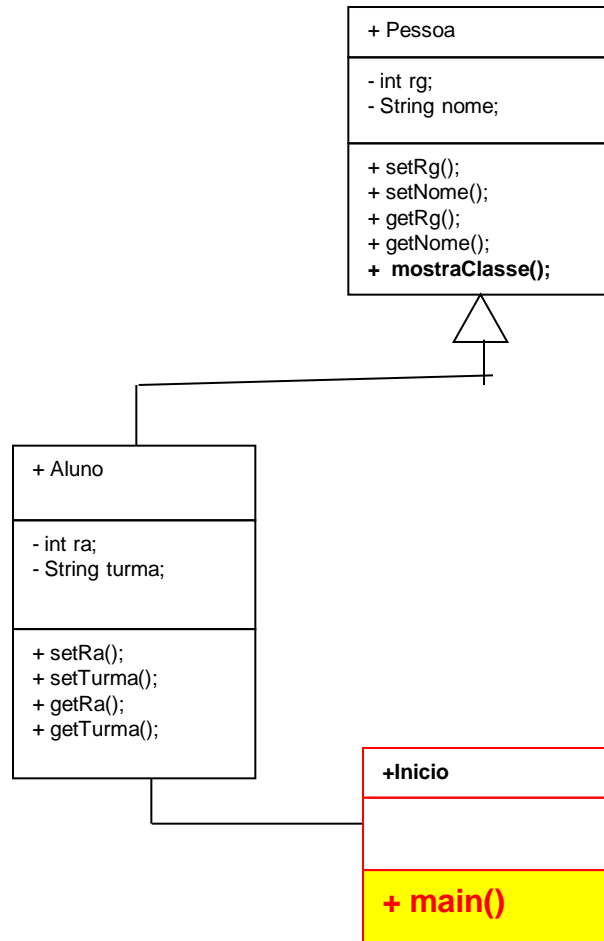
- Pode estar contido numa classe concreta ou abstrata;
- **Pode apresentar “corpo” na classe-mãe;**
- Sua re-implementação (sobrescrição) na classe filha é opcional.

Polimorfismo com Sobrescrita utilizando

# **Métodos Concretos**

## Polimorfismo Sobrescrita (*método concreto*)

Observe o diagrama de classes (vamos usar somente as classes Pessoa, Aluno e Inicio para exemplificar):



## Polimorfismo Sobrescrita (método concreto)

### classe Pessoa:

```
public class Pessoa{
    private int rg;
    private String nome;

    public void setRg(int rg){
        this.rg = rg;
    }
    public void setNome(String nome){
        this.nome = nome;
    }
    public int getRg(){
        return rg;
    }
    public int getNome(){
        return nome;
    }
    public void mostraClasse(){
        System.out.println("\n Estou na classe Pessoa");
    }
}
```

+ Pessoa

- int rg;  
- String nome;

+ setRg();  
+ setNome();  
+ getRg();  
+ getNome();  
+ **mostraClasse();**

## Polimorfismo Sobrescrita (método concreto)

classe Aluno:

```
public class Aluno extends Pessoa{  
    private int ra;  
    private String turma;  
  
    public void setRa(int ra){  
        this.ra = ra;  
    }  
    public void setTurma(String turma){  
        this.turma= turma;  
    }  
    public int getRa(){  
        return ra;  
    }  
    public int getTurma(){  
        return turma;  
    }  
}
```

|  |
|--|
| + Aluno  |
| - int ra;<br>- String turma;                               |
| + setRa();<br>+ setTurma();<br>+ getRa();<br>+ getTurma(); |

**Nota:** Ainda não há a sobrescrição do método mostraClasse( ) herdado da classe Pessoa.

## TESTANDO o Polimorfismo Sobrescrita (método concreto)

```
public class Inicio{  
    public static void main(String args[]){  
        Aluno a = new Aluno();  
  
        a.setRg(1);  
        a.setNome("Jesus");  
        a.setRa(10);  
        a.setTurma("A");  
        System.out.println("\n Rg: "+a.getRg());  
        System.out.println("\n Rg: "+a.getNome());  
        System.out.println("\n Rg: "+a.getRa());  
        System.out.println("\n Rg: "+a.getTurma());  
        a.mostraClasse();  
    }  
}
```

Ainda não há a sobrescrição. **Logo estará utilizando o método herdado** da classe Pessoa. Observe a mensagem que irá aparecer ao executar esta linha

## Polimorfismo Sobrescrita (método concreto) – sobrescrevendo na classe Aluno

```
public class Aluno extends Pessoa{  
    private int ra;  
    private String turma;  
  
    public void setRa(int ra){  
        this.ra = ra;  
    }  
    public void setTurma(String turma){  
        this.turma= turma;  
    }  
    public int getRa(){  
        return ra;  
    }  
    public int getTurma(){  
        return turma;  
    }  
    public mostraClasse(){  
        System.out.println("\nEstou na classe Aluno");  
    }  
}
```

|  |
|--|
| + Aluno  |
| - int ra;<br>- String turma;                               |
| + setRa();<br>+ setTurma();<br>+ getRa();<br>+ getTurma(); |

**Nota:** Atente para **a sobrescrição do método na classe Aluno**. Lembre-se este método já foi herdado da classe Pessoa e tinha uma implementação.

## TESTANDO o Polimorfismo Sobrescrita (método concreto) – depois de Aluno alterado

```
public class Inicio{  
    public static void main(String args[]){  
        Aluno a = new Aluno();  
  
        a.setRg(1);  
        a.setNome("Jesus");  
        a.setRa(10);  
        a.setTurma("A");  
        System.out.println("\n Rg: "+a.getRg());  
        System.out.println("\n Rg: "+a.getNome());  
        System.out.println("\n Rg: "+a.getRa());  
        System.out.println("\n Rg: "+a.getTurma());  
        a.mostraClasse();  
    }  
}
```

Agora há a sobrescrição. Então estará utilizando o método sobrescrito da própria classe Aluno. Novamente, **observe a mensagem** que irá aparecer ao executar esta linha.



Polimorfismo Sobrescrita (***SUPER*** – *acessando método da classe mãe*)

## Uma pergunta?

E se, mesmo tendo um método sobrecarregado, eu quiser utilizar o método “original” da classe mãe?

**R.:** Para isso poderá utilizar o objeto “**super**”. Este objeto é usado para “apontar” para um membro da classe-mãe (num mecanismo de herança).

Para exemplificar, veja o código do método `mostraMae()` . Que no caso deverá estar contido na classe filha `Aluno`:

## Alterando classe Aluno: colocando Super

```
public class Aluno extends Pessoa{
    private int ra;
    private String turma;

    public void setRa(int ra){
        this.ra = ra;
    }
    public void setTurma(String turma){
        this.turma= turma;
    }
    public int getRa(){
        return ra;
    }
    public int getTurma(){
        return turma;
    }
    public void mostraClasse(){
        System.out.println("\n RG da classe Aluno: "+(rg+100));
    }
    public void mostraMae(){
        super.motraClasse();
    }
}
```

|   |
|---|
| + Aluno   |
| - int ra;<br>- String turma;  |
| + setRa();<br>+ setTurma();<br>+ getRa();<br>+ getTurma();<br><b>+ mostraMae();</b> |

**Nota:** no método mostraMae() temos uma chamada do método mostraClasse() contido na classe-mãe Pessoa (através do apontamento do objeto **super**).

# TESTANDO o Polimorfismo Sobrescrita (SUPER)

```
public class Inicio{  
    public static void main(String args[]){  
        Aluno a = new Aluno();  
        a.setRg(1);  
        a.setNome("Jesus");  
        a.setRa(10);  
        a.setTurma("A");  
        System.out.println("\n Rg: "+a.getRg());  
        System.out.println("\n Rg: "+a.getNome());  
        System.out.println("\n Rg: "+a.getRa());  
        System.out.println("\n Rg: "+a.getTurma());  
        a.mostraClasse();  
        a.mostraMae();  
    }  
}
```

Executará o **mostraClasse()** da classe **Aluno**

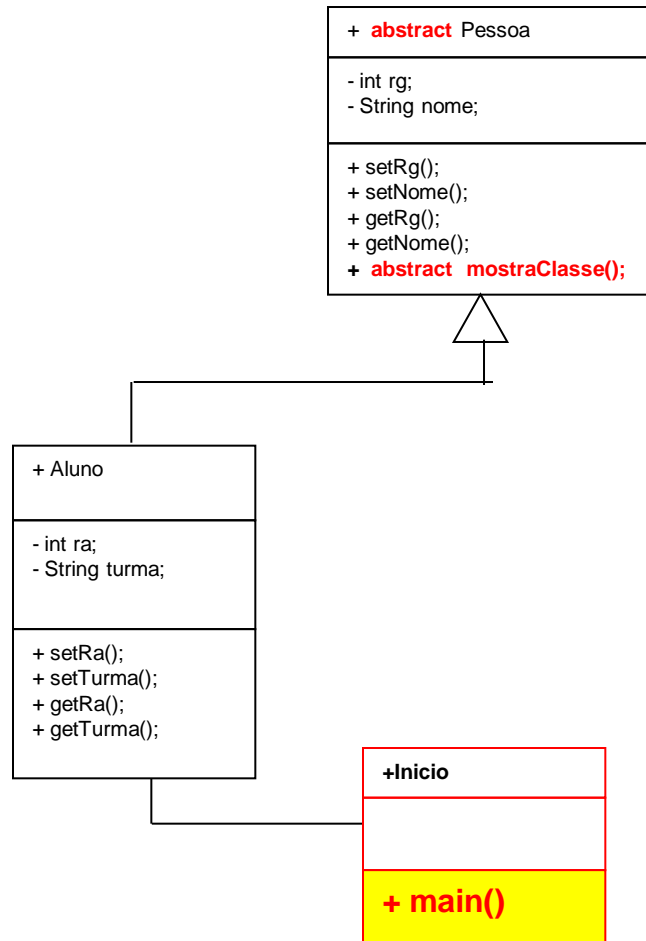
Executará, indiretamente, o **mostraClasse()** da classe **Pessoa**

Polimorfismo com Sobrescrita utilizando

# **Métodos Abstratos**

## Polimorfismo Sobrescrita (*método abstratos*)

Observe o diagrama de classes (vamos usar somente as classes Pessoa, Aluno e Inicio para exemplificar):



## Polimorfismo Sobrescrita (*método abstratos*)

**classe Pessoa:**

```
public abstract class Pessoa{  
    private int rg;  
    private String nome;  
  
    public void setRg(int rg){  
        this.rg = rg;  
    }  
    public void setNome(String nome){  
        this.nome = nome;  
    }  
    public int getRg(){  
        return rg;  
    }  
    public int getNome(){  
        return nome;  
    }  
    abstract void mostraClasse();  
}
```

+ **abstract** Pessoa

- int rg;  
- String nome;

+ setRg();  
+ setNome();  
+ getRg();  
+ getNome();  
+ **abstract** mostraClasse();

**Nota:**

**Método abstrato, somente assinatura**

## Polimorfismo Sobrescrita (método abstratos) – ERRO: Aluno sem a SOBRESCRIÇÃO

**classe Aluno:**

```
public class Aluno extends Pessoa{  
    private int ra;  
    private String turma;  
  
    public void setRa(int ra){  
        this.ra = ra;  
    }  
    public void setTurma(String turma){  
        this.turma= turma;  
    }  
    public int getRa(){  
        return ra;  
    }  
    public int getTurma(){  
        return turma;  
    }  
}
```

+ Aluno

- int ra;  
- String turma;

+ setRa();  
+ setTurma();  
+ getRa();  
+ getTurma();

**Nota: Ainda não há** a sobrescrição do método mostraClasse herdado da classe Pessoa.

## Polimorfismo Sobrescrita (método abstratos) – ERRO: Aluno sem a SOBRESCRIÇÃO

```
public class Inicio{  
    public static void main(String args[]){  
        Aluno a = new Aluno();  
  
        a.setRg(1);  
        a.setNome("Jesus");  
        a.setRa(10);  
        a.setTurma("A");  
        System.out.println("\n Rg: "+a.getRg());  
        System.out.println("\n Rg: "+a.getNome());  
        System.out.println("\n Rg: "+a.getRa());  
        System.out.println("\n Rg: "+a.getTurma());  
    }  
}
```

**Nota:** Mesmo o método mostraClasse() não sendo chamado aqui, apresentará um erro logo na compilação, informando que há um método abstrato (na classe Pessoa) que deve ser implementado na classe Aluno



## Polimorfismo Sobrescrita (método abstratos): CERTO – Aluno com sobrescrição

```
public class Aluno extends Pessoa{  
    private int ra;  
    private String turma;  
  
    public void setRa(int ra){  
        this.ra = ra;  
    }  
    public void setTurma(String turma){  
        this.turma= turma;  
    }  
    public int getRa(){  
        return ra;  
    }  
    public int getTurma(){  
        return turma;  
    }  
    public mostraClasse(){  
        System.out.println("\nSobrescrevi o método Abstrato de Pessoa. Estou na classe Aluno");  
    }  
}
```

|  |
|--|
| + Aluno  |
| - int ra;<br>- String turma;                               |
| + setRa();<br>+ setTurma();<br>+ getRa();<br>+ getTurma(); |

**Nota:** Atente para a sobrescrição do método abstrato na classe Aluno. Lembre-se este método foi herdado da classe Pessoa (onde é definido como abstrato), **não** tinha uma implementação.

## TESTANDO: Polimorfismo Sobrescrita (método abstratos)

```
public class Inicio{  
    public static void main(String args[]){  
        Aluno a = new Aluno();  
  
        a.setRg(1);  
        a.setNome("Jesus");  
        a.setRa(10);  
        a.setTurma("A");  
        System.out.println("\n Rg: "+a.getRg());  
        System.out.println("\n Rg: "+a.getNome());  
        System.out.println("\n Rg: "+a.getRa());  
        System.out.println("\n Rg: "+a.getTurma());  
        a.mostraClasse();  
    }  
}
```

Agora há a sobrescrição. Então estará utilizando o método sobrescrito da própria classe Aluno.

modificador de acesso

**FINAL**

## Modificador de acesso “FINAL” (*definição*)

Em Java a palavra reservada “FINAL” indica a impossibilidade de alteração à a estrutura a qual ela foi empregada. Porém esta indicação passa a ser contextual justamente pelos diferentes tipos de estruturas em que podemos empregá-la. Podemos utilizá-la na declaração de:

- **Atributos;**
- **Métodos;**
- **Classes;**

## Atributo FINAL *(definição e uso)*

### **Atributo final – envolve a questão de modificação de seu valor**

Indica que o atributo em questão será uma **constante**, isto é, uma estrutura de dados (assim como um variável) porém que não poderá ter seu valor modificado.

**Exemplo:** Imagine um código que deverá calcular as áreas de circunferências. Para isso deverá utilizar o “**PI radiano**”, cujo valor é sempre o mesmo (3,14159...), logo uma constante. Sendo assim poderíamos declarar o atributo da seguinte forma

```
final private float pi = 3.14159;
```

## Método FINAL *(definição e uso)*

### **Método final – envolve a questão do polimorfismo de sobrescrita**

Indica que o método **não pode ser sobrescrito**. Contrapondo-se ao conceito da **método abstrato** que é declarado em uma classe-mãe e deverá ser sobrescrito em uma classe-filha.

**Exemplo:** de acordo com os exemplos desta aula, que trataram da herança entre Pessoa e Aluno. Considere que o desenvolvedor da classe Aluno (que herdou a classe Pessoa) queira sobrescrever os métodos “set” (já definidos na classe Pessoa) . Isso poderia se uma tentativa de “driblar” o encapsulamento definido para a classe Pessoa. Pode-se evitar isso declarando que o método é final, isto é, não poderá ser sobrescrito nas classes-filhas:

```
final public void setRg(int rg){....}
```

## Classe FINAL *(definição e uso)*

### **Classe final – envolve a questão do mecanismo de herança**

Indica que a classe **não pode ser herdada**. Contrapondo-se ao conceito da **classe abstrata** que, geralmente, é projetada para ser utilizada como classe-mãe num mecanismo de herança, uma classe definida como “**final**” não poderá ser herdada.

**Exemplo:** Imagine que no desenvolvimento acadêmico o projeto não preveja (ou não deva existir) uma extensão de aluno.

**final public class Aluno extends Pessoa....**

Observe que aluno herda Pessoa, porém ela (a classe Aluno) não poderá ser herdada por nenhuma outra classe.