

Programação OO em Java

Profa Andréa Schwertner Charão
DELC/CT/UFSM

Sumário

■ Herança

- Visibilidade de atributos e métodos
- Extensão e sobreposição
- Uso de super
- Referências para objetos

■ Polimorfismo

- Conceitos
- Exemplo

Herança

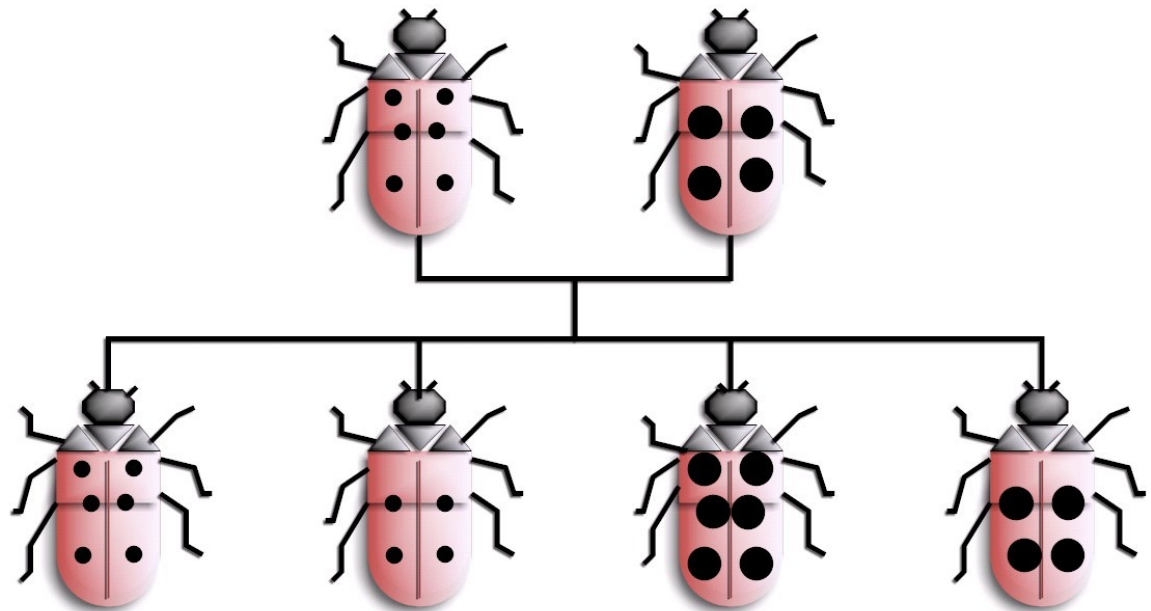
- "Um dia tudo isso será seu..."



Just think darling....one day, all of this will be yours!

Herança

- Inspiração no mundo real
- Pais transmitem aos filhos suas características e comportamento

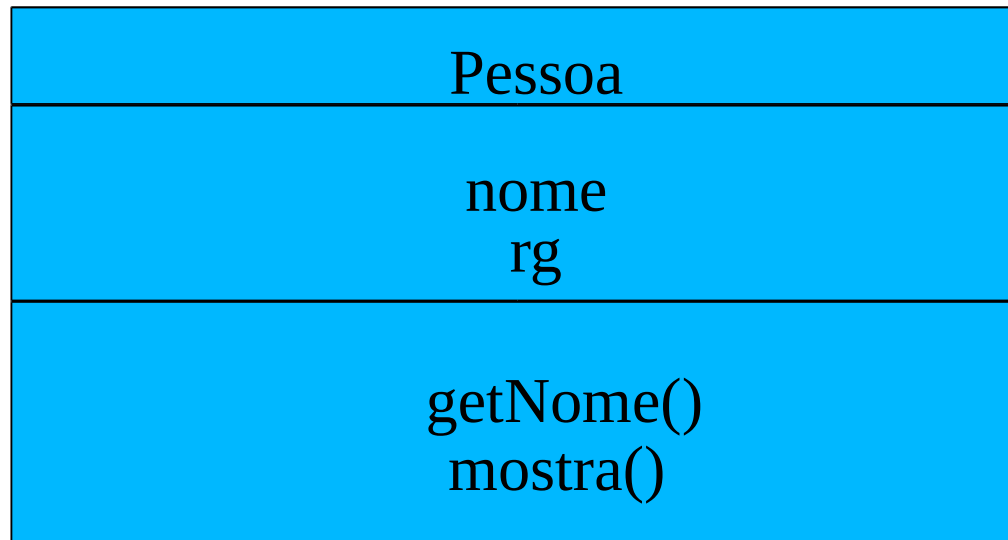


Herança e POO

- Permite criar classes que aproveitam atributos e métodos de classes existentes
- Motivação:
 - reutilização de código
 - com flexibilidade
- Usos:
 - especialização
 - extensão
 - sobreposição

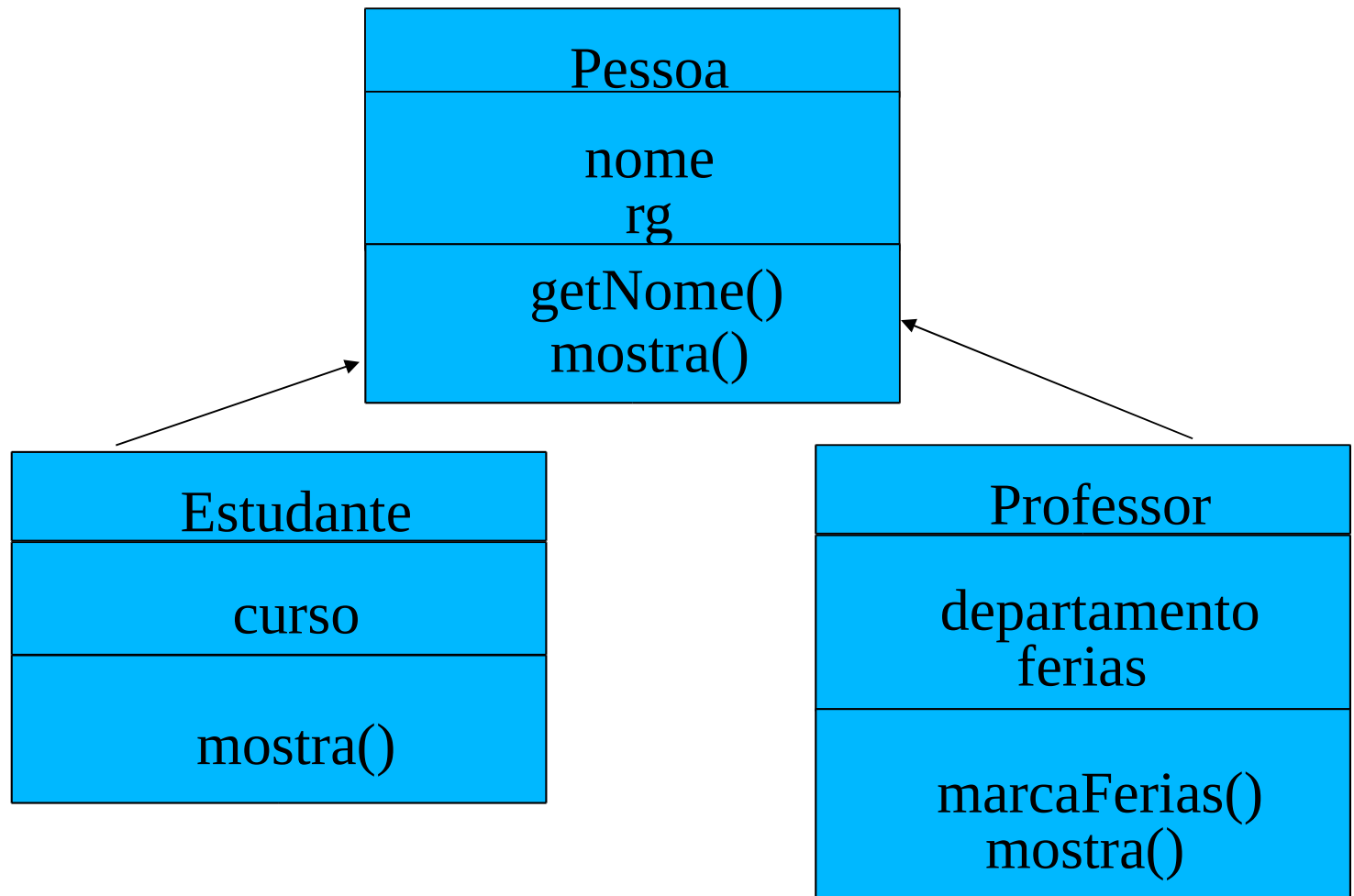
Herança

- Exemplo: classe Pessoa (existente)



Herança

- Novas classes: Estudante e Professor



Herança

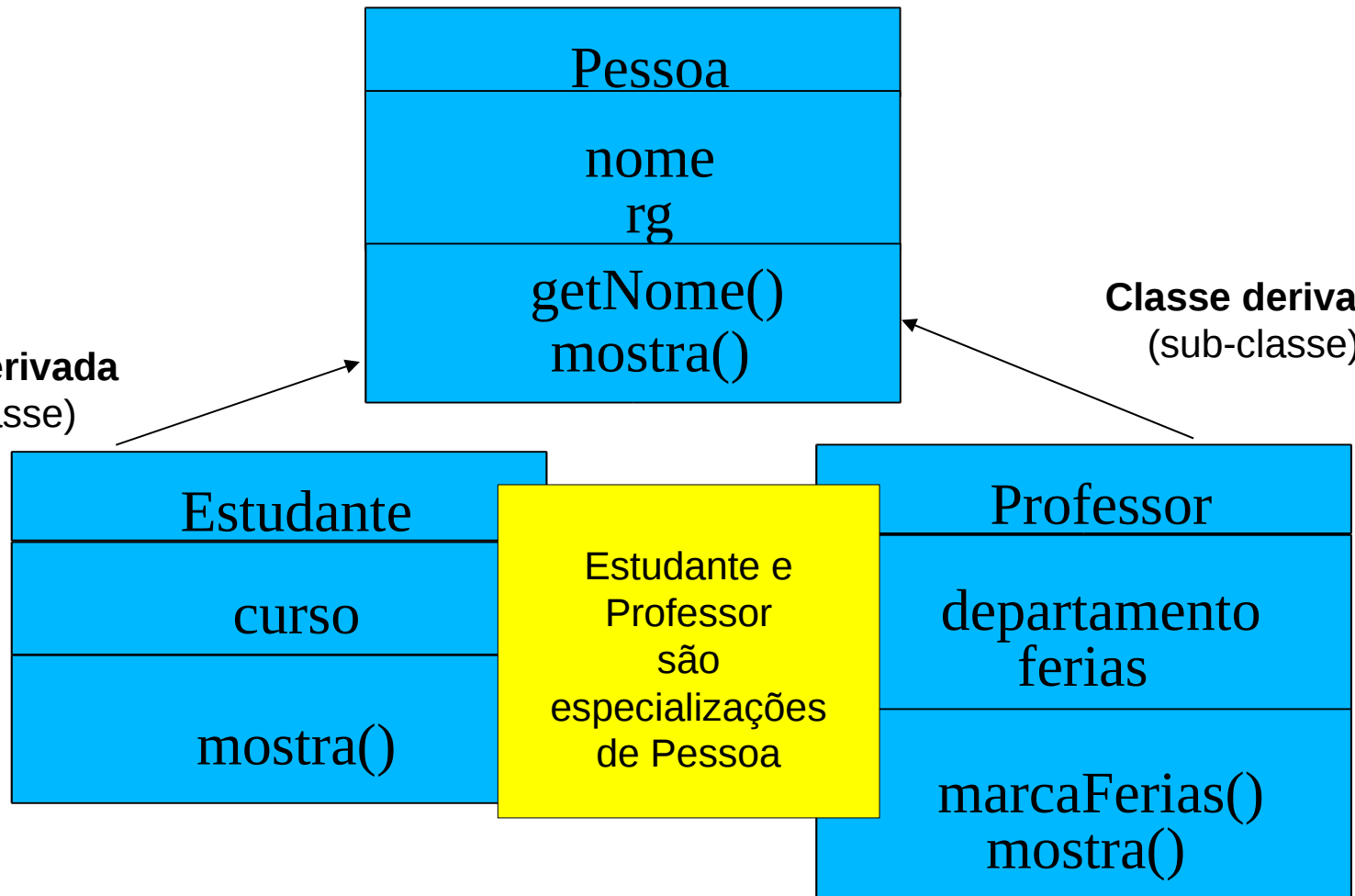
■ Terminologia

Super-classe

(classe-pai, classe-mãe,
classe-base)

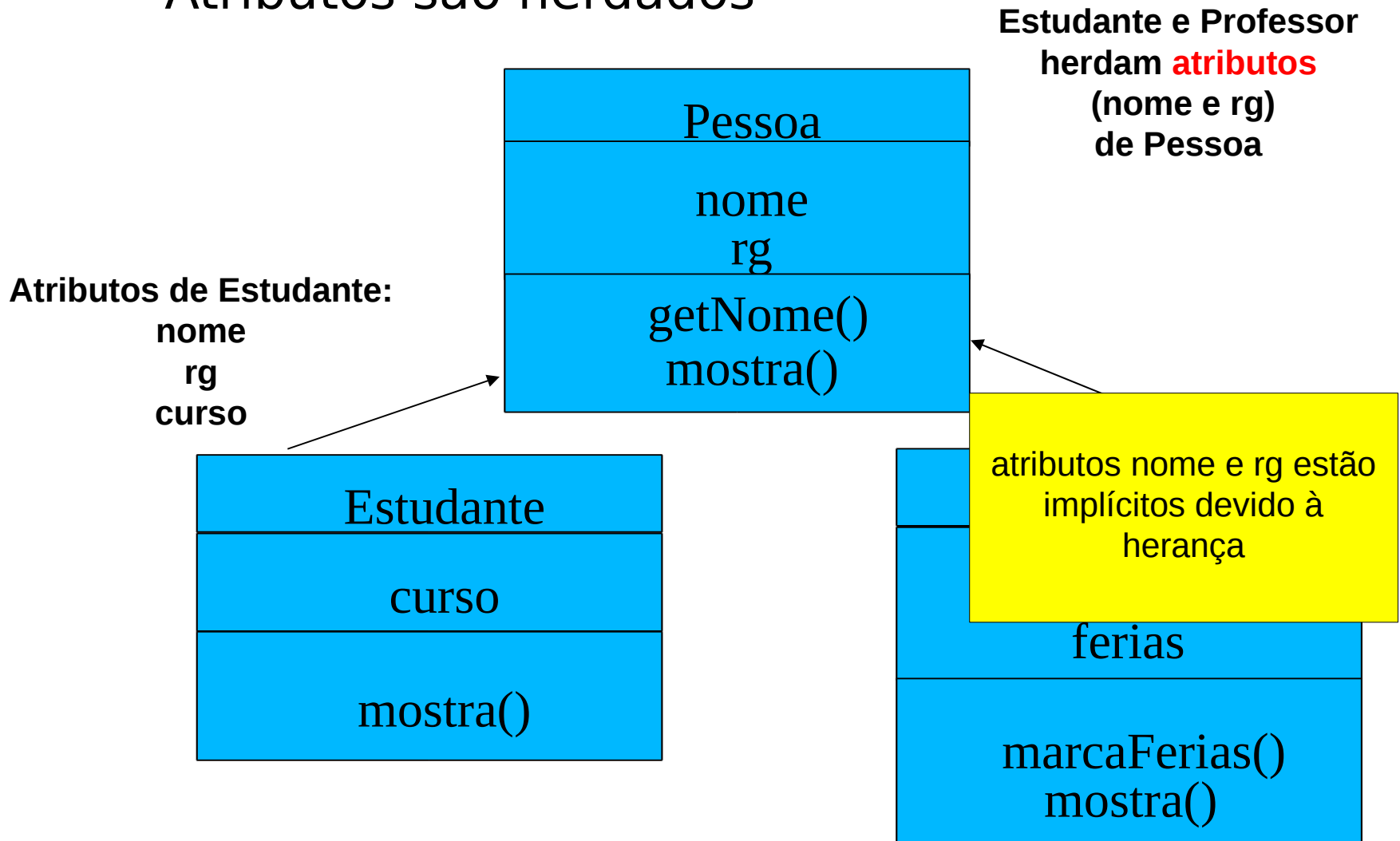
Classe derivada
(sub-classe)

Classe derivada
(sub-classe)



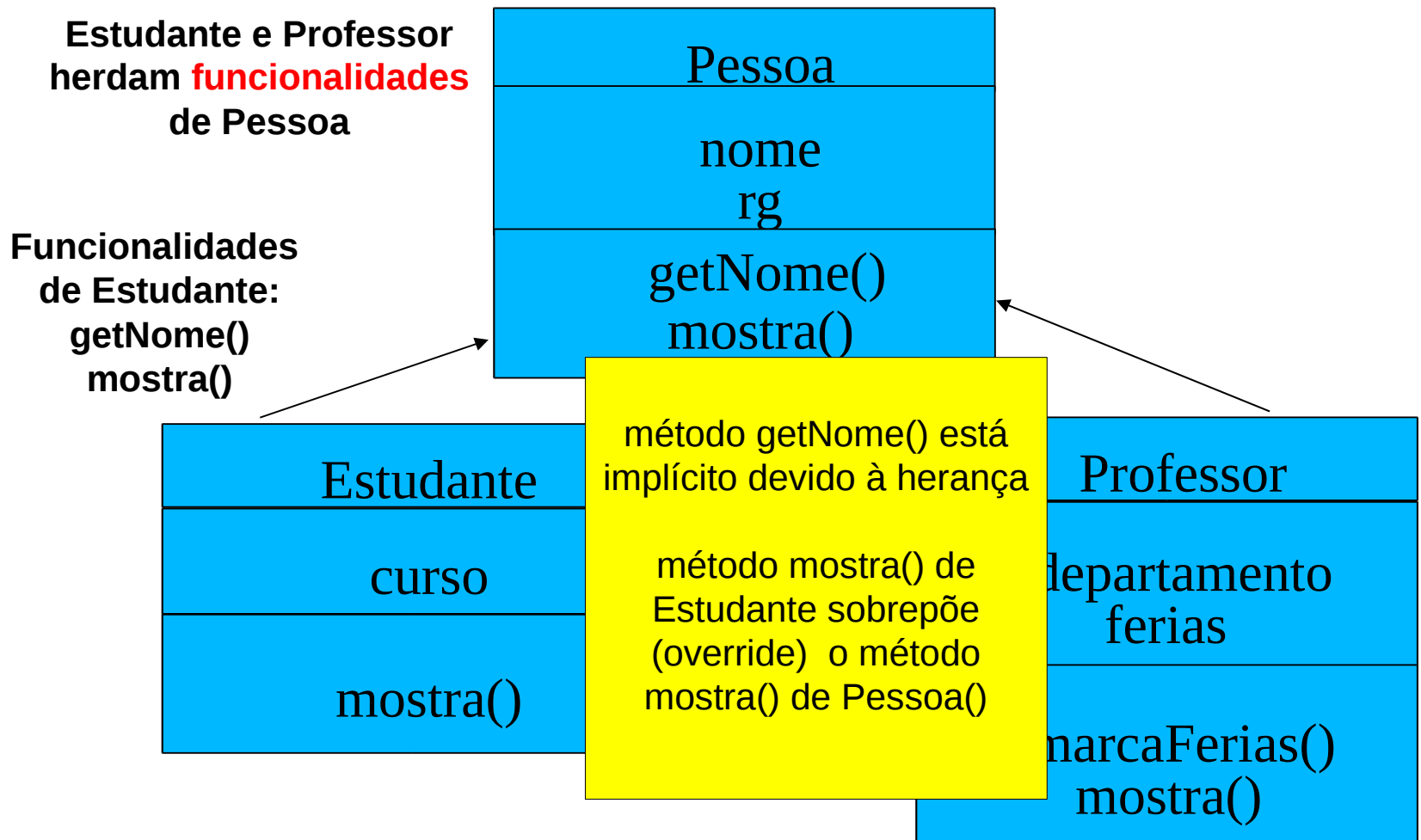
Herança

- Atributos são herdados



Herança

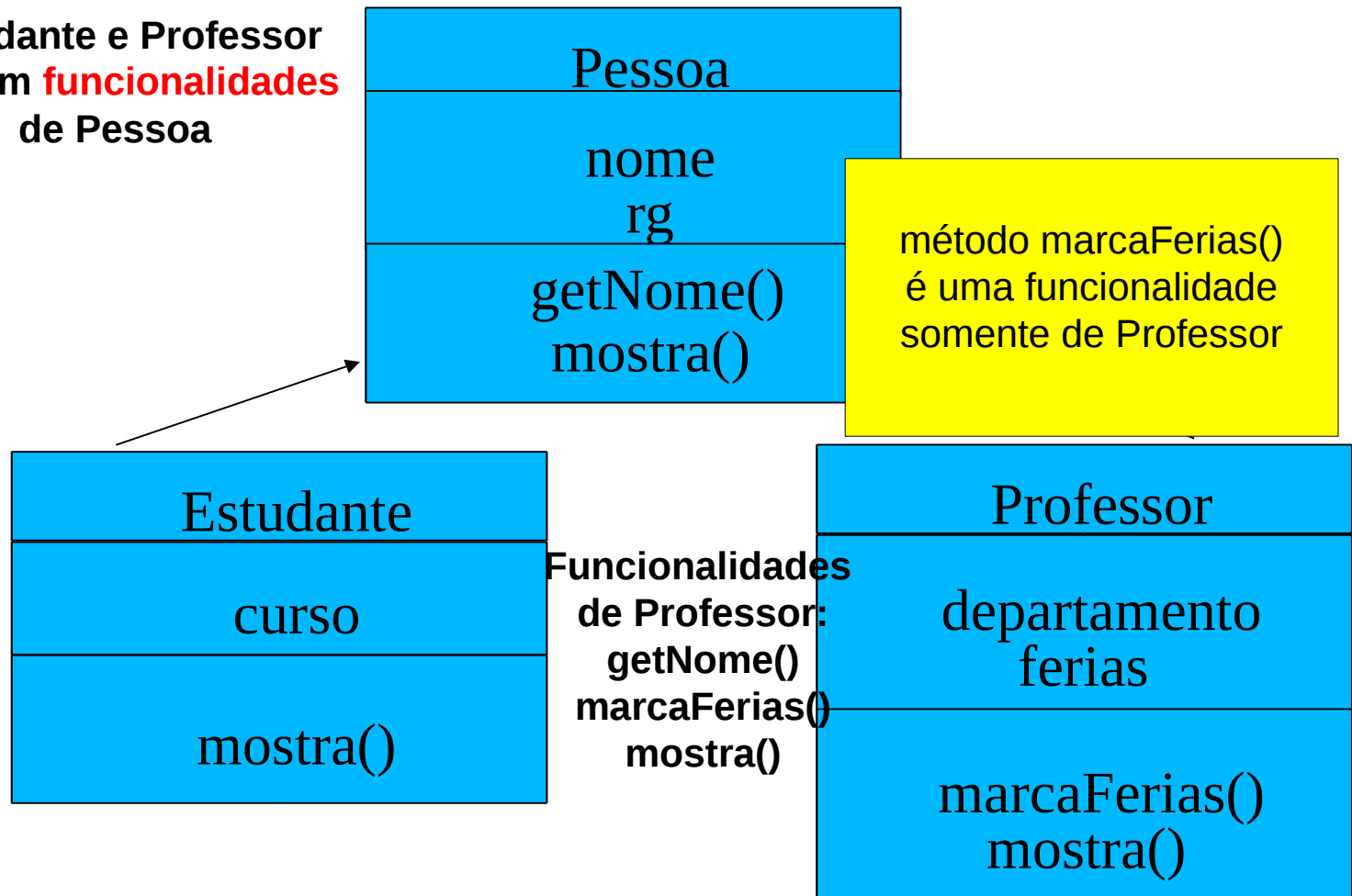
- Métodos (funcionalidades) são herdados



Herança

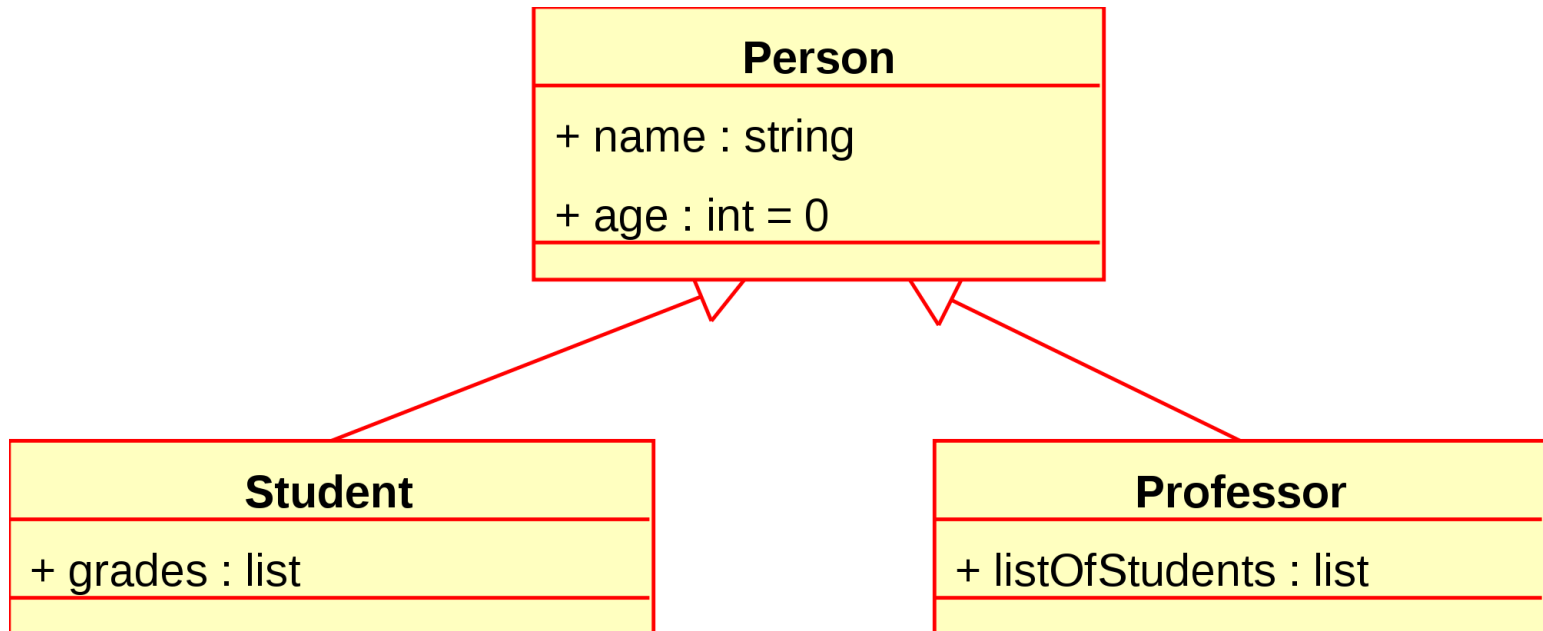
■ Novas classes: Estudante e Professor

Estudante e Professor
herdam **funcionalidades**
de Pessoa



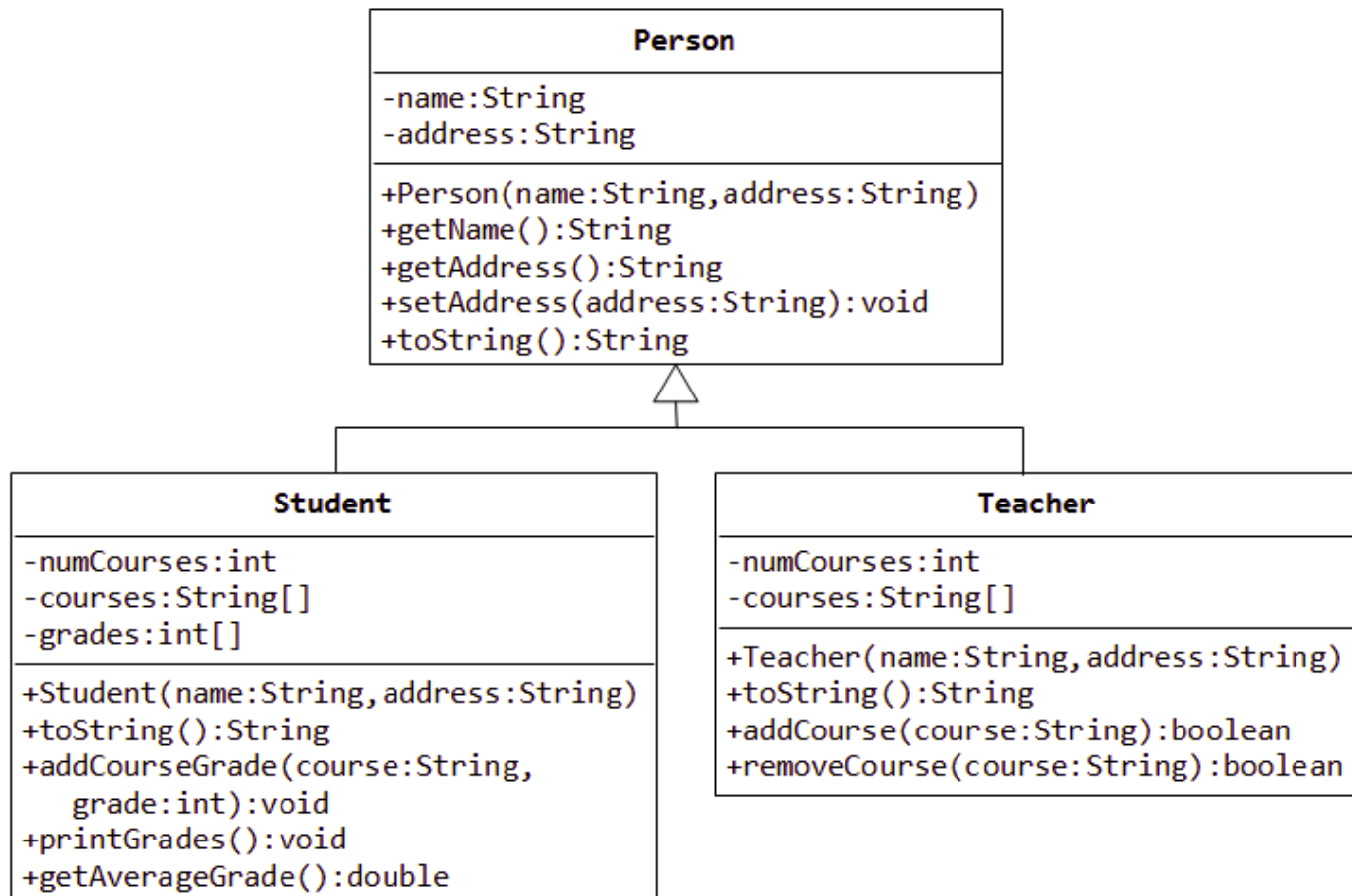
Herança e UML

- Diagrama de classes



Herança e UML

■ Diagrama de classes



Herança em Java

- Usa palavra-chave **extends** na declaração das sub-classes

```
class SuperClasse {  
    private int apriv;  
    public int apub;  
    private void metpriv();  
    public void metpub();  
}
```

```
class SubClasse  
    extends SuperClasse {  
}
```

Visibilidade de atributos e métodos

- Atributos e métodos **públicos** da super-classe podem ser acessados normalmente nas sub-classes
- O código abaixo é um exemplo disso:

```
SubClasse obj = new SubClasse();  
obj.apub = 99;  
obj.metpub();
```

Visibilidade de atributos e métodos

- Atributos e métodos **privados** da super-classe não podem ser acessados nas sub-classes
- O código abaixo vai dar **erro de compilação**:

```
SubClasse obj = new SubClasse();  
obj.apriv = 10;  
obj.metpriv();
```


Exemplo: Pessoa e Estudante

```
class Pessoa
{
    private String nome;

    public Pessoa()
    {
        nome = "";
    }

    public void setNome(String _nome)
    {
        nome = _nome;
    }

    public String getNome()
    {
        return nome;
    }

    public void mostra()
    {
        System.out.println("Nome: " + nome);
    }
}
```

```
class Estudante extends Pessoa
{
    private String curso;

    public Estudante()
    {
        curso = "";
    }

    public void setCurso(String _curso)
    {
        curso = _curso;
    }

    public void mostra()
    {
        // 
    }
}

class PessoaApp
{
    public static void main(String[] args)
    {
        Estudante e1 = new Estudante();
        e1.setNome("Jose");
        e1.setCurso("Direito");
        e1.mostra();
    }
}
```

Exemplo: Pessoa e Estudante

```
class Pessoa
{
    private String nome;

    public Pessoa()
    {
        nome = "";
    }

    public void setNome(String _nome)
    {
        nome = _nome;
    }

    public String getNome()
    {
        return nome;
    }

    public void mostra()
    {
        System.out.println("Nome: " + nome);
    }
}
```

Construtor de Pessoa
é invocado
implicitamente

```
class Estudante extends Pessoa
{
    private String curso;

    public Estudante()
    {
        curso = "";
    }

    public void setCurso(String _curso)
    {
        curso = _curso;
    }

    public void mostra()
    {
        System.out.println("Curso: " + curso);
    }
}

class PessoaApp
{
    public static void main(String[] args)
    {
        Estudante e1 = new Estudante();
        e1.setNome("Jose");
        e1.setCurso("Direito");
        e1.mostra();
    }
}
```

Exemplo: Pessoa e Estudante

```
class Pessoa
{
    private String nome;

    public Pessoa()
    {
        nome = "";
    }

    public void setNome(String _nome)
    {
        nome = _nome;
    }

    public String getNome()
    {
        return nome;
    }

    public void mostra()
    {
        System.out.println
    }
}
```

Método setNome()
herdado de Pessoa

```
class Estudante extends Pessoa
{
    private String curso;

    public Estudante()
    {
        curso = "";
    }

    public void setCurso(String _curso)
    {
        curso = _curso;
    }

    public void mostra()
    {
        // ...
    }
}

class PessoaApp
{
    public static void main(String[] args)
    {
        Estudante e1 = new Estudante();
        e1.setNome("Jose");
        e1.setCurso("Direito");
        e1.mostra();
    }
}
```

Exemplo: Pessoa e Estudante

```
class Pessoa
{
    private String nome;

    public Pessoa()
    {
        nome = "";
    }

    public void setNome(String _nome)
    {
        nome = _nome;
    }

    public String getNome()
    {
        return nome;
    }

    public void mostra()
    {
        System.out.println
    }
}
```

Método mostra()
deve mostrar
nome e curso

```
class Estudante extends Pessoa
{
    private String curso;

    public Estudante()
    {
        curso = "";
    }

    public void setCurso(String _curso)
    {
        curso = _curso;
    }

    public void mostra()
    {
        // ...
    }
}

class PessoaApp
{
    public static void main(String[] args)
    {
        Estudante e1 = new Estudante();
        e1.setNome("Jose");
        e1.setCurso("Direito");
        e1.mostra();
    }
}
```

Exemplo: Pessoa e Estudante

```
class Pessoa
{
    private String nome;

    public Pessoa()
    {
        nome = "";
    }

    public void setNome(String _nome)
    {
        nome = _nome;
    }

    public String getNome()
    {
        return nome;
    }

    public void mostra()
    {
        System.out.println("Nome: " + getNome());
    }
}
```

Mas nome é atributo privado de Pessoa!!!

Método mostra() deve mostrar nome e curso

```
class Estudante extends Pessoa
{
    private String curso;

    public Estudante()
    {
        curso = "";
    }

    public void setCurso(String _curso)
    {
        curso = _curso;
    }

    public void mostra()
    {
        // ...
    }
}

class PessoaApp
{
    public static void main(String[] args)
    {
        Estudante e1 = new Estudante();
        e1.setNome("Jose");
        e1.setCurso("Direito");
        e1.mostra();
    }
}
```

Uso de super (1)

- Referência para objeto da super-classe (para acessar seus atributos/métodos)

```
public void mostra()  
{  
      
}  
}
```

Veja isso em:

<http://download.oracle.com/javase/tutorial/java/landl/super.html>

Uso de super (1)

- Referência para objeto da super-classe (para acessar seus atributos/métodos)

```
public void mostra()  
{  
    super.mostra();  
    System.out.println("Curso: " + curso);  
}
```

Veja isso em:

<http://download.oracle.com/javase/tutorial/java/landl/super.html>

Uso de super (2)

- Para invocar explicitamente um construtor da super-classe
- Deve estar na primeira linha do construtor da sub-classe

```
class Pessoa
{
    private String nome;

    public Pessoa()
    {
        nome = "";
    }

    public Pessoa(String _nome)
    {
        nome = _nome;
    }
}
```

```
class Estudante extends Pessoa
{
    private String curso;

    public Estudante()
    {
        curso = "";
    }

    public Estudante(String _nome, String _curso)
    {
        super(_nome);
        curso = _curso;
    }
}
```

Veja isso em:

<http://download.oracle.com/javase/tutorial/java/landl/super.html>

Visibilidade protected

- Atributos e métodos **protected** da super-classe podem ser acessados nas sub-classes, mas não são acessíveis a outras classes

```
class Pessoa
{
    protected String nome;

    public Pessoa()
    {
        this("");
    }

    public Pessoa(String _nome)
    {
        nome = _nome;
    }
}
```

```
class Estudante extends Pessoa
{
    private String curso;

    public Estudante()
    {
        this("", "");
    }

    public Estudante(String _nome, String _curso)
    {
        nome = _nome;
        curso = _curso;
    }
}
```

Atributo nome
acessado
normalmente

Sobrescrita ou extensão de métodos

- Métodos da super-classe podem ser estendidos ou sobrescritos nas sub-classes

Método mostra()
de Estudante
estende o método
mostra() de Pessoa

```
public void mostra()  
{  
    super.mostra();  
    System.out.println("Curso: " + curso);  
}
```

Sobrescrita ou extensão de métodos

- Métodos da super-classe podem ser estendidos ou sobrescritos nas sub-classes

Método mostra()
de Estudante
sobrescreve o método
mostra() de Pessoa

```
public void mostra()  
{  
    ...  
    System.out.println("Curso: " + curso);  
}
```

Mais sobre herança em Java

- Ao contrário de C++, Java não permite herança múltipla
- Em Java, todas as classes derivam (implicitamente) da classe **Object**
- A classe Object possui alguns atributos e métodos úteis:
 - toString(): retorna uma string representando o objeto
 - clone(): cria e retorna uma cópia do objeto
 - etc.

Sobrescrita do método toString

```
class Pessoa {  
    private String nome;  
    ...  
    String toString() {  
        return nome;  
    }  
}
```

```
class PessoaApp {  
    public static void main(String[] args) {  
        Pessoa p = new Pessoa("Joao");  
        System.out.println(p);  
    }  
}
```

Uso de this

- Referência para o objeto corrente
- Pode ser usado para invocação explícita de um construtor

"this" não tem nada a ver com herança, mas assemelha-se a "super" em seu uso

```
class Relogio
{
    private int hora, minuto;

    Relogio(int hora, int minuto)
    {
        this.hora = hora;
        this.minuto = minuto;
    }

    Relogio()
    {
        this(0,0);
    }
}
```

Veja isso em:

<http://download.oracle.com/javase/tutorial/java/javaOO/thiskey.html>

Herança e referências para objetos

- Uma referência para a super-classe pode apontar para um objeto da sub-classe
- O contrário não é verdadeiro
- "Todo estudante é uma pessoa, mas nem toda pessoa é estudante"

```
Pessoa p;  
Estudante e = new Estudante();  
p = e;  
// e = p;    // Errado!
```

Polimorfismo

- Significa: muitas formas
- Um único nome de método, várias implementações
- Sobrecarga (overloading) e sobrescrita (override) são exemplos de polimorfismo
- Exemplos:
 - método toString()
 - método mostra()
 - construtores

Qual método será invocado?

```
class TesteVinculacaoDinamica
{
    public static void main(String[] args)
    {
        Pessoa refp;
        Estudante refe;
        Pessoa p = new Pessoa("Maria");
        Estudante e = new Estudante("Jose", "Direito");

        //refe = p;    // invalido

        refp = e;
        refp.mostra();

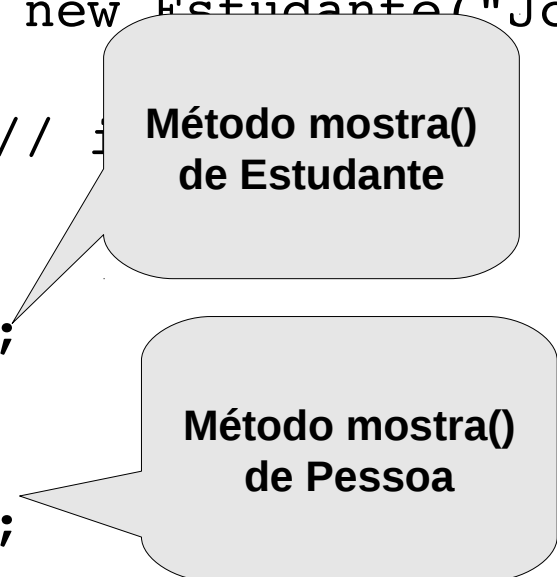
        refp = p;
        refp.mostra();
    }
}
```

Qual método será invocado?

```
class TesteVinculacaoDinamica
{
    public static void main(String[] args)
    {
        Pessoa refp;
        Estudante refe;
        Pessoa p = new Pessoa("Maria");
        Estudante e = new Estudante("Jose", "Direito");

        //refe = p; // 1
        refe = e;
        refp.mostra();

        refp = p;
        refp.mostra();
    }
}
```



Método mostra()
de Estudante

Método mostra()
de Pessoa

Polimorfismo dinâmico

- O exemplo do slide anterior ilustra polimorfismo dinâmico
- Vinculação dinâmica (**dynamic binding**): método a ser chamado é determinado em tempo de execução
- Mecanismo poderoso quando se tem herança e métodos com mesmo nome

Utilidade do polimorfismo

classes: Triangulo, Retangulo, etc.

objetos: Triangulo tris[], Retangulo rets[], etc.

```
for (Triangulo t : tris) {  
    desenhaTriangulo(t);  
}  
for (Retangulo r : rets) {  
    desenhaRetangulo(r);  
}  
// etc.
```

Código repetitivo.

**Semelhante à
programação
procedimental.**

Utilidade do polimorfismo

```
class Figura {
    public void desenhaFigura() {}
}
class Retangulo extends Figura {
    public void desenhaRetangulo() {}
}
class Triangulo extends Figura {
    public void desenhaTriangulo() {}
}
class Polimorfismo {
    public static void main(String[] args) {
        Figura figs[] = new Figura[2];
        figs[0] = new Retangulo();
        figs[1] = new Triangulo();
        for (Figura f : figs) {
            if (f instanceof Retangulo) {
                f.desenhaRetangulo();
            }
            if (f instanceof Triangulo) {
                f.desenhaTriangulo();
            }
        }
    }
}
```

Código repetitivo.

Desnecessário.

Utilidade do polimorfismo

```
class Figura {  
    public void desenha() {}  
}  
class Retangulo extends Figura {  
    public void desenha() {}  
}  
class Triangulo extends Figura {  
    public void desenha() {}  
}  
class Polimorfismo {  
    public static void main(String[] args) {  
        Figura figs[] = new Figura[2];  
        figs[0] = new Retangulo();  
        figs[1] = new Triangulo();  
        for (Figura f : figs) {  
            if (f instanceof Retangulo) {  
                f.desenha();  
            }  
            if (f instanceof Triangulo) {  
                f.desenha();  
            }  
        }  
    }  
}
```

Código repetitivo.

Desnecessário.

Utilidade do polimorfismo

```
class Figura {  
    public void desenha() {}  
}  
class Retangulo extends Figura {  
    public void desenha() {}  
}  
class Triangulo extends Figura {  
    public void desenha() {}  
}  
class Polimorfismo {  
    public static void main(String[] args) {  
        Figura figs[] = new Figura[2];  
        figs[0] = new Retangulo();  
        figs[1] = new Triangulo();  
        for (Figura f : figs) {  
            f.desenha();  
        }  
    }  
}
```

**Graças ao
polimorfismo,
o código
fica bem
mais simples!**