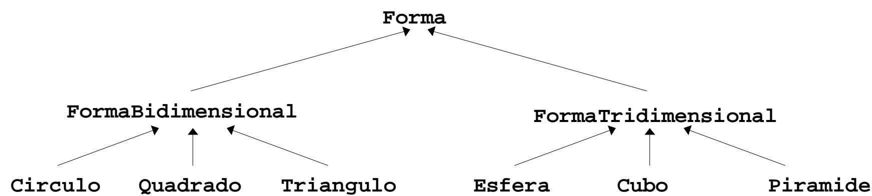
# 1. Herança

- Herança - forma de reutilização de software
- Novas classes são criadas a partir de classes já existentes
- Absorvem atributos e comportamentos, e incluem os seus próprios
  - Sobrescrevem métodos - redefinem métodos herdados
- <u>Subclasse</u> herda de uma <u>superclasse</u>
  - Superclasse direta - subclasse herda explicitamente
  - Superclasse indireta - subclasse herda de dois ou mais níveis acima na hierarquia de classes

---

- Exemplo de herança
  - Um retângulo é um quadrilátero
    - Retângulo é um <u>tipo específico</u> de quadrilátero
    - Quadrilátero é a superclasse, retângulo é a subclasse
    - <u>Não se pode dizer</u> que um quadrilátero é um retângulo
  - Nomes podem ser confusos porque a subclasse tem mais características que a superclasse
    - Subclasse é mais específica que a superclasse
    - Toda subclasse **é um** objeto da sua superclasse, <u>mas não vice-versa</u>
  - Formam estruturas hierárquicas (árvores)
    - Ex: criar uma hierarquia para a classe **`Forma`**




- Utiliza-se a palavra-chave **`extends`:**

```
class FormaBidimensional extends Forma
{
  ...
}
```

  - Obs: membros **`private`** da superclasse não estão acessíveis diretamente na subclasse!
  - Então, como acessá-los ???

- Em uma superclasse:
  - **`public`**
    - Acessível em qualquer classe
  - **`private`**
    - Accesível somente nos métodos da própria superclasse
  - **<u>`protected`</u>**
    - Proteção intermediária entre **`private`** and **`public`**
    - Somente acessível pelos métodos da superclasse ou de uma subclasse desta
- Métodos na subclasse
  - Podem se referir a membros **`public`** ou **`protected`** pelo nome

- Sobrescrevendo métodos
  - Subclasse pode redefinir métodos da superclasse
    - Quando um método é referenciado em uma subclasse, a versão escrita para a subclasse é utilizada
    - É possível acessar o método original da superclasse: `super.nomeDoMetodo(...)`
  - Para invocar o construtor da superclasse explicitamente:

    ```
    super(); // pode-se passar
      parâmetros se necessário
    ```

    - **Se for chamado dessa forma, tem que ser o primeiro comando no construtor da subclasse!**

---

- Exemplo prático: os professores de uma universidade dividem-se em 2 categorias
  - professores de regime integral
  - professores horistas
- Professores de regime possuem um salário fixo para 40 horas de atividade semanais
- Professores horistas recebem um valor estipulado por hora.
- O cadastro de professores desta universidade armazena o nome, idade, matrícula e informação de salário
- Pode ser resolvido através de uma modelagem de classes como segue:

**Classe ProfRegime:**

String nome, matricula
int idade
float salario

ProfRegime(String, String, int, float)
String retornaNome()
String retornaMatricula()
int retornaIdade()
float retornaSalario()

**Classe ProfHorista:**

String nome, matricula
int idade, int total_horas
float salario_hora

ProfHorista(String, String, int, int, float)
String retornaNome()
String retornaMatricula()
int retornaIdade()
float retornaSalario()
int retornaHoras()

- As classes têm alguns atributos e métodos iguais.
- O que acontece se precisarmos alterar a representação de algum atributo, como por exemplo, o número de matrícula para inteiros ao invés de uma string ?
- Será necessário alterar os construtores e os métodos retornaMatricula() nas duas classes, o que é ruim para a programação
- Motivo: **código redundante**
- Como resolver ? **Herança!**

---

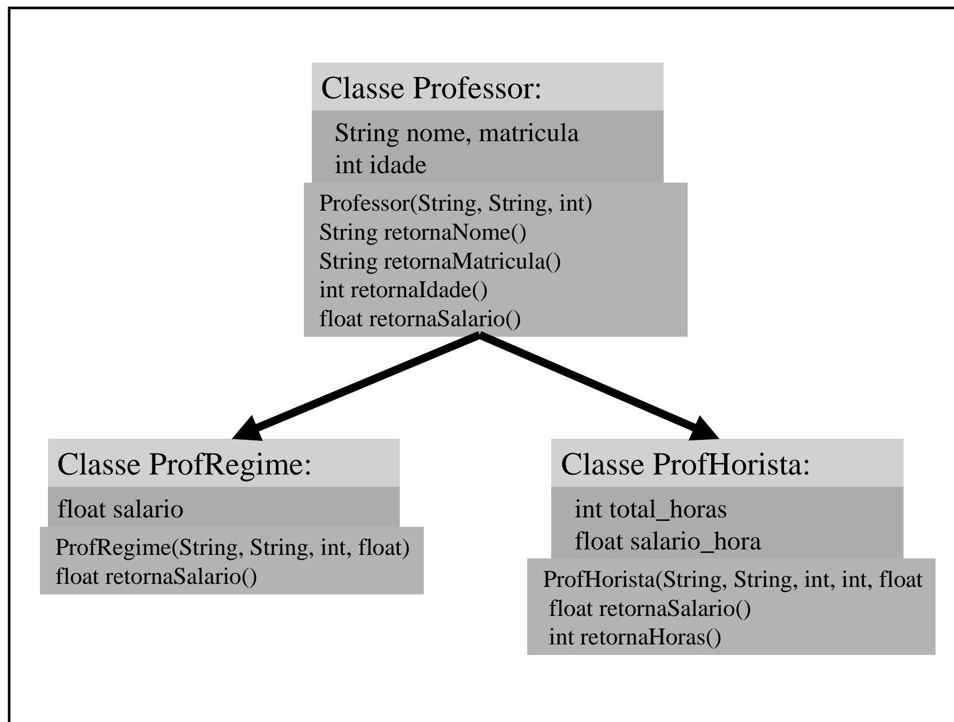
- **“Herança é a capacidade de reutilizar código pela especialização de soluções genéricas já existentes”**
- Neste caso, cria-se uma classe *Professor*, que contém os atributos e métodos comuns aos dois tipos de professor:

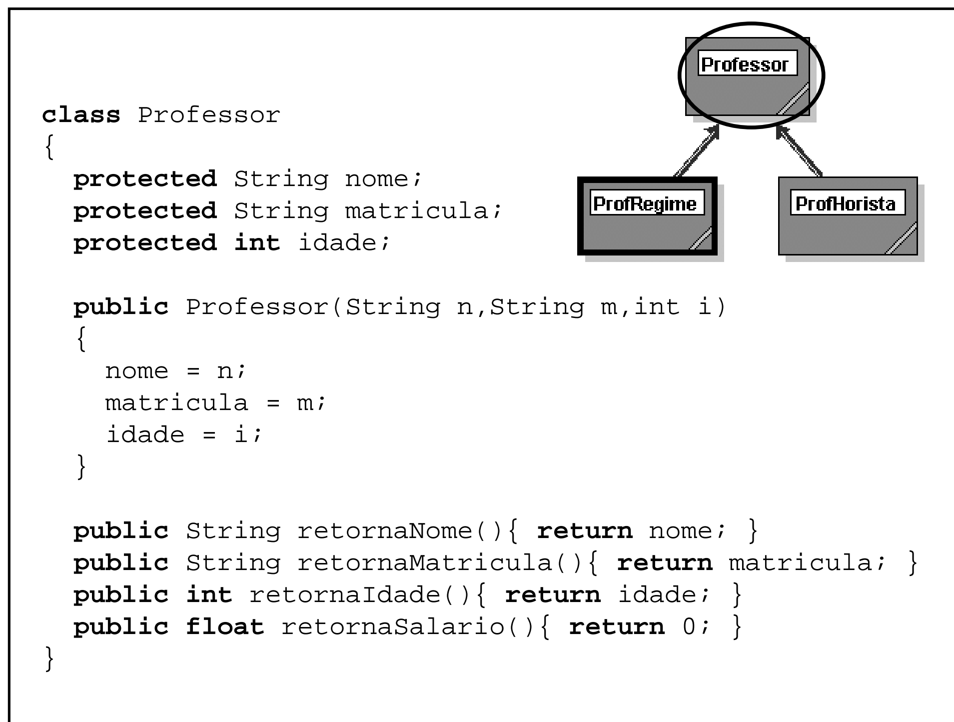
**Classe Professor:**

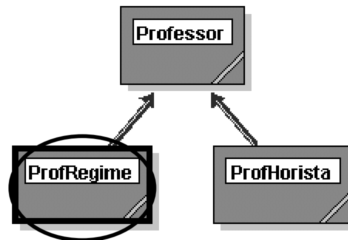
String nome, matricula
int idade

Professor(String, String, int)
String retornaNome()
String retornaMatricula()
int retornaIdade()
float retornaSalario()

- A partir dela, cria-se duas novas classes, que representarão os professores horistas e de regime.
- Para isso, essas classes deverão “**herdar**” os atributos e métodos declarados pela classe “pai”, Professor.


Classe Professor:
String nome, matricula
int idade
Professor(String, String, int)
String retornaNome()
String retornaMatricula()
int retornaIdade()
float retornaSalario()
Classe ProfRegime:
float salario
ProfRegime(String, String, int, float)
float retornaSalario()
Classe ProfHorista:
int total_horas
float salario_hora
ProfHorista(String, String, int, int, float
float retornaSalario()
int retornaHoras()


Professor
ProfRegime
ProfHorista
class Professor
{
protected String nome;
protected String matricula;
protected int idade;
public Professor(String n,String m,int i)
{
nome = n;
matricula = m;
idade = i;
}
public String retornaNome(){ return nome; }
public String retornaMatricula(){ return matricula; }
public int retornaIdade(){ return idade; }
public float retornaSalario(){ return 0; }
}

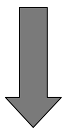



```
class ProfRegime extends Professor
{
  protected float salario;

  public ProfRegime(String n,String m,int i,float vs)
  {
    super(n,m,i);
    if (vs > 0) salario = vs;
  }

  public float retornaSalario()
  {
   return salario - (salario * 0.16);
  }
}
```

Lembrando que **extends** significa que a classe *ProfRegime* deve **herdar** os métodos e atributos da classe *Professor*

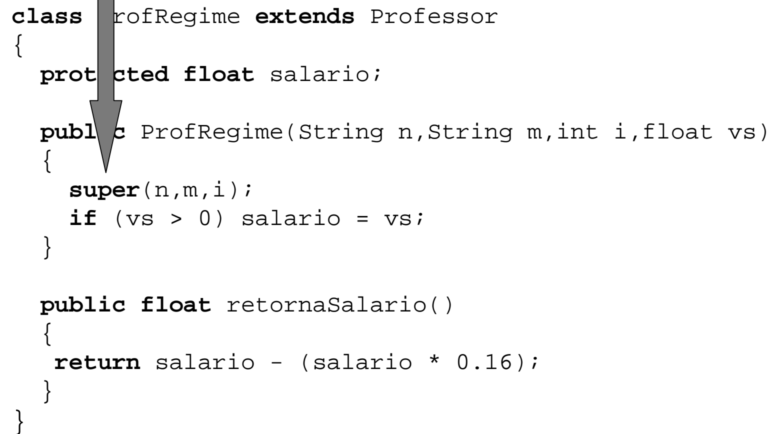



```
class ProfRegime extends Professor
{
  protected float salario;

  public ProfRegime(String n,String m,int i,float vs)
  {
    super(n,m,i);
    if (vs > 0) salario = vs;
  }

  public float retornaSalario()
  {
   return salario - (salario * 0.16);
  }
}
```

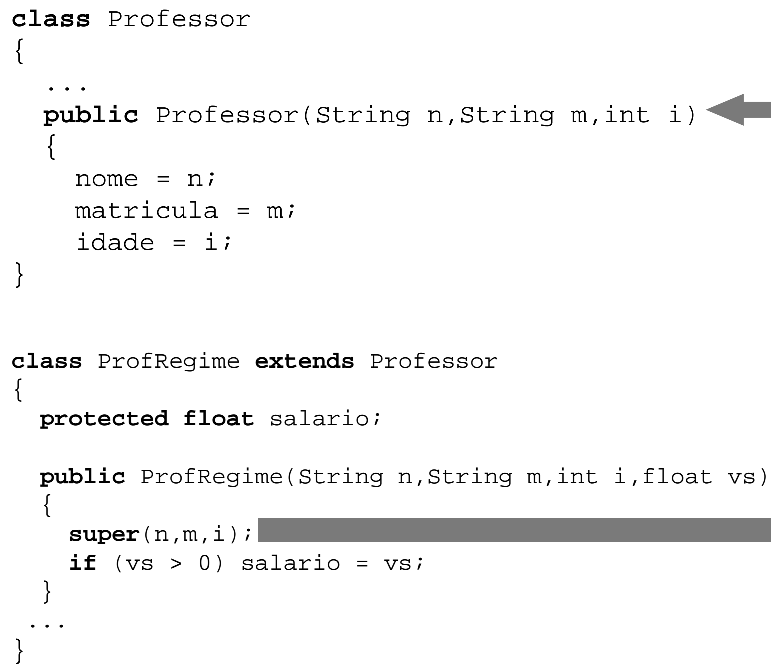
A construção **super(...)** é a chamada para o **método construtor** da classe pai, a *superclasse* de *ProfRegime*



```
class ProfRegime extends Professor
{
  protected float salario;

  public ProfRegime(String n,String m,int i,float vs)
  {
    super(n,m,i);
    if (vs > 0) salario = vs;
  }

  public float retornaSalario()
  {
   return salario - (salario * 0.16);
  }
}
```



```
class Professor
{
   ...
   public Professor(String n,String m,int i)
   {
     nome = n;
     matricula = m;
     idade = i;
}


class ProfRegime extends Professor
{
  protected float salario;

  public ProfRegime(String n,String m,int i,float vs)
  {
    super(n,m,i);
    if (vs > 0) salario = vs;
  }
 ...
}
```

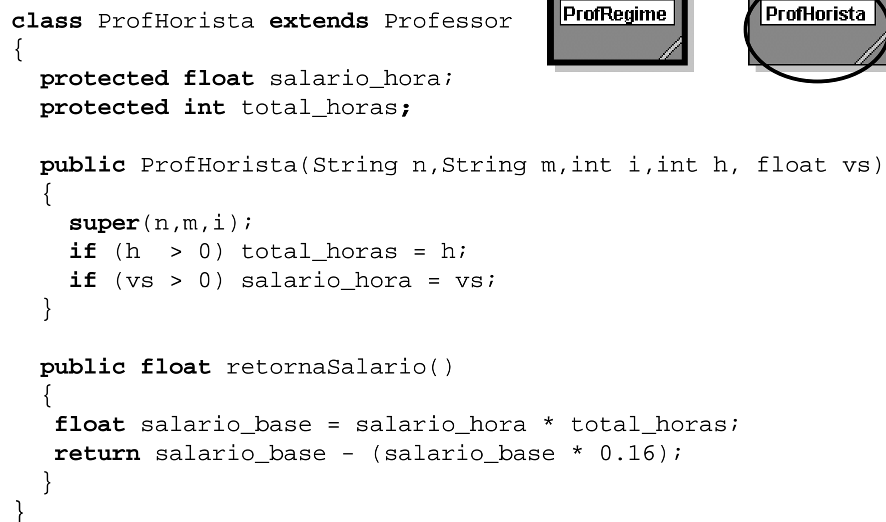
```java
class Professor
{
   ...
   public Professor(String n,String m,int i)
   {
     nome = n;
     matricula = m;
     idade = i;
}
```

executado *antes*

```java
class ProfRegime extends Professor
{
  protected float salario;

  public ProfRegime(String n,String m,int i,float vs)
  {
    super(n,m,i);
    if (vs > 0) salario = vs;
  }
 ...
}
```

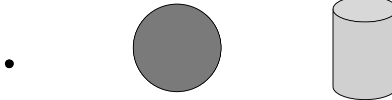
executado *depois*




```java
class ProfHorista extends Professor
{
  protected float salario_hora;
  protected int total_horas;

  public ProfHorista(String n,String m,int i,int h, float vs)
  {
    super(n,m,i);
    if (h  > 0) total_horas = h;
    if (vs > 0) salario_hora = vs;
  }

  public float retornaSalario()
  {
   float salario_base = salario_hora * total_horas;
   return salario_base - (salario_base * 0.16);
  }
}
```



- Outro exemplo: pontos, círculos, cilindros
  - Classe **`Ponto`**
    - variáveis **`protected x, y`**
    - métodos: **`setPonto, getX`**, **`getY`**, sobrescreve **`toString`**
  - Classe **`Circulo`** (**`extends Ponto`**)
    - variável **`protected raio`**
    - métodos: **`setRaio`**, **`getRaio`**, **`area`**, sobrescreve **`toString`**
  - Classe **`Cilindro`** (**`extends Circulo`**)
    - variável **`protected altura`**
    - métodos: **`setAltura`**, **`getAltura`**, **`area`** (superfície), **`volume`**, sobrescreve **`toString`**

## Classe Ponto

```
class Ponto
{
  protected float x,y;

  public Ponto()
  {
    x = y = 0;
  }

  public Ponto(float x,float y)
  {
    setPonto(x,y);
  }

  public void setPonto(float x,float y)
  {
    this.x = x;
    this.y = y;
  }
```

A palavra-chave **this** serve para referenciar o próprio objeto

## Classe Ponto (cont.)

```
  public float getX() { return x; }

  public float getY() { return y; }

  // O método toString() retorna uma
  // representação textual de um
  // objeto
  public String toString()
  {
    return "[" + x + ", " + y + "]";

  }
}
```

- O método **`toString`**`()` está sendo sobrescrito, pois existe na superclasse (**`Object`**)

## Classe Circulo

```
class Circulo extends Ponto
{
  protected float raio;

  public Circulo()
  {
    // construtor de Ponto é chamado implicitamente!
    setRaio(0);
  }

  public Circulo(float x,float y,float raio)
  {
    super(x,y); // construtor de Ponto é chamado explicitamente!
    setRaio(raio);
  }

  public void setRaio(float raio)
  {
    if(raio<=0) raio = 0;
    this.raio = raio;
  }
```

## Classe Circulo (cont.)

```
  public float getRaio() { return raio; }

  public float area()
  {
    return Math.PI * raio * raio;
  }

  // O método toString() retorna uma
  // representação textual de um
  // objeto
  public String toString()
  {
    return "Centro= " + super.toString()+
           "Raio= " + raio;
  }
}
```

- Observe o uso de **`super.toString()`** para chamar o método da superclasse **`Ponto`**

## Classe Cilindro

```
class Cilindro extends Circulo
{
  protected float altura;

  public Cilindro()
  {
    // construtor de Circulo é chamado implicitamente!
    setAltura(0);
  }

  public Cilindro(float x,float y,float raio,float alt)
  {
    // construtor de Circulo é chamado explicitamente!
    super(x,y,raio);
    setAltura(alt);
  }

  public void setAltura(float altura)
  {
    if(altura<=0) altura = 0;
    this.altura = altura;
  }
```

## Classe Cilindro (cont.)

```
  public float getAltura() { return altura; }

  public float area()
  {
    return 2 * super.area() + Math.PI * raio * altura;
  }

  public float volume()
  {
    return super.area() * altura;
  }

  public String toString()
  {
    return super.toString()+ " Altura= "+ altura;
  }
}
```

- Observe o uso de **`super.toString()`** para chamar o método da superclasse **`Circulo`**

Exercício:

- Escreva o código para implementar a seguinte hierarquia de classes:

**Atributos** da classe "**Animal**" :
- String nome

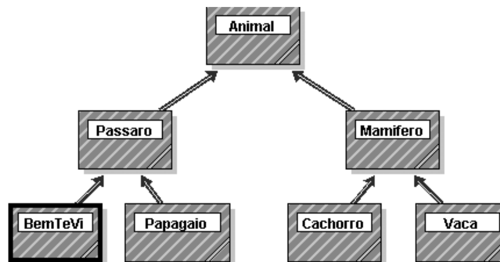
**Métodos** da classe "**Animal**:
- void imp() // imprime o nome e a classe do animal
- String getNome()
- void talk() // "Me not falar"

**Métodos** da classe "**Passaro**":
- void talk() // "piu, piu."

**Métodos** da classe "**BemTeVi**":
- void talk() // "bem-te-vi!"

**Atributos** da classe "**Papagaio**":
- String vocabulario // frase

**Métodos** da classe "**Papagaio**":
- void talk() // exibe vocabulario
- void setVoc(String v) //altera voc.




A classe mamífero não tem atributos ou métodos.

**Atributos** da classe "**Cachorro**":
- boolean lateMuito

**Métodos** da classe "**Cachorro**":
- void setLateAlto()
- void setLateBaixo()
- void talk() // "AU! AU!" ou "au, au..."

**Métodos** da classe "Vaca";
- void talk() // "Muuu..."