

**INSTITUTO FEDERAL**  
Goiás

Instituto Federal de Goiás  
Câmpus Goiânia

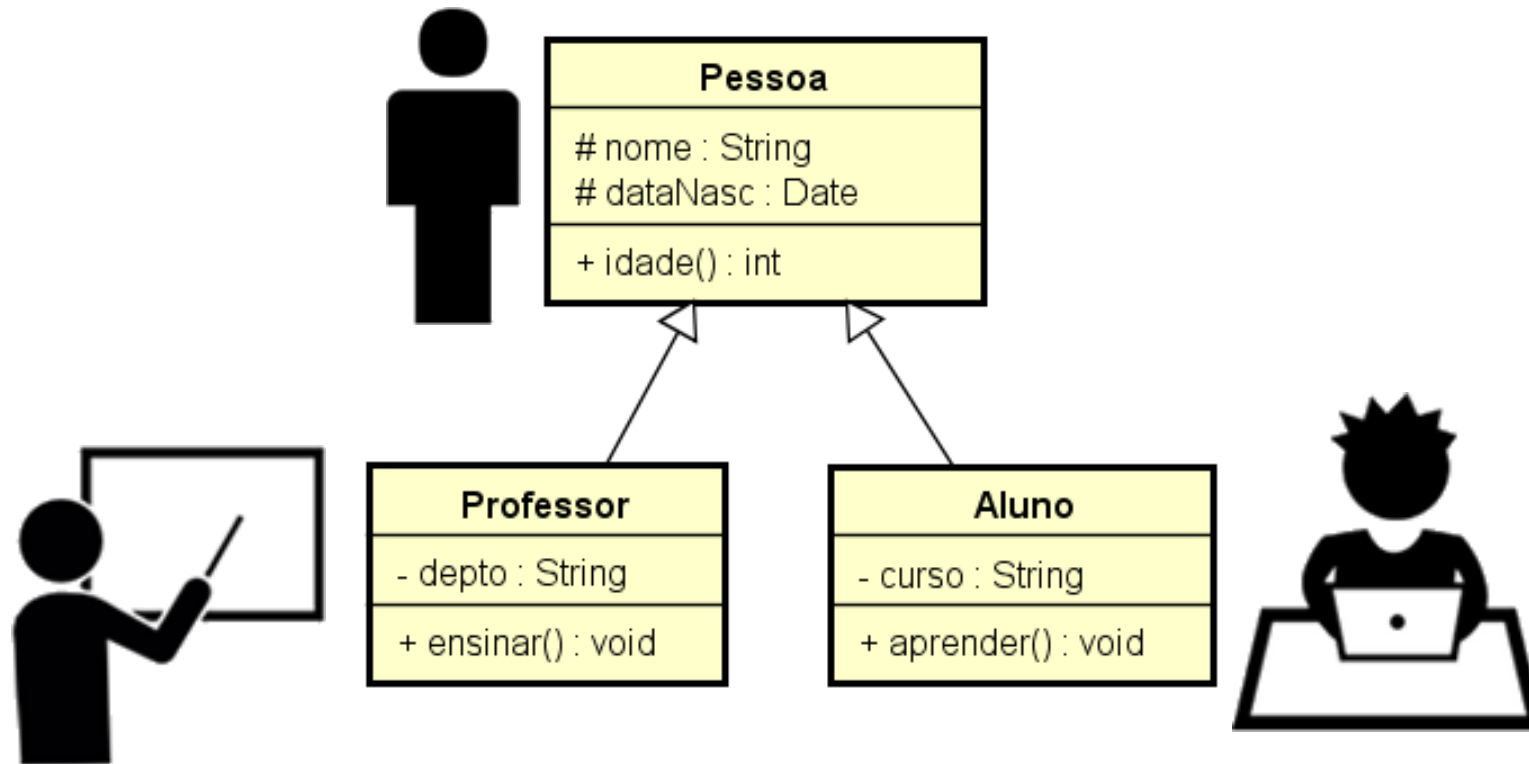
Bacharelado em Sistemas de Informação  
Disciplina: Programação Orientada a Objetos I

# Associação de Classes Herança

Prof. Ms. Renan Rodrigues de Oliveira  
Goiânia - GO

# Herança

A herança é um mecanismo em que a subclasse constitui uma especialização da superclasse. A superclasse pode ser vista como generalização das subclasses.



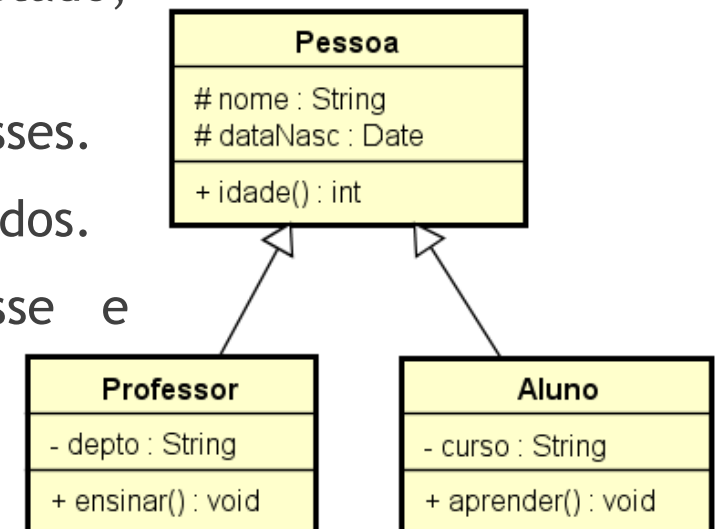
# Herança

Ao criar uma classe, em vez de declarar membros completamente novos, você pode designar que a nova classe deve herdar membros de uma classe existente.



**Aumenta a probabilidade de que um sistema será implementado e mantido eficientemente.**

- ▶ Permite economizar tempo durante o desenvolvimento de um programa baseando novas classes no software existente testado, depurado e de alta qualidade.
- ▶ Cada subclasse pode ser uma superclasse de futuras subclasses.
- ▶ Uma subclasse pode adicionar seus próprios campos e métodos.
- ▶ Uma subclasse é mais específica que sua superclasse e representa um grupo mais especializado de objetos.



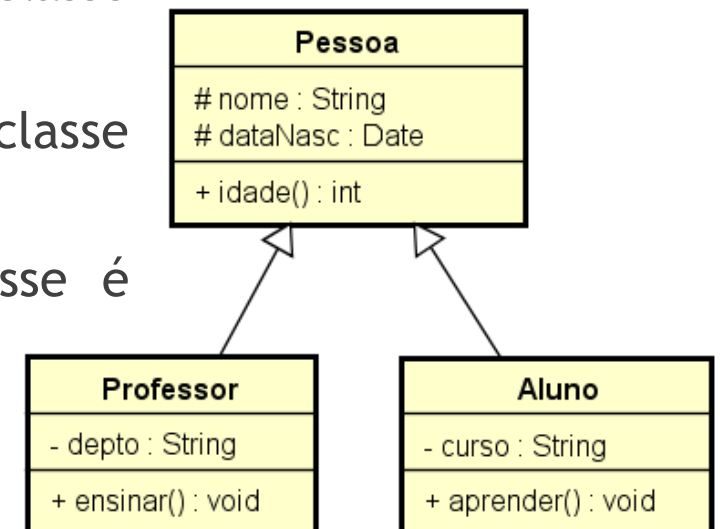
# Herança

A subclasse expõe os comportamentos da sua superclasse e pode adicionar comportamentos que são específicos à subclasse. É por isso que a herança é às vezes conhecida como especialização.



A hierarquia de classes inicia com a classe `Object` (no pacote `java.lang`). Toda classe em Java estende (ou herda de) `Object` direta ou indiretamente.

- ▶ A superclasse direta é a superclasse a partir da qual a subclasse herda explicitamente.
- ▶ Uma superclasse indireta é qualquer classe acima da superclasse direta na hierarquia de classes.
- ▶ O Java só suporta herança simples, na qual cada classe é derivada de exatamente uma superclasse direta.



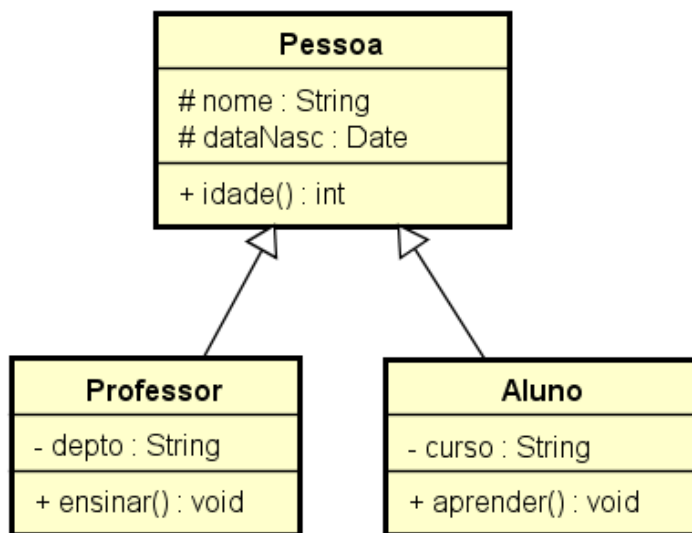
# Herança



Distinção entre o relacionamento é um e o relacionamento tem um:

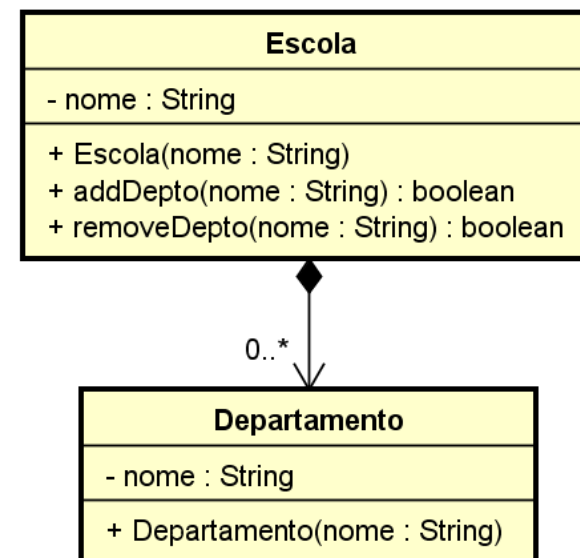
É um representa a herança.

Em um relacionamento É-Um, um objeto de uma subclasse também pode ser tratado como um objeto de sua superclasse.



Tem um representa composição.

Em um relacionamento Tem-Um, um objeto contém como membros referências a outros objetos.



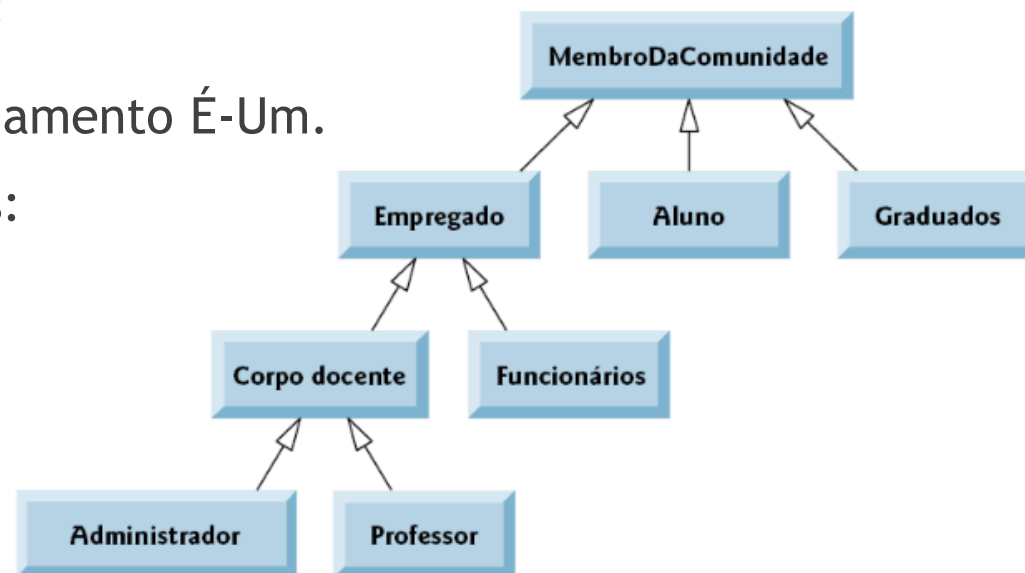
# Relacionamento É-Um e Tem-Um



**Exemplo:**

**Hierarquia de Herança na comunidade universitária:**

- ▶ Cada seta na hierarquia representa um relacionamento É-Um.
- ▶ Siga as setas para cima na hierarquia de classes:
  - ▶ “um Funcionário é um MembroDaComunidade”
  - ▶ “um Professor é um membro do CorpoDocente”.



**MembroDaComunidade é a superclasse direta de Funcionários, AlunoDeGraduação e AlunoDePósGraduação e é uma superclasse indireta de todas as outras classes no diagrama.**

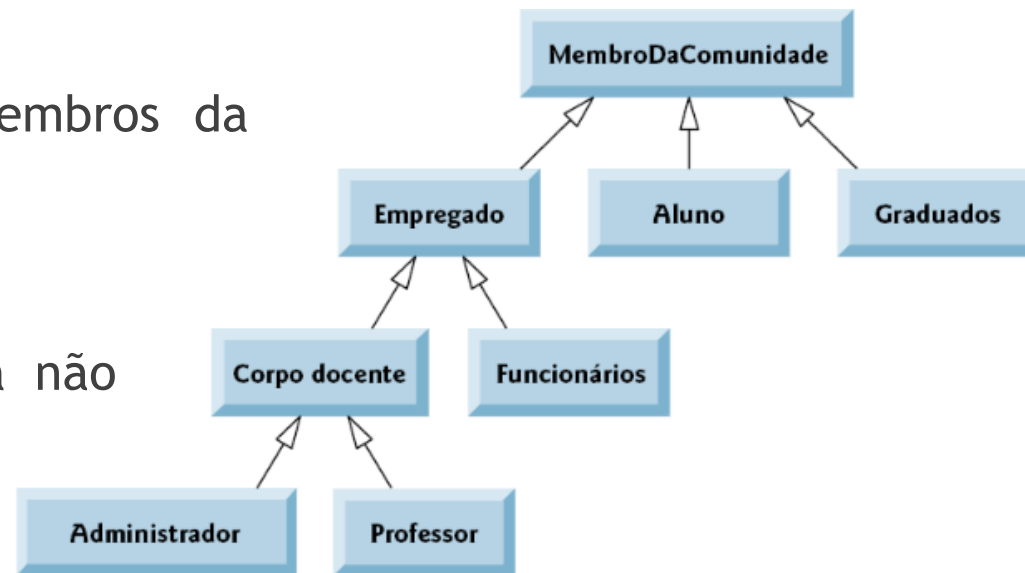
# Relacionamento É-Um e Tem-Um

✓ Os objetos de todas as classes que estendem uma superclasse comum podem ser todos tratados como objetos dessa superclasse.

- ▶ Seus aspectos comuns são expressos nos membros da superclasse.

✓ Problemas de herança.

- ▶ Uma subclasse pode herdar métodos que ela não necessita ou que não deveria ter.
- ▶ Mesmo quando um método de superclasse é adequado a uma subclasse, essa subclasse precisa frequentemente de uma versão personalizada do método.
- ▶ A subclasse pode sobrescrever (redefinir) o método de superclasse com uma implementação apropriada.



# Membros Protected



**O Java adota as seguintes estratégias na herança:**

- ▶ Só há herança simples de classes.
- ▶ Os membros public de uma classe são acessíveis onde quer que o programa tenha uma referência a um objeto dessa classe ou uma de suas subclasses.
- ▶ Os membros private de uma classe são acessíveis apenas dentro da própria classe.



**Membros Protected:**

- ▶ Os atributos private não são visíveis na subclasse, os atributos protected e public mantem a mesma visibilidade.
- ▶ Os métodos de subclasse podem referenciar membros public e protected herdados da superclasse simplesmente utilizando os nomes de membro.

**Os membros private de uma superclasse permanecem ocultos em suas subclasses.**  
Eles somente podem ser acessados pelos métodos public ou protected herdados da superclasse.



# Modificadores de Acesso

Os modificadores de acesso são utilizados para garantir o encapsulamento. Podem ser aplicados tanto a classes quanto a seus membros (atributos e métodos).

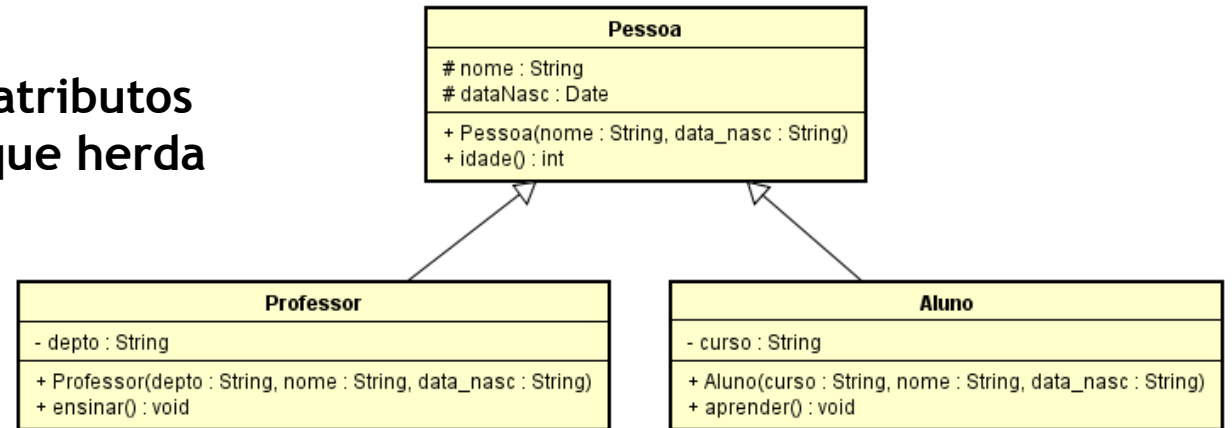


## O Java possui os seguintes modificadores de Acesso:

- ▶ **public:** é visível em qualquer lugar;
- ▶ **protected:** Só é visível na mesma classe, em classes do mesmo pacote e em suas subclasses;
- ▶ **package:** Default. Só é visível em classes do mesmo pacote. Em Java não existe modificador com este nome. A ausência de modificador o torna package.
- ▶ **private:** Só é visível dentro da mesma classe.

# Construtores em Subclasses

É da responsabilidade da subclasse inicializar os atributos definidos na sua classe, assim como os atributos que herda das suas superclasses.



## Construtores em Subclasses

- ▶ O construtor da subclasse pode delegar a inicialização dos atributos herdados para a superclasse, chamando, implícita ou explicitamente, o construtor da superclasse.
- ▶ Um construtor da subclasse pode fazer uma chamada explícita dum construtor da superclasse através do `super()`.
- ▶ Se existir, a chamada explícita do construtor da superclasse deve ser a primeira instrução no construtor.

```
public Professor(String nome, String dataNasc, String depto) {
    super(nome, dataNasc);
    this.depto = depto;
}
```

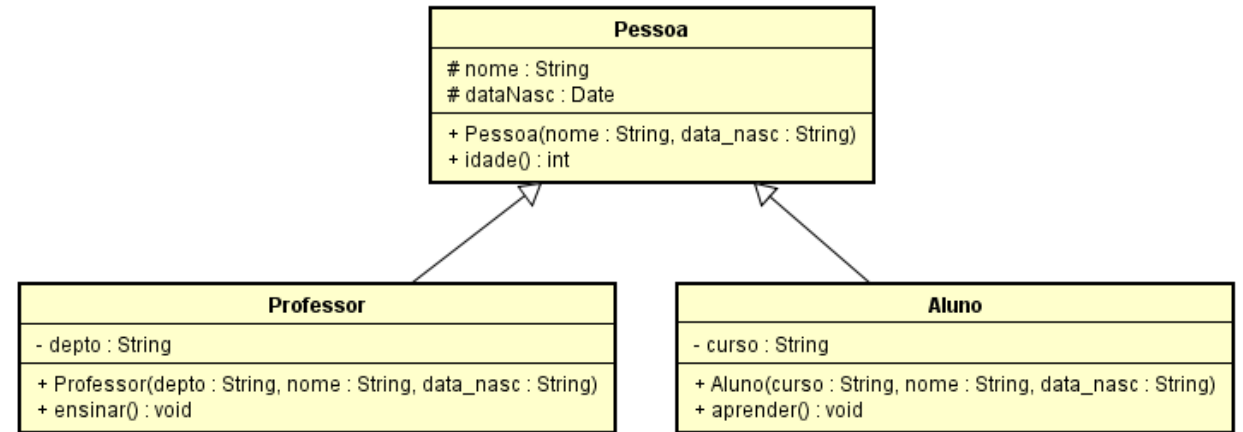
# Construtores em Subclasses

- ▶ Se nenhum construtor da superclasse é chamado, ou se nenhum construtor da classe é chamado, como primeira instrução do construtor, o construtor sem argumentos da superclasse é implicitamente chamado antes de qualquer instrução no construtor.
- ▶ Se a superclasse não tiver um construtor sem argumentos, é necessário chamar explicitamente um construtor da superclasse.
  - ▶ Note que um construtor implícito é automaticamente criado se não existir mais nenhum construtor, mas apenas neste caso.
- ▶ Pode se chamar outro construtor da classe usando o `this(parametros)`.

# Herança

## Implementando a Classe Pessoa

```
1 package heranca;
2
3 import java.text.ParseException;
4
5 public class Pessoa {
6
7     protected String nome;
8     protected Date dataNasc;
9
10    public Pessoa(String nome, String dataNasc) throws ParseException {
11        SimpleDateFormat format = new SimpleDateFormat("dd/MM/yyyy");
12
13        this.nome = nome;
14        this.dataNasc = (Date)format.parse(dataNasc);
15    }
16
17    public int idade() {
18        Calendar c1 = Calendar.getInstance();
19        Calendar c2 = Calendar.getInstance();
20
21        c2.setTime(dataNasc);
22
23        return c1.get(Calendar.YEAR) - c2.get(Calendar.YEAR);
24    }
25 }
```

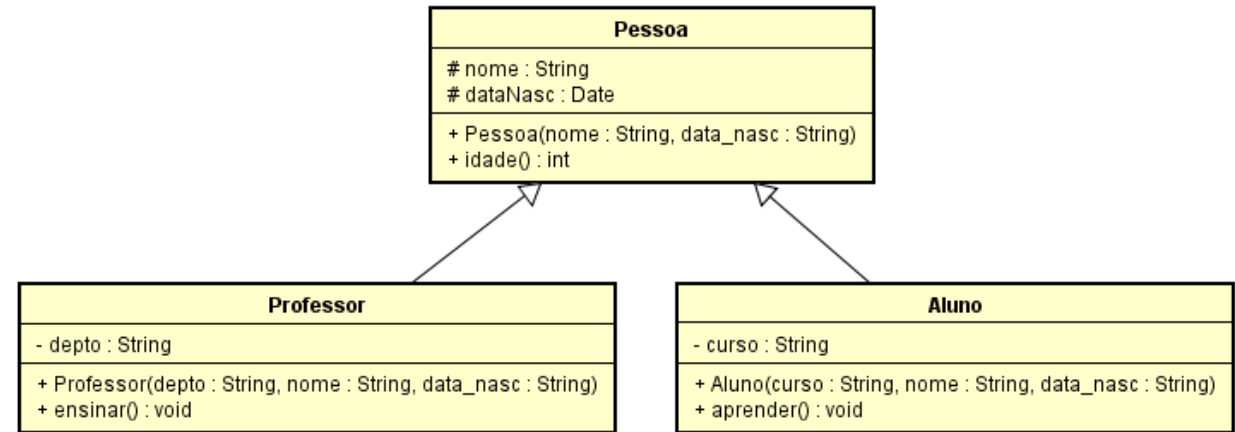


CONTINUA >>

# Herança

## Implementando a Classe Pessoa

```
29 public String getNome() {  
30     return nome;  
31 }  
32  
33 public void setNome(String nome) {  
34     this.nome = nome;  
35 }  
36  
37 public Date getDataNasc() {  
38     return dataNasc;  
39 }
```



```
41 public void setDataNasc(Date dataNasc) {  
42     this.dataNasc = dataNasc;  
43 }  
44  
45 @Override  
46 public String toString() {  
47     return "Pessoa [nome=" + nome + ", dataNasc=" + dataNasc + " ]";  
48 }  
49 }
```

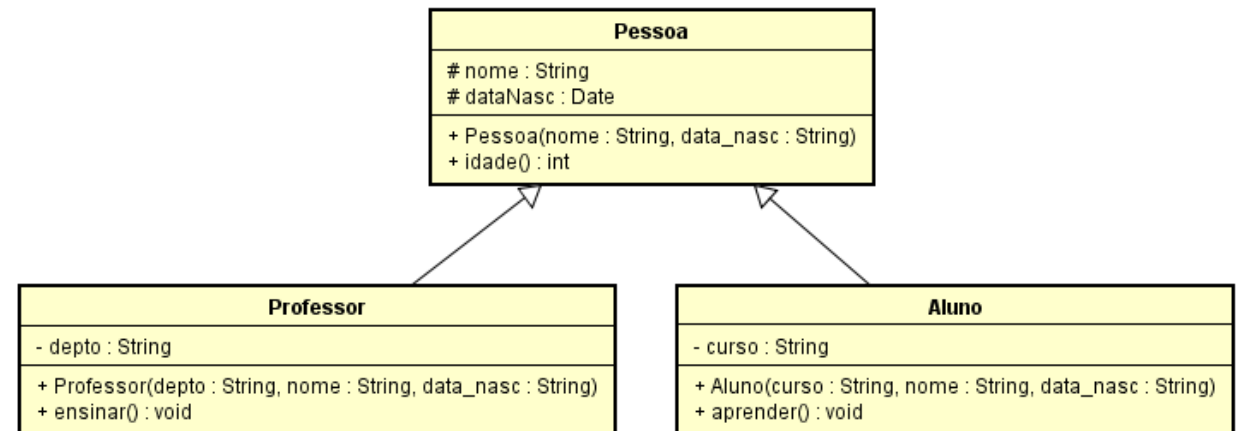
# Herança

## Implementando a Classe Pessoa

```
1 package heranca;
2
3 import java.text.ParseException;
4
5 public class Professor extends Pessoa {
6
7     private String depto;
8
9     public Professor(String nome, String dataNasc, String depto) throws ParseException {
10         super(nome, dataNasc);
11         this.depto = depto;
12     }
13
14     public void ensinar() {
15         System.out.println("O professor está ensinando!");
16     }
```

```
26 @Override
27 public String toString() {
28     return "Professor [depto=" + depto + ", nome=" + nome + ", dataNasc="
29         + dataNasc + "]";
30 }
31 }
```

```
18 public String getDepto() {
19     return depto;
20 }
21
22 public void setDepto(String depto) {
23     this.depto = depto;
24 }
```



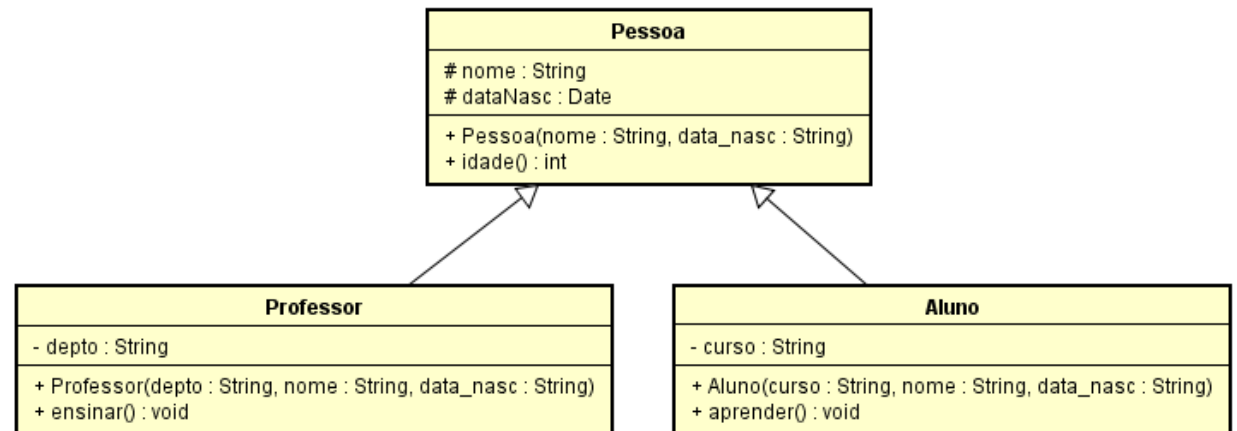
# Herança

## Implementando a Classe Pessoa

```
1 package heranca;
2
3 import java.text.ParseException;
4
5 public class Aluno extends Pessoa {
6
7     private String curso;
8
9     public Aluno(String nome, String dataNasc, String curso) throws ParseException {
10         super(nome, dataNasc);
11         this.curso = curso;
12     }
13
14     public void aprender() {
15         System.out.println("O aluno está aprendendo!");
16     }
```

```
26 @Override
27 public String toString() {
28     return "Aluno [curso=" + curso + ", nome=" + nome + ", dataNasc="
29         + dataNasc + "]";
30 }
31 }
```

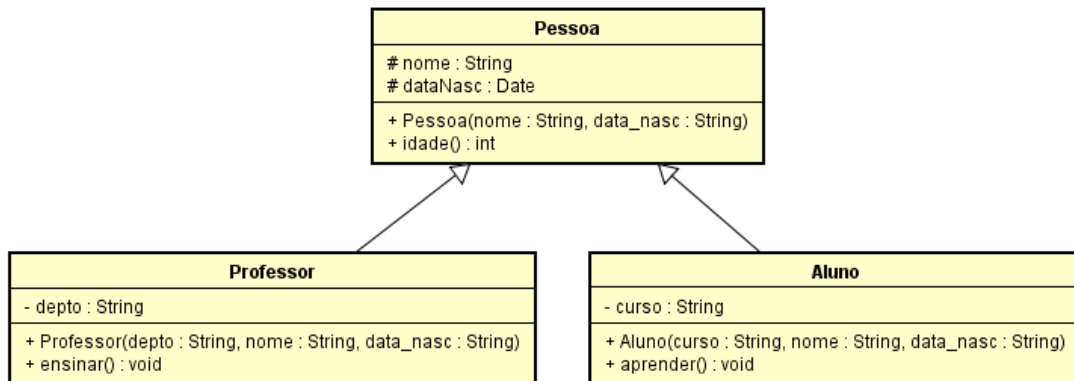
```
18 public String getCurso() {
19     return curso;
20 }
21
22 public void setNome(String curso) {
23     this.curso = curso;
24 }
```



# Herança

## Implementando a Classe Pessoa

```
1 package heranca;
2
3 import java.text.ParseException;
4
5 public class TestePessoa {
6
7     public static void main(String[] args) throws ParseException {
8
9         Pessoa pessoa = new Pessoa("Thiago", "15/09/1975");
10        System.out.println(pessoa);
11
12        Professor professor = new Professor("Renan", "27/04/1984", "Depto IV");
13        System.out.println(professor);
```



```
15 Aluno aluno = new Aluno("Carla", "05/08/1997", "Sistemas de Informação");
16 System.out.println(aluno);
17
18 System.out.println(
19     "Idade das Pessoas: " + "\n" +
20     pessoa.getNome() + "(" + pessoa.idade() + ") " +
21     professor.getNome() + "(" + professor.idade() + ") " +
22     aluno.getNome() + "(" + aluno.idade() + ")"
23 );
24
25 professor.ensinar();
26 aluno.aprender();
27 }
28 }
```

### Saída do Programa

```
Pessoa [nome=Thiago, dataNasc=Mon Sep 15 00:00:00 BRT 1975]
Professor [depto=Depto IV, nome=Renan, dataNasc=Fri Apr 27 00:00:00 BRT 1984]
Aluno [curso=Sistemas de Informação, nome=Carla, dataNasc=Tue Aug 05 00:00:00 BRT 1997]
Idade das Pessoas:
Thiago(40) Renan(31) Carla(18)
O professor está ensinando!
O aluno está aprendendo!
```



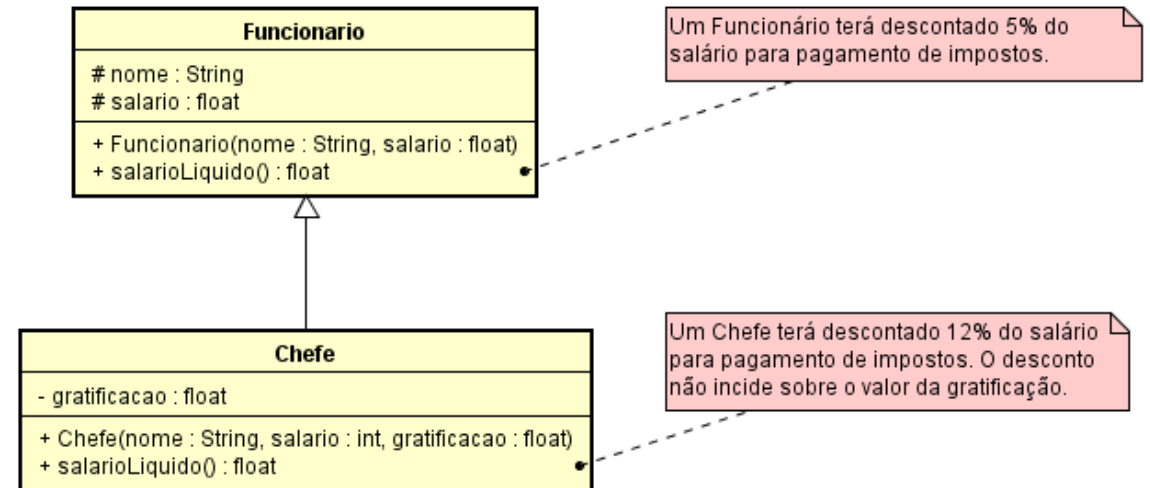
# Sobrescrita de Métodos

Sempre que herdamos um método da superclasse, podemos sobrescrevê-lo. Fazendo isto temos a oportunidade de determinar um comportamento específico do método para a subclasse.



A sobrescrita (Overriding) está diretamente relacionada à orientação a objetos, mais especificamente com a herança.

- Com a sobrescrita, conseguimos especializar os métodos herdados das superclasses, alterando o seu comportamento nas subclasses por um mais específico.

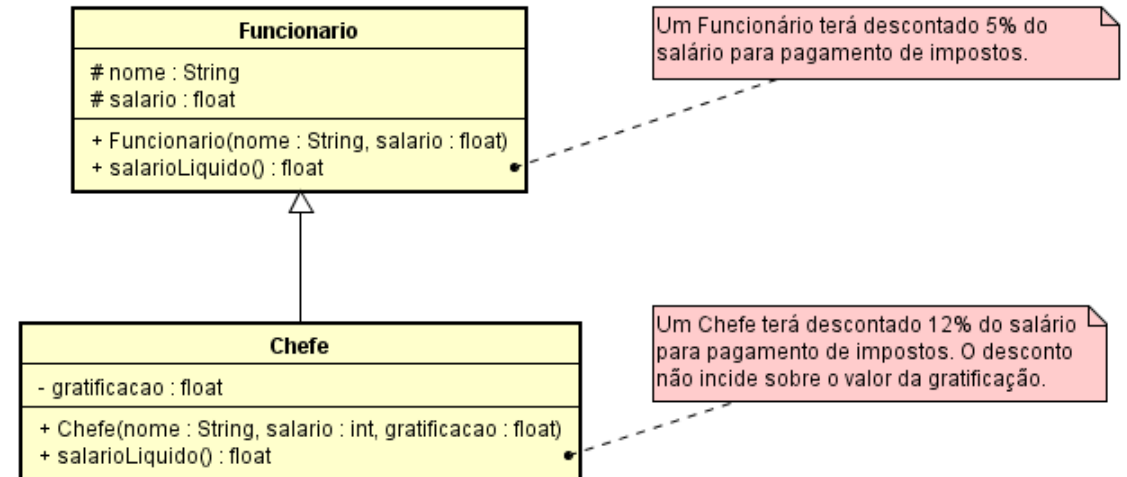


# Sobrescrita de Métodos

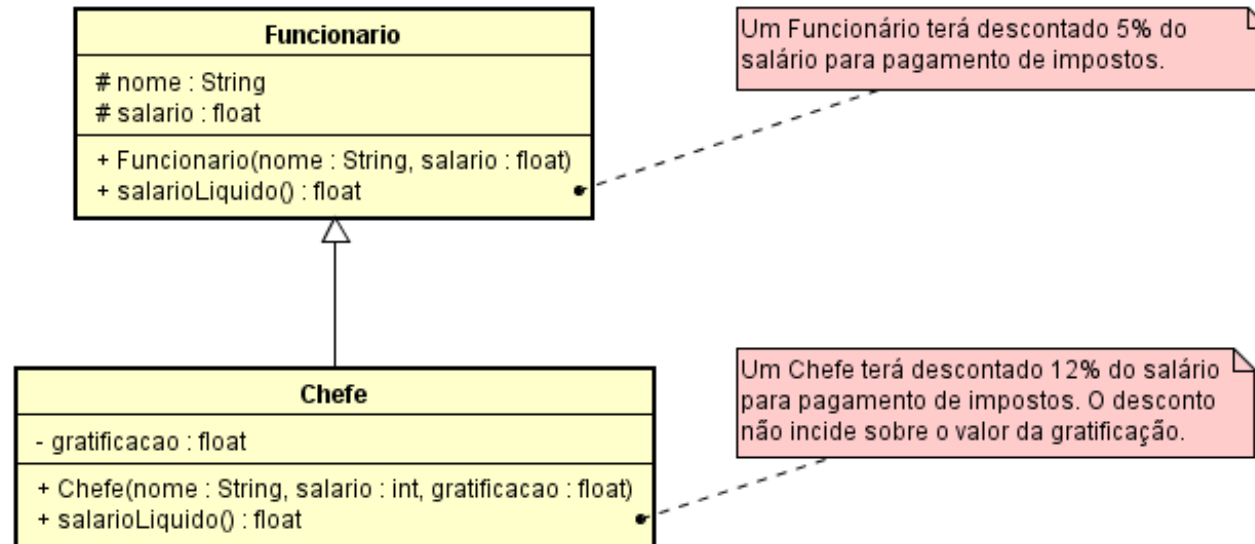


Um método da subclasse é considerado uma redefinição de um método da superclasse se:

- ▶ Ambos têm o mesmo identificador e parâmetros (número e tipo).
- ▶ O tipo de retorno é covariante:
  - ▶ Se o tipo de retorno é uma referência, então o método redefinido pode declarar como tipo de retorno um subtipo do tipo de retorno do método da superclasse.
  - ▶ Se o tipo de retorno é um tipo primitivo, então o tipo de retorno do método redefinido tem que ser idêntico ao tipo de retorno do método da superclasse.



# Sobrescrita de Métodos



```
3 public class Funcionario {
4
5     public float salarioLiquido() {
6         return salario * 0.95f;
7     }
}

3 public class Chefe extends Funcionario {
4
5     public float salarioLiquido() {
6         return salario * 0.88f + gratificacao;
7     }
}
```

# Métodos Sobrescritos



**Um método só pode ser redefinido na subclasse se for visível da superclasse para a subclasse.**

- ▶ Se um método definido na superclasse não é visível na subclasse então não é herdado.
- ▶ Se um método não é herdado, mesmo que um método com o mesmo identificador, mesmos parâmetros (número e tipo) e retorno covariante seja definido na subclasse, este método não é uma redefinição do método na superclasse.



**A visibilidade dos métodos redefinidos pode ser diferente dos métodos da superclasse, mas apenas para dar mais acesso.**

- ▶ Por exemplo, um método declarado na superclasse como `protected` pode ser redefinido `protected` ou `public`, mas não `private` ou com visibilidade de pacote.

# Métodos Sobrescritos

## Implementando a Classe Funcionário

```
1 package br.com.renanrodrigues.heranca;
```

```
2  
3 public class Funcionario {
```

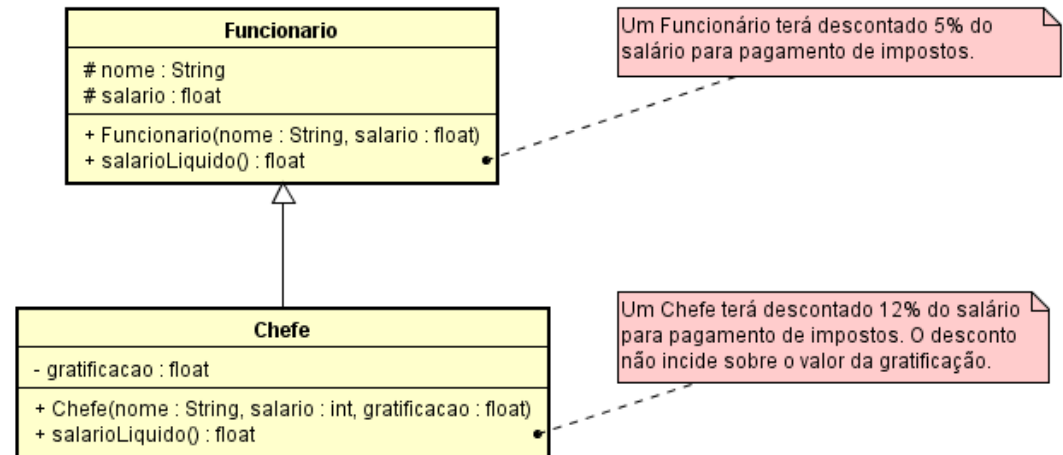
```
4  
5     protected String nome;  
6     protected float salario;
```

```
7  
8     public Funcionario(String nome, float salario) {  
9         this.nome = nome;  
10        this.salario = salario;  
11    }  
12
```

```
13    public float salarioLiquido() {  
14        return salario * 0.95f;  
15    }
```

```
17    public String getNome() {  
18        return nome;  
19    }  
20  
21    public void setNome(String nome) {  
22        this.nome = nome;  
23    }
```

```
25    public float getSalario() {  
26        return salario;  
27    }  
28  
29    public void setSalario(float salario) {  
30        this.salario = salario;  
31    }
```

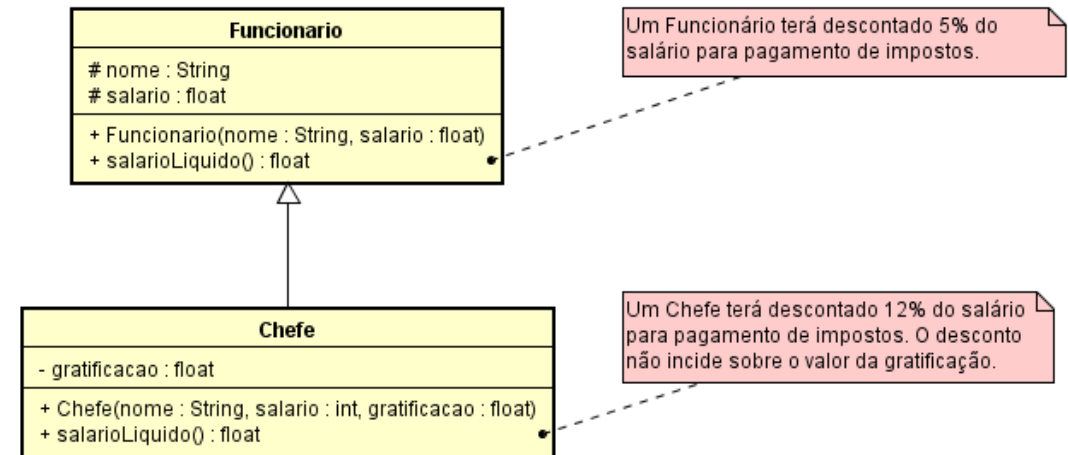


```
33    @Override  
34    public String toString() {  
35        return "Funcionario [nome=" + nome + ", salario=" + salario + "];  
36    }  
37 }
```

# Métodos Sobrescritos

## Implementando a Classe Chefe

```
1 package br.com.renanrodrigues.heranca;
2
3 public class Chefe extends Funcionario {
4
5     private float gratificacao;
6
7     public Chefe(String nome, float salario, float gratificacao) {
8         super(nome, salario);
9         this.gratificacao = gratificacao;
10    }
11
12    public float salarioLiquido() {
13        return salario * 0.88f + gratificacao;
14    }
15
16    public float getGratificacao() {
17        return gratificacao;
18    }
19
20    public void setGratificacao(float gratificacao) {
21        this.gratificacao = gratificacao;
22    }
23
24    @Override
25    public String toString() {
26        return "Chefe [gratificacao=" + gratificacao + ", nome=" + nome + ", salario=" + salario +
27    }
28 }
```



# Métodos Sobrescritos

## Principal

```
1 package br.com.renanrodrigues.heranca;
2
3 public class TesteFuncionario {
4
5     public static void main(String[] args) {
6
7         Funcionario f = new Funcionario("Pedro", 2000);
8         Chefe c = new Chefe("Marcos", 5000, 3000);
9
10        System.out.println(f);
11        System.out.println("Salário Líquido do " + f.getNome() +
12            " = " + f.salarioLiquido());
13
14        System.out.println(c);
15        System.out.println("Salário Líquido do " + c.getNome() +
16            " = " + c.salarioLiquido());
17    }
18 }
```

### Saída do Programa

```
Funcionario [nome=Pedro, salario=2000.0]
Salário Líquido do Pedro = 1900.0
Chefe [gratificacao=3000.0, nome=Marcos, salario=5000.0]
Salário Líquido do Marcos = 7400.0
```

# Métodos Sobrecarregados

Se um método definido numa subclasse tiver o mesmo identificador, mas diferente número ou tipo de parâmetros, que um método (visível) da superclasse então é uma sobrecarga.



**Para que seja permitida a sobrecarga (Overloading), os nomes dos métodos devem ser iguais mas as assinaturas devem ser diferentes.**

- ▶ A assinatura de um método é composta por seu nome e pelo número e tipos de argumentos que são passados para esse método, independentemente dos nomes das variáveis usadas na declaração do método.
- ▶ O tipo de retorno do método não é considerado parte da assinatura.

Calculadora
+ somar(n1 : int, n2 : int) : int
+ somar(n1 : int, n2 : int, n3 : int) : int
+ somar(n1 : int, n2 : float) : float
+ somar(n1 : float, n2 : float) : float



# Métodos Sobrecarregados

## Implementando a Classe Funcionário

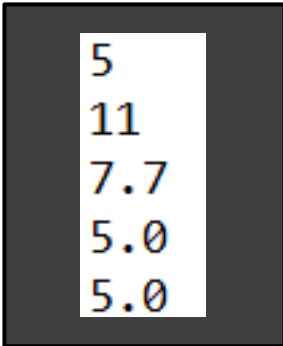
```
1 package br.com.renanrodrigues.calculo;
2
3 public class Calculadora {
4
5     public int somar(int n1, int n2) {
6         return n1 + n2;
7     }
8
9     public int somar(int n1, int n2, int n3) {
10        return n1 + n2 + n3;
11    }
12
13    public float somar(int n1, float n2) {
14        return n1 + n2;
15    }
16
17    public float somar(float n1, float n2) {
18        return n1 + n2;
19    }
20 }
```

Calculadora
+ somar(n1 : int, n2 : int) : int
+ somar(n1 : int, n2 : int, n3 : int) : int
+ somar(n1 : int, n2 : float) : float
+ somar(n1 : float, n2 : float) : float

# Programa Principial

```
1 package br.com.renanrodrigues.calculo;
2
3 public class TesteCalculo {
4
5     public static void main(String[] args) {
6
7         Calculadora c = new Calculadora();
8
9         System.out.println(c.somar(2,3));
10        System.out.println(c.somar(4,2,5));
11        System.out.println(c.somar(2,5.7f));
12        System.out.println(c.somar(2.0f,3));
13        System.out.println(c.somar(2.0f,3.0f));
14    }
15 }
```

Saída do Programa



```
5
11
7.7
5.0
5.0
```

# Herança e Redefinição

## Atributo



**Um atributo da superclasse que é declarado na subclasse com o mesmo nome (independentemente do tipo) é escondido.**

- ▶ Não há redefinição de atributos, estes são sempre escondidos.
- ▶ O atributo da superclasse continua a existir, mas deixa de ser possível à subclasse acessar diretamente.
- ▶ É necessário usar a referência super, ou outra referência para o objeto da superclasse, para acessar o atributo escondido.

# Herança e Redefinição

## Atributo

```
1 package br.com.renanrodrigues.heranca;
2
3 public class Veiculo {
4
5     protected String modelo;
6
7     public Veiculo(String modelo) {
8         this.modelo = modelo;
9     }
10
11     public String getModelo() {
12         return modelo;
13     }
14
15     public void setModelo(String modelo) {
16         this.modelo = modelo;
17     }
18
19     @Override
20     public String toString() {
21         return "Veiculo [modelo=" + modelo + "]";
22     }
23 }
```

```
1 package br.com.renanrodrigues.heranca;
2
3 public class Carro extends Veiculo {
4
5     protected String modelo;
6
7     public Carro(String modelo) {
8         super("XWQ");
9         this.modelo = modelo;
10    }
11
12    public String getModelo() {
13        return modelo;
14    }
15
16    public void setModelo(String modelo) {
17        this.modelo = modelo;
18    }
19
20    @Override
21    public String toString() {
22        return "Carro [modelo=" + modelo + " " + super.modelo + "]";
23    }
24 }
```

```
1 package br.com.renanrodrigues.heranca;
2
3 public class TesteCarro {
4
5     public static void main(String[] args) {
6         Veiculo v = new Veiculo("Gol");
7         Veiculo c = new Carro("Palio");
8
9         System.out.println(v);
10        System.out.println(c);
11    }
12 }
```

Saída do Programa

```
Veiculo [modelo=Gol]
Carro [modelo=Palio XWQ]
```