

**INSTITUTO FEDERAL**  
Goiás

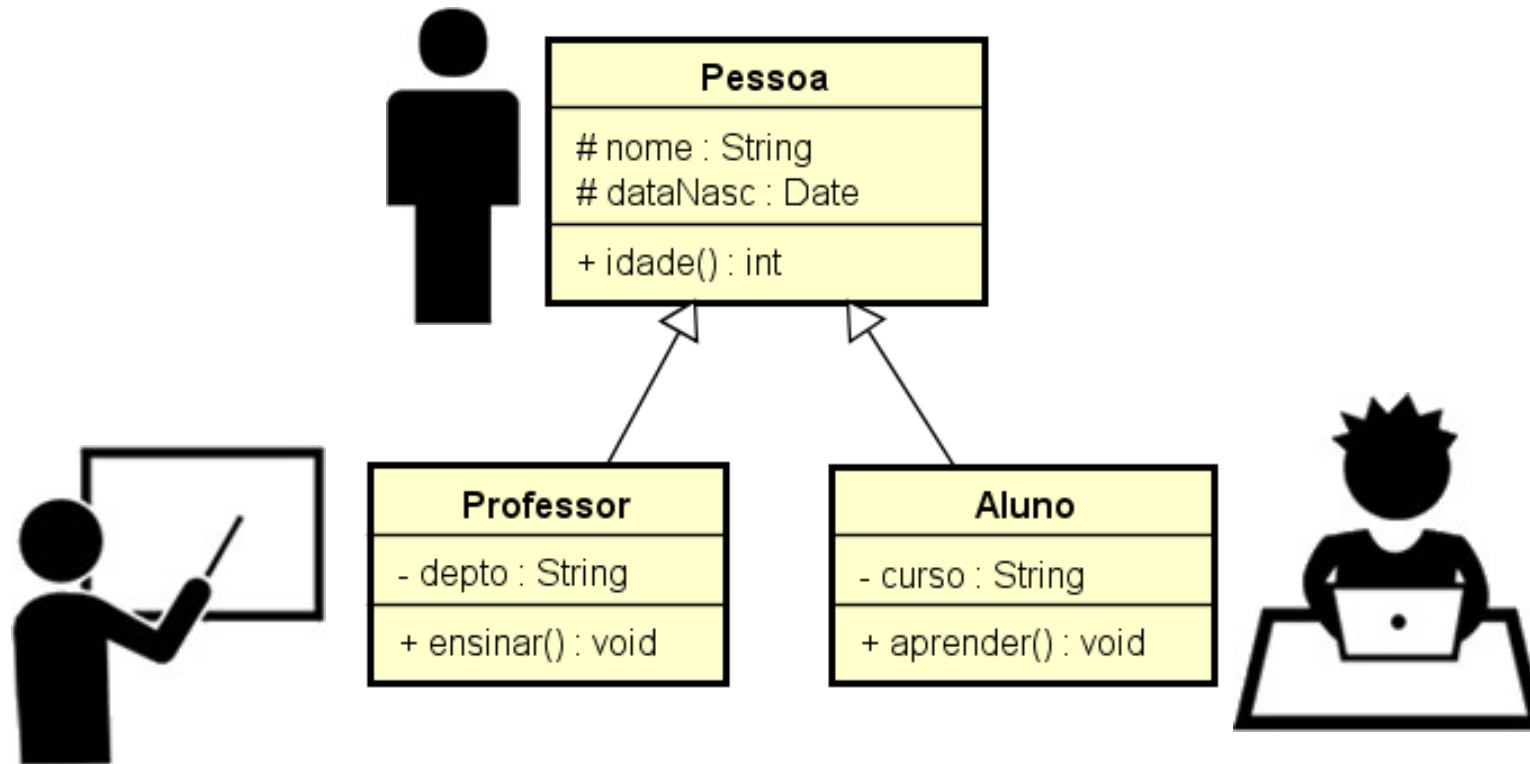
Bacharelado em Sistemas de Informação  
Disciplina: Programação Orientada a Objetos I

# Associação de Classes Herança (Sobrescrita, Sobrecarga)

Prof. Ms. Dory Gonzaga Rodrigues  
Goiânia - GO

# Herança

Em JAVA, podemos criar classes que herdem atributos e métodos de outras classes, evitando rescrita de código. Este tipo de relacionamento é chamado de HERANÇA



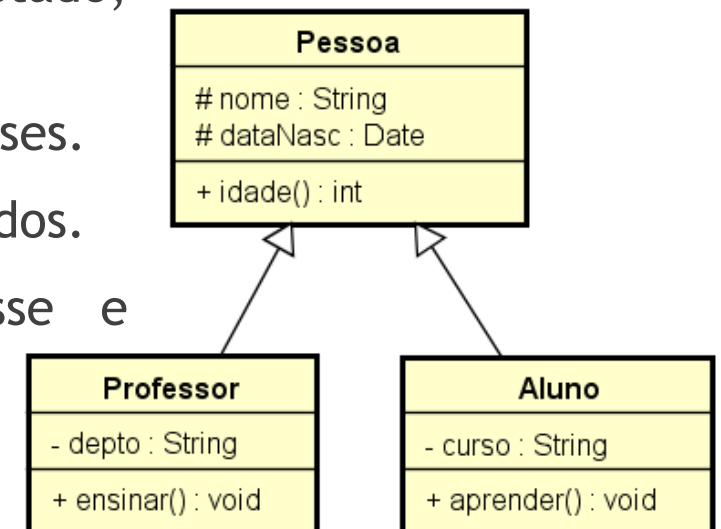
# Herança

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



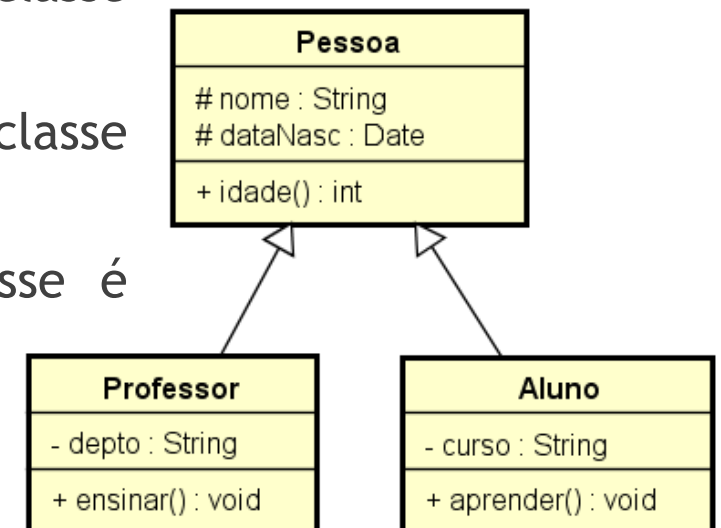
# Herança

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.



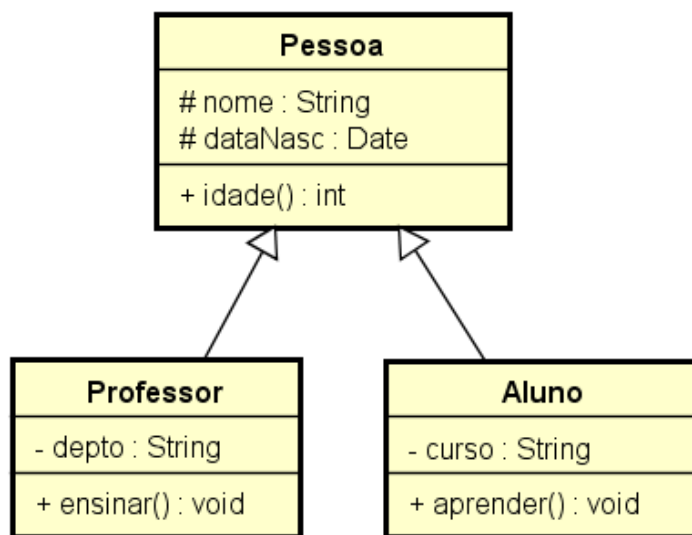
# Herança



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

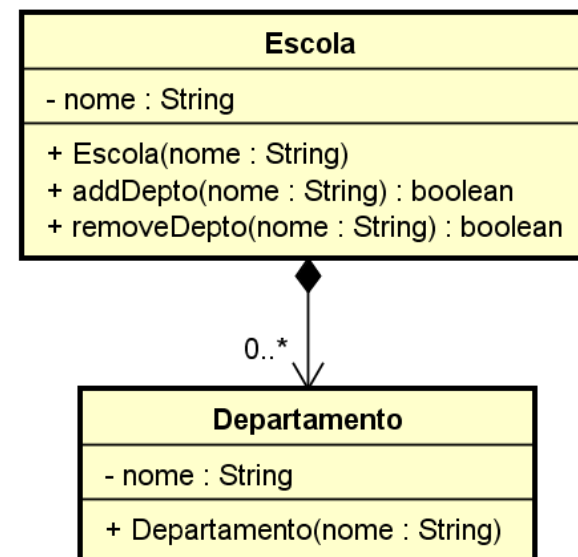
É um representa a herança.

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.

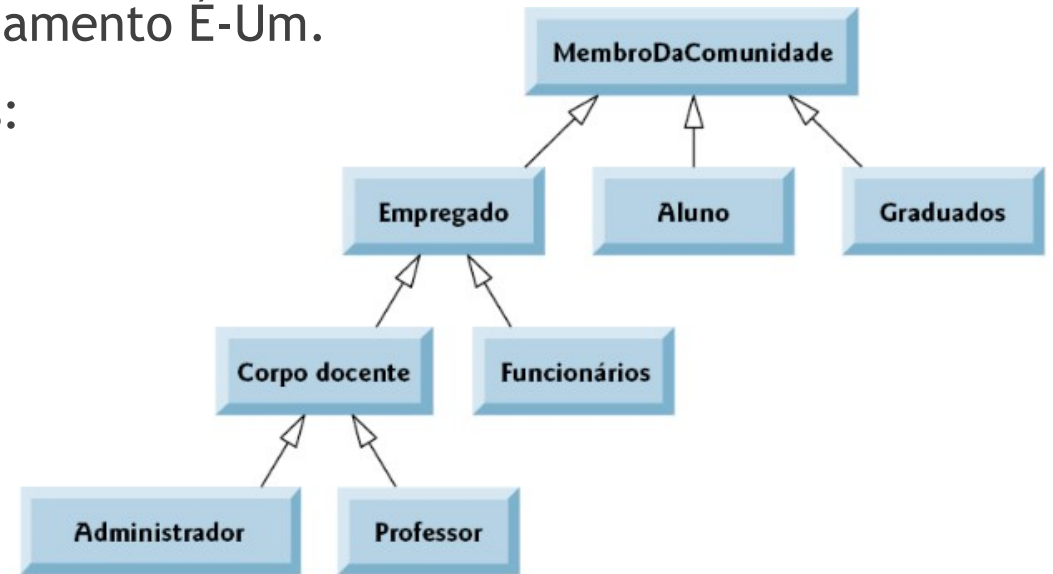




### Exemplo:

#### 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 Empregado.**

**MembroDaComunidade é a superclasse indireta de Administrador e Professor.**

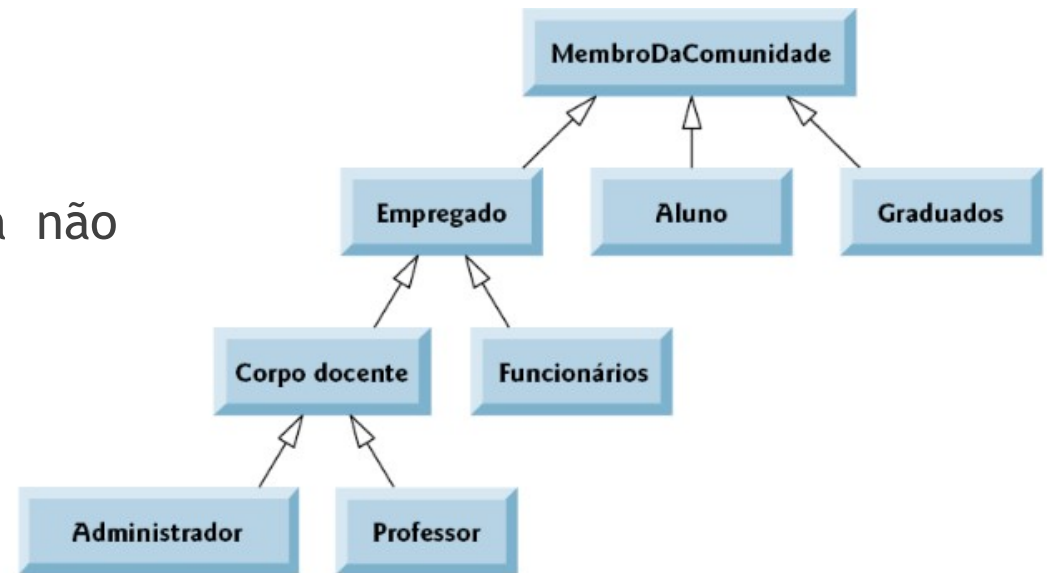
## Relacionamento É-Um e Tem-Um

✓ 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.



# Visibilidade na herança



**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.
- ▶ 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

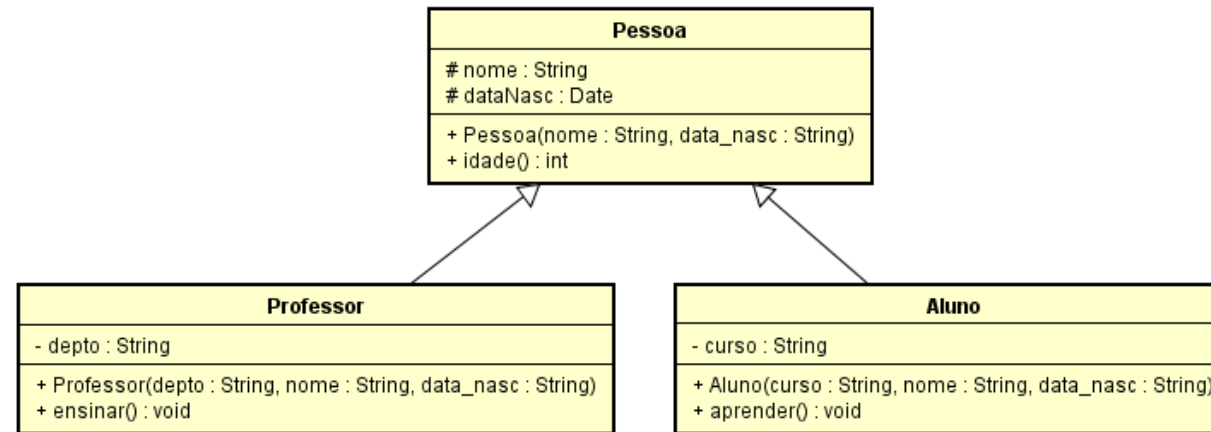
Os modificadores de acesso são utilizados para garantir o encapsulamento.  
Podem ser aplicados tanto a classes quanto a seus membros  
(atributos e métodos).



## 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.

# Criando o relacionamento entre as classes



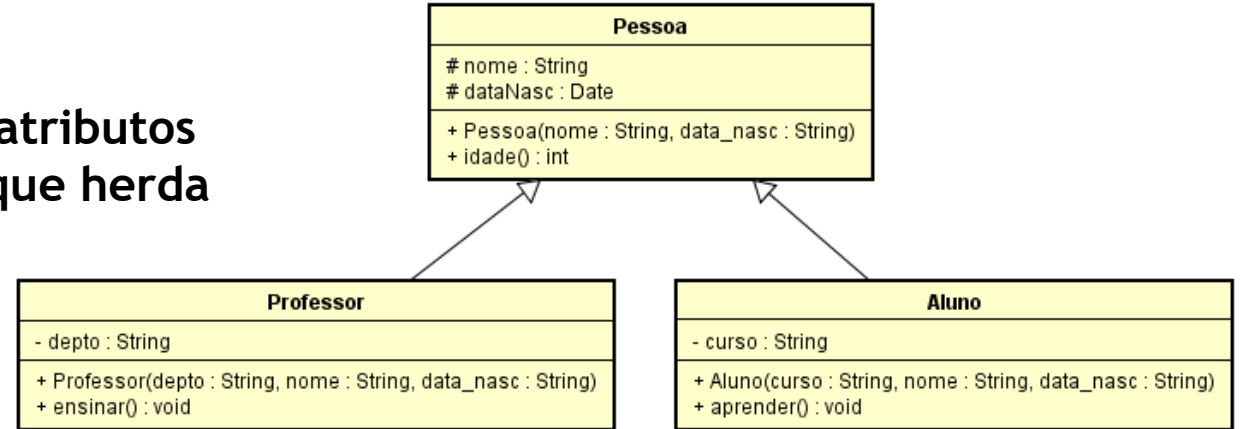
## Palavra reservada «EXTENDS»

- Para representar este tipo de relacionamento na linguagem Java, deve-se utilizar a palavra reservada **extends**.

```
public class Professor extends Pessoa {  
    private String depto;  
}
```

# Construtores em Subclasses

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



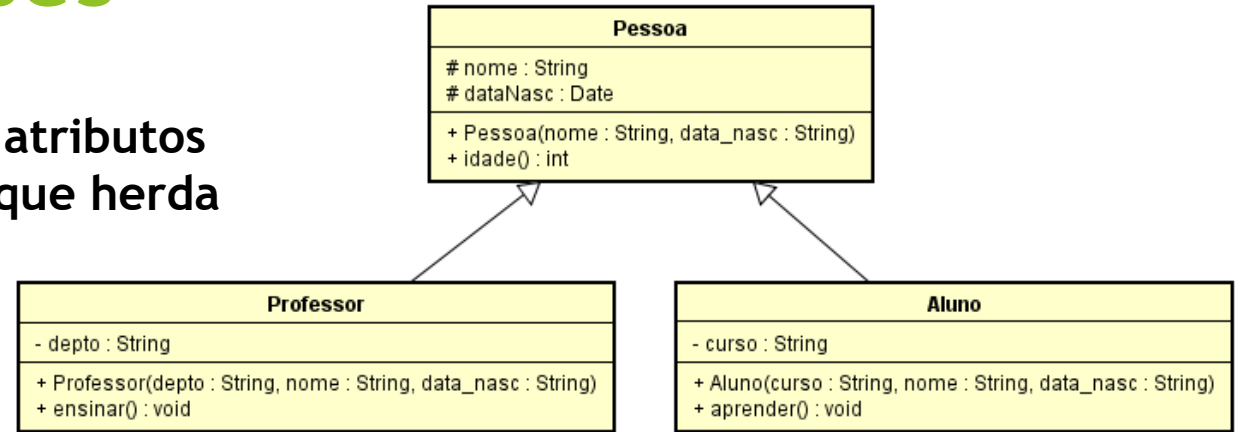
## Construtores em Subclasses

- ▶ 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 ao 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

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



## 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

```
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Calendar;
import java.util.Date;

public class Pessoa {

    protected String nome;
    protected Date dataNasc;

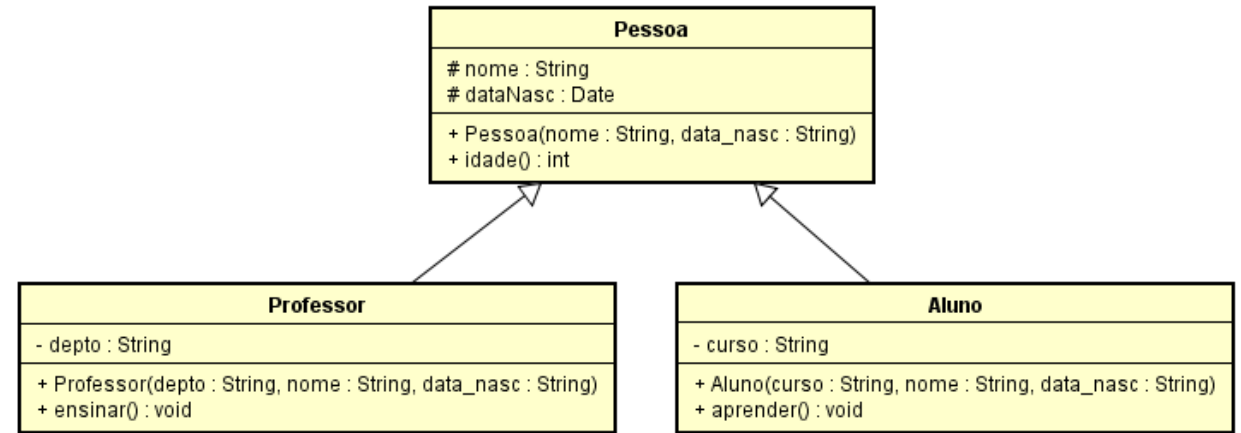
    public Pessoa(String nome, String dataNasc) throws ParseException {
        SimpleDateFormat format = new SimpleDateFormat("dd/MM/yyyy");

        this.nome = nome;
        this.dataNasc = (Date) format.parse(dataNasc);
    }

    public int idade() {
        Calendar c1 = Calendar.getInstance();
        Calendar c2 = Calendar.getInstance();

        c2.setTime(dataNasc);

        return c1.get(Calendar.YEAR) - c2.get(Calendar.YEAR);
    }
}
```



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

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

public Date getDataNasc() {
    return dataNasc;
}

public void setDataNasc(Date dataNasc) {
    this.dataNasc = dataNasc;
}

@Override
public String toString() {
    return "Pessoa [nome=" + nome
        + ", dataNasc=" + dataNasc + "];"
}
}
```

# Herança

## Implementando a Classe Professor

```
import java.text.ParseException;

public class Professor extends Pessoa {
    private String depto;

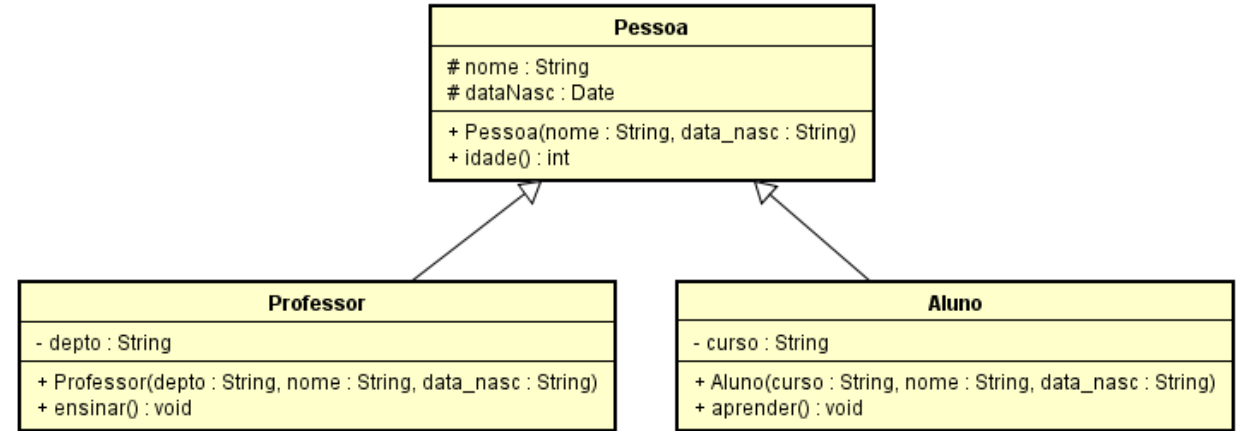
    public Professor(String nome, String dataNasc, String depto) throws ParseException {
        super(nome, dataNasc);
        this.depto = depto;
    }

    public void ensinar() {
        System.out.println("O professor est□ ensinando!");
    }

    public String getDepto() {
        return depto;
    }

    public void setDepto(String depto) {
        this.depto = depto;
    }

    @Override
    public String toString() {
        return "Professor [depto=" + depto + ", nome=" + nome + ", dataNasc=" + dataNasc + "];"
    }
}
```



# Herança

## Implementando a Classe Aluno

```
package Aula12;

import java.text.ParseException;

public class Aluno extends Pessoa {
    private String curso;

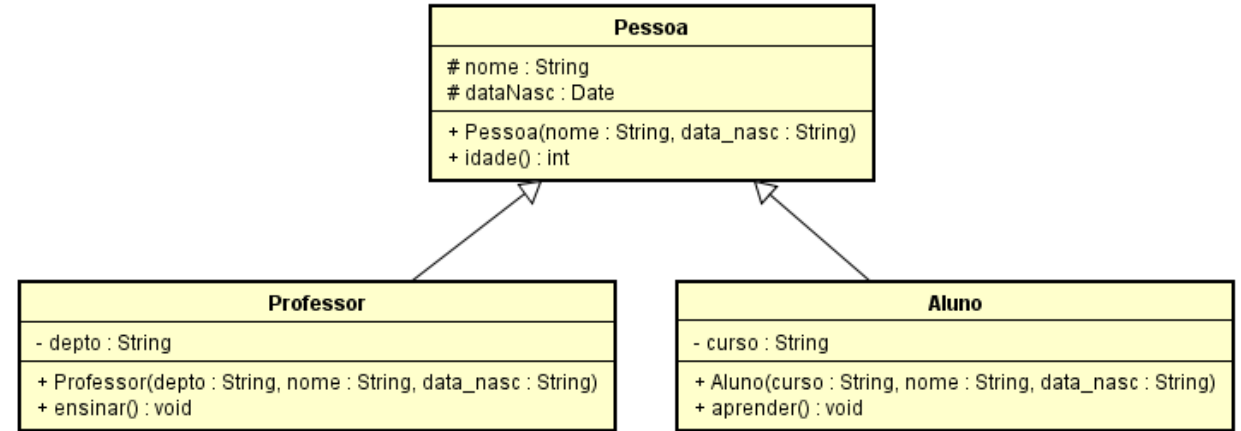
    public Aluno(String nome, String dataNasc, String curso) throws ParseException {
        super(nome, dataNasc);
        this.curso = curso;
    }

    public void aprender() {
        System.out.println("O aluno está aprendendo!");
    }

    public String getCurso() {
        return curso;
    }

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

    @Override
    public String toString() {
        return "Aluno [curso=" + curso + ", nome=" + nome + ", dataNasc=" + dataNasc + "]";
    }
}
```



# Herança

## Implementando a Classe Principal

```
import java.text.ParseException;

public class TestePessoa {

    public static void main(String[] args) throws ParseException {

        Pessoa pessoa = new Pessoa("Thiago", "15/09/1975");
        System.out.println(pessoa);

        Professor professor = new Professor("Renan", "27/04/1984", "Depto IV");
        System.out.println(professor);

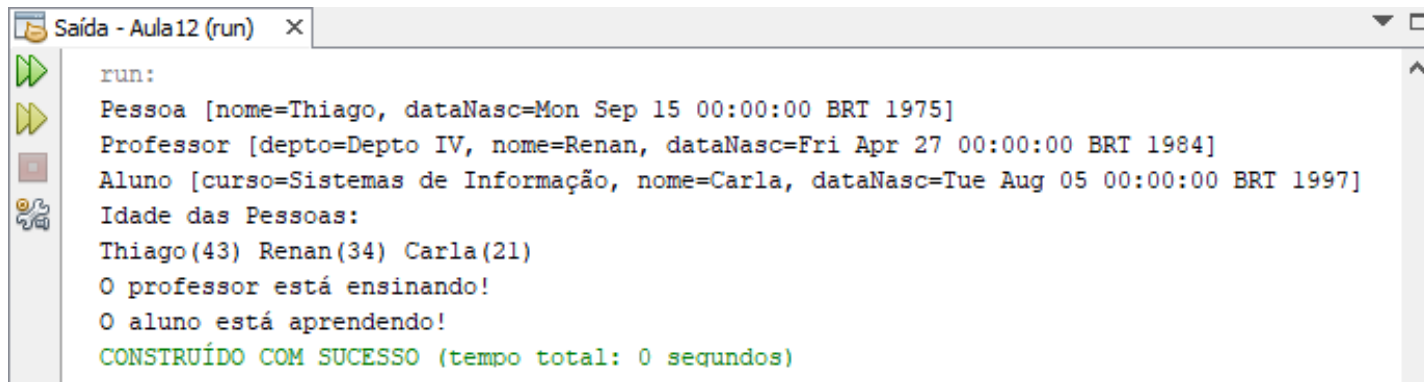
        Aluno aluno = new Aluno("Carla", "05/08/1997", "Sistemas de Informação");
        System.out.println(aluno);

        System.out.println(
            "Idade das Pessoas: " + "\n" +
            pessoa.getNome() + "(" + pessoa.idade() + ") " +
            professor.getNome() + "(" + professor.idade() + ") " +
            aluno.getNome() + "(" + aluno.idade() + ") "
        );

        professor.ensinar();
        aluno.aprender();

    }

}
```



```
run:
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(43) Renan(34) Carla(21)
O professor está ensinando!
O aluno está aprendendo!
CONSTRUÍDO COM SUCESSO (tempo total: 0 segundos)
```



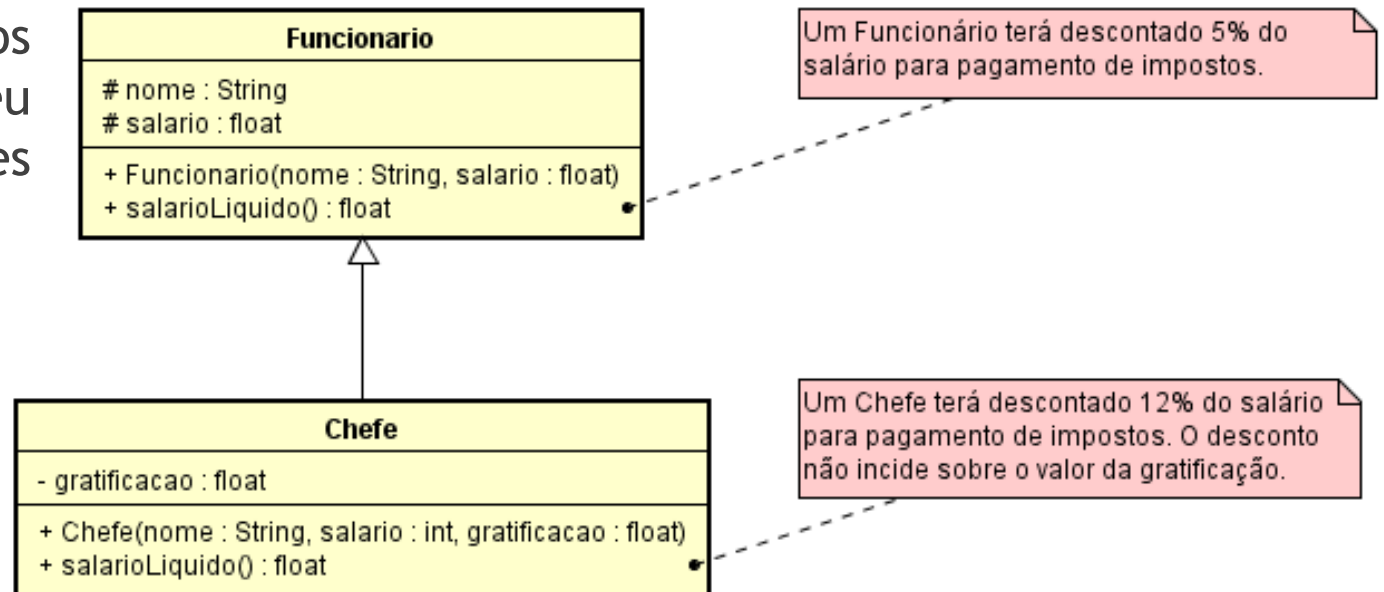
# Sobrescrita de Métodos

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.

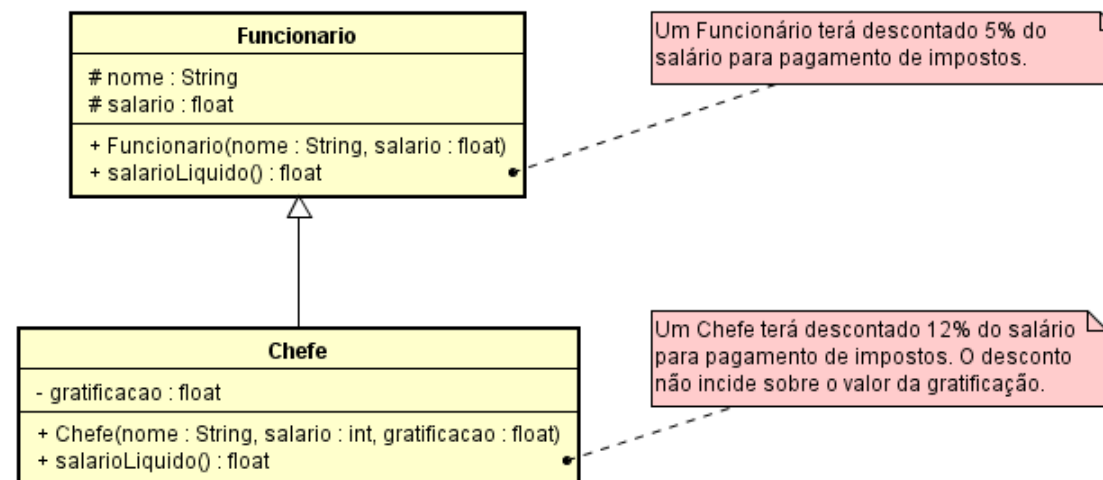


# Sobrescrita de Métodos

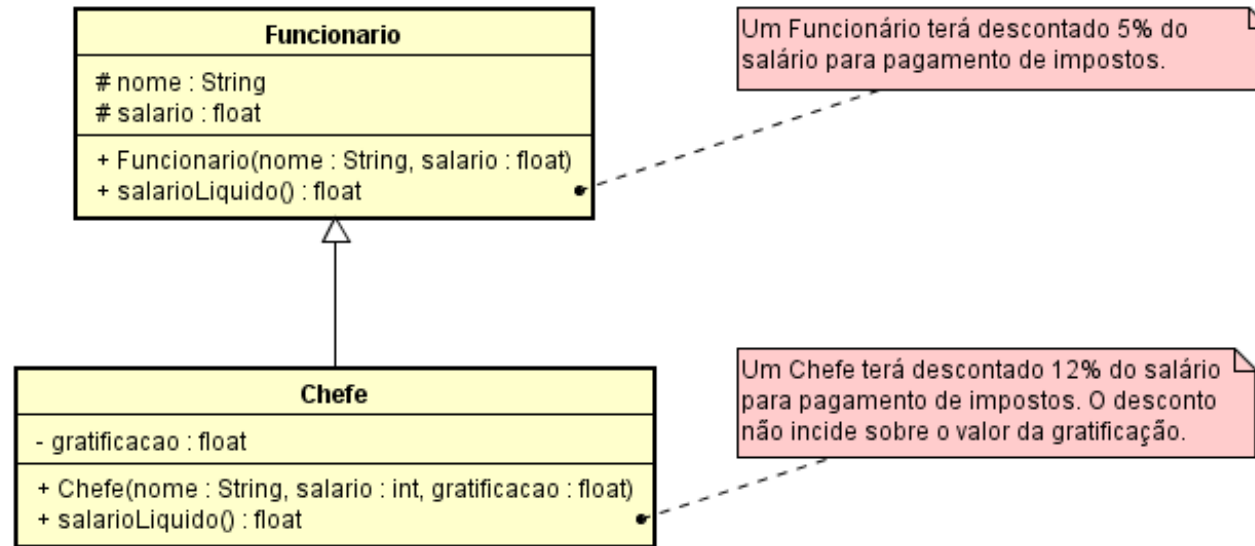


Um método da subclasse é considerado uma redefinição de um método da superclasse se:

- ▶ 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.



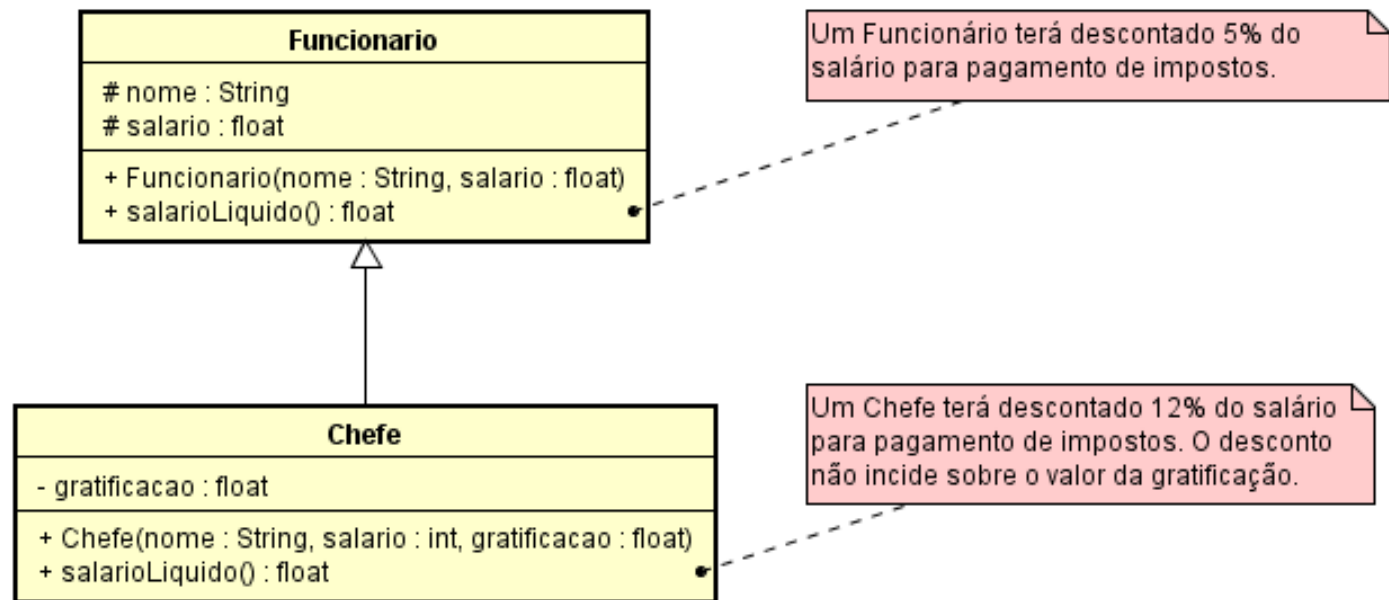
**A visibilidade dos métodos redefinidos pode ser diferente dos métodos da superclasse, mas apenas para dar mais acesso.**

- ▶ 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

```
public class Funcionario {  
  
    protected String nome;  
    protected float salario;  
  
    public Funcionario(String nome, float salario) {  
        this.nome = nome;  
        this.salario = salario;  
    }  
  
    public float salarioLiquido() {  
        return salario * 0.95f;  
    }  
  
    public String getNome() {  
        return nome;  
    }  
  
    public void setNome(String nome) {  
        this.nome = nome;  
    }  
  
    public float getSalario() {  
        return salario;  
    }  
  
    public void setSalario(float salario) {  
        this.salario = salario;  
    }  
}
```



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

# Métodos Sobrescritos

## Implementando a Classe Chefe

```
public class Chefe extends Funcionario {
```

```
    private float gratificacao;
```

```
    public Chefe(String nome, float salario, float gratificacao) {  
        super(nome, salario);  
        this.gratificacao = gratificacao;  
    }
```

```
    public float salarioLiquido() {  
        return salario * 0.88f + gratificacao;  
    }
```

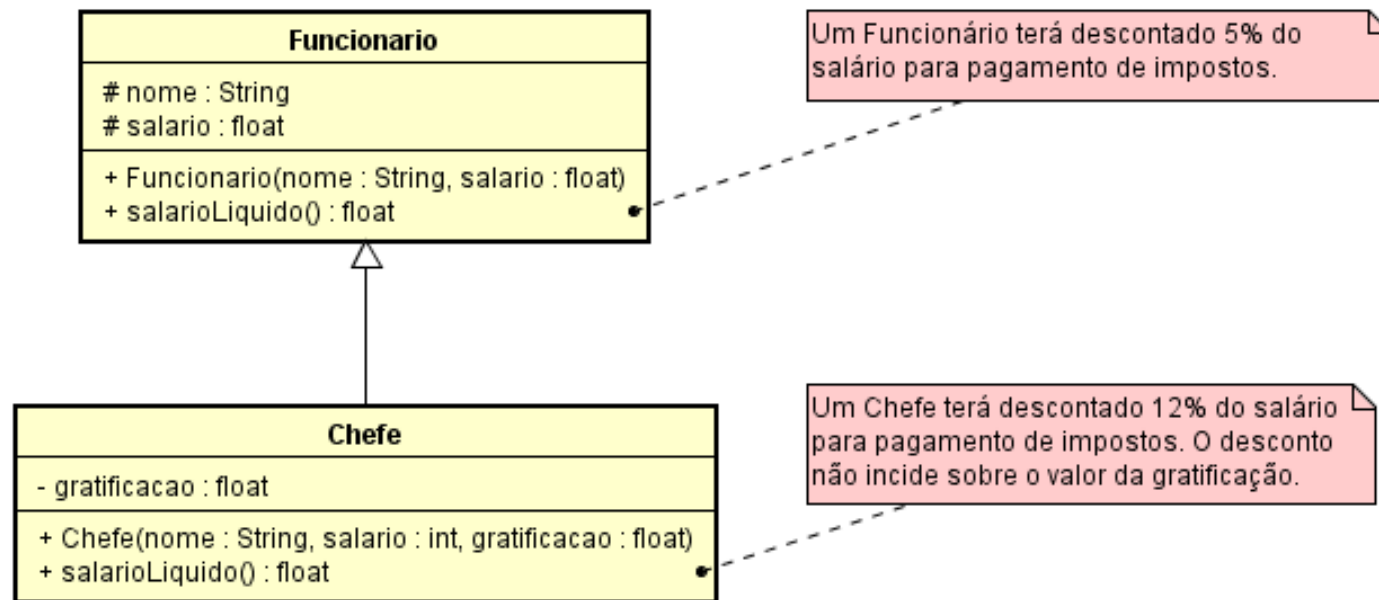
```
    public float getGratificacao() {  
        return gratificacao;  
    }
```

```
    public void setGratificacao(float gratificacao) {  
        this.gratificacao = gratificacao;  
    }
```

```
    @Override
```

```
    public String toString() {  
        return "Chefe [gratificacao=" + gratificacao + ", nome=" + nome + ", salario=" + salario +
```

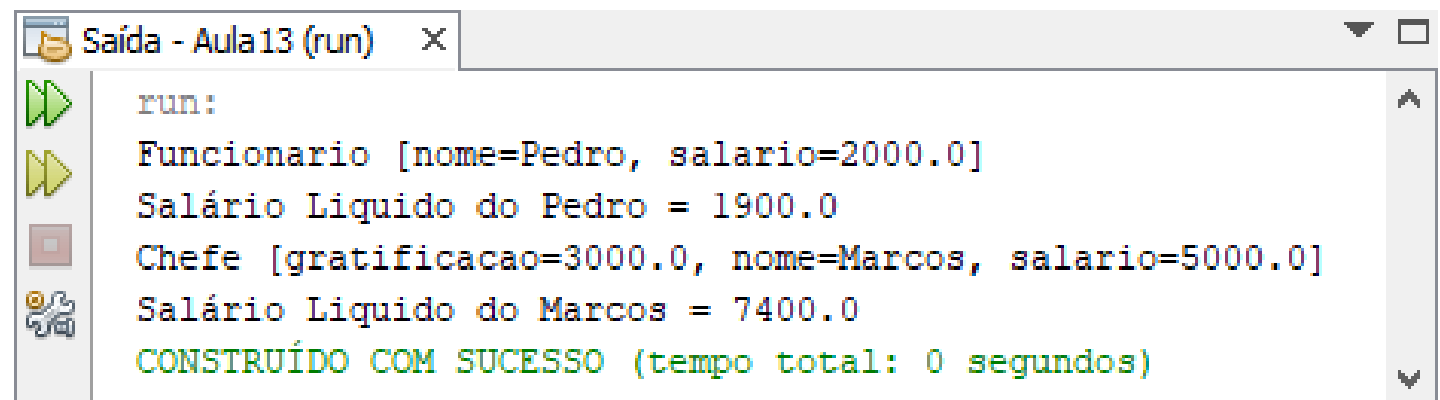
```
    }
```



## Métodos Sobrescritos

### Implementando a Classe Principal

```
public class TesteFuncionario {  
  
    public static void main(String[] args) {  
  
        Funcionario f = new Funcionario("Pedro", 2000);  
        Chefe c = new Chefe("Marcos", 5000, 3000);  
  
        System.out.println(f);  
        System.out.println("Salário Líquido do " + f.getNome() +  
            " = " + f.salarioLiquido());  
  
        System.out.println(c);  
        System.out.println("Salário Líquido do " + c.getNome() +  
            " = " + c.salarioLiquido());  
    }  
}
```



Saída - Aula13 (run) X

```
run:  
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  
CONSTRUÍDO COM SUCESSO (tempo total: 0 segundos)
```

# 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
```

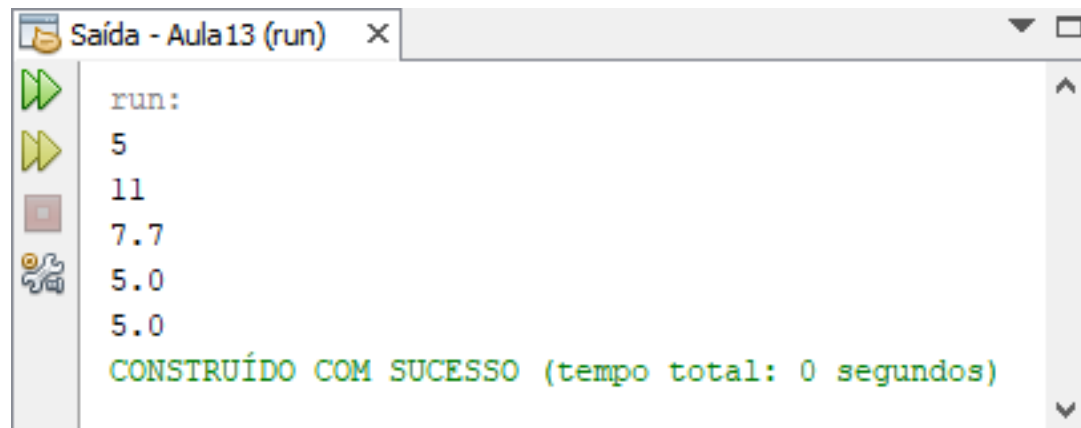


## Métodos Sobrecarregados

### Implementando a Classe Calculadora

```
public class Calculadora {  
  
    public int somar(int n1, int n2) {  
        return n1 + n2;  
    }  
  
    public int somar(int n1, int n2, int n3) {  
        return n1 + n2 + n3;  
    }  
  
    public float somar(int n1, float n2) {  
        return n1 + n2;  
    }  
  
    public float somar(float n1, float n2) {  
        return n1 + n2;  
    }  
}
```

```
public class TesteCalculo {  
  
    public static void main(String[] args) {  
  
        Calculadora c = new Calculadora();  
  
        System.out.println(c.somar(2,3));  
        System.out.println(c.somar(4,2,5));  
        System.out.println(c.somar(2,5.7f));  
        System.out.println(c.somar(2.0f,3));  
        System.out.println(c.somar(2.0f,3.0f));  
    }  
}
```



The screenshot shows a window titled "Saída - Aula13 (run)" with a list of output values and a success message. The output values are 5, 11, 7.7, 5.0, and 5.0, which correspond to the five println statements in the TesteCalculo class. The message "CONSTRUÍDO COM SUCESSO (tempo total: 0 segundos)" indicates that the program was compiled and run successfully.

```
run:  
5  
11  
7.7  
5.0  
5.0  
CONSTRUÍDO COM SUCESSO (tempo total: 0 segundos)
```

### 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.

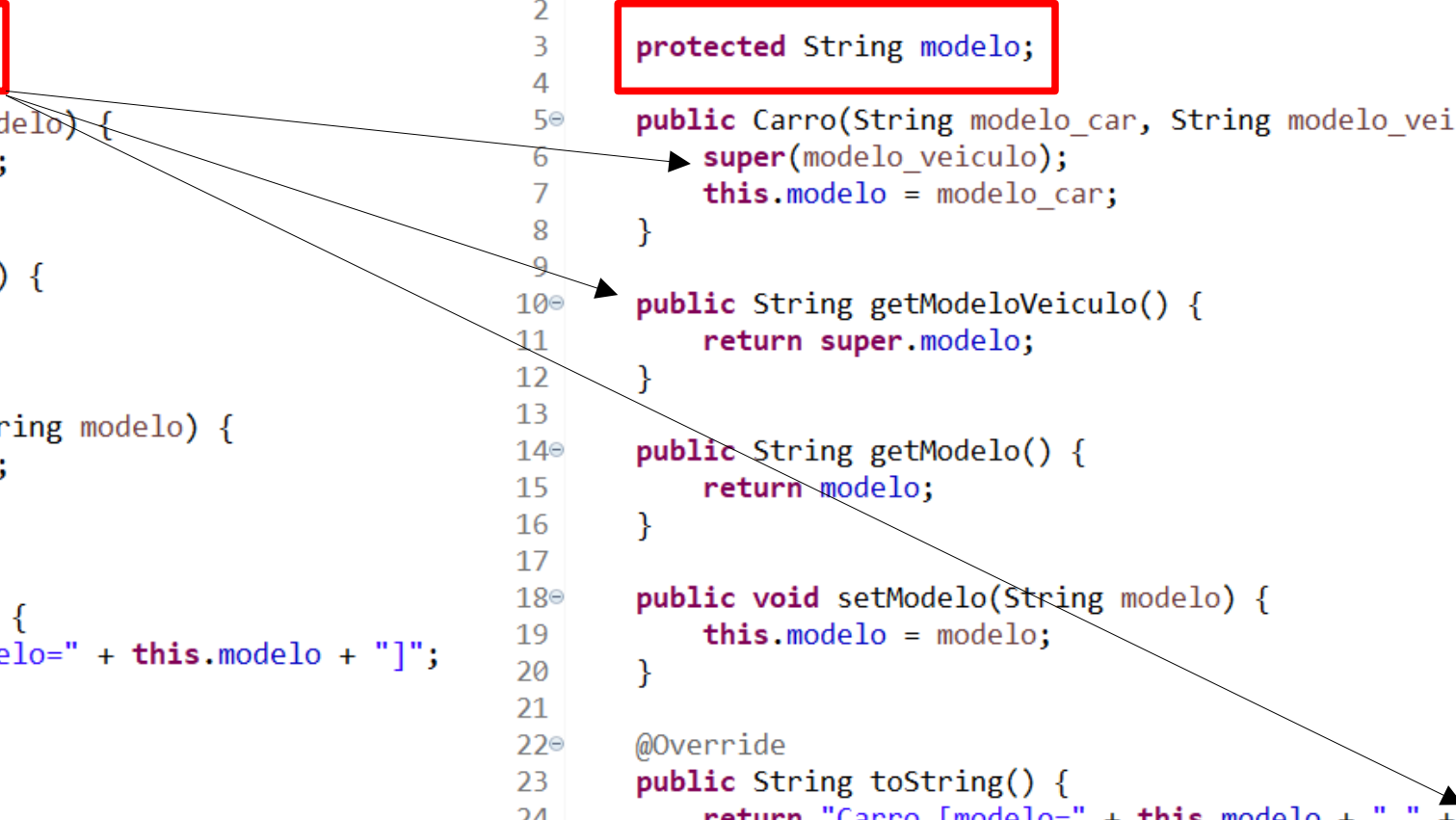
# Herança e Redefinição

## Atributo

### Implementando a Classe Veículo e Carro

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

```
1 public class Carro extends Veiculo {  
2  
3     protected String modelo;  
4  
5     public Carro(String modelo_car, String modelo_veiculo) {  
6         super(modelo_veiculo);  
7         this.modelo = modelo_car;  
8     }  
9  
10    public String getModeloVeiculo() {  
11        return super.modelo;  
12    }  
13  
14    public String getModelo() {  
15        return modelo;  
16    }  
17  
18    public void setModelo(String modelo) {  
19        this.modelo = modelo;  
20    }  
21  
22    @Override  
23    public String toString() {  
24        return "Carro [modelo=" + this.modelo + " " + super.modelo + "];"  
25    }  
26 }
```



# Herança e Redefinição

## Atributo

### Implementando a Classe Veículo e Carro

```
1 public class TesteCarro {  
2  
3     public static void main(String[] args) {  
4  
5         Veiculo v = new Veiculo("Gol");  
6  
7         System.out.println(v);  
8  
9         Veiculo c = new Carro("Gol", "G3");  
10  
11        System.out.println(c);  
12    }  
13 }
```

Console

<terminated> TesteCarro [Java Application] C:\Program Files\Java\jdk-11.0.4\bin\javaw.exe (

Veiculo [modelo=Gol]

Carro [modelo=Gol G3]