

# Programação orientada a objetos em Java

## Herança

---

Profª. Heloisa Moura



# Programação orientada a objetos em Java

## Herança

---

Princípio da orientação a objetos que permite a criação de novas classes a partir de outras previamente criadas. Essas novas classes são chamadas de subclasses, ou classes filhas. As classes já existentes, que deram origem às subclasses, são chamadas de superclasses, ou classes pai. Uma classe pode ter várias filhas, mas apenas uma mãe. É a chamada herança simples do Java.



# Programação orientada a objetos em Java

## Herança

---

É possível criar uma hierarquia de classes, definindo classes mais gerais e classes mais específicas. Uma sub-classe (+específica) herda métodos e atributos de sua super-classe (+geral); A sub-classe pode reescrever métodos, dando uma forma mais específica para um método herdado.



# Programação orientada a objetos em Java

## Herança

---

Observe as duas classes

```
class Cliente {  
    private String nome  
    private String endereco;  
    private String telefone;  
    private float debito;  
    private float credito;  
    private int limiteCredito;  
  
    ... metodos  
}
```

```
class Funcionario {  
    private String nome  
    private String endereco;  
    private String telefone;  
    private float salario;  
    private int horasExtras;  
  
    ... metodos  
}
```



# Programação orientada a objetos em Java

## Herança

---

Em java:

```
class Pessoa{  
    String nome;  
    String endereco;  
    String telefone;  
    ...métodos  
}
```



# Programação orientada a objetos em Java

## Herança

---

Em java: utilizamos a palavra chave **extends** para representar a Herança

```
import java.util.*;  
class Cliente extends  
Pessoa{  
    private float debito;  
    private float credito;  
    private int limiteCredito;  
    ...  
}
```

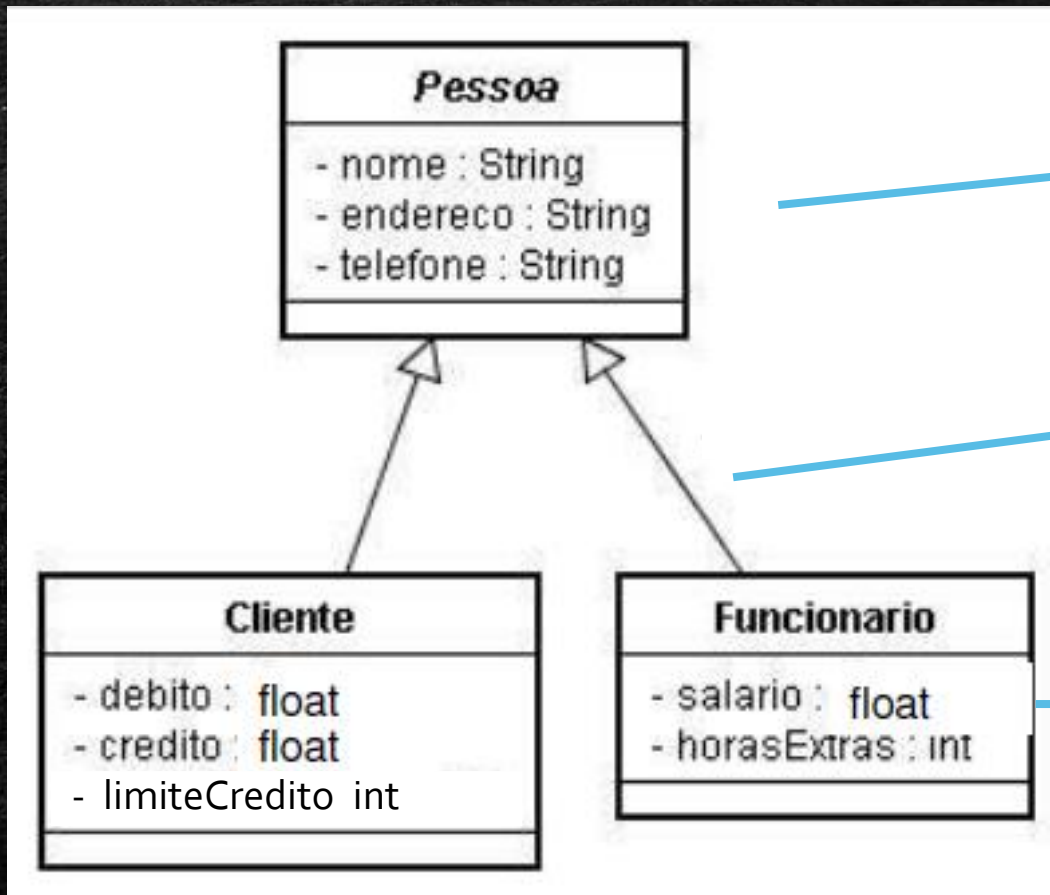
```
class Funcionario extends  
Pessoa{  
    private float salario;  
    private int horasExtras;  
}
```



# Programação orientada a objetos em Java

## Herança

Exemplo em UML



Super-classe:  
Generalização ou  
abstração

Herança:  
relacionamento "é um"

Sub-classe:  
Especialização



# Programação orientada a objetos em Java

## Herança

---

Exemplo Pratico



# Programação orientada a objetos em Java

## Herança

---

### Uso correto de herança

### Outro Exemplo:

```
package Herancaoutroexemplo
class Pessoa {
    String nome;
    String cpf;
    String endereco;
    String telefone;
}
```

```
package Herancaoutroexemplo
class Professor extends Pessoa {
    private float salario;
    private Date data_admissao;
    private String[] disciplinas;
}
```



# Programação orientada a objetos em Java

## Herança

---

```
class Professor extends Pessoa {  
    private float salario;  
    private Date dt_admissao;  
    private String[] disciplinas;  
}
```

```
class Tecnico extends Pessoa {  
    private float salario;  
    private Date dt_admissao;  
    private String cargo;  
}
```



# Programação orientada a objetos em Java

## Herança

---

```
class Tecnico extends Pessoa {
```


```
    private float salario;
```

```
    private Date data_admissao;
```

```
    private String cargo;
```

```
}
```

PROBLEMA:  
Repetição de código



Fazer:

```
class Tecnico extends Professor{...}
```

seria uma solução??

Na dúvida, pergunte: Técnico é um Professor?



# Programação orientada a objetos em Java

## Herança

---

Não, Técnico não é um Professor.

Neste caso é errado fazer

- class Tecnico extends Professor

Solução?

- Criar uma nova abstração conceitual capaz de fornecer o que é comum a ambas as classes (Tecnico e Professor)



# Programação orientada a objetos em Java

## Herança

---

Uma saída seria:

```
class Funcionario extends Pessoa{
```

```
    private float salario;
```



Comum a ambas as classes

```
    private Date data_admissao;
```

```
    //..métodos
```

```
}
```



# Programação orientada a objetos em Java

## Herança

---

```
class Tecnico extends Funcionario {
```

```
    private String cargo; ←
```

Específico de Técnico

```
    //...métodos
```

```
}
```

```
class Professor extends Funcionario {
```

```
    private String[] disciplinas; ←
```

Específico de Professor

```
    //...métodos
```

```
}
```



# Programação orientada a objetos em Java

## Herança

---

Desta forma,

- Técnico e Professor possuem o relacionamento é um. Nesse caso é um Funcionário.
- Elimina-se a replicação de código;
- Cria-se classes mais coesas e de reuso mais fácil;
- Cria-se um projeto:
  - mais flexível
  - de fácil entendimento
  - de fácil manutenção



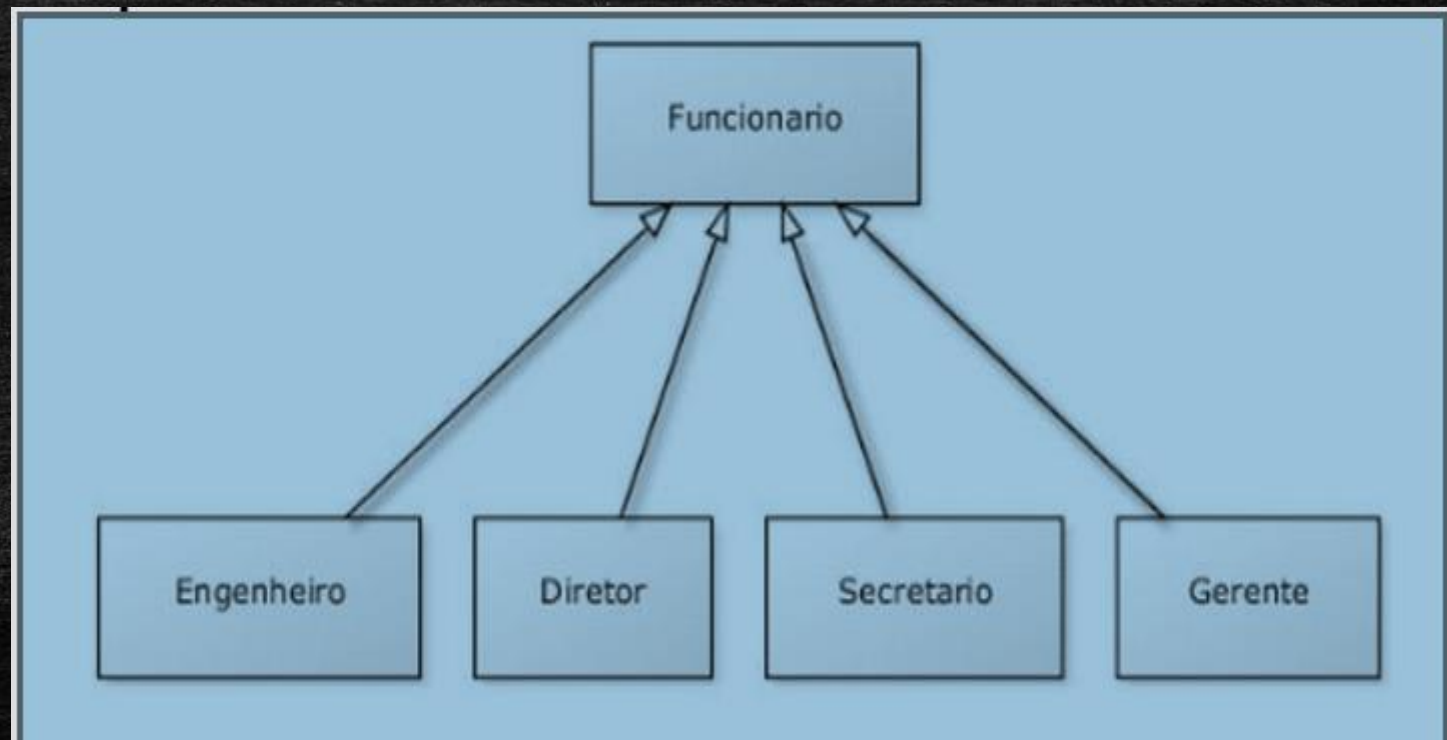
# Programação orientada a objetos em Java

## Herança

---

Além disto, pode-se estender o projeto com maior facilidade no futuro.

Exemplo:





# Programação orientada a objetos em Java

## Herança

---

Exemplo Prático



# Programação orientada a objetos em Java

## Herança - Construtor superclasse/subclasse

---

Um fato importante: os construtores são algo único a cada classe, portanto não são herdados.

A primeira coisa que um construtor faz é rodar o construtor de sua superclasse, pois ações importantes podem estar acontecendo lá - como inicialização de variáveis que poderão ser usadas na subclasse.

Porém, é possível invocar os construtores de uma superclasse através da subclasse.

Exemplo prático



# Programação orientada a objetos em Java

## Herança - Palavra reservada Super

---

Uso do **this** – referência a elementos da classe (atributos/métodos).  
Exemplo:

```
class Funcionario extends Pessoa{  
    private float salario;  
    private int horasExtras;  
  
    public float calcSalario(float vHora){  
        return this.salario+(this.horasExtras * vHora);  
    }  
}
```



# Programação orientada a objetos em Java

## Herança - Palavra reservada Super

---

Uso do **super** – referência a elementos da super-classe.

Exemplo:

```
class Pessoa {
```

```
    String nome;  
    String endereco;  
    String telefone;
```

```
    void mostrar() {
```

```
        System.out.println("Nome:" + this.nome);  
        System.out.println("Endereço: " + this.endereco);  
        System.out.println("Telefone: " + this.telefone);
```

```
    }
```

```
}
```



# Programação orientada a objetos em Java

## Herança - Palavra reservada Super

---

```
class Funcionario extends Pessoa{  
    private float salario;  
    private int horasExtras;  
    void mostrarInfo(){  
        System.out.println("Nome:" + super.nome);  
        System.out.println("Endereco: " + super.endereco);  
        System.out.println("Telefone: " + super.telefone);  
        System.out.println("Salario: " + this.salario);  
        System.out.println("H.extras: " + this.horasExtras);  
    }  
}
```



# Programação orientada a objetos em Java

## Herança - Palavra reservada Super

---

Uma solução melhor...

```
class Funcionario extends Pessoa{  
    private float salario;  
    private int horasExtras;  
    void mostrar(){  
        super.mostrar();  
        System.out.println("Salario: " + this.salario);  
        System.out.println("H.extras: " + this.horasExtras);  
    }  
}
```



# Programação orientada a objetos em Java

## Herança - Palavra reservada Super

---

Exemplo Prático



# Programação orientada a objetos em Java

## Herança - Modificador de acesso protected

---

### Modificador de acesso protected

Fica entre o private e o public . Um atributo protected só pode ser acessado (visível) pela própria classe, suas subclasses e classes encontradas no mesmo pacote.

Modificador usado em Herança, por sua visibilidade em subclasses.

Sintaxe:

```
protected String testeVisibilidade;
```



# Programação orientada a objetos em Java

## Herança - Modificador de acesso protected

Tabela de visibilidade dos modificadores de acesso para membros da classe

	<b>private</b>	<b>default</b>	<b>protected</b>	<b>public</b>
mesma classe	sim	sim	sim	sim
mesmo pacote	não	sim	sim	sim
pacotes diferentes (subclasses)	não	não	sim	sim
pacotes diferentes (sem subclasses)	não	não	não	sim



Programação orientada a objetos em Java

Herança - Modificador de acesso protected

---

Exemplo Pratico



# Programação orientada a objetos em Java

## Herança – Polimorfismo (sobrescrita/overriding) de métodos

---

Quando uma classe é estendida, a subclasse herda todos os métodos "não-privados" da classe pai. Algumas vezes, é desejável modificar o comportamento de um desses métodos na nova classe. Para fazer isso basta reescrever o método da classe pai na classe filha, utilizando o mesmo nome e a mesma lista de parâmetros (com algumas restrições quanto ao tipo de retorno, uso de modificadores de acesso e utilização de exceções). Nesse caso dizemos que o método está sendo sobrescrito.



# Programação orientada a objetos em Java

## Herança – Polimorfismo (sobrescrita/overriding) de métodos

---

Sendo alterado para atender o comportamento daquele objeto em questão na mesma operação herdada. A isso damos o nome de polimorfismo, ou seja, várias formas de implementar a mesma operação.

Exemplo prático



# Programação orientada a objetos em Java

## Herança – Classes abstratas

Classe abstrata é aquela que não criamos objetos dela. São classes muito genéricas. Isso quer dizer que criamos objetos de suas subclasses concretas. Ao colocar a palavra chave **abstract** na declaração da classe, não podemos mais instanciar-la.

Se ela não pode ser instanciada, para que serve? Serve para o polimorfismo e herança dos atributos e métodos, que são recursos muito poderosos. A palavra **abstract** não pode ser aplicada a atributos.



# Programação orientada a objetos em Java

## Herança – Classes abstratas

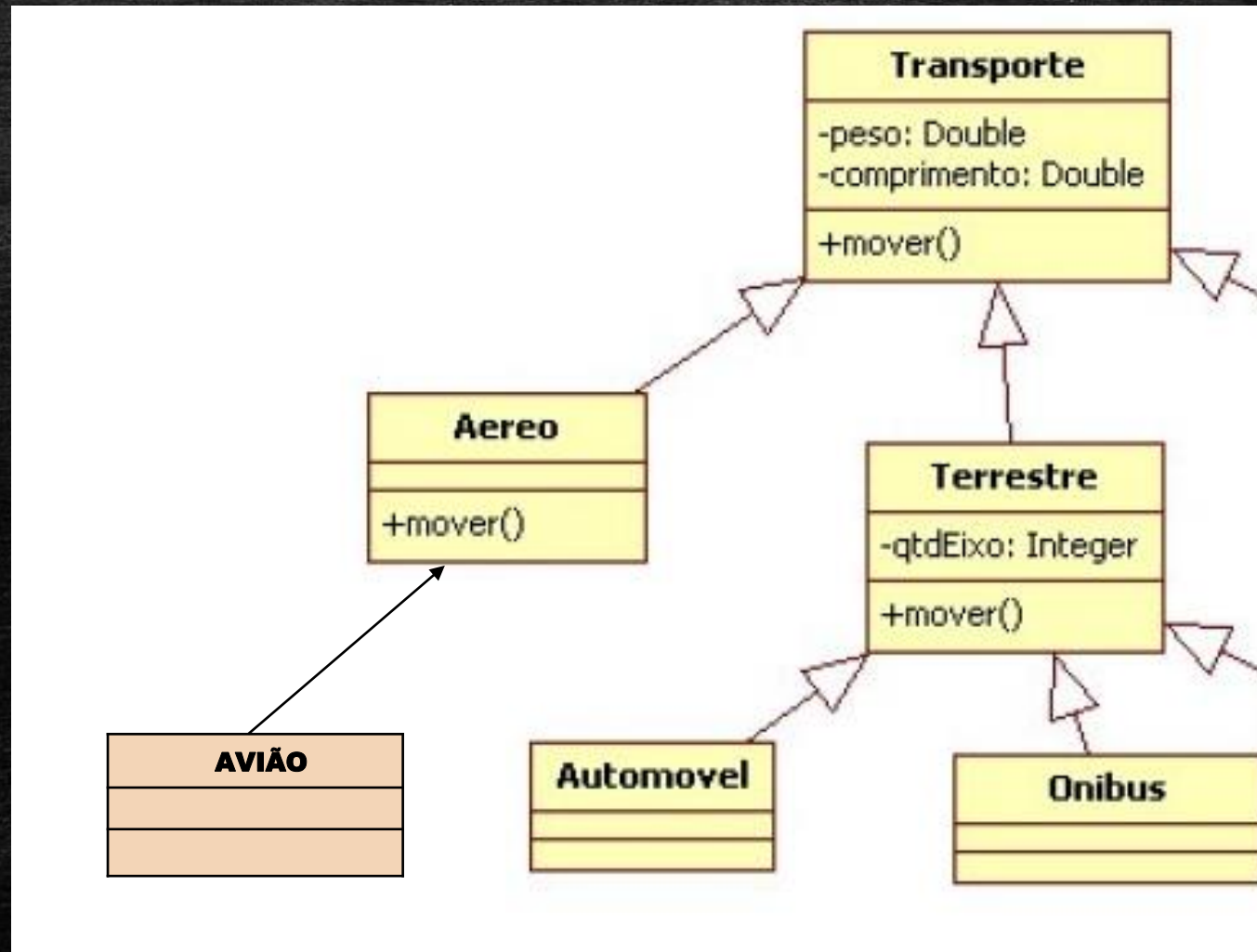
Exemplo pratico

```
public abstract class Transporte{
```



# Programação orientada a objetos em Java

## Herança – Classes abstratas





# Programação orientada a objetos em Java

## Herança – Palavra reservada final

---

A palavra reservada **final** quando é aplicada na classe, não permite estende-la, nos métodos impede que os mesmos sejam sobrescrito (overriding) na subclasse, e nos valores de atributos, não permite que os mesmos sejam alterados, depois que já tenha sido atribuído um valor.

A finalidade de se ter uma classe **final** é pra que a mesma seja preservada. Seu comportamento não seja alterado. Um exemplo de classes **final** é a Classe String e a Classe Math do pacote java.lang. Classes essas bastante utilizada. Imagine se qualquer outra classe pudesse mudar seus comportamentos.

    O que aconteceria???



# Programação orientada a objetos em Java

## Herança – Palavra reservada final

---

Os atributos marcados com **final** se tornam variáveis CONSTANTES dentro da classe e não pode ter seus valores alterados. Por isso do nome constante.

As constantes por convenção do Java são declaradas com letras maiúsculas. É comum ter o modificador static em sua declaração para facilitar o seu acesso direto, através do nome da classe.

Exemplo de constante da Classe Math

```
public static final double PI = 3.14159265358979323846;
```



# Programação orientada a objetos em Java

## Herança – Palavra reservada final

---

Exemplo prático

EXERCÍCIOS