# Herança

## Programação Orientada a Objetos — QXD0007

Prof. Atílio Gomes Luiz
gomes.atilio@ufc.br

Universidade Federal do Ceará
Campus Quixadá

2º semestre/2021

# Leituras para esta aula



- **Capítulo 9** (Herança, reescrita e polimorfismo) da apostila da Caelum – Curso FJ-11, Disponível no link: https://www.caelum.com.br/apostila/apostila-java-orientacao-objetos.pdf

- **Capítulo 9** (Herança) do livro Java Como Programar, Décima Edição, Disponível no link: http://libgen.lc/ads.php?md5=728636A04ACA056038BB5F079403AC96

- **Capítulo 8** (Reutilização de classes) do livro Introdução à Programação Orientada a Objetos usando Java, Rafael Santos, Disponível no link: https://www.academia.edu/6227746/Introdu%C3%A7%C3%A3o_%C3%A0_Programa%C3%A7%C3%A3o_Orientada_a_Objetos_Usando_Java

# Introdução

# Reutilização de Classes



- Vimos 3 relacionamentos entre classes que permitem o reuso de classes:
  - Associação, Agregação e Composição

- Esses mecanismos permitem a reutilização de classes já existentes como instâncias de novas classes. As classes originais ficam *contidas* na nova classe.

- Esses mecanismos são úteis quando consideramos que a classe que reutiliza instâncias de outras é composta das outras classes ou as usa.

# Reutilização de Classes



- Vimos 3 relacionamentos entre classes que permitem o reuso de classes:
  - Associação, Agregação e Composição

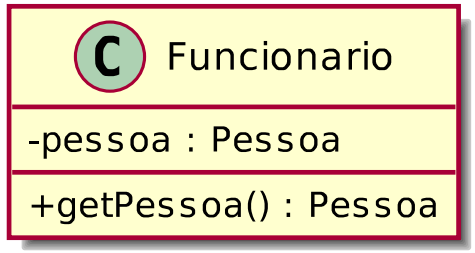
- Esses mecanismos permitem a reutilização de classes já existentes como instâncias de novas classes. As classes originais ficam *contidas* na nova classe.

- Esses mecanismos são úteis quando consideramos que a classe que reutiliza instâncias de outras é composta das outras classes ou as usa.

- Nem sempre o mecanismo de delegação é o mais natural para reutilização de classes já existentes. Em especial, quando queremos usar uma classe para servir de base à criação de outra mais especializada, a relação de composição imposta pelo uso do mecanismo de delegação acaba por criar soluções pouco naturais.

# Reutilização de Classes – Exemplo

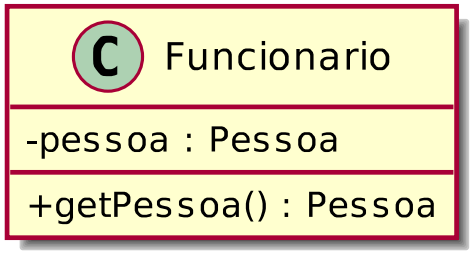
Temos duas classes `Pessoa` e `Funcionario` tal que uma instância de `Pessoa` é declarada dentro da classe `Funcionario` para representar os dados da pessoa/funcionário.



# Reutilização de Classes – Exemplo



Temos duas classes `Pessoa` e `Funcionario` tal que uma instância de `Pessoa` é declarada dentro da classe `Funcionario` para representar os dados da pessoa/funcionário.

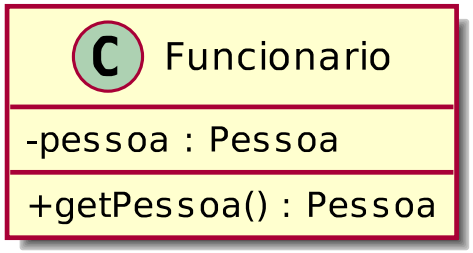



Queremos criar uma classe `ChefeDeDepartamento`. Um chefe de departamento é um funcionário que é responsável por um departamento.

# Reutilização de Classes – Exemplo

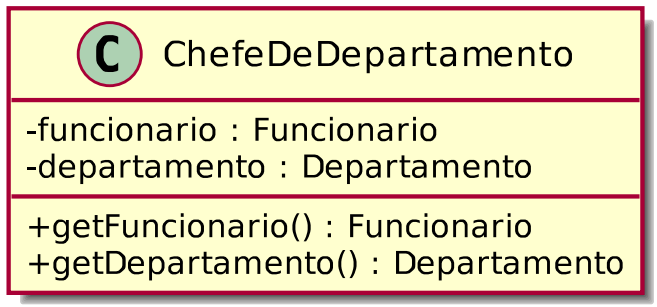


Temos duas classes `Pessoa` e `Funcionario` tal que uma instância de `Pessoa` é declarada dentro da classe `Funcionario` para representar os dados da pessoa/funcionário.




Queremos criar uma classe `ChefeDeDepartamento`. Um chefe de departamento é um funcionário que é responsável por um departamento.

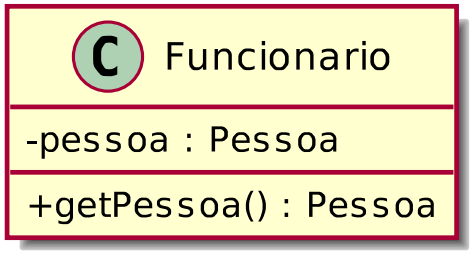
Usando o mecanismo de *delegação*, podemos declarar uma instância de `Funcionario` dentro da classe `ChefeDeDepartamento` e acrescentar alguns campos que diferenciam `ChefeDeDepartamento` de `Funcionario`.



# Reutilização de Classes – Exemplo

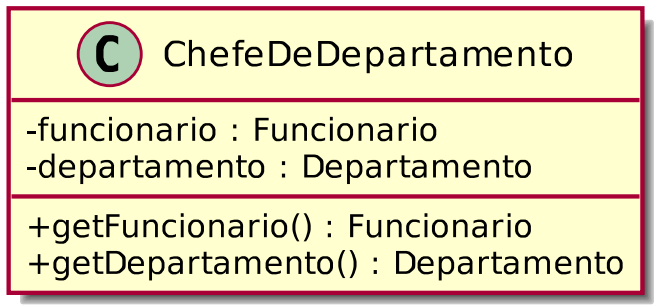


Temos duas classes `Pessoa` e `Funcionario` tal que uma instância de `Pessoa` é declarada dentro da classe `Funcionario` para representar os dados da pessoa/funcionário.




Queremos criar uma classe `ChefeDeDepartamento`. Um chefe de departamento é um funcionário que é responsável por um departamento.

Usando o mecanismo de *delegação*, podemos declarar uma instância de `Funcionario` dentro da classe `ChefeDeDepartamento` e acrescentar alguns campos que diferenciam `ChefeDeDepartamento` de `Funcionario`.




**Problema:** Declarar que `ChefeDeDepartamento` *contém um* funcionário soa artificial — um chefe de departamento é um tipo de funcionário, que tem campos adicionais para representar dados específicos de um chefe de departamento, e métodos para manipular esses campos.

# Herança



- Herança é um relacionamento entre duas ou mais classes.
- Esse relacionamento permite que criemos uma classe usando outra como base e descrevendo ou implementando as diferenças e adições da classe usada como base, reutilizando os atributos e métodos não-privados da classe base.
- O mecanismo de herança é o mais apropriado para criar relações *é-um-tipo-de* entre classes.

# Herança

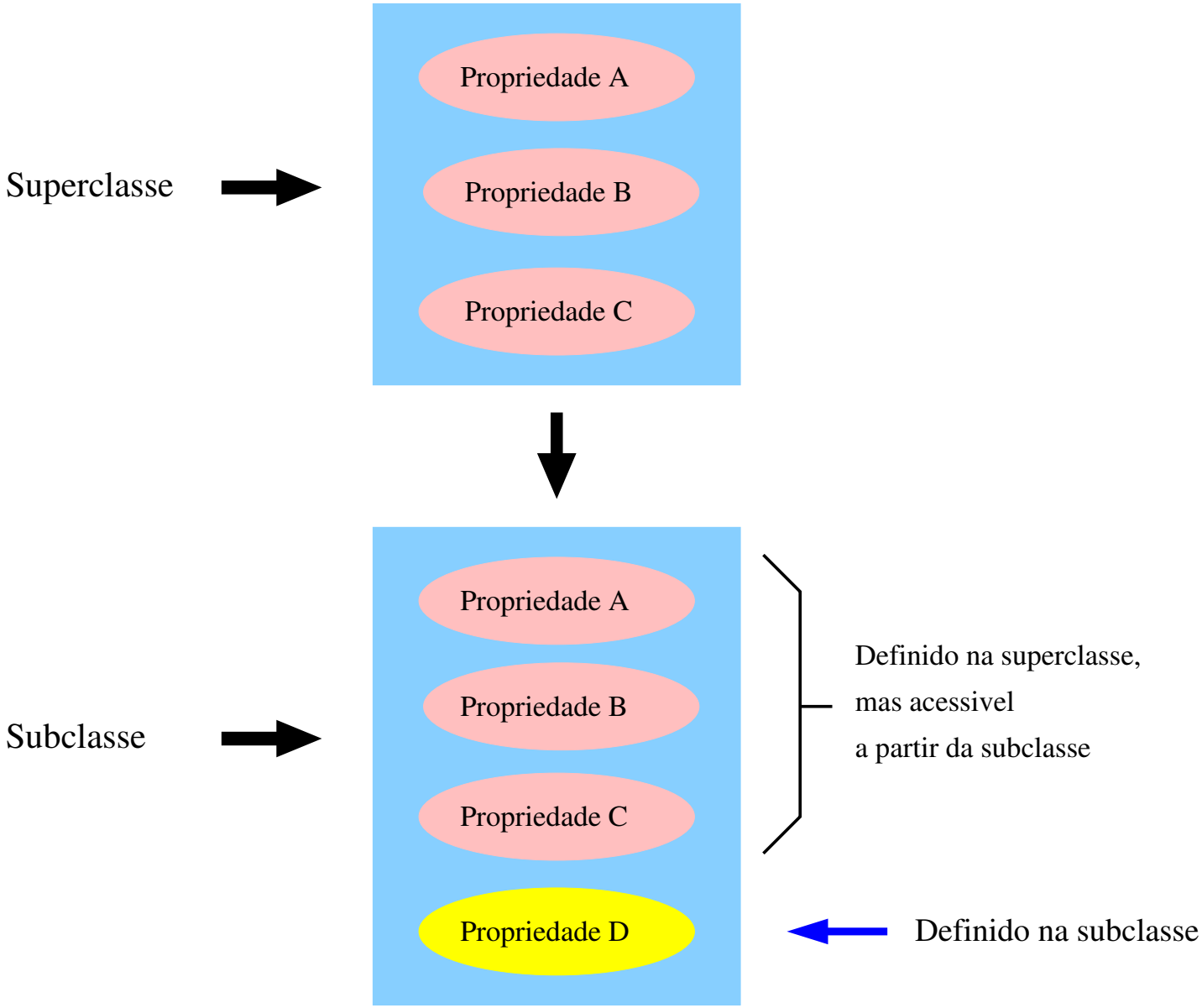


- Herança é um relacionamento entre duas ou mais classes.
- Esse relacionamento permite que criemos uma classe usando outra como base e descrevendo ou implementando as diferenças e adições da classe usada como base, reutilizando os atributos e métodos não-privados da classe base.
- O mecanismo de herança é o mais apropriado para criar relações *é-um-tipo-de* entre classes.

- Superclasse: é a classe cujas propriedades são herdadas por outra classe. É também chamada de classe base ou classe pai.
- Subclasse: é a classe que herda propriedades da classe base. É também chamada de classe derivada ou classe filha.

# Conceito de Herança

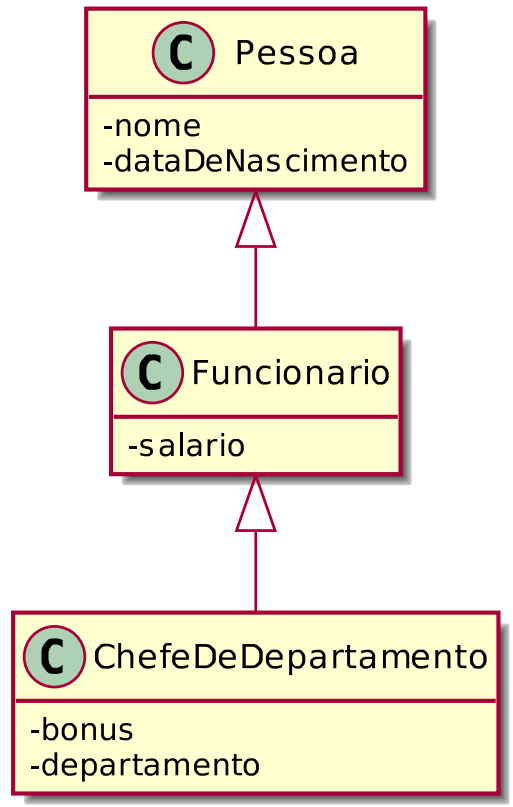




# Herança — Superclasses e subclasses

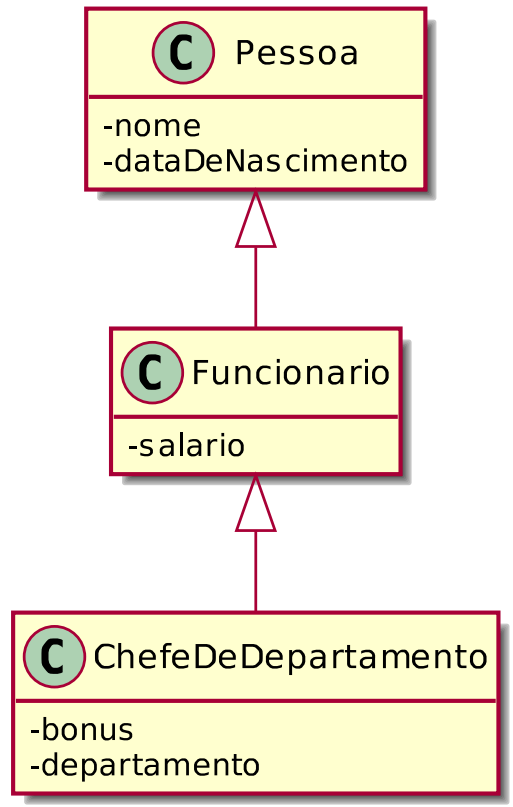
UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

- Uma subclasse é uma forma especializada da superclasse.
- Uma subclasse também pode vir a ser uma superclasse.



# Herança — Superclasses e subclasses

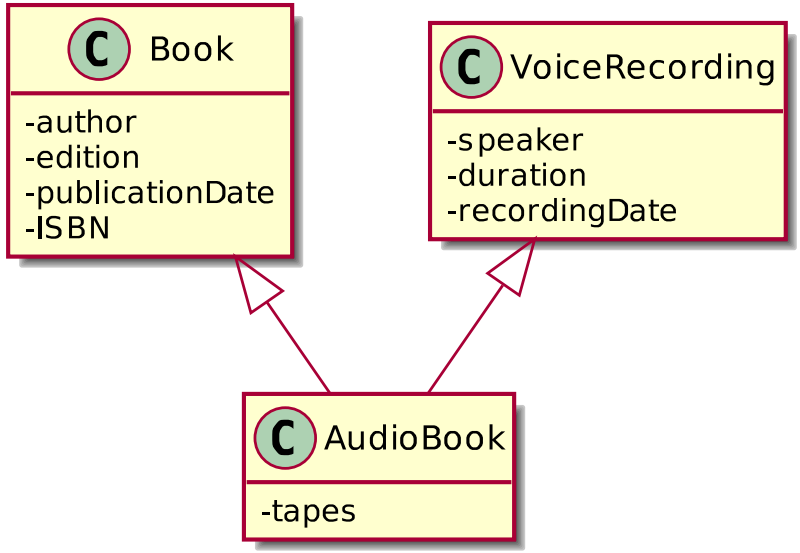
- Uma subclasse é uma forma especializada da superclasse.
- Uma subclasse também pode vir a ser uma superclasse.
- A superclasse direta é a superclasse da qual a subclasse herda explicitamente.
  - As outras são consideradas superclasses indiretas.



# Herança Única × Herança Múltipla



- Herança múltipla é quando uma subclasse pode herdar de mais de uma subclasse direta.
- Na Herança única, uma subclasse herda somente de uma superclasse direta.

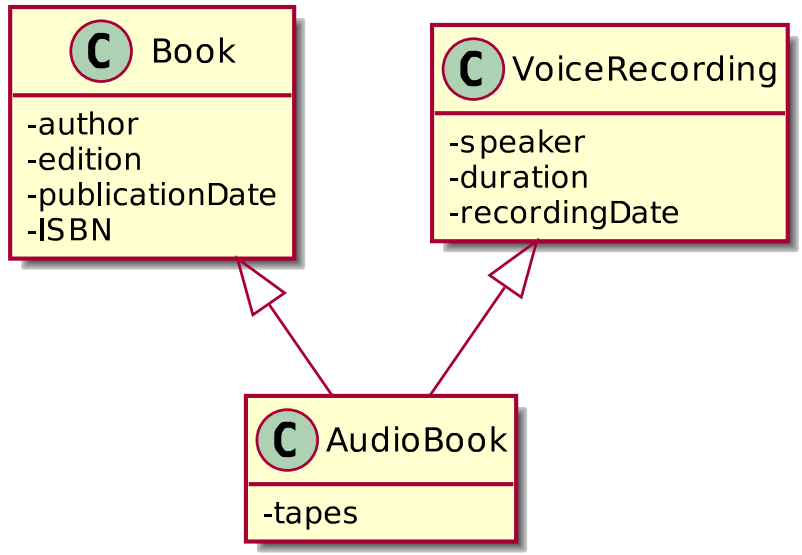



Exemplo de herança múltipla

# Herança Única × Herança Múltipla



- Herança múltipla é quando uma subclasse pode herdar de mais de uma subclasse direta.
- Na Herança única, uma subclasse herda somente de uma superclasse direta.




Exemplo de herança múltipla

- C++ permite herança múltipla, porém Java não permite herança múltipla.
- No entanto, é possível utilizar interfaces para desfrutar de alguns dos benefícios da herança múltipla.

# Problemas com a herança



- Um problema com a herança é que a subclasse pode herdar métodos que não precisa ou que não deveria ter.
  - Solução: Podemos declarar um método como `final` a fim de forçar com que este método não seja herdado pelas subclasses de uma superclasse.

# Problemas com a herança



- Um problema com a herança é que a subclasse pode herdar métodos que não precisa ou que não deveria ter.
  - Solução: Podemos declarar um método como `final` a fim de forçar com que este método não seja herdado pelas subclasses de uma superclasse.

- Um outro problema é que o método herdado pode ser necessário na subclasse, mas inadequado.
  - Solução: A classe pode sobrescrever/sobrepor (*override*) um método herdado para adequá-lo.
  - Exemplo: o método `toString()`

# Herança em Java



Em Java, para se estabelecer que uma classe é herdeira de outra, após o nome da subclasse que está sendo declarada coloca-se a cláusula `extends` seguido do nome da superclasse. Por exemplo:

```
class Funcionario extends Pessoa {...}
```

# Herança em Java



Em Java, para se estabelecer que uma classe é herdeira de outra, após o nome da subclasse que está sendo declarada coloca-se a cláusula `extends` seguido do nome da superclasse. Por exemplo:

```
class Funcionario extends Pessoa {...}
```

- Com o mecanismo de herança, podemos declarar a classe `Funcionario` como sendo um tipo de `Pessoa`, e a classe `Funcionario` herdará todos os campos e métodos da classe `Pessoa`, não sendo necessária a sua redeclaração.

# Herança em Java



Em Java, para se estabelecer que uma classe é herdeira de outra, após o nome da subclasse que está sendo declarada coloca-se a cláusula `extends` seguido do nome da superclasse. Por exemplo:

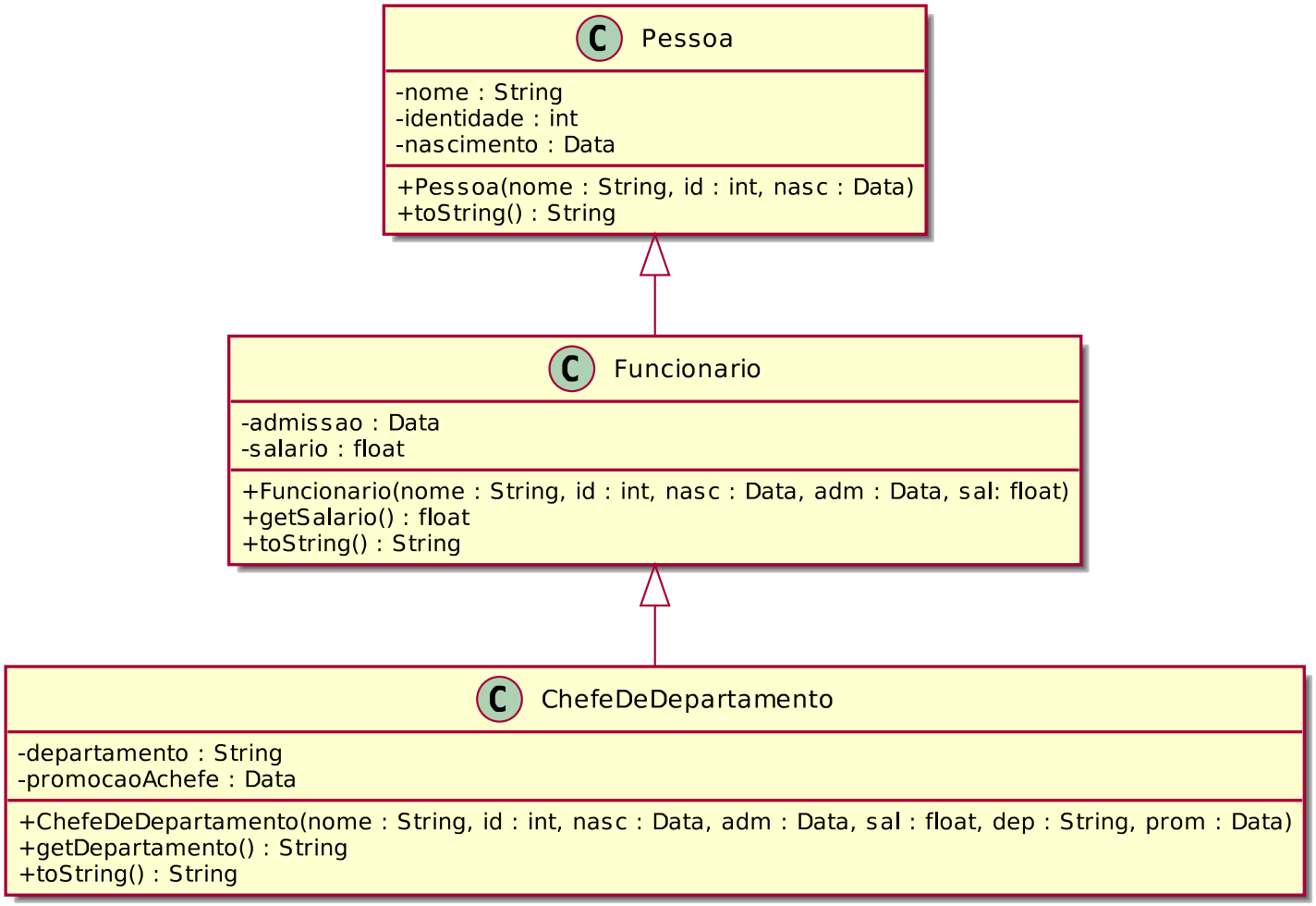
```
class Funcionario extends Pessoa {...}
```

- Com o mecanismo de herança, podemos declarar a classe `Funcionario` como sendo um tipo de `Pessoa`, e a classe `Funcionario` herdará todos os campos e métodos da classe `Pessoa`, não sendo necessária a sua redeclaração.

- **Atenção:** Os atributos privados são herdados, mas, como só podem ser acessados e modificados pelas classes que os declararam diretamente, não podem ser acessados diretamente pela subclasse.

# Exemplo de herança






Analisar o ProjetoHerança01

# Herança e Construtores



- A subclasse `Funcionario` invoca o construtor da superclasse explicitamente através da instrução

```
super(nome, id, nasc);
```

# Herança e Construtores



- A subclasse `Funcionario` invoca o construtor da superclasse explicitamente através da instrução

```
super(nome, id, nasc);
```

- Construtores não são herdados.
  - A primeira tarefa de qualquer construtor é invocar o construtor da superclasse direta de forma implíta ou explícita.

- Se não houver uma chamada explícita ao construtor da superclasse direta, o compilador invoca o construtor default.
  - *Construtor default*: construtor sem argumentos.
  - **Atenção:** Se a superclasse direta não tiver um construtor default, o compilador lançará uma exceção.

# A anotação @Override



- Para sobrescrever um método de superclasse, uma subclasse deve declarar um método com a mesma assinatura (nome de método, número de parâmetros, tipos de parâmetro e ordem dos tipos de parâmetro), como o método de superclasse.

- O método `toString` da classe `Pessoa` sobrescreve (redefine) o método toString da classe Object.

- o método `toString` de `Object` não aceita nenhum parâmetro, então `Pessoa` e as demais classes declaram `toString` sem parâmetros.

- Nas classes do exemplo anterior, usamos a anotação `@Override` opcional antes da declaração do método `toString` para indicar que a declaração atual do toString deve sobrescrever o método toString da superclasse existente.
  - Essa anotação ajuda o compilador a capturar alguns erros comuns.

# A anotação @Override

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

**Dica de prevenção de erro:** Embora seja opcional, declare métodos sobrescritos com `@Override` para assegurar em tempo de compilação que suas assinaturas foram definidas corretamente.

Sempre é melhor encontrar erros em tempo de compilação em vez de em tempo de execução.

# A palavra-chave `super`

# A palavra-chave `super`



- As subclasses podem ter acesso a métodos das superclasses, usando a palavra-chave `super`.

- O acesso a métodos de classes ancestrais é útil para aumentar a reutilização de código.
  - **Atenção:** Métodos `private` não são acessíveis nas subclasses.

# A palavra-chave `super`



- Existem duas maneiras de se reutilizar métodos de classes que não tenham sido declarados como `private`:

# A palavra-chave `super`



- Existem duas maneiras de se reutilizar métodos de classes que não tenham sido declarados como `private`:
  1. Se a execução do método for a mesma para a superclasse e a subclasse, então instâncias da subclasse podem chamar diretamente o método como se fosse delas mesmas — é o caso do método `getSalario()` definido na classe Funcionario, e que também pode ser invocado por instâncias da subclasse ChefeDeDepartamento.

# A palavra-chave `super`



- Existem duas maneiras de se reutilizar métodos de classes que não tenham sido declarados como `private`:
  1. Se a execução do método for a mesma para a superclasse e a subclasse, então instâncias da subclasse podem chamar diretamente o método como se fosse delas mesmas — é o caso do método `getSalario()` definido na classe Funcionario, e que também pode ser invocado por instâncias da subclasse ChefeDeDepartamento.
  2. Se um método na classe ancestral realiza operações necessárias, é preferível que ele seja chamado, ao invés de duplicarmos o código. Isso reduz a manutenção de código.
     - No exemplo, o método toString da classe ChefeDeDepartamento invoca o método toString da classe Funcionario; este, por sua vez, invoca o método toString da classe Pessoa.
     - O comando usado para invocar em ambos os casos foi: `super.toString()`

# Regras para uso de `super`



Algumas regras para uso da palavra-chave `super` para chamar métodos de classes ancestrais como sub-rotinas são:

1. Construtores são chamados pela palavra-chave `super` seguida dos argumentos a serem passados para o construtor entre parênteses. Se não houver argumentos, a chamada deve ser feita como `super()`.

# Regras para uso de `super`



Algumas regras para uso da palavra-chave `super` para chamar métodos de classes ancestrais como sub-rotinas são:

1. Construtores são chamados pela palavra-chave `super` seguida dos argumentos a serem passados para o construtor entre parênteses. Se não houver argumentos, a chamada deve ser feita como `super()`.
   - **Obs.:** O construtor de uma subclasse **SEMPRE** chama o construtor de uma superclasse, mesmo que a chamada não seja explícita.

     Quando a chamada não é explícita (através da palavra-chave super), o construtor chamado é o construtor vazio (sem argumentos) – se este construtor não estiver definido, haverá um erro de compilação.

# Regras para uso de `super`



2. Construtores de superclasses só podem ser chamados de dentro de construtores de subclasses e **DEVEM** ser declarados na primeira linha de código do construtor da subclasse.
   - **Obs.:** Métodos não podem chamar construtores de superclasses.

# Regras para uso de `super`



2. Construtores de superclasses só podem ser chamados de dentro de construtores de subclasses e **DEVEM** ser declarados na primeira linha de código do construtor da subclasse.
   - **Obs.:** Métodos não podem chamar construtores de superclasses.

3. Métodos são chamados pela palavra-chave `super` seguida de um ponto e do nome do método (seguido dos possíveis argumentos).
   - Exemplo: `super.toString()`

# Regras para uso de `super`



2. Construtores de superclasses só podem ser chamados de dentro de construtores de subclasses e **DEVEM** ser declarados na primeira linha de código do construtor da subclasse.
   - **Obs.:** Métodos não podem chamar construtores de superclasses.

3. Métodos são chamados pela palavra-chave `super` seguida de um ponto e do nome do método (seguido dos possíveis argumentos).
   - Exemplo: `super.toString()`

4. Somente os métodos e construtores da superclasse direta podem ser chamados usando a palavra-chave `super`

# Regras para uso de `super`



2. Construtores de superclasses só podem ser chamados de dentro de construtores de subclasses e **DEVEM** ser declarados na primeira linha de código do construtor da subclasse.
   - **Obs.:** Métodos não podem chamar construtores de superclasses.

3. Métodos são chamados pela palavra-chave `super` seguida de um ponto e do nome do método (seguido dos possíveis argumentos).
   - Exemplo: `super.toString()`

4. Somente os métodos e construtores da superclasse direta podem ser chamados usando a palavra-chave `super`

5. Se um método de uma classe ancestral for herdado pela classe descendente, ele pode ser chamado diretamente sem necessidade da palavra `super`.

# Modificadores de acesso

# Modificadores de acesso



- **public:** Os membros `public` de uma classe são acessíveis em qualquer parte de um programa em que haja uma referência a um objeto da classe ou das subclasses.

- **private:** Membros `private` são acessíveis apenas dentro da própria classe.

- **protected:** Membros `protected` podem ser acessados por por membros da própria classe, de subclasses e de classes do mesmo pacote.

# Modificadores de acesso

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

| | Atributos e métodos com visibilidade: | | | |
|---|---|---|---|---|
| **Classes que têm acesso** | **private** | **protected** | ***default*** | **public** |
| A mesma classe | sim | sim | sim | sim |
| Classes herdeiras | não | sim | sim* | sim |
| Demais classes no mesmo pacote | não | sim | sim | sim |
| Demais classes em outro pacote | não | não | não | sim |

* se estiverem no mesmo pacote que a superclasse

# Sobreposição e Ocultação

# Sobreposição



- **Definição:** Sobreposição ou superposição: é a declaração de métodos com a mesma assinatura que métodos de classes ancestrais.

# Sobreposição



- **Definição:** Sobreposição ou superposição: é a declaração de métodos com a mesma assinatura que métodos de classes ancestrais.
  - A razão de sobrepormos métodos é que métodos de classes herdeiras geralmente executam tarefas adicionais que os métodos das classes ancestrais não executam.
  - Um exemplo comum são os métodos que imprimem atributos na tela.

# Ocultação



- **Definição:** Ocultação é a declaração de atributos em uma classe descendente com o mesmo nome de atributos declarados na classe ancestral.

- Ao contrário da sobreposição de métodos, que é bastante útil e comum em classes herdeiras, a ocultação de atributos não oferece muitas vantagens, e as poucas oferecidas podem facilmente ser implementadas através de métodos que retornam valores e são superpostos de acordo com a necessidade.

# Regras para sobreposição de métodos



1. A sobreposição de um método em uma subclasse não elimina o acesso ao método de mesma assinatura na classe ancestral – este pode ser acessado, de dentro da classe herdeira, com a palavra-chave `super`, contanto que não tenha sido declarado como `private`.

# Regras para sobreposição de métodos



1. A sobreposição de um método em uma subclasse não elimina o acesso ao método de mesma assinatura na classe ancestral – este pode ser acessado, de dentro da classe herdeira, com a palavra-chave `super`, contanto que não tenha sido declarado como `private`.

2. Métodos declarados em uma subclasse com o mesmo nome mas assinaturas diferentes (por exemplo, número de argumentos diferentes) dos métodos da superclasse não sobrepõem estes métodos.

# Regras para sobreposição de métodos



1. A sobreposição de um método em uma subclasse não elimina o acesso ao método de mesma assinatura na classe ancestral – este pode ser acessado, de dentro da classe herdeira, com a palavra-chave `super`, contanto que não tenha sido declarado como `private`.

2. Métodos declarados em uma subclasse com o mesmo nome mas assinaturas diferentes (por exemplo, número de argumentos diferentes) dos métodos da superclasse não sobrepõem estes métodos.

3. Métodos podem ser sobrepostos com diferentes modificadores de acesso, contanto que os métodos sobrepostos tenham modificadores de acesso menos restritivos.
   - Exemplo: podemos declarar um método na superclasse com o modificador de acesso private e sobrepor este método em uma subclasse com o modificador de acesso public, mas não podemos fazer o contrário.

# Regras para sobreposição de métodos



4. Métodos estáticos declarados em classes ancestrais não podem ser sobrepostos em classes descendentes, nem mesmo se não forem declarados como estáticos.

5. Qualquer método da classe herdeira pode chamar qualquer método da classe ancestral que tenha sido declarado como `public`, `protected` ou sem declaração explícita de modificador. Métodos declarados como `private` não são acessíveis diretamente.

# Regras para sobreposição de métodos



6. Métodos declarados como `final` são herdados por subclasses, mas não podem ser sobrepostos (a não ser que a sua assinatura seja diferente). Por exemplo, a classe `ChefeDeDepartamento`, vista anteriormente, não pode declarar um método `getSalario` pois este foi declarado como `final` na classe ancestral `Funcionario`.

# Regras para sobreposição de métodos



6. Métodos declarados como `final` são herdados por subclasses, mas não podem ser sobrepostos (a não ser que a sua assinatura seja diferente). Por exemplo, a classe `ChefeDeDepartamento`, vista anteriormente, não pode declarar um método `getSalario` pois este foi declarado como `final` na classe ancestral `Funcionario`.

7. Se um atributo é declarado em uma superclasse e oculto em subclasses, e métodos que acessam este campo são herdados, estes métodos farão referência ao campo da superclasse onde foram declarados.

Analisar o ProjetoHerança02

# Atributos, Métodos e Classes `final`

# Atributos, Métodos e classes `final`



- Uma variável ou atributo declarado com o modificador `final` é constante
  - Ou seja, depois de inicializada não pode ser modificada.
- Um método declarado com o modificador `final` não pode ser sobrescrito.
- Uma classe declarada com o modificador `final` não pode ser herdada.
  - A declaração de uma classe como final efetivamente impede o mecanismo de herança — o compilador não compilará uma classe declarada como herdeira de uma classe final.

# Atributos, Métodos e classes `final`



- Uma variável ou atributo declarado com o modificador `final` é constante
  - Ou seja, depois de inicializada não pode ser modificada.
- Um método declarado com o modificador `final` não pode ser sobrescrito.
- Uma classe declarada com o modificador `final` não pode ser herdada.
  - A declaração de uma classe como final efetivamente impede o mecanismo de herança — o compilador não compilará uma classe declarada como herdeira de uma classe final.

Analisar o ProjetoHerança03

# Polimorfismo

# Polimorfismo



- Na herança, vimos que todo `ChefeDeDepartamento` é um tipo de `Funcionario`, pois é uma extensão deste.
  - Podemos nos referir a um `ChefeDeDepartamento` como sendo um `Funcionario`.

# Polimorfismo



- Na herança, vimos que todo `ChefeDeDepartamento` é um tipo de `Funcionario`, pois é uma extensão deste.
  - Podemos nos referir a um `ChefeDeDepartamento` como sendo um `Funcionario`.

- A relação é-um-tipo-de entre classes permite a existência de outra característica fundamental de linguagens de programação orientadas a objetos: *polimorfismo*.

**Definição:** Em programação orientada a objetos, polimorfismo é a capacidade de uma referência de classe se associar a instâncias de diferentes classes em tempo de execução.

# Polimorfismo



Usando herança, podemos escrever métodos que recebam instâncias de uma superclasse $C$, e os mesmos métodos serão capazes de processar instâncias de qualquer classe que herde da classe $C$, já que qualquer classe que herde de $C$ é um tipo de $C$.

# Polimorfismo



Usando herança, podemos escrever métodos que recebam instâncias de uma superclasse $C$, e os mesmos métodos serão capazes de processar instâncias de qualquer classe que herde da classe $C$, já que qualquer classe que herde de $C$ é um tipo de $C$.

Analisar o arquivo ConcessionariaDeAutomoveis.java do ProjetoHerança02

# Exercício



- Crie uma classe **Equipamento** com o atributo ligado (tipo boolean) e com os métodos *liga* e *desliga*. O método *liga()* torna o atributo ligado true e o método *desliga()* torna o atributo ligado false.

- Crie também uma classe **EquipamentoSonoro** que herda as características de **Equipamento** e que possui os atributos volume (tipo short) que varia de 0 a 10 e stereo (tipo boolean).

- A classe ainda deve possuir métodos getters e setters, além dos métodos *mono()* e *stereo()*. O método *mono()* torna o atributo stereo falso e o método *stereo()* torna o atributo stereo verdadeiro. Ao ligar o **EquipamentoSonoro** através do método liga, seu volume é automaticamente ajustado para 5.

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

# FIM