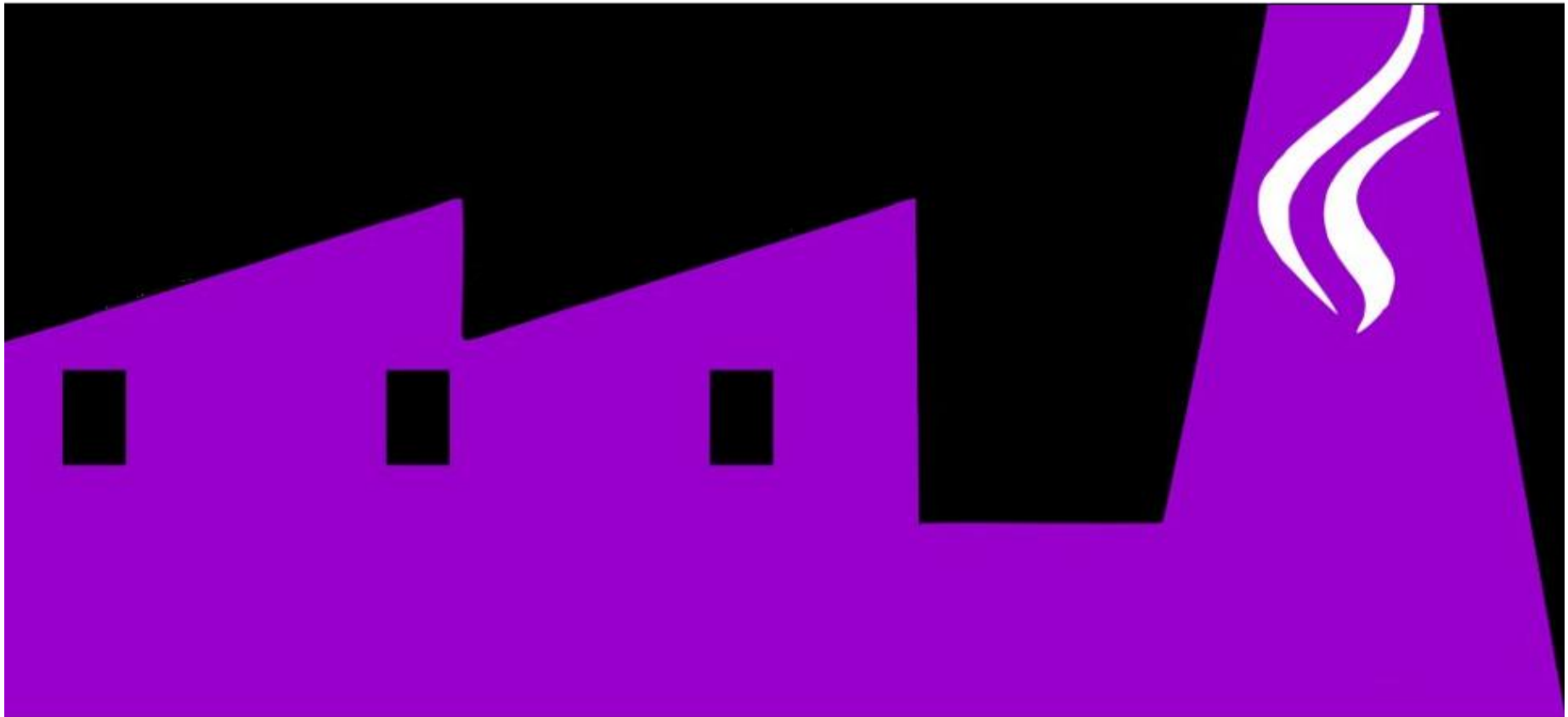


Fábrica de Software

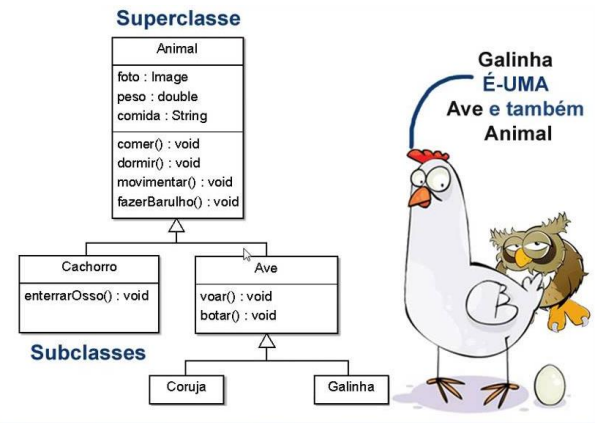
Profes. Ivan L. Süptitz e Evandro Franzen

Herança de Classes (reuso de código, reescrita)



Sumário – O que veremos nesta aula?

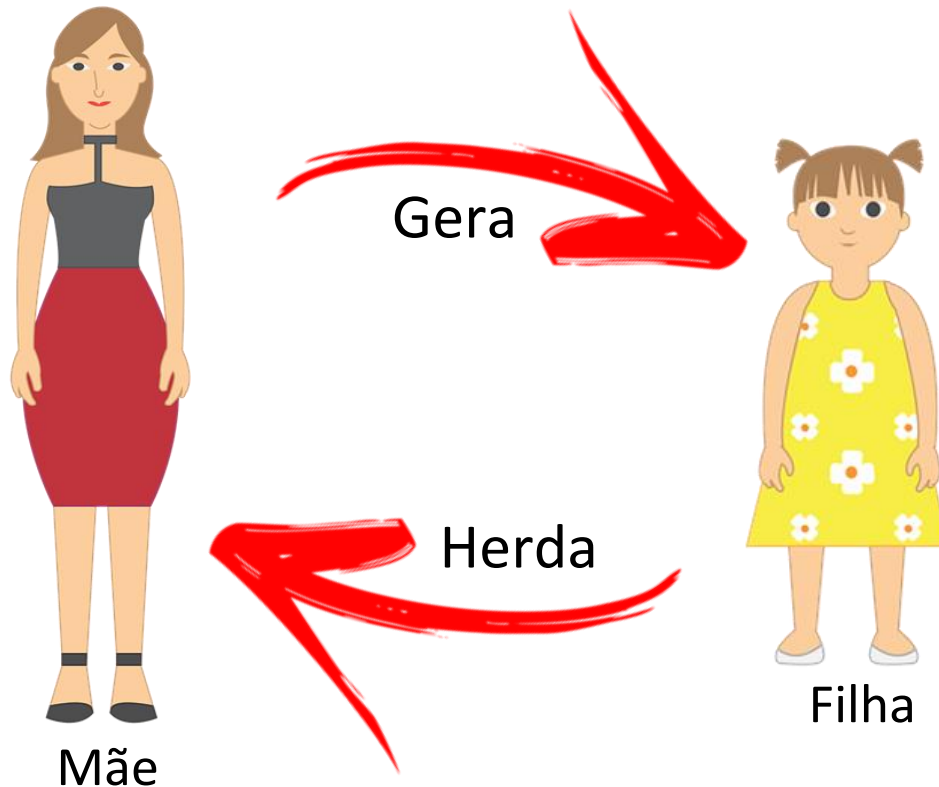
- Herança de classes
- Mãe (superclasse), filha (subclasse),
- Modificador de visibilidade protegido
- Reescrita de método: @Override
- O comando “super”
- Exercícios



Objetivos de aprendizagem:

Conhecer o conceito de Herança em POO e compreender sua utilidade no reuso de software. Entender e aplicar a técnica de reescrita

Herança de classe – O que é?



Herança

Características da mãe
Comportamentos da mãe

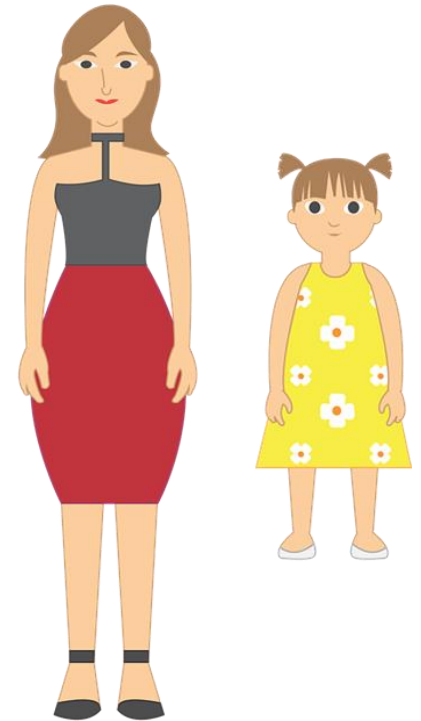
Herança de classe – O que é?

Conceito:

“permitir criar uma nova classe com a definição de uma outra classe previamente existente.”

“Relacionar uma classe de tal maneira que uma delas herda tudo que a outra tem”

“A herança será aplicada tanto para as características quanto para os comportamentos.”



O que estas classes têm em comum (compartilhado)?

Aluno

- nome
- idade
- sexo
- matricula
- curso

+ fazerAniversario ()
+ cancelarMatricula ()



Professor

- nome
- idade
- sexo
- especialidade
- salário

+ fazerAniversario ()
+ receberAumento ()



Funcionario

- nome
- idade
- sexo
- setor
- trabalhando

+ fazerAniversario ()
+ mudarTrabalho ()



O que estas classes têm em comum (compartilhado)?

Aluno	Professor	Funcionario
<ul style="list-style-type: none">- nome- idade- sexo	<ul style="list-style-type: none">- nome- idade- sexo	<ul style="list-style-type: none">- nome- idade- sexo
<ul style="list-style-type: none">- matricula- curso	<ul style="list-style-type: none">- especialidade- salário	<ul style="list-style-type: none">- setor- trabalhando
<ul style="list-style-type: none">+ fazerAniversario ()+ cancelarMatricula ()	<ul style="list-style-type: none">+ fazerAniversario ()+ receberAumento ()	<ul style="list-style-type: none">+ fazerAniversario ()+ mudarTrabalho ()



Mais alguma coisa em
comum?

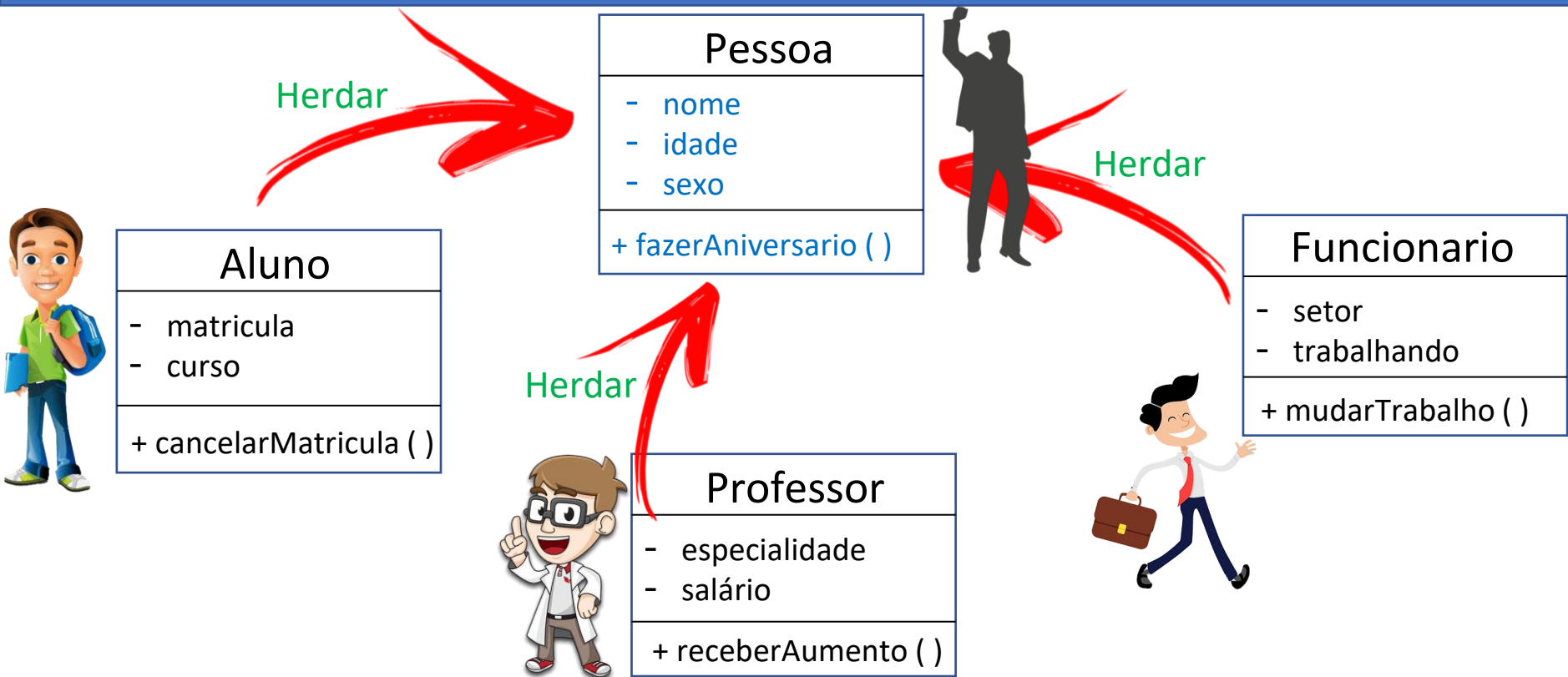


O que estas classes tem em comum (compartilhado)?

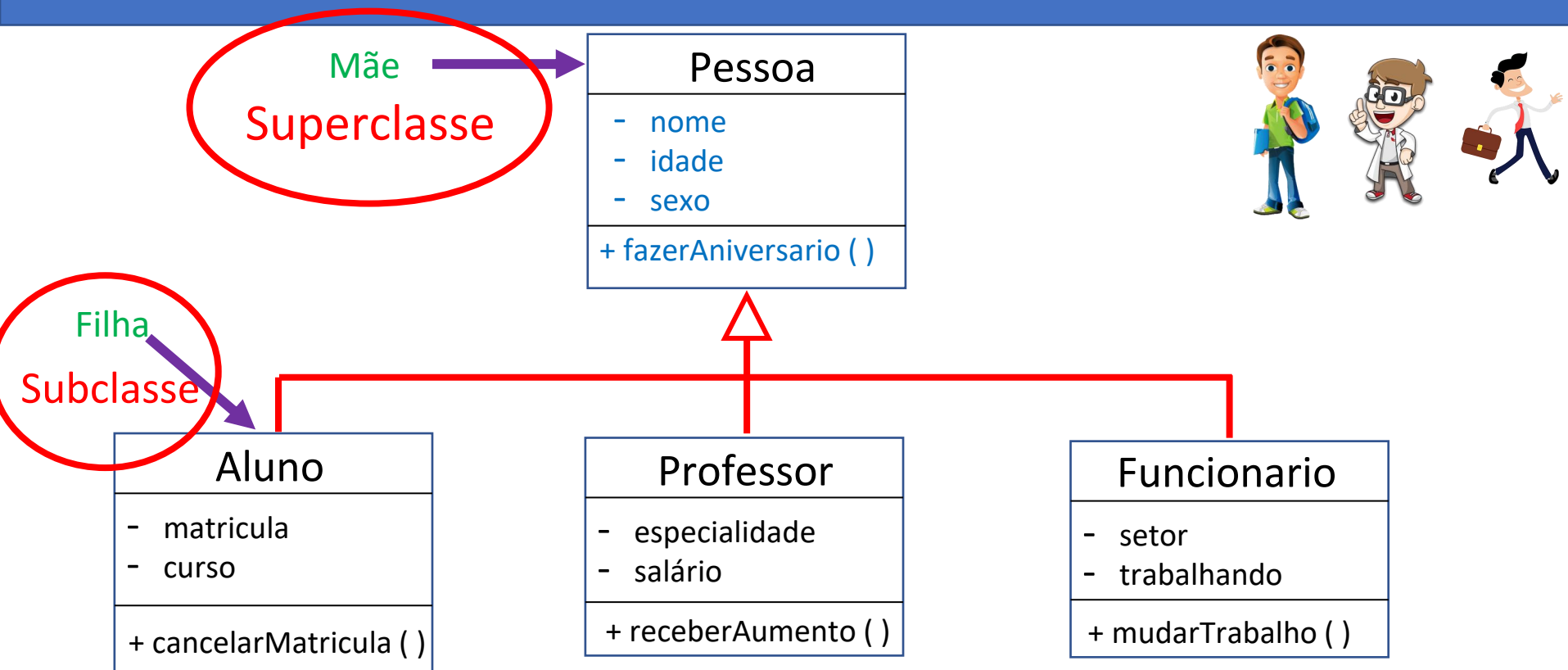
Aluno	Professor	Funcionario
<ul style="list-style-type: none">- nome- idade- sexo- matricula- curso	<ul style="list-style-type: none">- nome- idade- sexo- especialidade- salário	<ul style="list-style-type: none">- nome- idade- sexo- setor- trabalhando
+ fazerAniversario ()	+ fazerAniversario ()	+ fazerAniversario ()
+ cancelarMatricula ()	+ receberAumento ()	+ mudarTrabalho ()



Assim, criando uma classe mãe... e suas filhas...



Como ficaria em UML?



Sem o uso de herança...

```
public class Pessoa {  
    private String nome;  
    private int idade;  
    private String sexo;  
  
    public void fazerAniversario() {  
        this.idade++;  
    }  
}
```

```
public class Aluno {  
    private String nome;  
    private int idade;  
    private String sexo;  
    private int matricula;  
    private String curso;  
  
    public void fazerAniversario() {  
        this.idade++;  
    }  
  
    public void cancelarMatricula() {  
        System.out.println("Matricula será cancelada!!!");  
    }  
}
```

Com o uso de herança?

```
public class Pessoa {  
    private String nome;  
    private int idade;  
    private String sexo;  
  
    public void fazerAniversario()  
    {  
        this.idade++;  
    }  
}
```

?

```
public class Aluno {  
    private int matricula;  
    private String curso;  
  
    public void cancelarMatricula() {  
        System.out.println("Matricula será cancelada!!!");  
    }  
  
    public int getMatricula() {  
        return this.matricula;  
    }  
  
    public void setMatricula(int matricula) {  
        this.matricula = matricula;  
    }  
  
    public String getCurso() {  
        return this.curso;  
    }  
  
    public void setCurso(String curso) {  
        this.curso = curso;  
    }  
}
```

Estende

```
public class Aluno extends Pessoa {  
    private int matricula;  
    private String curso;  
  
    public void cancelarMatricula() {  
        System.out.println("Matricula será cancelada!!!");  
    }  
  
    public int getMatricula() {  
        return this.matricula;  
    }  
  
    public void setMatricula(int matricula) {  
        this.matricula = matricula;  
    }  
  
    public String getCurso() {  
        return this.curso;  
    }  
  
    public void setCurso(String curso) {  
        this.curso = curso;  
    }  
}
```

* Fazemos isto através da palavra chave *extends*.

Reescrita de método - @override


Quando herdamos um método, podemos alterar seu comportamento.

Assim, podemos reescrever (reescrever, sobrescrever, *override*) este método.


Como assim?

Reescrita de método - @override

```
public class Pessoa {  
    private String nome;  
    private int idade;  
    private String sexo;  
  
    public String getInfo(){  
        return "Nome: "+this.getNome()+  
            ", idade: "+this.getIdade();  
    }  
}
```



```
public class Aluno extends Pessoa{  
    private int matricula;  
    private String curso;  
  
    @Override  
    public String getInfo(){  
        return "Nome: "+this.getNome()+  
            ", idade: "+this.getIdade()+  
            ", curso : "+this.getCurso();  
    }  
  
    public void cancelarMatricula(){  
        System.out.println("Matricula será cancelada!!!");  
    }  
}
```



E se precisar incluir uma informação no método getInfo do Aluno? Teria que criar um novo método? Não, uso a **reescrita**!

Quando herdamos um método, podemos alterar seu comportamento: Podemos reescrever (reescrever, sobrescrever, *override*) este método.

Reescrita de método - @override

OK, mas qual a diferença para a sobrecarga?

A diferença é que a assinatura do método PRECISA SER IGUAL, enquanto que na sobrecarga apenas o nome é igual mas a assinatura precisa ser diferente

Palavra chave - Super

Ao utilizar a **reescrita**, **caso** precisássemos acessar o método da classe mãe, **para não** ter que copiar e colocar o conteúdo desse método **e depois** incluir alguma informação extra, utilizamos a palavra:

super

Palavra chave Super

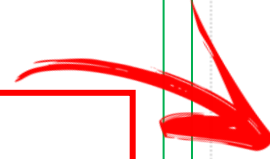
```
public class Aluno extends Pessoa{
    private int matricula;
    private String curso;

    @Override
    public String getInfo(){
        return "Nome: "+this.getNome()+
            ", idade: "+this.getIdade()+
            ", curso : "+this.getCurso();
    }

    public void cancelarMatricula(){
        System.out.println("Matricula será cancelada!!!");
    }
}
```

```
public class Aluno extends Pessoa{
    private int matricula;
    private String curso;

    @Override
    public String getInfo(){
        return super.getInfo() +
            ", curso : "+this.getCurso();
    }
}
```



Com isso eu evito o problema que seria gerado se um dia precisar adicionar algum dado relevante no getInfo da Pessoa e esquecer de alterar o método da classe Aluno.

Modificadores de visibilidade

Lembram quais são?

Lembram qual a diferença?

Protected agora tem um papel especial, permitindo que as classes "filhas" acessem métodos e atributos da classe "mãe".

Exercícios

5.1. Pensar e Implementar um programa usando uma classe mãe Animal (com dois atributos e dois métodos).

- a. Esta classe terá duas filhas, cada uma com ao menos um atributo novo. Cada filha deverá gerar outras classes (de livre escolha com ao menos um atributo e um método cada).
- b. Demonstrar um uso da reescrita de ao menos um método da classe mãe. Demonstrar também o uso da palavra chave super.
- c. Criar um programa de teste (main) para testar os métodos e imprimir os resultados.

Exercícios

- 5.2.** Crie uma classe chamada Ingresso que possui um valor em reais e um método retornaValor()
- a. crie uma classe IngressoNormal, que herda de Ingresso e possui um método toString que retorna: "Ingresso Normal" e o seu valor;
 - b. crie uma classe IngressoVip que é mais caro (possui valor adicional de 50% se for CAMAROTE_SUPERIOR e 70 % se for CAMAROTE_INFERIOR (utilize enum). Essa classe também herda de Ingresso e possui ainda a localização do ingresso. Crie um método que retorne o valor do ingresso VIP (com o adicional incluído).
 - c. Ambas as classes devem ter um método toString que retorna uma descrição adequada.

Exercícios

5.3. Criar uma classe Pessoa:

- a. Os atributos devem ser encapsulados, com seus respectivos seletores (getters) e modificadores (setters), e ainda o construtor padrão e pelo menos mais duas opções de construtores conforme sua percepção. Atributos: String nome; String endereco; String telefone;
- b. Considere, como subclasse da classe Pessoa (desenvolvida no exercício anterior) a classe Fornecedor. Considere que cada instância da classe Fornecedor tem, para além dos atributos que caracterizam a classe Pessoa, os atributos valorCredito (correspondente ao crédito máximo atribuído ao fornecedor) e valorDivida (montante da dívida para com o fornecedor). Implemente na classe Fornecedor, para além dos usuais métodos seletores e modificadores, um método obterSaldo() que devolve a diferença entre os valores dos atributos valorCredito e valorDivida. Depois de implementada a classe Fornecedor, crie um programa de teste adequado que lhe permita verificar o funcionamento dos métodos implementados na classe Fornecedor e os herdados da classe Pessoa.

Exercícios

5.3.

c. Considere, como subclasse da classe Pessoa, a classe Empregado. Considere que cada instância da classe Empregado tem, para além dos atributos que caracterizam a classe Pessoa, os atributos `codigoSetor` (inteiro), `salarioBase` (vencimento base) e `imposto` (porcentagem retida dos impostos). Implemente a classe Empregado com métodos seletores e modificadores e um método `calcularSalario`. Escreva um programa de teste adequado para a classe Empregado.

d. Implemente a classe Administrador como subclasse da classe Empregado. Um determinado administrador tem como atributos, para além dos atributos da classe Pessoa e da classe Empregado, o atributo `ajudaDeCusto` (ajudas referentes a viagens, estadias, ...). Note que deverá redefinir na classe Administrador o método herdado `calcularSalario` (o salário de um administrador é equivalente ao salário de um empregado usual acrescido das ajuda de custo). Escreva um programa de teste adequado para esta classe.

Exercícios

5.3.

e. Implemente a classe Operario como subclasse da classe Empregado. Um determinado operário tem como atributos, para além dos atributos da classe Pessoa e da classe Empregado, o atributo valorProducao (que corresponde ao valor monetário dos artigos efetivamente produzidos pelo operário) e comissao (que corresponde à percentagem do valorProducao que será adicionado ao vencimento base do operário). Note que deverá redefinir nesta subclasse o método herdado calcularSalario (o salário de um operário é equivalente ao salário de um empregado usual acrescido da referida comissão). Escreva um programa de teste adequado para esta classe.

f. Implemente a classe Vendedor como subclasse da classe Empregado. Um determinado vendedor tem como atributos, para além dos atributos da classe Pessoa e da classe Empregado, o atributo valorVendas (correspondente ao valor monetário dos artigos vendidos) e o atributo comissao (percentagem do valorVendas que será adicionado ao vencimento base do Vendedor). Note que deverá redefinir nesta subclasse o método herdado calcularSalario (o salário de um vendedor é equivalente ao salário de um empregado usual acrescido da referida comissão). Escreva um programa de teste adequado para esta classe.

Exercícios

5.4.

- Criar uma superclasse Funcionario (nome, cpf, salario), no mínimo dois métodos.
- Criar quatro subclasses (Gerente, Engenheiro, Secretaria, Diretor), criar no mínimo dois métodos específicos para cada subclasse.

Regra 1: Todo ano os funcionários do banco recebem uma bonificação. Os funcionários comuns recebem 10% do salário e os gerentes 15%. (criar o método getBonificacao() na classe Funcionario(mae)).

- Como fazer para o gerente receber 15% ?

OBS: (uma opção é criar o método na getBonificaçãoGerente() na classe gerente. **Qual o problema?** Dois métodos de bonificação em gerente (confunde quem usar o método, gera dois resultados diferente). Resolver o problema com reescrita (override)

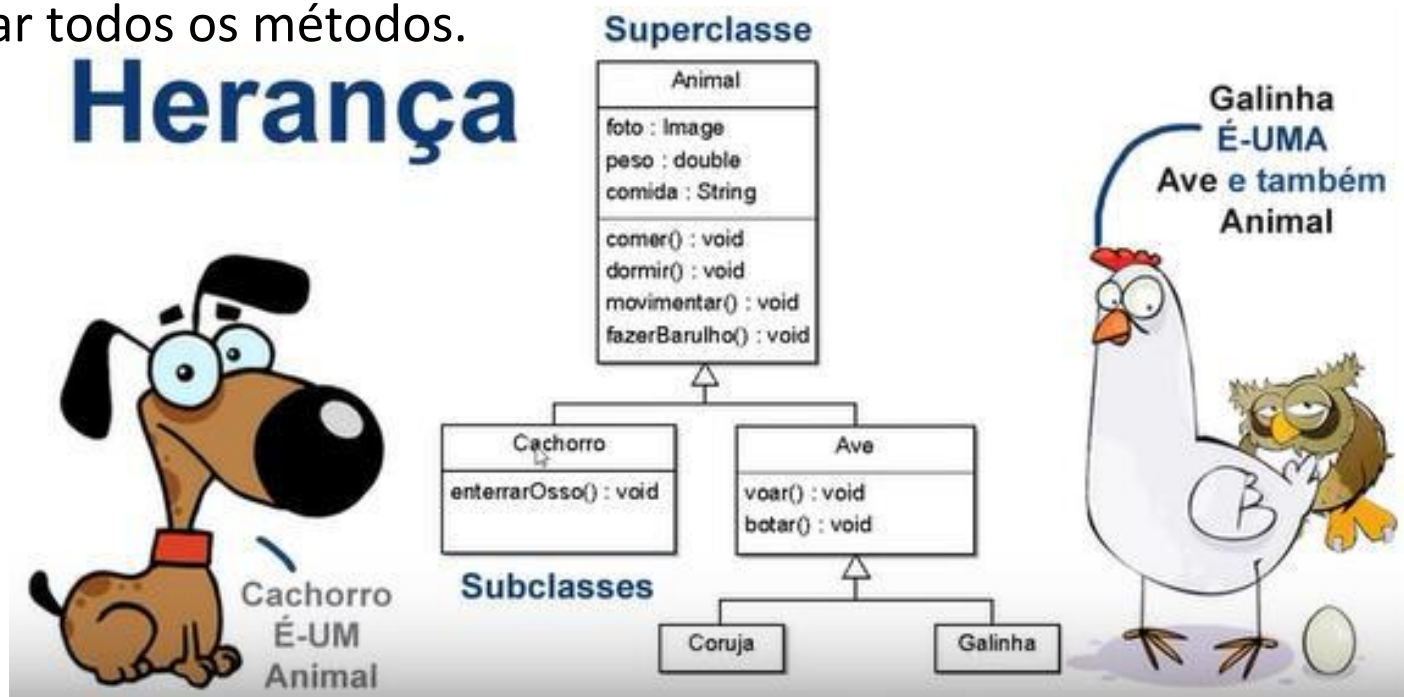
Regra 2: Mudar a forma de calcular a bonificação do gerente, agora é da mesma forma que o funcionário mas adicionando mais R\$ 1000,00. Já para a secretária, adicionar mais R\$ 100,00, para o diretor mais R\$ 2000,00 , para o engenheiro R\$ 550,00. **Como fazer isso** (faça da mesma forma sugerida na questão 1, @override)?

Regra 3: Aumentar o percentual de bonificação do funcionário para 20%, **como ficaria o código?**

Regra 4: Usando a palavra super, modifique seu código nos métodos de bonificação (depois altere a bonificação do funcionário para 30%), **consegue ver a diferença e em quantos lugares precisou modificar seu código?**

Exercícios

5.5. Implementar o diagrama abaixo e criar um programa de teste para testar todos os métodos.



Referências

<https://app.diagrams.net/>