

DCC205 – PROGRAMAÇÃO ORIENTADA A OBJETOS

Aula 08 – Classes Abstratas

Carlos Bruno Oliveira Lopes
carlosbrunocb@gmail.com

Vamos recordar
em como pode
estar nossa
classe
Funcionario:

```
class Funcionario {  
  
    protected String nome;  
    protected String cpf;  
    protected double salario;  
  
    public double getBonificacao() {  
        return this.salario * 1.2;  
    }  
  
    // outros métodos aqui  
  
}
```

Considere o nosso ControleDeBonificacao:

```
class ControleDeBonificacoes {  
  
    private double totalDeBonificacoes = 0;  
  
    public void registra(Funcionario f) {  
        System.out.println("Adicionando bonificação do funcionario: " + f);  
        this.totalDeBonificacoes += f.getBonificacao();  
    }  
  
    public double getTotalDeBonificacoes() {  
        return this.totalDeBonificacoes;  
    }  
}
```

Um pouco mais de herança e polimorfismo...

- Porém, em alguns sistemas (como é o nosso caso) usamos uma classe com apenas esses intuitos:
 - Reutilizar trechos de código (herança)
 - Ganhar polimorfismo para criar métodos mais genéricos, que se encaixem a diversos objetos.
- Ou seja, não queremos receber um objeto do tipo Funcionario, mas sim que aquela referência seja ou um Gerente, ou um Diretor, etc. Queremos algo mais concreto que um Funcionario.

Faz sentido ter um objeto do tipo Funcionario?

- Essa pergunta é diferente de saber se faz sentido ter uma referência do tipo Funcionario: nesse caso, faz sim e é muito útil.

```
ControleDeBonificacoes cdb = new ControleDeBonificacoes();  
Funcionario f = new Funcionario();  
cdb.adiciona(f); // faz sentido?
```

Precisamos de algo mais **concreto** que o
Funcionário...

Outro Exemplo

- Imagine a classe Pessoa e duas filhas, PessoaFisica e PessoaJuridica.
- Quando puxamos um relatório de nossos clientes (uma array de Pessoa por exemplo), queremos que cada um deles seja ou uma PessoaFisica, ou uma PessoaJuridica.
- A classe Pessoa, nesse caso, estaria sendo usada apenas para ganhar o polimorfismo e herdar algumas coisas:
 - **Não faz sentido permitir instanciá-la!!!!**

Classes abstratas

- Usadas quando não faz sentido termos instâncias de determinadas classes
- Deixar o programa mais coerente e consistente
- Utilizar o modificador **abstract** na declaração da classe

```
public abstract class Animal {  
    ...  
}
```

Classes abstratas

- Não é permitida a existência de objetos da classe se ela for abstrata

```
Animal a = new Animal();
```

Este código não
compila

- É permitido criar referências à classe

```
Animal a = new Cachorro();
```

A instância é do tipo
Cachorro

Classe Abstrata

- O que é a nossa classe Funcionario?
- Nossa empresa tem apenas diretores, gerentes, secretárias, etc.
- Ela é uma classe que apenas idealiza um tipo, um rascunho.
- Para o nosso sistema, é inadmissível que um objeto seja apenas do tipo Funcionario
- Usamos a palavra chave **abstract** para impedir que ela possa ser instanciada.

O modificador **abstract** na declaração de uma classe:

```
abstract class Funcionario {  
  
    protected double salario;  
  
    public double getBonificacao() {  
        return this.salario * 1.2;  
    }  
  
    // outros atributos e métodos comuns a todos Funcionarios  
  
}
```

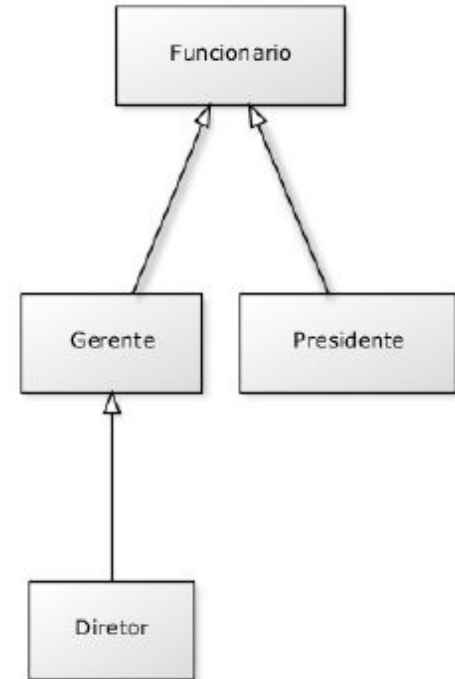
E, no meio de um código:

```
Funcionario f = new Funcionario(); // não compila!!!
```

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:  
    Cannot instantiate the type Funcionario  
  
    at br.com.caelum.empresa.TestaFuncionario.main(TestaFuncionario.java:5)
```

Vamos então herdar dessa classe, reescrevendo o método `getBonificacao`:

```
class Gerente extends Funcionario {  
  
    public double getBonificacao() {  
        return this.salario * 1.4 + 1000;  
    }  
}
```



Métodos Abstratos

- Se o método `getBonificacao` não fosse reescrito, ele seria herdado da classe mãe, fazendo com que devolvesse o salário mais 20%.
- Cada funcionário em nosso sistema tem uma regra totalmente diferente para ser bonificado.
- **Faz algum sentido ter esse método na classe `Funcionario`?**
- Queremos que cada pessoa que escreve a classe de um `Funcionario` diferente (subclasses de `Funcionario`) reescreva o método `getBonificacao` de acordo com as suas regras.

Classe abstrata

Poderíamos, então, jogar fora esse método da classe Funcionario?

- O problema é que, se ele não existisse, não poderíamos chamar o método apenas com uma referência a um Funcionario, pois ninguém garante que essa referência aponta para um objeto que possui esse método.

Será que então devemos retornar um código, como um número negativo?

- Isso não resolve o problema: se esquecermos de reescrever esse método, teremos dados errados sendo utilizados como bônus.

- Existe um recurso em Java que, em uma classe abstrata, podemos escrever que determinado método será **sempre** escrito pelas classes filhas. Isto é, um **método abstrato**.
- Ele indica que todas as classes filhas (concretas, isto é, que não forem abstratas) devem reescrever esse método ou não compilarão.
 - Para tanto basta escrever a palavra chave **abstract** na assinatura do mesmo e colocar um ponto e vírgula em vez de abre e fecha chaves:

```
abstract class Funcionario {  
  
    abstract double getBonificacao();  
  
    // outros atributos e métodos  
  
}
```

Métodos abstratos

- Repare que não colocamos o corpo do método e usamos a palavra chave **abstract** para definir o mesmo.

Por que não colocar corpo algum?

- Porque esse método nunca vai ser chamado, sempre que alguém chamar o método `getBonificacao`, vai cair em uma das suas filhas, que realmente escreveram o método.
- Qualquer classe que estender a classe `Funcionario` será obrigada a reescrever este método, tornando-o “concreto”

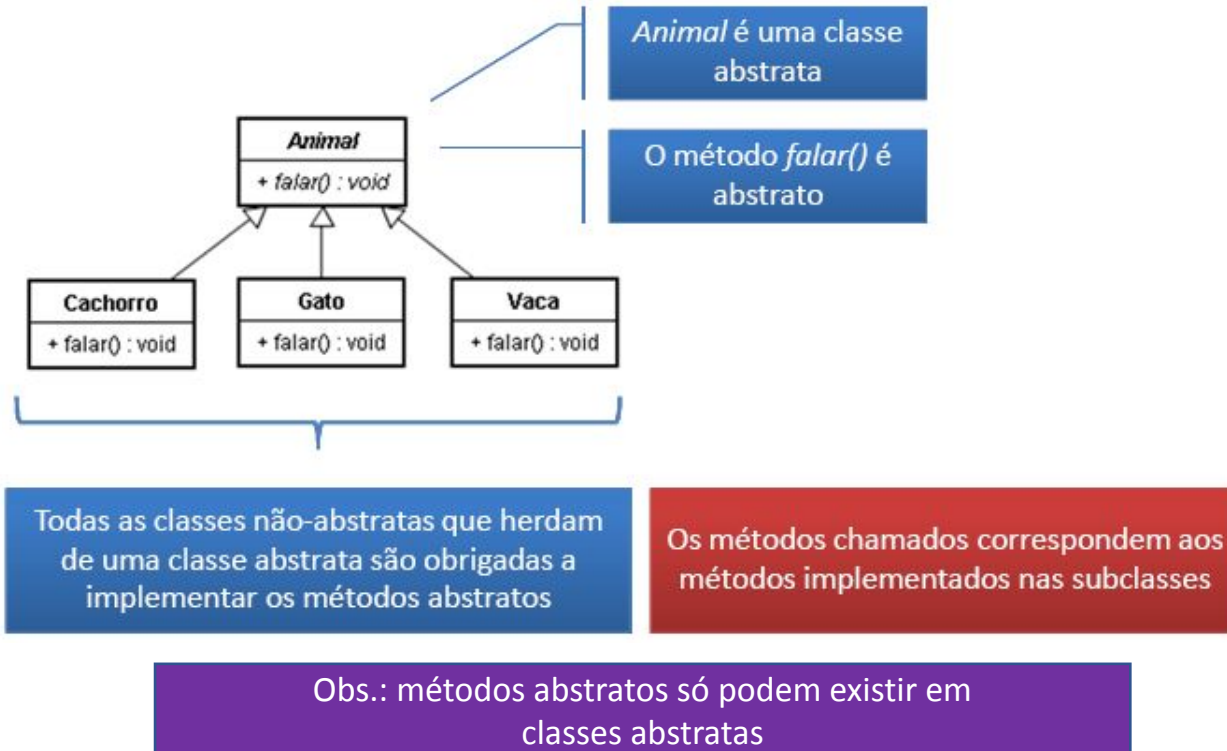
O método do ControleDeBonificacao estava assim:

```
public void registra(Funcionario f) {  
    System.out.println("Adicionando bonificação do funcionario: " + f);  
    this.totalDeBonificacoes += f.getBonificacao();  
}
```

Como posso acessar o método getBonificacao se ele não existe na classe Funcionario?

- Já que o método é abstrato, **com certeza** suas subclasses têm esse método, o que garante que essa invocação de método não vai falhar.

Outro exemplo:



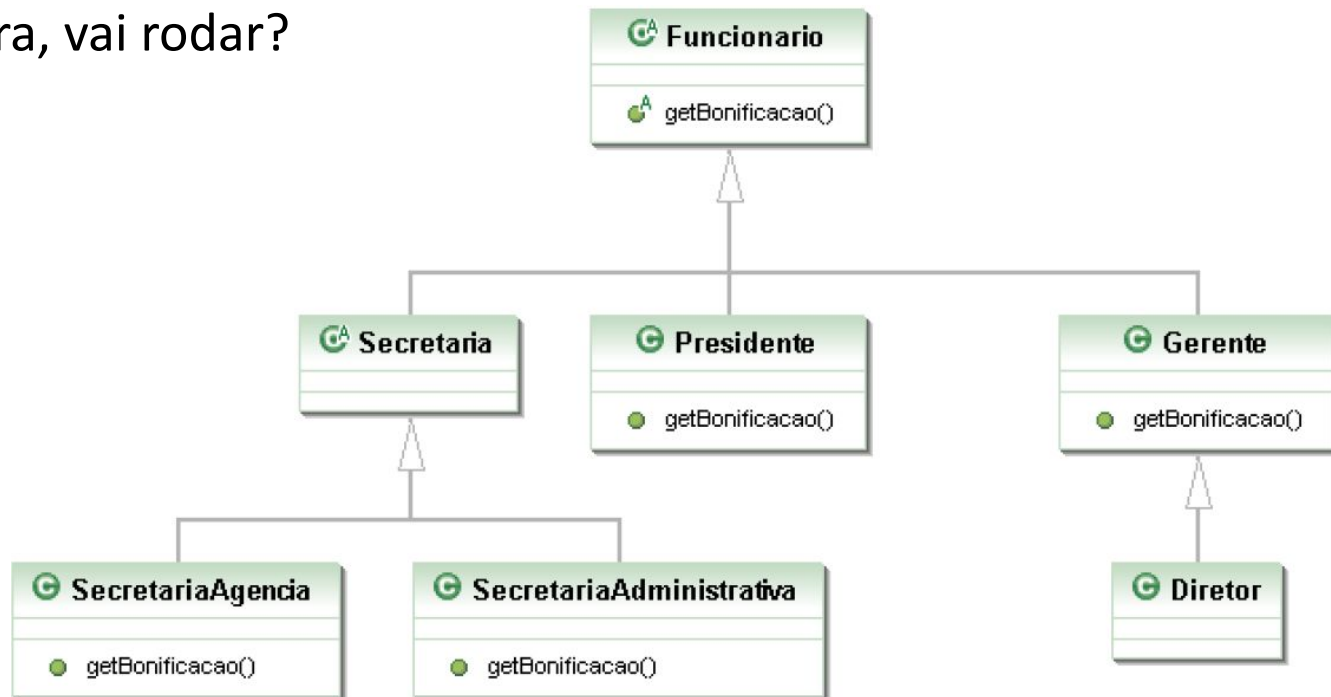
- E se, no nosso exemplo de empresa, tivéssemos o seguinte diagrama de classes com os seguintes métodos:

Essas classes vão compilar?
Vão rodar?



Sim! Gerente e Presidente possuem os métodos implementados, a classe Diretor, que não possui o método implementado, vai usar a implementação herdada de Gerente.

E agora, vai rodar?



De novo, a resposta é sim, pois **Secretaria** é uma classe abstrata e, por isso, o Java tem certeza de que ninguém vai conseguir instanciá-la e, muito menos, chamar o método `getBonificacao` dela. Lembrando que, nesse caso, não precisamos nem ao menos escrever o método abstrato `getBonificacao` na classe **Secretaria**.

Resumindo

- Em uma hierarquia, quanto mais alta a classe, mais abstrata é sua definição.
 - A classe *Animal* apresenta o método *locomover()*, mas ela não tem como implementar este método pois não sabe o tipo de animal tratado.
 - Java permite definir métodos sem implementá-los!
 - Métodos Abstratos:
 - Não possui corpo (implementação).
 - Apresenta apenas a definição seguida de “.”
 - Apresenta o modificador *abstract*.
- ```
public abstract class Animal {
 public int peso;
 public abstract void locomover();
}
```
- Se uma classe apresentar pelo menos um método abstrato, ela deve ser declarada como *abstract*.

# Resumindo


- Classes abstratas não podem ser instanciadas!
  - São geralmente utilizadas como superclasses (são estendidas).
  - Uma subclasse de uma classe abstrata pode ser instanciada se ela sobrepor todos os métodos abstratos e fornecer implementação para cada um deles.
  - Se a subclasse não implementar **todos** os métodos abstratos da superclasse, ela também terá que ser abstrata.

```
public abstract class FiguraGeometrica {
 public abstract double area();
 public abstract double perimetro();
}

public class Retangulo extends FiguraGeometrica {
 protected double w, h;
 public Retangulo() { this(0.0,0.0); }
 public Retangulo(double l, double a) { w = l; h = a; }
 public double area() { return w*h; }
 public double perimetro() { return 2*(w+h); }
}
```



# Para saber mais...

- Uma classe que estende uma classe normal também pode ser abstrata! No entanto, ela não poderá ser instanciada!
  - Uma classe abstrata não precisa necessariamente ter um método abstrato.
- 

# Pequeno Joguinho

- Criar um jogo em que podemos pegar(posX, posY): Moedas, Cogumelos, Diamantes e Estrelas. Todas essas classes devem possuir o método pegar e os atributos posX, posY.
- À medida em que você pegar qualquer um dos Itens você deverá receber uma mensagem no console avisando da conquista.

# Exercício em Sala

Cria as classes:

Animal,

- Cachorro;
- Gato;
- Galinha;
- Vaca
- Porco

Regras das classes acima:

- Os métodos *String falar()*, *void falarCom (Animal)* devem estar em todas as classes. Decida quais classes devem ser abstratas e quais métodos devem ser abstratos.
- Todos os animais deverão ter um nome, o qual deve ser passado quando os objetos são instanciados;
- Sobrescrevam o método *toString* com a identificação de cada animal. Ex.: Eu sou o Max e sou um cachorro.
- Sobrescrevam o método *equals* para compor dois animais pelo nome deles.



# Exercício em Sala

- Crie uma classe chamada Fazenda. Métodos da classe Fazenda:
  - apresentar(Animal a);
    - Neste método você deverá chamar o toString do bicho.
  - registrar(Animal a);
    - Neste método você deverá registrar todos os animais da fazenda. Antes de registrar o animal, verifique se ele não foi registrado anteriormente. Nesse caso o identificador dos animais será o nome deles.

# Referências Bibliográficas

- DEITEL, Harvey M. e DEITEL, Paul J. Java - Como Programar, 8ª edição. Pearson. 2010.
- BLOCH, Joshua. Effective Java, 2ª edição. Addison-Wesley, 2008.
- CAELUM. Java e Orientação a Objetos. Disponível em: <https://www.caelum.com.br/apostila-java-orientacao-objetos/>
- SOFTBLUE. Professor Carlos Eduardo Gusso Tosin. Fundamentos de Java. <http://www.softblue.com.br/>.
- K19. Java e Orientação a Objetos. Disponível em: <http://www.k19.com.br/cursos/orientacao-a-objetos-em-java>.
- HORSTMANN, CORNELL. Core Java Volume I – Fundamentos, 8ª Edição. São Paulo, Pearson Education, 2010.
- BRAUDE, E. J. Projeto de software - da programação à arquitetura: uma abordagem baseada em Java. Porto Alegre: Bookman, 2005.
- SANTOS, R. Introdução à Programação Orientada a Objetos usando Java. São Paulo: Campus, 2003.
- Slides do Professor Doutor Horácio Fernandes da UFAM.