

Programação Orientada a Objetos

Professor Filipe Dwan Pereira

Aula 6 – Herança, Reescrita e Polimorfismo

“Tens visto um homem precipitado nas suas palavras? Maior esperança a no tolo do que nele.”
Provérbios 28:20

Disclaimer

- Este slide foi baseado nas seguintes fontes principais:
 - SOFTBLUE. Professor Carlos Eduardo Gusso Tosin. Fundamentos de Java. <http://www.softblue.com.br/>.
 - Slides professor Horácio Oliveira – UFAM.
 - CAELUM. Java e Orientação a Objetos. Disponível em: <https://www.caelum.com.br/apostila-java-orientacao-objetos/>
 - K19. Java e Orientação a Objetos. Disponível em: <http://www.k19.com.br/cursos/orientacao-a-objetos-em-java>.

Objetivo desta aula

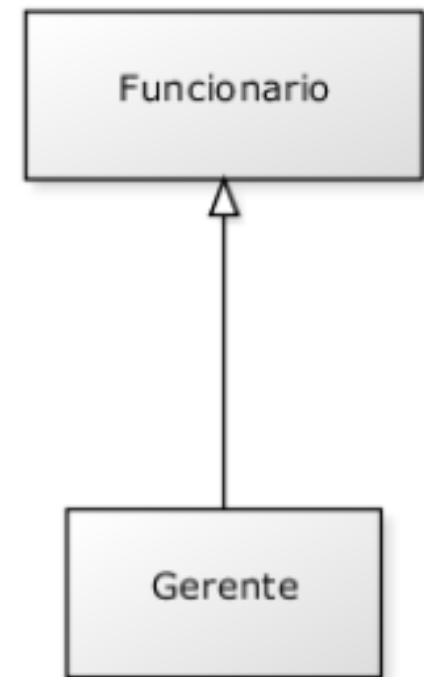
- dizer o que é herança e quando utilizá-la;
- reutilizar código escrito anteriormente;
- criar classes filhas e reescrever métodos;
- usar todo o poder que o polimorfismo dá.

Repetindo código?

```
class Funcionario {  
    String nome;  
    String cpf;  
    double salario;  
}  
  
class Gerente {  
    String nome;  
    String cpf;  
    double salario;  
    int senha;  
    int numeroDeFuncionariosGerenciados;  
  
    public boolean autentica(int senha) {  
        if (this.senha == senha) {  
            System.out.println("Acesso Permitido!");  
            return true;  
        } else {  
            System.out.println("Acesso Negado!");  
            return false;  
        }  
    }  
}
```

- Podemos relacionar uma classe de tal maneira que ela **herda** tudo que a outra tem.
- Fazer com que o Gerente tivesse tudo que um Funcionario tem, gostaríamos que ela fosse uma **extensão** de Funcionario.

Em todo momento que criarmos um objeto do tipo Gerente, este objeto possuirá também os atributos e comportamentos (métodos) definidos na classe Funcionario, pois um Gerente **é um** Funcionario:



Fazemos isto através da palavra chave **extends**:

```
class Gerente extends Funcionario {
    int senha;
    int numeroDeFuncionariosGerenciados;

    public boolean autentica(int senha) {
        if (this.senha == senha) {
            System.out.println("Acesso Permitido!");
            return true;
        } else {
            System.out.println("Acesso Negado!");
            return false;
        }
    }

    // setter da senha omitido
}
```

A classe Gerente **herda** todos os atributos e métodos da classe mãe, no nosso caso, a Funcionario:

```
class TestaGerente {  
    public static void main(String[] args) {  
        Gerente gerente = new Gerente();  
  
        // podemos chamar métodos do Funcionario:  
        gerente.setNome("João da Silva");  
  
        // e também métodos do Gerente!  
        gerente.setSenha(4231);  
    }  
}
```

Herança

- Para ser mais preciso, uma **subclasse** (Classe Filha) também herda os atributos e métodos privados da **superclasse** (Classe Mãe), porém não consegue acessá-los diretamente.
- Para acessar um membro privado na Classe filha indiretamente, seria necessário que a mãe expusesse um outro método visível que invocasse esse atributo ou método privado.

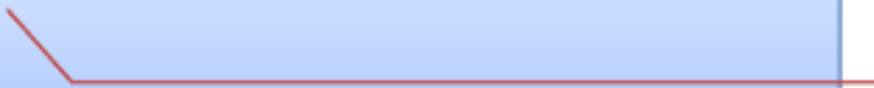
E se precisarmos acessar os atributos que herdamos?

- Usaríamos o modificador **protected** que só pode ser acessado (visível) pela própria classe e por suas subclasses (e mais algumas outras classes, mas veremos isso mais a frente).

```
class Funcionario {  
    protected String nome;  
    protected String cpf;  
    protected double salario;  
    // métodos devem vir aqui  
}
```


Outro exemplo: protected

```
class Telefone {  
    protected String numero;  
    ...  
}
```



O atributo é
declarado como
protected na
superclasse

```
class Celular extends Telefone {  
  
    public void adicionarDDD(String ddd) {  
        String n = ddd + this.numero;  
    }  
}
```

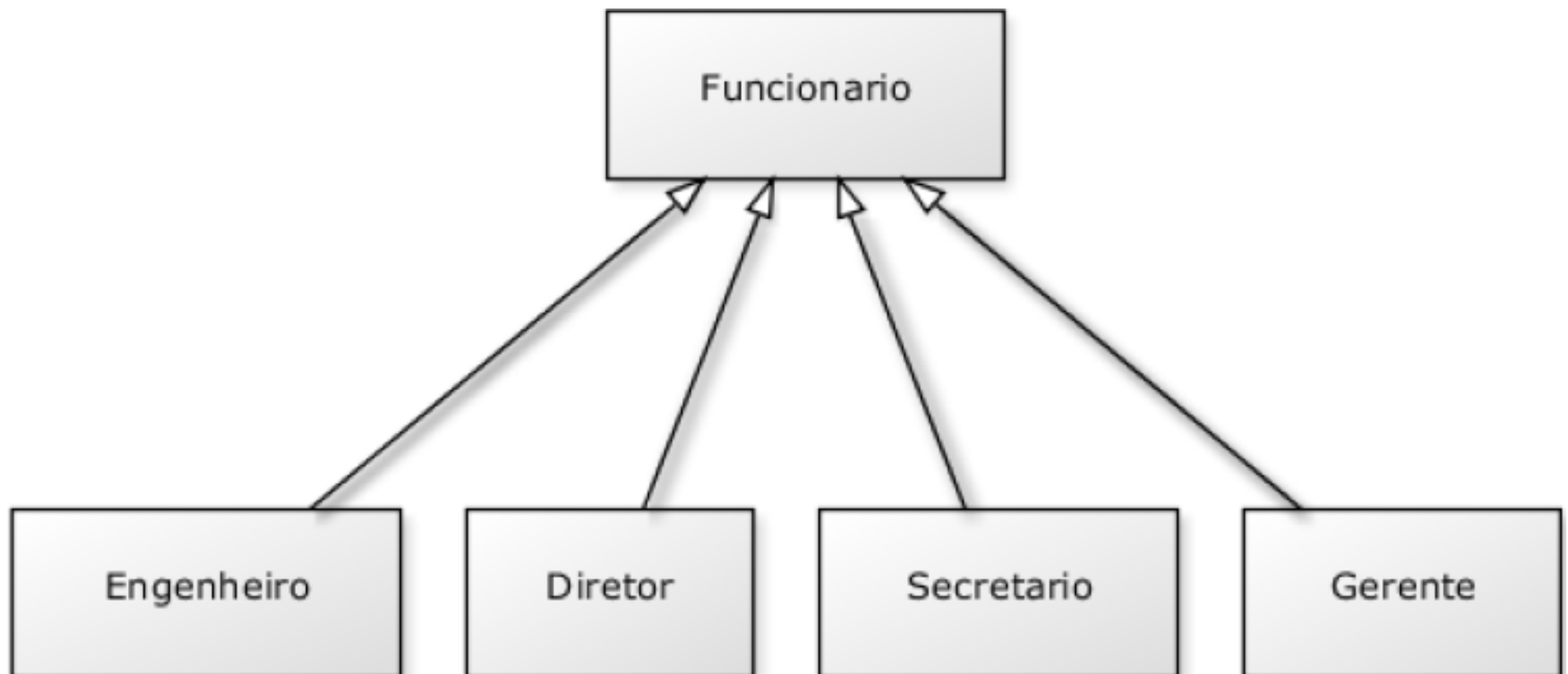


Métodos da
subclasse possuem
acesso ao atributo
declarado na
superclasse

Sempre usar protected?

- Nem sempre é uma boa ideia deixar que a classe filha acesse os atributos da classe mãe, pois isso quebra um pouco a ideia de que só aquela classe deveria manipular seus atributos.
- Além disso, não só as subclasses, mas também as outras classes, podem acessar os atributos protected, que veremos mais a frente (mesmo pacote).

Uma classe pode ter várias filhas, mas pode ter apenas uma mãe, é a chamada herança simples do java:



Reescrita de método

- Todo fim de ano, os funcionários do nosso banco recebem uma bonificação. Os funcionários comuns recebem 10% do valor do salário e os gerentes, 15%.

```
class Funcionario {  
    protected String nome;  
    protected String cpf;  
    protected double salario;  
  
    public double getBonificacao() {  
        return this.salario * 0.10;  
    }  
    // métodos  
}
```

Se deixarmos a classe Gerente como ela está, ela vai herdar o método getBonificacao.

```
Gerente gerente = new Gerente();  
gerente.setSalario(5000.0);  
System.out.println(gerente.getBonificacao());
```

O resultado aqui será 500. Não queremos essa resposta, pois o gerente deveria ter 750 de bônus nesse caso.

- No Java, quando herdamos um método, podemos alterar seu comportamento. Podemos **reescrever** (reescrever, sobrescrever, **override**) este método:

```
class Gerente extends Funcionario {  
    int senha;  
    int numeroDeFuncionariosGerenciados;  
  
    public double getBonificacao() {  
        return this.salario * 0.15;  
    }  
    // ...  
}
```

Agora o método está correto para o Gerente. Se refizéssemos o teste o valor impresso seria o correto (750):

```
Gerente gerente = new Gerente();  
gerente.setSalario(5000.0);  
System.out.println(gerente.getBonificacao());
```


Reescrita (sobrescrever) métodos

- Observações importantes:
 - Os métodos sobrescritos substituem os métodos da superclasse
 - A assinatura do método sobrescrito deve ser a mesma do método original

Outro exemplo:

```
class Telefone {  
    public void telefonar() {  
        //código para telefonar  
    }  
}
```

```
class Orelhao extends Telefone {  
    public void telefonar() {  
        //código para telefonar do orelhão  
    }  
}
```

```
Orelhao o = new Orelhao();  
o.telefonar();
```



Como o método foi sobrescrito, é chamado o método da subclasse

Outro exemplo:

```
class Telefone {  
    public void telefonar() {  
        //código para telefonar  
    }  
}
```

```
class Orelhao extends Telefone {  
    public void telefonar(int numero) {  
        //código para telefonar do orelhão  
    }  
}
```

```
Orelhao o = new Orelhao();  
o.telefonar();
```

Não há sobrescrita de método. Métodos sobrescritos devem ter a mesma assinatura (tipo de retorno, nome do método e parâmetros)



Invocando o método reescrito

- Imagine que para calcular a bonificação de um Gerente devemos fazer igual ao cálculo de um Funcionário, porém adicionando R\$ 1000. Poderíamos fazer assim:

```
class Gerente extends Funcionario {  
    int senha;  
    int numeroDeFuncionariosGerenciados;  
  
    public double getBonificacao() {  
        return this.salario * 0.10 + 1000;  
    }  
    // ...  
}
```

No Slide anterior teríamos um problema : o dia que o `getBonificacao` do `Funcionario` mudar, precisaremos mudar o método do `Gerente` para acompanhar a nova bonificação. Para evitar isso, o `getBonificacao` do `Gerente` pode chamar o do `Funcionario` utilizando a palavra chave **super**.

```
class Gerente extends Funcionario {  
    int senha;  
    int numeroDeFuncionariosGerenciados;  
  
    public double getBonificacao() {  
        return super.getBonificacao() + 1000;  
    }  
    // ...  
}
```

Resumindo Herança

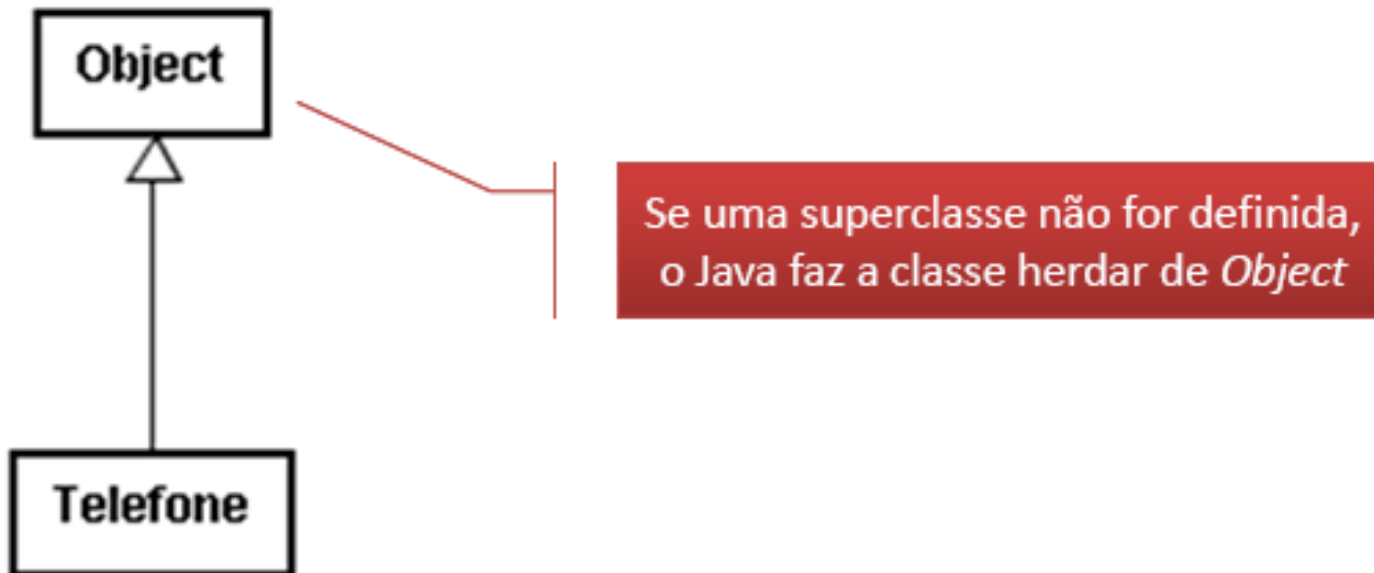
- Mecanismo simples e poderoso do paradigma OO que permite que uma nova classe seja descrita a partir de uma classe já existente.
 - Classe mãe (ou pai): superclasse, classe base;
 - Classe filha: subclasse, classe derivada;
 - Classe filha (mais específica) herda atributos e métodos da classe mãe (mais geral);
 - Classe filha possui atributos e métodos próprios.
- Possibilidades
 - incluir dados e códigos em uma classe sem ter de mudar a classe original.
 - usar o código novamente (reusabilidade).
 - alterar o comportamento de uma classe.

Resumindo Herança. Ex.:

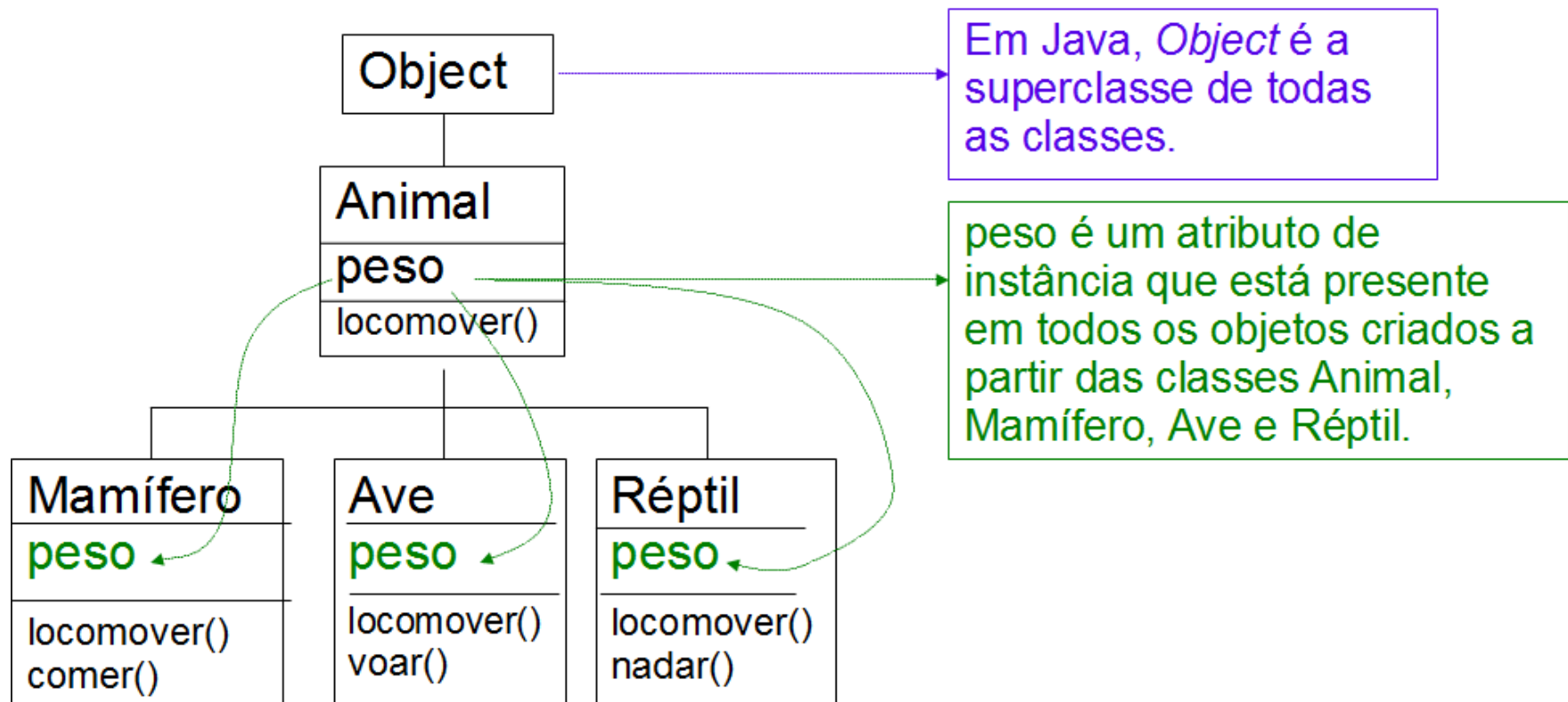
- Classe Carro:
 - Carro Básico
 - Carro Luxuoso
- Classe Estudante:
 - Estudante Graduação
 - Estudante Pós Graduação
 - Mestrado
 - Doutorado
- Classe FormaGeométrica:
 - Círculo
 - Quadrado
 - Triângulo

Herança da Classe Object

- Toda classe em Java herda de apenas uma superclasse



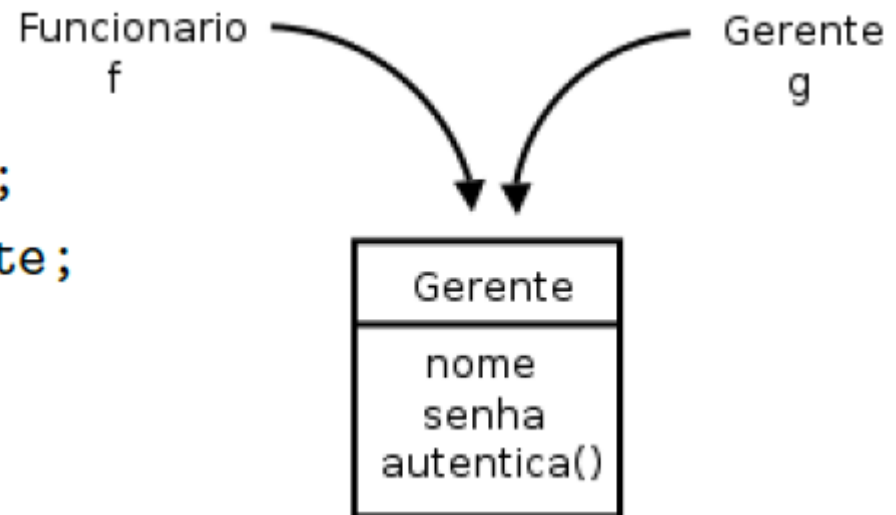
Resumindo Herança: Hierarquia de Classes



Polimorfismo

- O que guarda uma variável do tipo Funcionario?
 - **Resposta: Uma referência para um Funcionario, nunca o objeto em si.**
- Na herança, vimos que todo Gerente **é um** Funcionario, pois é uma extensão deste. Podemos nos referir a um Gerente como sendo um Funcionario.

```
Gerente gerente = new Gerente();  
Funcionario funcionario = gerente;  
funcionario.setSalario(5000.0);
```



Polimorfismo

- Polimorfismo é a capacidade de um objeto poder ser referenciado de várias formas.

***Cuidado**, polimorfismo não quer dizer que o objeto fica se transformando, muito pelo contrário, um objeto nasce de um tipo e morre daquele tipo, o que pode mudar é a maneira como nos referimos a ele.*

```
Gerente gerente = new Gerente();  
Funcionario funcionario = gerente;  
funcionario.setSalario(5000.0);
```

Até aqui tudo bem, mas e se eu tentar:

```
funcionario.getBonificacao();
```

- Qual o retorno desse método? 500 ou 1500?
 - *No Java, a invocação de método sempre vai ser decidida em tempo de execução. O Java vai procurar o objeto na memória e, aí sim, decidir qual método deve ser chamado, sempre relacionando com sua classe de verdade, e não coma que estamos usando para referenciá-lo. Apesar de estarmos nos referenciando a esse Gerente como sendo um Funcionario, o método executado é do Gerente e o retorno 750.*

Por que criar um gerente referenciá-lo como funcionário?

Um método pode receber como argumento um funcionário:

```
class ControleDeBonificacoes {  
    private double totalDeBonificacoes = 0;  
  
    public void registra(Funcionario funcionario) {  
        this.totalDeBonificacoes += funcionario.getBonificacao();  
    }  
  
    public double getTotalDeBonificacoes() {  
        return this.totalDeBonificacoes;  
    }  
}
```

- No main poderia ser feito:

```
ControleDeBonificacoes controle = new ControleDeBonificacoes();
```

```
Gerente funcionario1 = new Gerente();
```

```
funcionario1.setSalario(5000.0);
```

```
controle.registra(funcionario1);
```

```
Funcionario funcionario2 = new Funcionario();
```

```
funcionario2.setSalario(1000.0);
```

```
controle.registra(funcionario2);
```

```
System.out.println(controle.getTotalDeBonificacoes());
```

Conseguimos passar um Gerente para um método que recebe um Funcionario como argumento.

Um outro exemplo de herança e polimorfismo

- Imagine que vamos modelar um sistema para a faculdade que controle as despesas com funcionários e professores. Nosso funcionário fica assim:

```
class EmpregadoDaFaculdade {  
    private String nome;  
    private double salario;  
    double getGastos() {  
        return this.salario;  
    }  
    String getInfo() {  
        return "nome: " + this.nome + " com salário " + this.salario;  
    }  
    // métodos de get, set e outros  
}
```

- O gasto que temos com o professor não é apenas seu salário. Temos de somar um bônus de 10 reais por hora/aula. O que fazemos então?
 - Reescrevemos o método. Assim como o `getGastos` é diferente, o `getInfo` também será, pois temos de mostrar as horas/aula também.

```
class ProfessorDaFaculdade extends EmpregadoDaFaculdade {  
    private int horasDeAula;  
    double getGastos() {  
        return this.getSalario() + this.horasDeAula * 10;  
    }  
    String getInfo() {  
        String informacaoBasica = super.getInfo();  
        String informacao = informacaoBasica + " horas de aula: "  
                                + this.horasDeAula;  
        return informacao;  
    }  
    // métodos de get, set e outros que forem necessários  
}
```


Como tiramos proveito do polimorfismo?

Imagine que temos uma classe de relatório:

```
class GeradorDeRelatorio {  
    public void adiciona(EmpregadoDaFaculdade f) {  
        System.out.println(f.getInfo());  
        System.out.println(f.getGastos());  
    }  
}
```

Podemos passar para nossa classe qualquer EmpregadoDaFaculdade! Vai funcionar tanto para professor, quanto para funcionário comum.

Queremos colocar agora no nosso programa a classe Reitor.

Como ele também é um EmpregadoDaFaculdade, será que vamos precisar alterar algo na nossa classe de Relatorio?

Não. Essa é a ideia!

```
class Reitor extends EmpregadoDaFaculdade {  
    // informações extras  
    String getInfo() {  
        return super.getInfo() + " e ele é um reitor";  
    }  
    // não sobrescrevemos o getGastos!!!  
}
```

Herança Versus Acoplamento

- Note que o uso de herança **aumenta** o acoplamento entre as classes, isto é, o quanto uma classe depende de outra.
- Esse é um problema da herança, e não do polimorfismo, que resolveremos mais tarde com a ajuda de Interfaces.

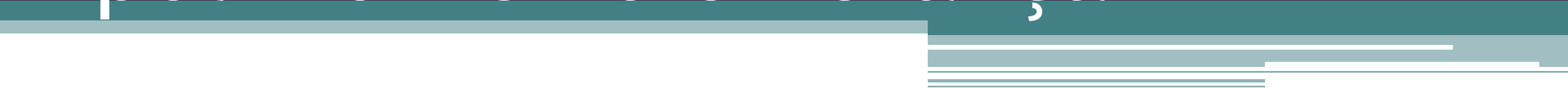
Herança versus Composição:

- Na composição temos uma instância da classe existente sendo usada como componente da outra classe.
- Enquanto a **herança** usa o termo **É UM** a **composição** usa o termo **TEM UM**.
 - Por exemplo, podemos dizer que um carro TEM UM pneu, neste exemplo temos a definição de uma composição.

Exercício em sala

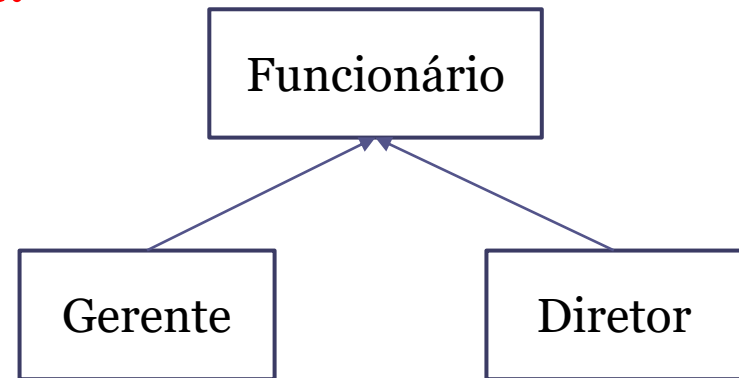
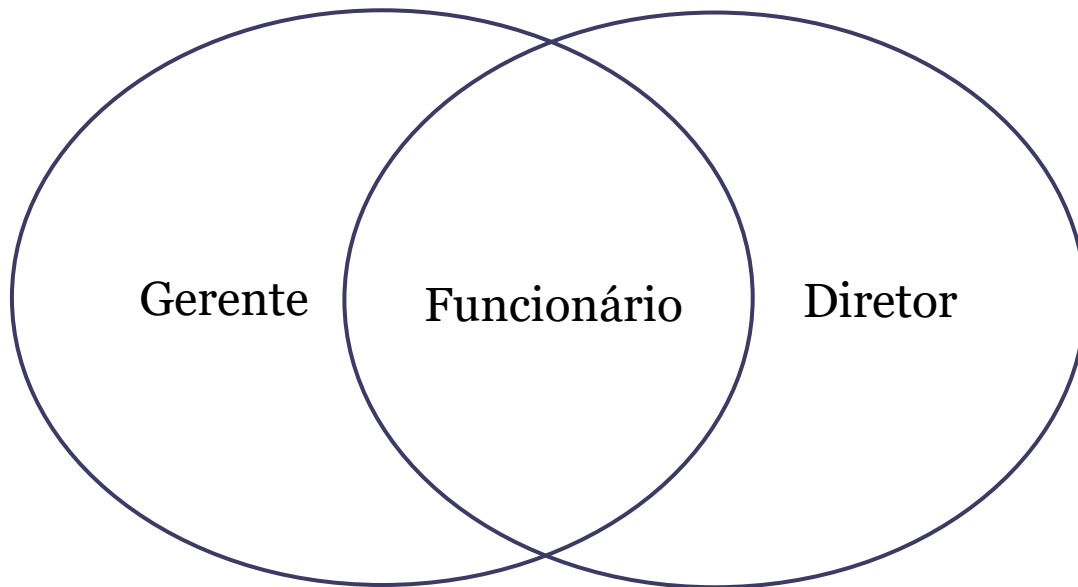
- Crie a classe Figura que representa figuras geométricas, representadas pelas classes Quadrado e Retangulo. Uma figura pode ter sua área calculada a partir do método calcularArea(), que retorna a área calculada da figura em forma de um double.
- Crie também a classe FiguraComplexa. Uma figura complexa é também uma figura, mas a diferença é que ela é composta por várias figuras (quadrados, retângulos ou até outras figuras complexas). Para calcular a área de uma figura complexa, basta somar a área de todas as figuras que a compõem.
- Para executar a aplicação, crie a classe Calculador, que é responsável por criar uma figura complexa e calcular a sua área. Esta figura deve ser composta por:
 - 1 quadrado com 3 de lado
 - 1 quadrado com 10 de lado
 - 1 retângulo com lados 2 e 7
 - 1 retângulo com lados 5 e 3
- Dica: Perceba a diferença entre uma classe ser uma figura e ter uma ou mais figuras. A primeira relação é de herança, enquanto a segunda implica em uma composição.

Um pouco mais sobre polimorfismo e herança

A series of horizontal lines in teal and light blue colors, with varying lengths and slight offsets, creating a modern, layered effect across the middle of the slide.

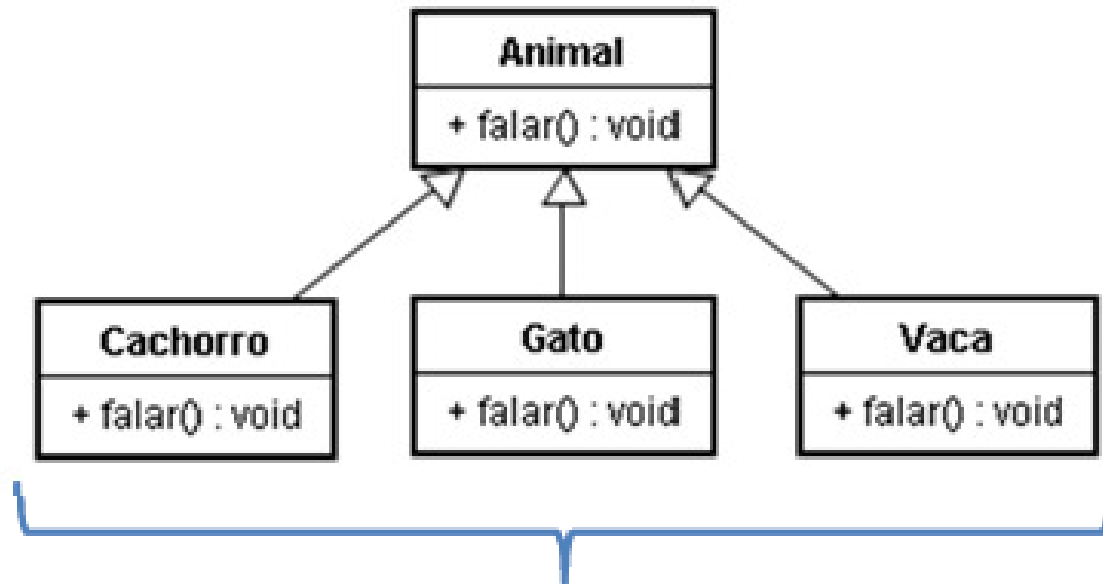
Vamos analisar a seguinte linha:

```
Gerente funcionario1 = new Gerente();
```



Quando nós referenciarmos um Gerente como Funcionário, então podemos acessar todos os métodos do objeto gerente que estão definidos no Funcionário. Caso o método tenha sido sobrescrito, então será chamado o comportamento da classe mais especializada.

Outro exemplo Polimorfismo



As subclasses sobreescrevem o método *falar()*

Outro exemplo Polimorfismo

```
class Animal {  
    public void falar() {  
    }  
}
```

```
class Cachorro extends Animal {  
    public void falar() {  
        System.out.println("Au");  
    }  
}
```

```
class Gato extends Animal {  
    public void falar() {  
        System.out.println("Miau");  
    }  
}
```

```
class Vaca extends Animal {  
    public void falar() {  
        System.out.println("Mu");  
    }  
}
```

Cada animal implementa o método *falar()* do seu modo

Outro exemplo Polimorfismo

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

Resultado: "Au"

```
Animal a = new Gato();  
a.falar();
```

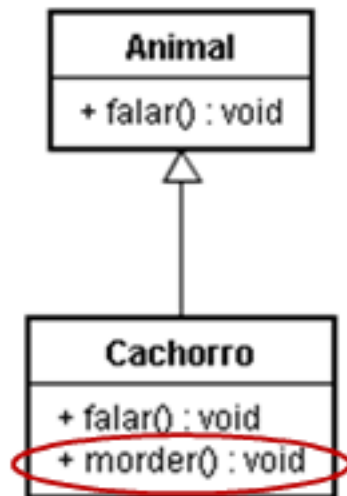
Resultado: "Miau"

```
Animal a = new Vaca();  
a.falar();
```

Resultado: "Mu"

O método invocado é determinado pelo tipo do objeto que está armazenado na memória

Outro exemplo Polimorfismo



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

Resultado: "Au"

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

Método
inexistente

```
Animal a = new Cachorro();
Cachorro c = (Cachorro) a;
c.morder();
```

OK

O tipo pelo qual o objeto é referenciado determina
quais métodos e/ou atributos podem ser invocados

Operador *instanceOf*

- Utilizado para verificar se um objeto pertence à determinada classe

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

```
a instanceof Cachorro
```

```
true
```

```
a instanceof Animal
```

```
true
```

```
a instanceof Gato
```

```
false
```

```
a instanceof Object
```

```
true
```

Normalmente é utilizado antes de realizar um cast, para garantir que a operação é válida

Sobrescrevendo métodos do Object

- Método **toString()**

- O comportamento padrão do toString() é retornar o nome da classe @ hashCode.
- As classes podem sobrescrever este método para mostrarem uma mensagem que as representem
- O método toString sempre é chamado de forma automática quando um objeto é passado para System.out.println()

- Método **equals(Object)**

- É a forma que o Java tem de comparar objetos pelo seu conteúdo ao invés de comparar as referências (como acontece ao usarmos "==")

Exemplo de uso equals()

```
public class Conta {  
    private double saldo;  
    // outros atributos...  
  
    public Conta(double saldo) {  
        this.saldo = saldo;  
    }  
  
    public boolean equals(Object object) {  
        Conta outraConta = (Conta) object;  
        if (this.saldo == outraConta.saldo) {  
            return true;  
        }  
        return false;  
    }  
}
```

Exemplo de uso equals()

- Poderíamos fazer assim:

```
public boolean equals(Object object) {  
    if (!(object instanceof Conta))  
        return false;  
    Conta outraConta = (Conta) object;  
    return this.saldo == outraConta.saldo;  
}
```

Benefícios de sobrepor toString()

- o que é melhor de ver ao exibir um mapeamento: “Jenny=Cep@163b91” ou “Jenny=69300000”?
- o método toString deve retornar todas as informações interessantes contidas no objeto, como no exemplo acima do número do CEP.
- Quando temos um objeto grande demais o ideal é criar um resumo como “O funcionário Fulano mora no CEP 92110300, Av. Ataíde Teive no Estado de Roraima em Boa Vista”.
- O ideal é que a String sempre seja auto-explicativa.

Exemplo de uso toString()

```
public class Endereco {  
  
    private Integer cep;  
    private String cidade;  
    private String estado;  
    private String rua;  
    private Integer numero;  
  
    //métodos set e get implementados aqui|  
    @Override  
    public String toString() {  
        return "Rua: " + this.rua + ", Número: " + this.numero +  
            ", Cidade: " + this.cidade + ", Estado: " + this.estado +  
            ", CEP: " + this.cep;  
    }  
}
```

Exemplo de uso toString()

```
public class ExibeDados {  
  
    public static void main(String args []) {  
        Endereco endereco = new Endereco();  
        endereco.setCEP(92110300);  
        endereco.setCidade("Porto Alegre");  
        endereco.setEstado("Rio Grande do Sul");  
        endereco.setNumero(700);  
        endereco.setRua("Chacara Barreto");  
  
        System.out.println(endereco);  
    }  
}
```

Métodos e Classes Finais

- Se um método for declarado com o modificador *final*, ele não pode ser sobreposto.
 - Todos os **métodos** estáticos (*static*) e privados (*private*) são finais por definição, da mesma forma que todos os métodos de uma classe *final*.
 - Obs.: ele ainda pode ser sobrecarregado (overloading).
- Classes como Constantes
 - Quando uma classe é declarada com o modificador *final*, significa que ela não pode ser estendida.
 - `java.lang.System` é um exemplo de uma classe *final*.
 - Declarar uma classe como sendo final previne extensões não desejadas da mesma.

Sugestão de Leitura

- Leia o capítulo 7 da apostila fj11 – Herança, reescrita e polimorfismo:
 - <http://www.caelum.com.br/apostila-java-orientacao-objetos/>
- Lesson: Object-Oriented Programming Concepts:
 - <http://docs.oracle.com/javase/tutorial/java/concepts/index.html>
- Leia o capítulo 9 e 10 – Herança e Polimorfismo:
 - DEITEL, Harvey M. e DEITEL, Paul J. Java - Como Programar, 8ª edição. Pearson. 2010.

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.

“Seja a Mudança que você quer ver no mundo”. Ghandi



filipedwan@gmail.com

 filipedwan

 @filipedwan