PROGRAMAÇÃO ORIENTADA A OBJETOS

Prof. Milton Palmeira Santana

Anhanguera

Definição e utilização de métodos construtores

- Construtores são “métodos especiais” que implementam as ações necessárias para inicializar um objeto
  - Tem o mesmo nome da classe
  - Não possuem tipo de retorno (nem void)
  - Parâmetros são opcionais

```
public Cliente(string nome, string cpf) {
     this.nome = nome;
     this.cpf = cpf;
}
```

Anhanguera

Definição e utilização de métodos construtores

- Todo new chama um construtor obrigatoriamente.

- Até agora não vimos nenhum construtor porque existem construtores default. O construtor default não contém nenhum parâmetro.

- Não somos obrigados a escrever o construtor default, exceto quando temos outros construtores.

```
public Conta()
{
}
```

Definição e utilização de métodos construtores

- O construtor pode ser utilizado para que seja efetuado um determinado código quando o objeto for instanciado.

- **Ex**: Queremos que toda vez que uma conta for iniciada ela tenha um valor inicial de limite = 100.

```
public Conta()
{
  this.limite = 100;
}
```

Definição e utilização de métodos construtores

- Vamos agora pensar no seguinte: Quando criamos uma conta, já temos o Cliente cadastrado, correto? Então, não seria melhor que quando a Conta fosse criada já iniciasse com o nosso Cliente? Vejamos:

```
public Conta(Cliente dono)
{
   this.donoDaConta = dono;
}
```

Anhanguera

Modificadores de Acesso

- Modificadores de acesso são utilizados para definir níveis de acesso a membros das classes

Declaração	Definição
public	Acesso ilimitado
private	Acesso limitado à classe e seus membros
protected	Acesso limitado à classe, seus membros e a tipos derivados da mesma

Anhanguera

Encapsulamento

- Quando liberamos o acesso aos atributos e métodos de uma classe, permitimos que qualquer um acesse os campos dela e os altere da maneira que quiser.

- O ideal é que esse acesso seja restrito para que apenas a própria classe tenha a possibilidade de alterar seus atributos ou então que outra classe apenas altere seus atributos através de métodos públicos.

- Esse processo de “esconder” os detalhes de implementação de outra classe é chamado de Encapsulamento.

Encapsulamento

- Fazendo isso, não permitimos aos outros saber COMO a classe faz o trabalho dela, mas O QUÊ ela faz.

- Utilizaremos o modificador de acesso **private** para “esconder” o atributo.

- Fazendo isso, poderemos acessar esse atributo apenas da própria classe ou através de um método público se tentar acessar de outra classe.

```
private double saldo;
```

Encapsulamento

- Ótimo! Agora resolvemos o problema da alteração do saldo. Não é mais permitido acessar o saldo nem alterar de fora da classe.

- O único detalhe é que precisaremos de métodos para poder acessar cada atributo. Para isso, utilizamos os getters e setters.

```
public double getSaldo() {
    return this.saldo;
}

public void setSaldo(double saldo) {
    this.saldo = saldo;
}
```

Exercício

- Modifique as classes criadas e encapsule todos os atributos.

Anhanguera

Herança

- Como toda empresa, nosso banco possui funcionários.

- Precisamos adicionar esse funcionários.
- Crie uma classe funcionário:
  - String nome
  - String cpf
  - double salario

Herança

- Além do funcionário comum, temos também o gerente. O gerente é um funcionário que têm funções e responsabilidades diferentes, portanto, se criarmos uma classe gerente precisamos de alguns outros atributos e métodos, além dos do funcionário.
  - String nome
  - String cpf
  - double salario
  - int senha
  - int numeroDeFuncionariosGerenciados
  - Método autentica

Herança

- Pergunta: precisamos de outra classe? Por que?

- Temos duas classes com alguns atributos que são iguais e, no caso do gerente, outros a mais. Imagine a seguinte situação: em uma empresa temos diversos tipos de funcionários. Precisamos criar uma classe para cada e copiar todos os atributos?

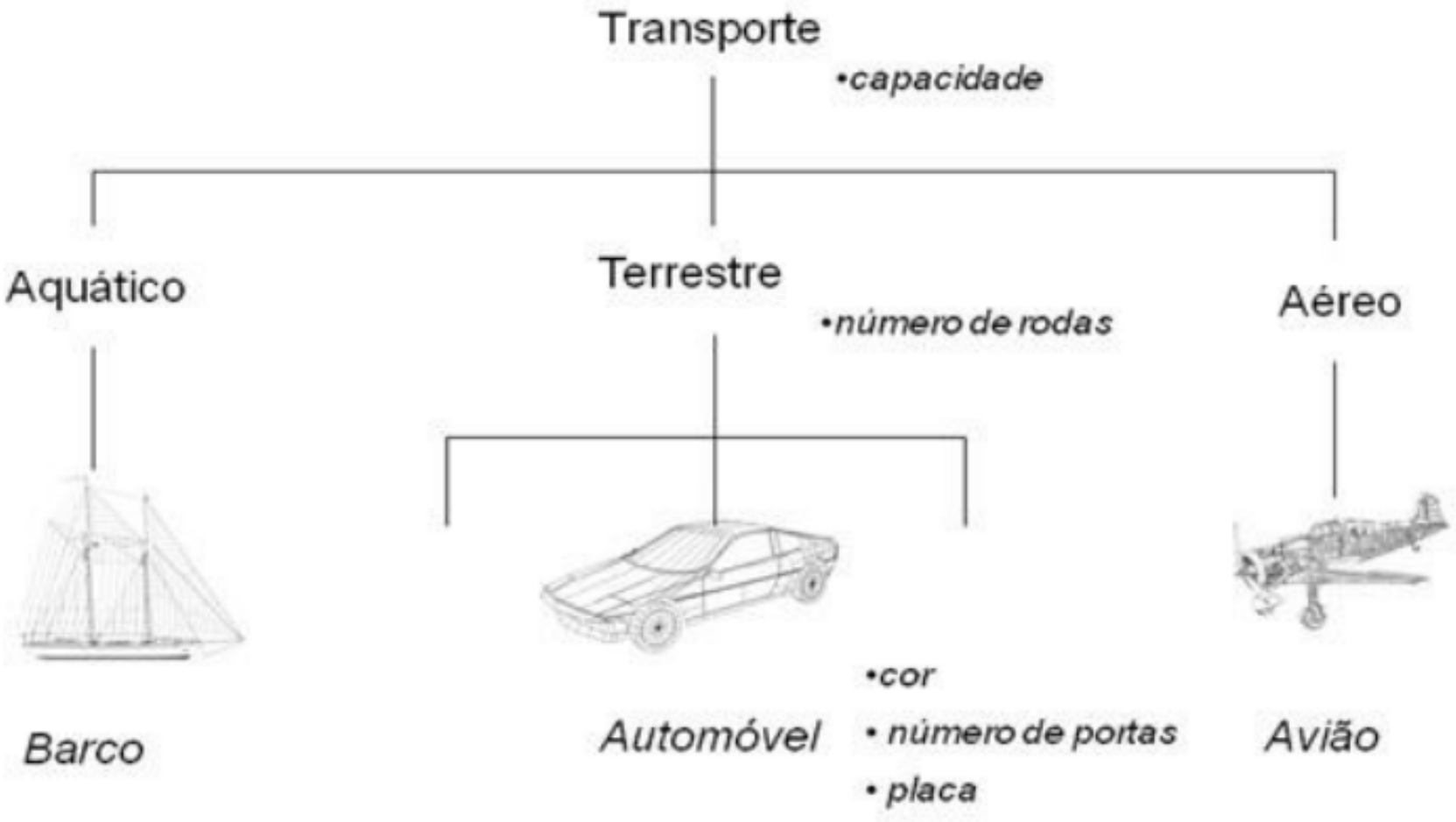
- O problema piora se tivermos que adicionar um atributo para todos os funcionários. Seria necessário entrar em cada classe e adicionar o mesmo atributo para todos eles.

Herança

- Existe um processo que utilizamos em Orientação a Objetos chamado Herança. Através desse processo, podemos “herdar” todos os atributos de uma classe. Assim, não precisamos reescrever em todas as classes os mesmos atributos, apenas adicionar o que é diferente da outra.

- Em nosso exemplo, podemos pensar assim: **TODO** gerente É um funcionário. Portanto, o gerente têm todos os atributos de um funcionário.

Herança


Transporte
•capacidade
Aquático
Terrestre
•número de rodas
Aéreo
Barco
Automóvel
•cor
• número de portas
• placa
Avião

Anhanguera

Herança

» Como seriam as classes da imagem anterior?

```
public class Transporte{
        private int capacidade;
}
```

```
public class Terrestre extends Transporte{
        private int numRodas;
}
```

```
public class Automovel extends Terrestre{
        private String cor;
        private int numPortas;
        private String placa;
}
```

Anhanguera

Herança

```
public class Gerente extends Funcionario {
    private int senha;
    private 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
}
```

Herança

- Quando utilizamos herança no exemplo a classe Gerente herdou todos os atributos da classe Funcionario, inclusive os atributos privados.

- Como os atributos estão privados temos acesso a elas apenas na própria classe (nesse caso Funcionario).

- Se quisermos acessar esses atributos temos que utilizar as propriedades (não aconselhável) ou utilizar o atributo fazendo referência à classe “mãe”.

- A nomenclatura mais encontrada é que Funcionario é **superclasse** de Gerente e Gerente é **subclasse** de Funcionario.

Herança

- Se quisermos acessar os atributos da classe mãe na classe filha podemos utilizar o modificador **protected**.

- Altere todos os modificadores atributos da classe Funcionario para protected.

- Atributos protegidos são acessíveis dentro de sua classe e classes derivadas.

- Por que utilizar **Protected**? Por que utilizar **Private**?

Anhanguera

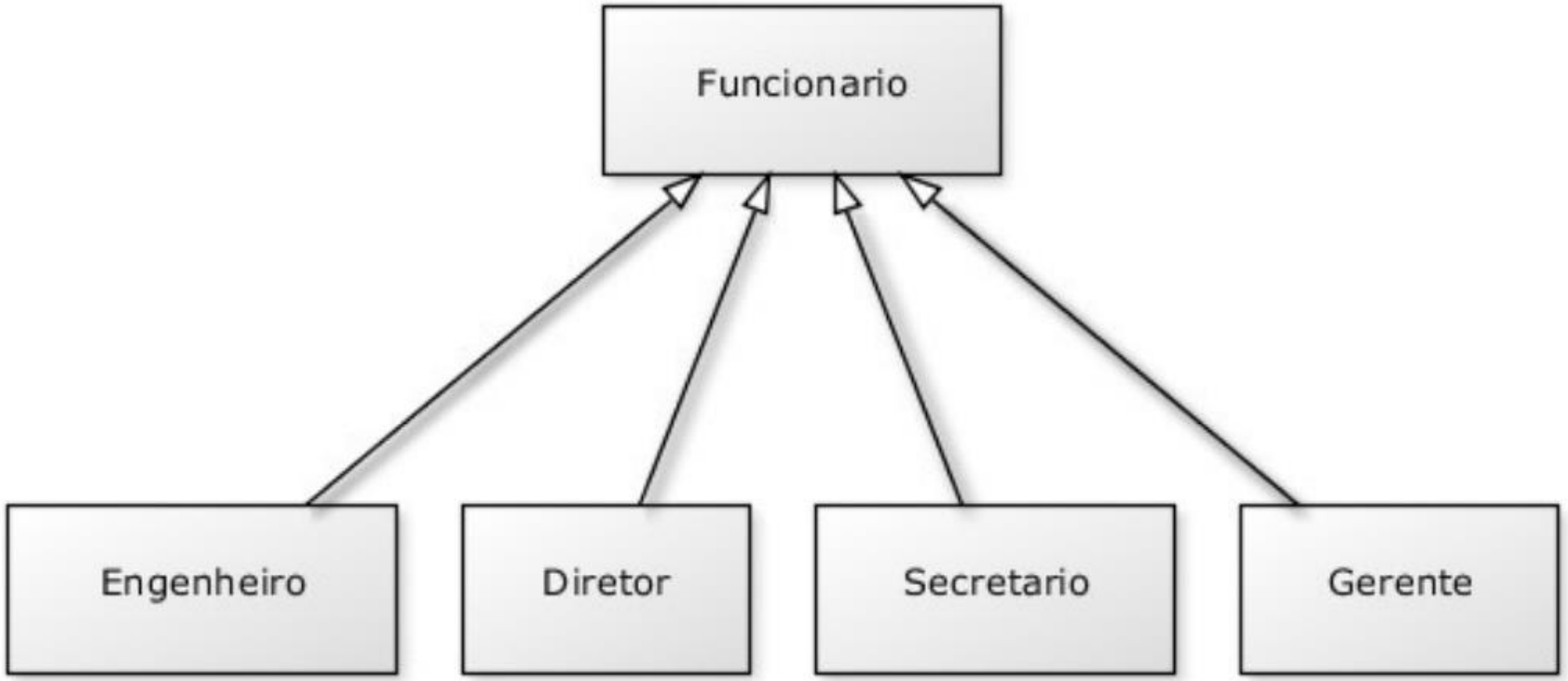
Herança

- Da mesma maneira, podemos ter uma classe Diretor que estenda Gerente, e a classe Presidente pode estender diretamente de Funcionario.

- Fique claro que essa é uma decisão de negócio. Se Diretor estenderá de Gerente ou não, dependerá se, para você, Diretor é um Gerente.

- Uma classe pode ter várias filhas, mas apenas uma mãe. É a chamada herança simples do Java.

Herança



Anhanguera

Reescrita ou sobrecarga de métodos

» Nossa empresa todo final de ano dá uma bonificação aos funcionários. O Funcionário recebe 10% do seu salário e o Gerente recebe 15% (injusto...).

» Vamos então criar um método para calcular a bonificação:

```
public class Funcionario {
    protected String nome;
    protected String cpf;
    protected double salario;

    public double getBonificacao() {
        return this.salario * 0.10;
    }
    // métodos
}
```

Reescrita ou sobrecarga de métodos

- A classe Gerente que é filha também herda esse método, portanto, pode utilizá-la também. Porém, o valor da bonificação é 15% e não 10. Crio outro método?

- Em JAVA podemos simplesmente reescrever o método na classe Gerente:

```
public double getBonificacao() {
    return this.salario * 0.15;
}
```

Anhanguera

Reescrita ou sobrecarga de métodos

» Há como deixar explícito no seu código que determinado método é a reescrita de um método da classe mãe dele. Podemos fazê-lo colocando **@Override** em cima do método. Isso é chamado anotação. Existem diversas anotações, e cada uma terá um efeito diferente sobre seu código. Isso NÃO É OBRIGATÓRIO.

```
@Override
public double getBonificacao() {
    return this.salario * 0.15;
}
```

Anhanguera

Reescrita ou sobrecarga de métodos

» Imagine que, para calcular a bonificação de um Gerente , devamos fazer igual ao cálculo de um **Funcionario** , porém adicionando R$ 1000. Poderíamos fazer assim:

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

    public double getBonificacao() {
        return this.salario * 0.10 + 1000;
    }
     // ...
}
```

Anhanguera

Reescrita ou sobrecarga de métodos

» Aqui teríamos um problema: o dia que o **getBonificacao** do **Funcionario** mudar, precisaremos mudar o método do Gerente a fim de acompanhar a nova bonificação. Para evitar isso, o **getBonificacao** do Gerente pode chamar o do **Funcionario** utilizando a palavra-chave **super**.

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

    public double getBonificacao() {
        return super.getBonificacao() + 1000;
    }
     // ...
}
```

Anhanguera

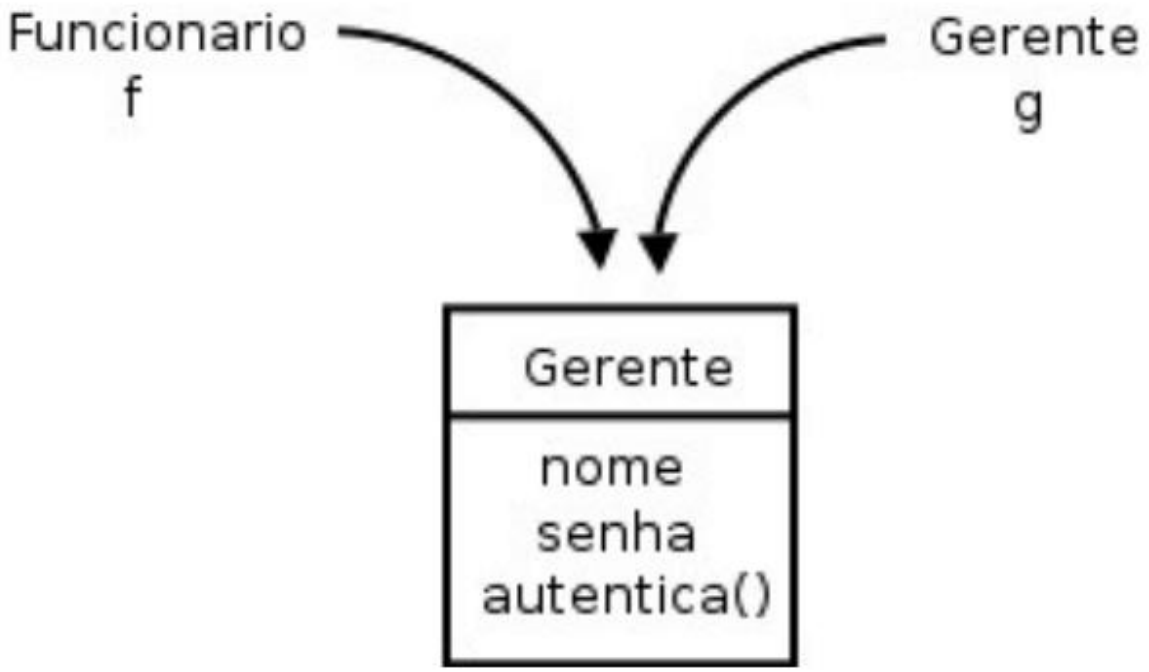
Polimorfismo

- O que guarda uma variável do tipo Funcionario? 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. Se alguém precisa falar com um Funcionario do banco, pode falar com um Gerente! Por quê? Pois, Gerente é um Funcionario. Essa é a semântica da herança.

Anhanguera

Polimorfismo



Anhanguera

Atributos e métodos estáticos

- Ao criarmos instâncias de uma classe por meio da palavra reservada new, cada instância da classe terá uma cópia de todos os campos declarados na classe.

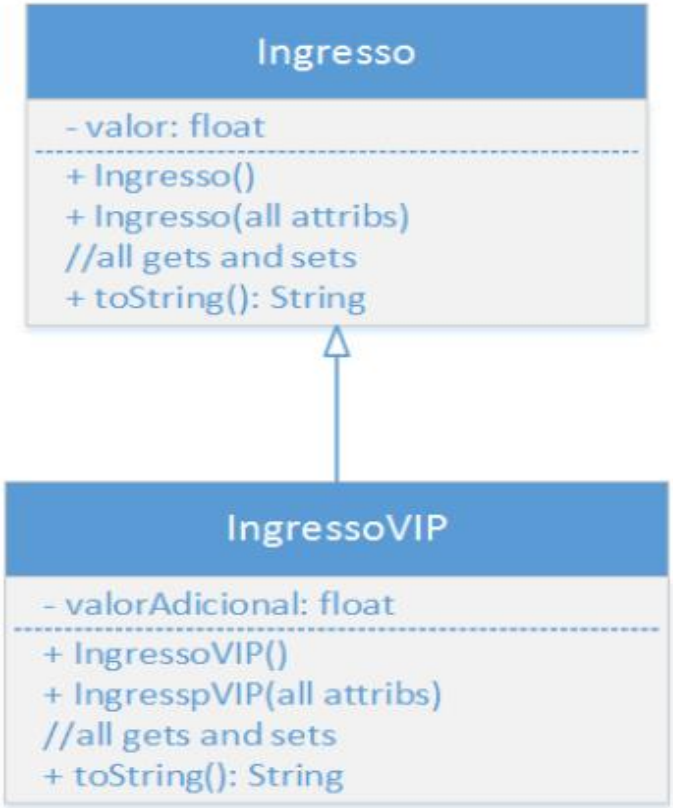
- Para acessar atributos ou métodos estáticos de uma classe não há necessidade de instanciar a classe.

Exercícios

- **1)** Utilizando projeto Calculadora, crie três sobrecargas do método subtrair, uma recebendo como parâmetros dois tipos double, outra recebendo um int e um double e uma terceira recebendo um double e um int.

Exercícios

» **2)** Crie as classes conforme o diagrama. IngressoVIP herda de Ingresso e possui um atributo valor Adicional. O método toString da classe IngressoVIP deve considerar que o valor do ingresso é o valor da superclasse somado ao valor Adicional do IngressoVIP.




Anhanguera