Programação Orientada a Objetos

Aula 04: um pouco mais de sintaxe

Ivo Calado

ivo.calado@ifal.edu.br

Instituto Federal de Educação, Ciência e Tecnologia de Alagoas

3 de Novembro de 2016

Roteiro

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA ALAGOAS

Comentário de linha

- Java fornece dois tipos de comentários, que são os comentários de linha e de bloco
- Comentários de linha iniciam com // enquanto que comentários de bloco iniciam com /* e terminam com */

```
public class Pessoa {
  //Este é um comentário de linha

  /*
     Este é um comentário de bloco
  */

}
```

Gerando documentação

- Além da adição de comentários, a JDK oferece o suporte para geração automática de documentação com base nos comentários
- A geração dos comentários se dá a partir dos comentários de bloco adicionados imediatamente antes das classes, métodos e atributos
- Todavia, para que um comentário de bloco possa ser incluído como parte da documentação ele deve iniciar com /** ao invés de simplesmente /*

Ex 1: Adição de documentação à classe

```
package br.edu.ifal.poo;

/**
 * Classe tem o objetivo de representar uma pessoa.
    Cada pessoa possui um nome,
 * cpf e endereço
 *
 * @author ivocalado
 * @version 1.5
 * @since 1.1
 * @see br.edu.ifal.poo.Endereco
 */
public class Pessoa {
//Métodos e atributos
}
```

Parâmetros utilizados

- @autor: geralmente utilizado para indicar o autor do arquivo. Podem ser utilizadas várias entradas para indicar os autores da aplicação
- @version: indica a versão atual do software
- @since: indica em qual versão a classe, método ou atributo foi inserido na aplicação
- @see adiciona um link para uma determinada classe

Ex 2: Adição de documentação aos atributos

```
public class Pessoa {
        /**
         * representa a propriedade nome da entidade
         */
        private String nome;
        /**
         * representa o cpf
         */
        private String cpf;
        /**
         * representa o endereço da pessoa
         * @see br.edu.ifal.poo.Endereco
         */
        private Endereco endereco;

}
```

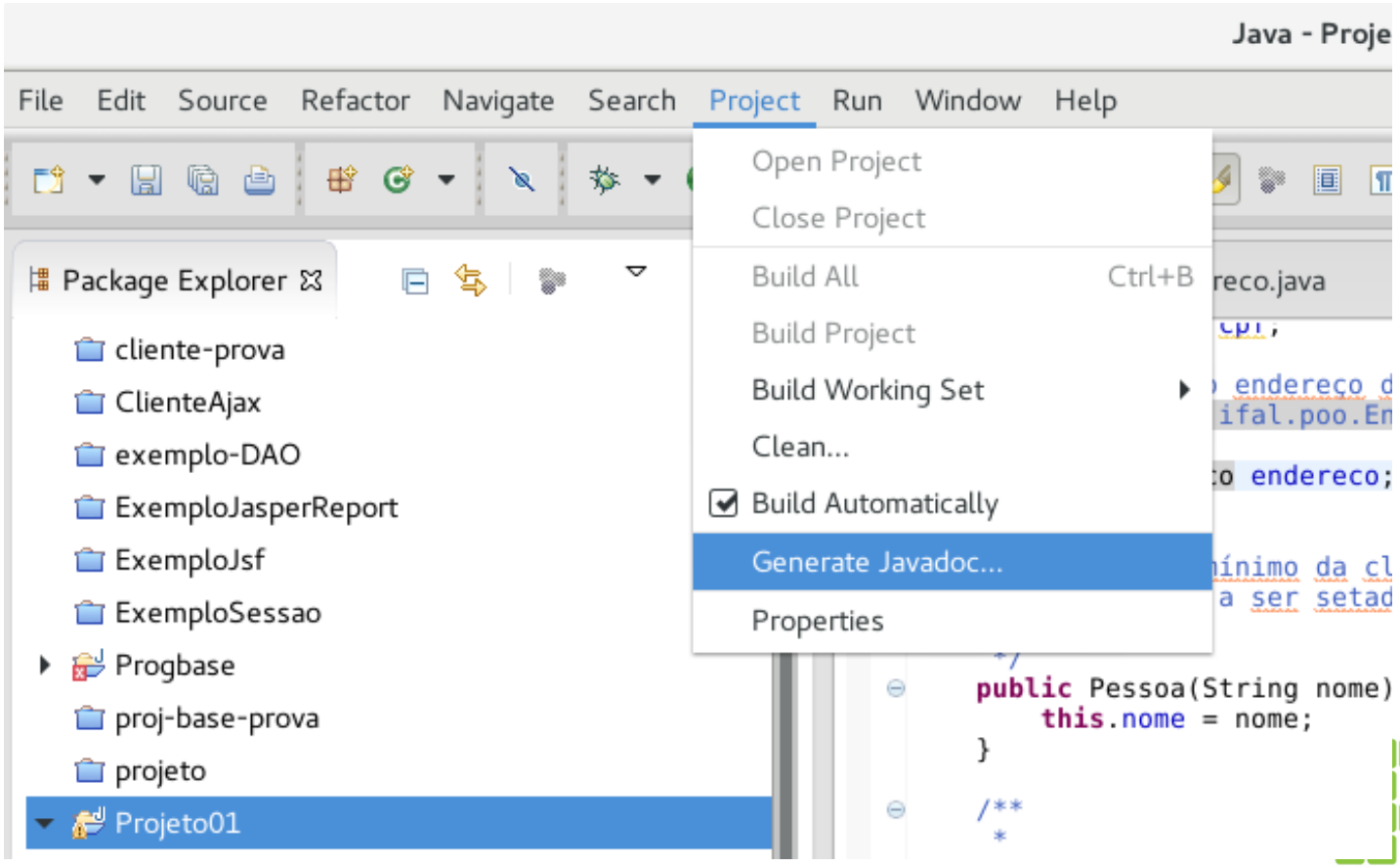
Ex 3: Adição de documentação aos métodos

```
// Definições da classe e atributos
        /**
         * Construtor mínimo da classe Pessoa.
         * @param nome a ser setado
         * @param cpf a ser setado
         */
        public Pessoa(String nome, String cpf) {
                this.nome = nome;
        }
        /**
         * @return o endereco da pessoa ( @see {@link
            Endereco})
         */
        public Endereco getEndereco() {
                return endereco;
        }
```

Parâmetros utilizados

- @param: indica os parâmetros de entrada do método
- @return: o parâmetro de retorno do método
- @link uma outra forma de referenciar uma classe

Gerando javadoc a partir do Eclipse



java.util.Random

- Objetivos: gerar números **pseudo-randomicos**

```
import java.util.Random;
//...
   Random r = new Random();
   Random r2 = new Random(10);//Seed predefinida
   int v1 = r.nextInt();
   int v2 = r.nextInt(10);
   float v3 = r.nextFloat();
   double v4 = r.nextDouble();
   boolean v5 = r.nextBoolean();
   long v6 = r.nextLong();
```

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA ALAGOAS

java.lang.Math

```
double e = Math.E;
double pi = Math.PI;
int v1 = Math.abs(-10); //retorna o número absoluto
int v2 = Math.max(10, 5); //retorna o valor máximo
int v3 = Math.min(10, -2); //retorna o valor mínimo
double v4 = Math.ceil(10.4); //calcula o proximo
    inteiro superior
double v5 = Math.floor(10.4); //calcula o proximo
    inteiro inferior
double v6 = Math.pow(2, 3); // 2 elevado ao cubo
double v7 = Math.sqrt(25);//Raiz quadrada de 25
double v8 = Math.round(2.4999);//Arredondamento
//Funções relacionadas à trigonometria
```

java.util.Date e java.util.SimpleDateFormat

```
SimpleDateFormat pr = new SimpleDateFormat("dd/MM/
    yyyy");
try {
        Date d = pr.parse("23/07/2016");
        String resultado = pr.format(d);
} catch (ParseException e) {
}
```

O problema da repetição de código I

- Considere que foi solicitada a criação de uma classe Funcionario nos seguintes moldes:

```
public class Funcionario {
  String nome;
  String cpf;
  double salario;
 // setters e getters
}
```

O problema da repetição de código II

- Considere agora que é necessário criar um tipo específico de funcionário, denominado gerente, que deveria ter como campos extras a senha, o número de funcionários gerenciados e um método de autenticação. Uma possível implementação seria:

```
public class Gerente {
  String nome;
  String cpf;
  double salario;
  int senha;
  int numeroDeFuncionariosGerenciados;

  public boolean autentica(int senha) {
```

O problema da repetição de código III

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

    // outros métodos
}
```

Qual o problema?

- Duplicação de código!
- Toda vez que precisarmos criar um novo tipo de funcionário os métodos e atributos seriam duplicados

Poderíamos definir uma única classe funcionário com todos os atributos possíveis. Mas quais os problemas dessa abordagem?

Qual o problema?

- Duplicação de código!
- Toda vez que precisarmos criar um novo tipo de funcionário os métodos e atributos seriam duplicados

Poderíamos definir uma única classe funcionário com todos os atributos possíveis. Mas quais os problemas dessa abordagem?

- Vários atributos ficariam vazios
- Conforme precisarmos criar novos tipos de funcionário, a classe acaba por ficar "inchada"

Qual a solução? Herança!

- Herança é uma forma de relacionamento entre duas classes de tal maneira que uma delas (**subclasse**) herda atributos e métodos da outra (**superclasse**)
- No nosso exemplo, a classe **Gerente** seria uma **extensão** (ou especialização) da classe **Funcionario**

Quais informações são herdadas?

- métodos e atributos públicos
- métodos e atributos protegidos (**protected**)
- métodos e atributos com encapsulamento *default* são herdados apenas se as classes estiverem no mesmo pacote

Como ficaria nossa classe Gerente?

- Para indicarmos que uma classe herda da outra fazemos uso da palavra-chave **extends**

```
public class Gerente extends Funcionario {
  int senha;
  int numeroDeFuncionariosGerenciados;
  public boolean autentica(int senha) {
    String nomeGerente = getNome();
    if (this.senha == senha) {
      System.out.println("Acesso Permitido a " +
          nomeGerente);
      return true;
    } else {
      System.out.println("Acesso Negado a "+
          nomeGerente);
      return false;     } } }
```

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA ALAGOAS

Subclasse x superclasse

Super e Sub classe

A nomenclatura mais encontrada é que **Funcionario** é a superclasse de Gerente, e Gerente é a **subclasse** de Funcionario. Dizemos também que todo Gerente **é um** Funcionário. Outra forma é dizer que Funcionario é classe **mãe** de Gerente e Gerente é classe **filha** de Funcionario.

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA ALAGOAS

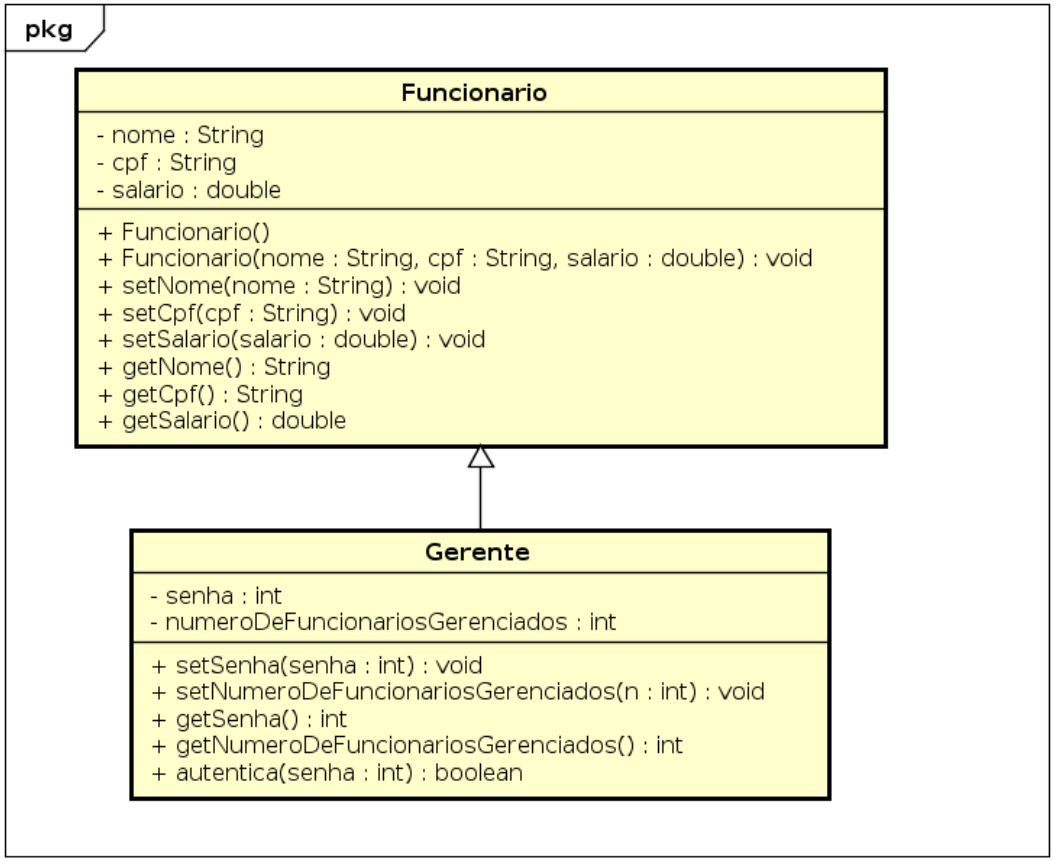
Como utilizar a classe Gerente?

- Da mesma forma que a classe **Funcionario!**

```
public class TestaGerente {
  public static void main(String[] args) {
    Gerente gerente = new Gerente();

    // podemos chamar métodos do Funcionario:
    gerente.setNome("João da Silva"); //método
        pertencente à Funcionário

    // e também métodos do Gerente!
    gerente.setSenha(4231);
  }
}
```

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA ALAGOAS

Qual a representação UML da herança?



INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA ALAGOAS

O modificador de acesso protected I

Considere a situação

Suponha que se deseja construir uma classe **Funcionario** na qual alguns métodos deveriam ser acessados **apenas** pelas classes filhas. É possível com o modificador **protected**!

```
public class Funcionario {
    private int v1;
    protected int v2;
    protected void metodo1() {
    }
}

public class Gerente extends Funcionario {
```

O modificador de acesso protected II

```
    public void metodo2() {
      metodo1(); //Ok!, Gerente herda o metodo1 e pode
          utilizá-lo!
      v1 = 10; //Erro, não é possível acessar um
          atributo privado
      v2 = 20; //Ok! É possível acessar um atributo
          protegido
    }
}
```

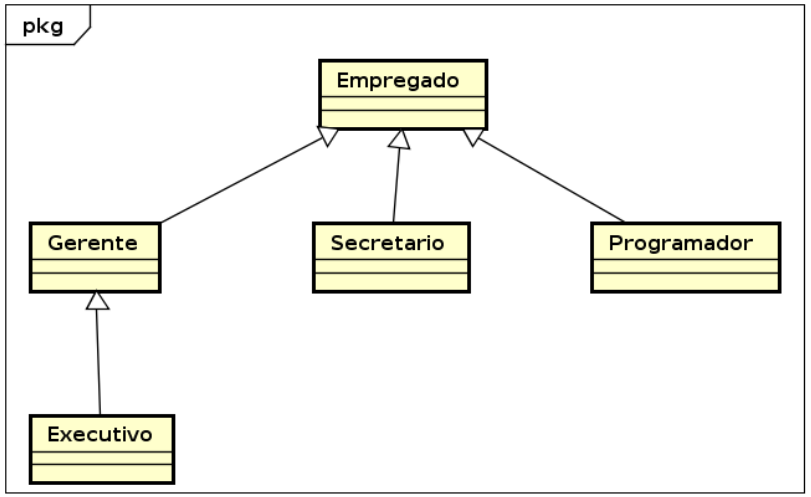
Palavra-chave super

- De maneira semelhante ao **this**, a palavra-chave **super** possibilita fazer referências a métodos, atributos e construtores da superclasse

```
public classe Gerente extends Funcionario {
  public Gerente(String nome, String cpf, double
      salario) {
    super(nome, cpf, salario); //invoca o construtor
        de Funcionário!
  }
  public void metodo1() {
  //
    String nome = super.getNome();
  //
  }
}
```

Hierarquia de classes

- O processo de herança não se restringe a uma única camada. Podem existir um número indefinido de camadas

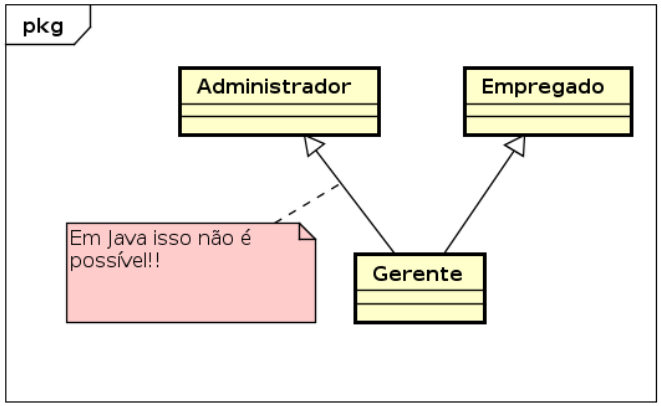


- A classe **Executivo** irá herdar os métodos e atributos de **Gerente** e **Empregado**

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA ALAGOAS

Java: herança única!

- Em Java, cada classe pode herdar diretamente de apenas uma **única classe**



Curiosidade: alguma linguagem dá suporte à herança múltipla?

Sim! C++ é um exemplo de uma linguagem orientada a objetos com suporte à herança múltipla

A classe java.lang.Object

- Em Java, a classe **Object** do pacote **java.lang** é superclasse de qualquer outra classe
- Se uma classe não herda explicitamente de nenhuma outra classe então ela **herda** de Object
- Significa dizer que qualquer classe **é um** Object

Principais métodos	**Descrição**
boolean equals(Object)	Compara dois objetos e retorna **true** se forem iguais
int hashCode()	Gera um identificador inteiro com base nos atributos do objetos
String toString()	Gera uma representação textual do objeto

Sobreposição de métodos I

- Um dos recursos do Java é a possibilidade de reescrever métodos para adicionar novas funcionalidades às subclasses
- A subclasse passa a ter a nova funcionalidade

```
public class Funcionario {
  protected double salario;

  public Funcionario(double salario) {
    this.salario = salario;
  }
  public double getBonificacao() {
    return this.salario * 0.10;
  }
```

Sobreposição de métodos II

```
  // métodos
}

public class Gerente extends Funcionario {

  public Gerente(double salario) {
    super(salario);
  }

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

Utilização:

Sobreposição de métodos III

```
//...
  Funcionario f = new Funcionario(1000);
  double grat1 = f.getBonificacao(); // == 100


  Gerente g = new Gerente(1000);
  double grat2 = g.getBonificacao();// == 150

//...
```

Como reutilizar o método da classe base?

Considere a seguinte situação

A empresa possui a seguinte regra para bonificação dos seus funcionários: funcionários comuns recebem uma bonificação de 10% enquanto Gerentes devem receber R$ 100,00 além da bonificação padrão. Como fazer?

Como reutilizar o método da classe base?

Considere a seguinte situação

A empresa possui a seguinte regra para bonificação dos seus funcionários: funcionários comuns recebem uma bonificação de 10% enquanto Gerentes devem receber R$ 100,00 além da bonificação padrão. Como fazer?

- Implementar o método getBonificacao com a funcionalidade completa (cálculo dos 10% e somatório do valor extra). Qual o problema?
- Reutilização do método de cálculo de bonificação: soma-se a bonificação geral com o valor da bonificação específica

Como implementar? I

```
public class Funcionario {
  protected double salario;

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

public class Gerente extends Funcionario {
```

Como implementar? II

```
    public Gerente(double salario) {
      super(salario);
    }

    public double getBonificacao() {
      return super.getBonificacao() + 100;
    }
}
```

Sobrecarga x Sobreposição

- **Sobrecarga (overload):** criação de diferentes métodos na mesma classe, com mesmo nome e assinaturas diferentes (parâmetros de entrada e retorno)
- **Sobreposição (reescrita ou override):** reescrita de um método na subclasse com a mesma assinatura

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA ALAGOAS

Classes final e métodos final I

Situação 1

Seu sistema possui uma classe Departamento que não deve, sob hipótese alguma, ser modificado seu comportamento na forma de especialização. Em outras palavras, você deseja **impedir que qualquer outra classe herde** da classe Departamento

Classes final e métodos final II

Situação 2

Seu sistema possui uma classe Funcionario e que por sua vez possui um método para cálculo dos descontos de impostos. Você deseja impedir que qualquer classe que venha a herdar de Funcionario modifique a implementação desse método. Em outras palavras, **você deseja impedir a sobreposição do método**

- Para ambos os casos devemos fazer uso da palavra chave **final** na classe e no método respectivamente
- A utilização da palavra chave **final** na classe impede que qualquer outra classe possa herdar dela

Classes final e métodos final III

```
public final class Departamento {
 //Métodos e atributos
}

//ERRO! Não é possível herdar de uma classe final
public class DepartamentoEspecializado extends
    Departamento {

}
```

- A utilização da palavra chave **final** no método impede que qualquer outra classe que venha a herdar dela sobrescreva o método **final**

Classes final e métodos final IV

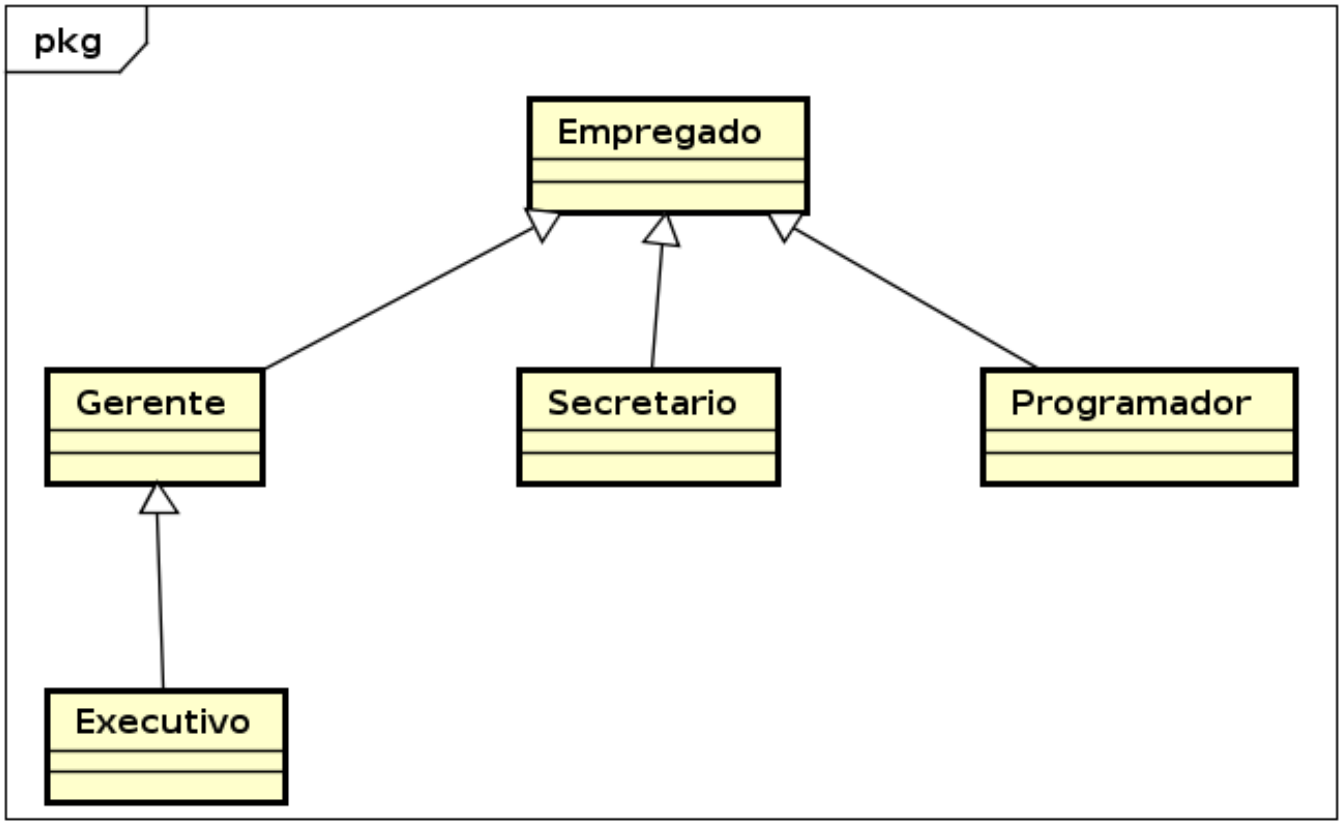
```
public class Funcionario {
  public final double calcularDescontos() {
    return 300;
  }
}

public class Gerente extends Funcionario {
  public double calcularDesconto() {
    return 100;//ERRO! Não é possível sobrescrever um
        método final!
  }
}
```

Atribuindo uma subclasse à uma superclasse I

- Tendo em vista que herança é um relacionamento do tipo **é um** pode-se entender que um objeto da classe especializado **também** é um objeto da classe base
- Com base nisso, a atribuição de tipos especializados para tipos base pode ser feito de maneira direta

Considere a seguinte hierarquia:

Atribuindo uma subclasse à uma superclasse II



Atribuindo uma superclasse à uma subclasse

E no código abaixo, o que acontece?

```
Gerente g1 = new Gerente();
Empregado e1 = g1;
Gerente g2 = e1;
```

Atribuindo uma superclasse à uma subclasse

E no código abaixo, o que acontece?

```
Gerente g1 = new Gerente();
Empregado e1 = g1;
Gerente g2 = e1;
```

- Erro! Não é possível atribuir um tipo geral para um tipo específico diretamente **mesmo que ele seja do tipo destino**
- Para realizar a atribuição deve-se fazer um **cast**

```
Gerente g1 = new Gerente();
Empregado e1 = g1;
Gerente g2 = (Gerente) e1;
```

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA ALAGOAS

Verificando o tipo real do objeto via instanceof

O que acontece quando testamos o seguinte código?

```
Empregado e1 = new Gerente();//Ok
Gerente g1 = (Gerente)e1; //Ok
Programador p1 = (Programador)g1;
```

Verificando o tipo real do objeto via instanceof

O que acontece quando testamos o seguinte código?

```
Empregado e1 = new Gerente();//Ok
Gerente g1 = (Gerente)e1; //Ok
Programador p1 = (Programador)g1;
```

- Não é possível forçar um cast pois um Gerente **não é** um **Programador**
- Para verificar se um objeto é de um tipo específico utiliza-se **instanceof**

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA ALAGOAS

Usando o instanceof

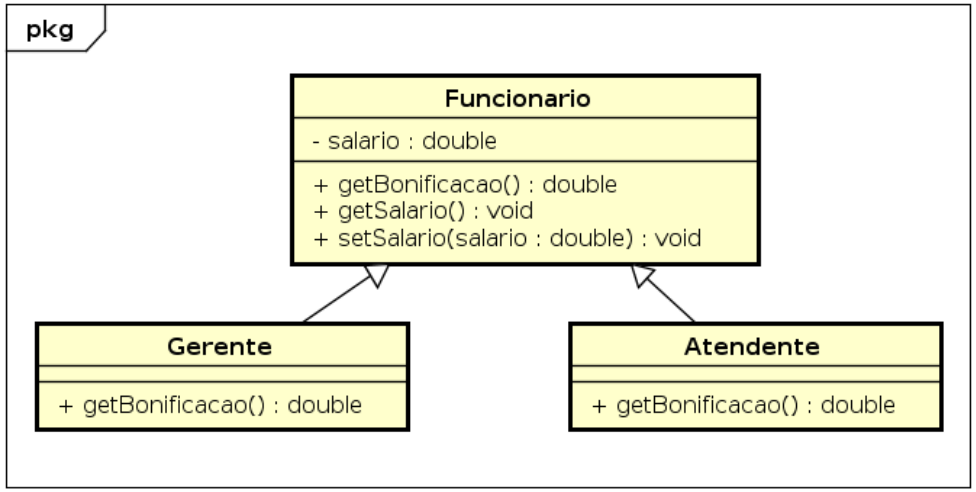
```
Funcionario f = ...//algum tipo de instanciação
if (f instanceof Gerente) {
//É seguro fazer o cast
   Gerente ge = (Gerente) f;
}
```

- A palavra chave **instanceof** retorna **true** se o objeto testado for do tipo a ser testado

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA ALAGOAS

Polimorfismo

Considere o seguinte diagrama de classes:



Como seria a implementação dessas classes se um Funcionário padrão tivesse 5% de bonificação, o Gerente tivesse 20% e o Atendente tivesse 15% de bonificação?

O que acontece se invocarmos o getBonificação

Agora considere o seguinte código:

```
Funcionario f = new Gerente();
f.setSalario(100);
double s = f.getSalario();
double gratificacao = f.getBonificacao();
```

- Qual será o valor da bonificação? 5 ou 20?

O que acontece se invocarmos o getBonificação

Agora considere o seguinte código:

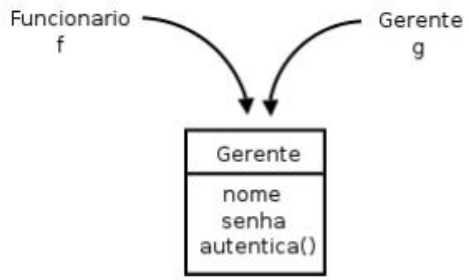
```
Funcionario f = new Gerente();
f.setSalario(100);
double s = f.getSalario();
double gratificacao = f.getBonificacao();
```

- Qual será o valor da bonificação? 5 ou 20?

20! Pois o objeto alocado é de um **Gerente ainda que** esteja sendo representado como um **Funcionario**

Definição de Polimorfismo

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



Definição

Polimorfismo é a capacidade de um objeto poder ser referenciado de várias formas. A invocação do método 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 com a que estamos referenciando.

Quando é útil utilizar polimorfismo?

- Sempre que for necessário tratar os objetos de maneira uniforme
- Suponha que precisamos criar uma classe para acumular as bonificações de todos os funcionários

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

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA ALAGOAS

Quando é útil utilizar o polimorfismo?

```
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());
```

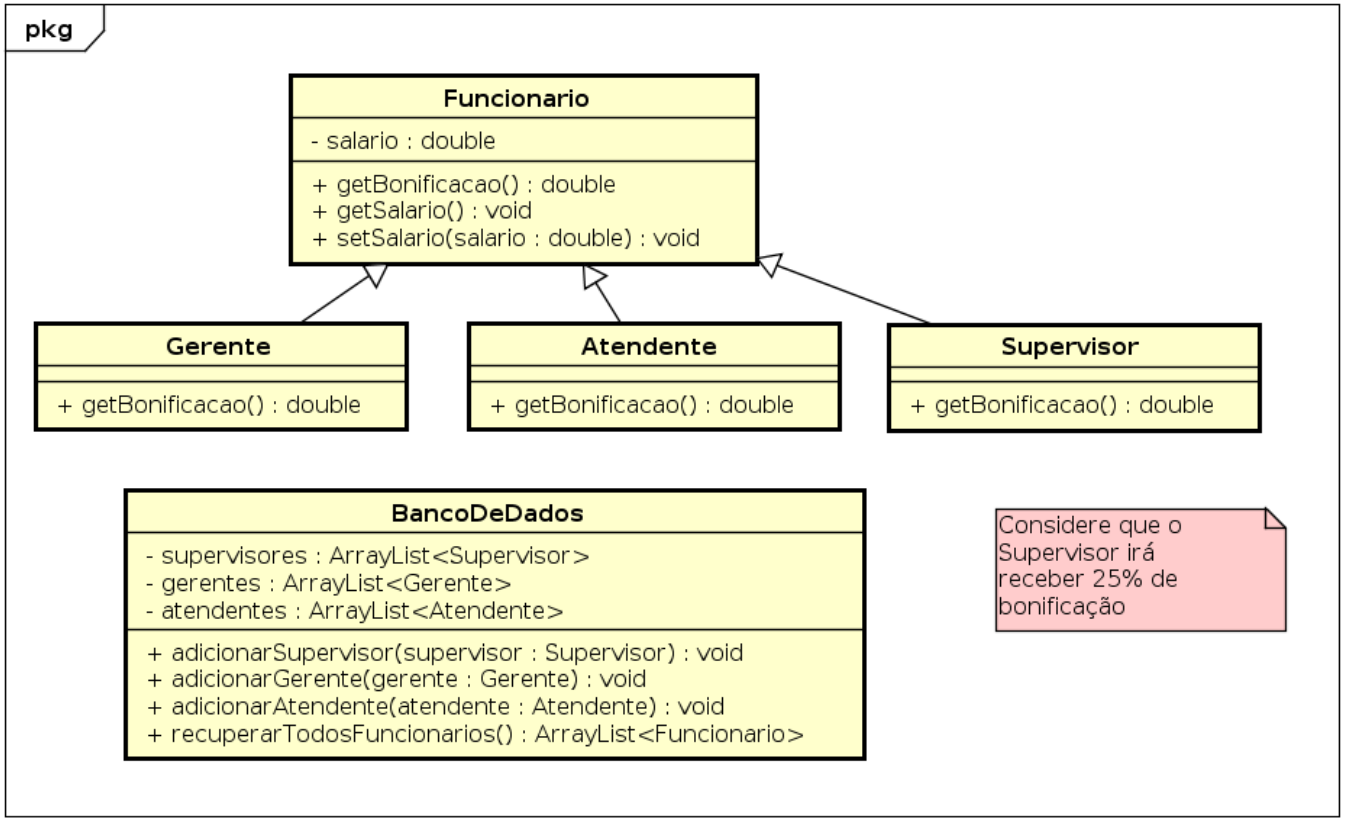
Aplicação prática do polimorfismo: System.out.println()

- Não há uma implementação do método println para todos as classes complexas (nem seria possível)
- A única coisa que é necessário é definir o método println para superclasse e utilizar o polimorfismo!

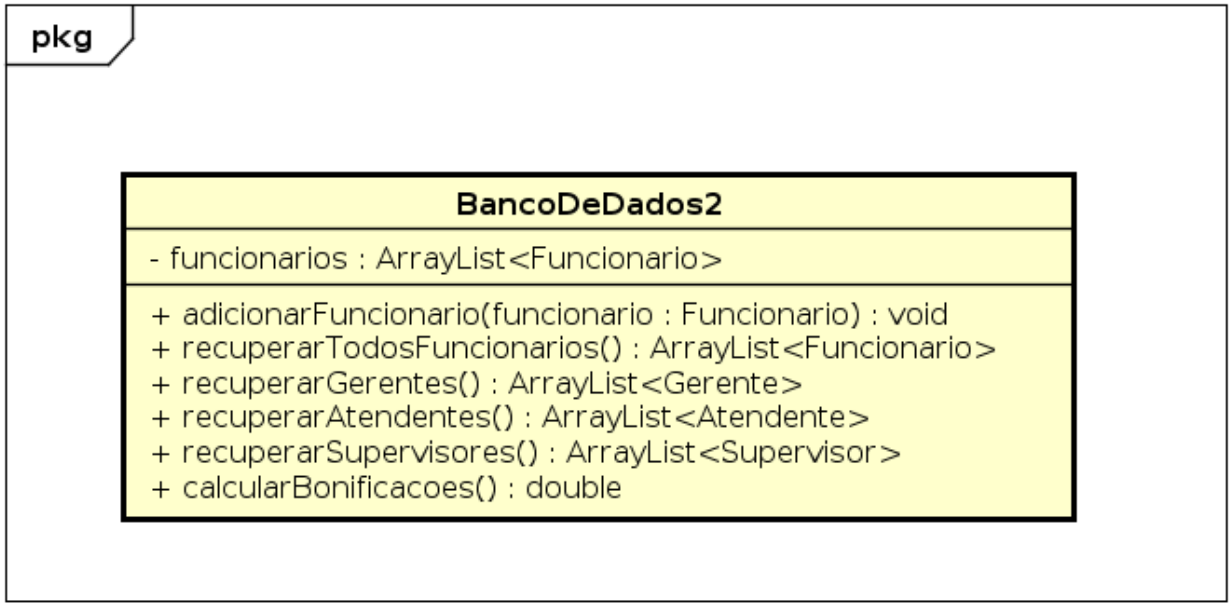
```
//...
public void println(Object o) {
   String mensagem = o.toString(); //Irá recuperar a
       implementação correta dado o tipo real do objeto
   print(mensagem);
   print("\n")
}
```

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA ALAGOAS

Exercício 1


pkg
Funcionario
- salario : double
+ getBonificacao() : double
+ getSalario() : void
+ setSalario(salario : double) : void
Gerente
+ getBonificacao() : double
Atendente
+ getBonificacao() : double
Supervisor
+ getBonificacao() : double
BancoDeDados
- supervisores : ArrayList<Supervisor>
- gerentes : ArrayList<Gerente>
- atendentes : ArrayList<Atendente>
+ adicionarSupervisor(supervisor : Supervisor) : void
+ adicionarGerente(gerente : Gerente) : void
+ adicionarAtendente(atendente : Atendente) : void
+ recuperarTodosFuncionarios() : ArrayList<Funcionario>
Considere que o Supervisor irá receber 25% de bonificação
powered by Astah

Exercício 2



Dica: utilizar o **instanceof** para identificar os tipos concretos nos métodos de recuperação

Reescrevendo o toString() I

Considere o seguinte código:

```
Funcionario f = new Funcionario();
System.out.println(f);
```

- Ao executar esse código é impresso um texto semelhante a "pac1.Funcionario@4f93b604"
- Isto acontece pois o método **println** invoca **toString** de cada objeto que por padrão retorna uma representação do endereço de memórica do objeto
- No entanto, é possível sobrescrever o método toString para fornecer uma saída customizada

Reescrevendo o toString() II

```
public class Funcionario {
  private String nome;
  private String cpf;
  //getters e setters

  public String toString() {
    return "nome=" + nome + ", cpf=" + cpf;
  }
}

public class Main {
  public static void main(String[] args) {
    Funcionario f2 = new Funcionario();
    f2.setNome("Maria");
```

Reescrevendo o toString() III

```
        f2.setCpf("1234");
        System.out.println(f2);
    }
}
```

Reescrevendo o equals I

- Para compararmos objetos em Java utilizamos o método **equals**
- Por padrão este método verifica se as duas refereências apontam para o mesmo endereço de memória

```
Funcionario f1 = new Funcionario();
Funcionario f2 = new Funcionario();
if(f1.equals(f2))  {

} else {//Sempre entrará aqui pois os objetos são
    diferentes

}
```

Reescrevendo o equals II

- E se quisermos testar se dois objetos são iguais a partir do seu cpf?
- Devemos sobrescrever o método equals

```
public class Funcionario {
  private String nome;
  private String cpf;

  public boolean equals(Object obj) {
    //compara-se o objeto recebido com o objeto testado
  }
}
```

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA ALAGOAS

Reescrevendo o equals III

- A assinatura do método equals recebe por padrão um Object. O que devemos fazer?
- Utilizar o instanceof e cast para converter para o tipo pretendido

```
public boolean equals(Object obj) {
  if (obj instanceof Funcionario) {
    Funcionario f = (Funcionario) obj;
    return cpf.equals(f.cpf); //Por que não é
        necessário chamar getCpf?
  } else {
    return false;
  }
}
```

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA ALAGOAS

Reescrevendo o equals IV

Curiosidade: como um ArrayList remove um elemento?

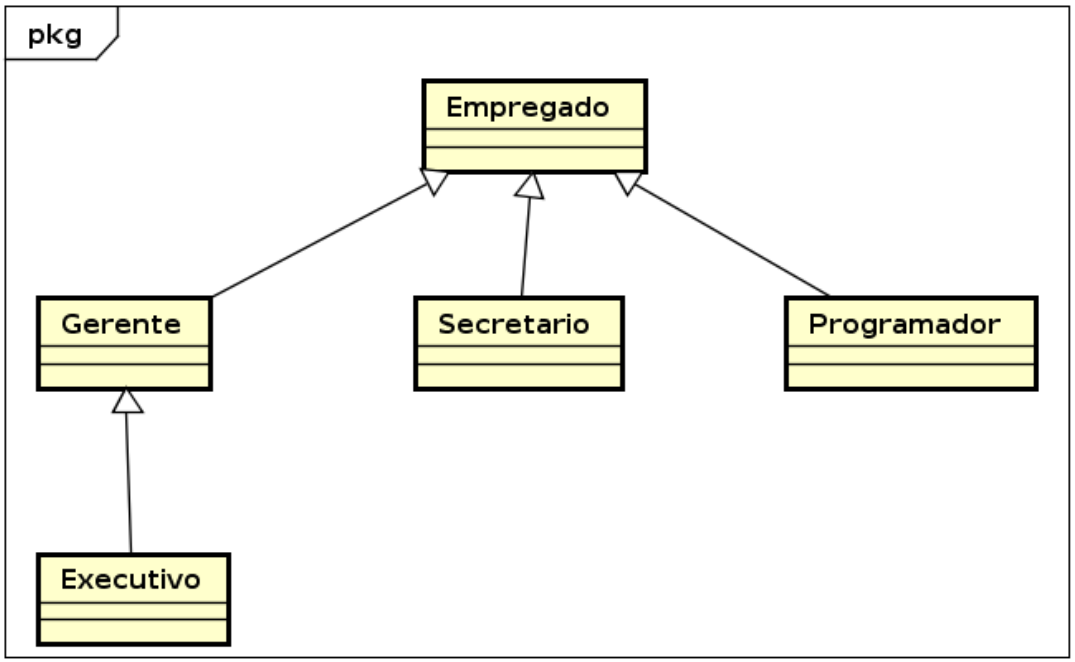
Ele faz uso do método equals para remover objetos

```
ArrayList<Funcionario> lista = new ArrayList<>();
lista.add(f2);
System.out.println(lista.size()); //==1
Funcionario f3 = new Funcionario();
f3.setCpf("1234");
lista.remove(f3);
System.out.println(lista.size()); //==0
```

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA ALAGOAS

Classes abstratas I

Considere a hierarquia abaixo



Classes abstratas II

Agora considere o seguinte cenário

O sistema possui um tipo geral **Empregado** em que **parte** dos seus atributos e a implementação dos seus métodos servem a maioria dos casos. No entanto, alguns métodos **não possuem** uma implementação padrão (ex.: o cálculo da bonificação) devendo sempre ser redefinidas. O que fazer?

- Definir a classe Empregado como **abstrata**
- Manter o método de cálculo de bonificação sem implementação

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA ALAGOAS

Classes abstratas III

```
public abstract class Empregado {
   private String nome;
   private String cpf;
   private double salario;

   public abstract double calcularBonificacao();//método
        abstrato. Sem implementação!

   public String setNome(String nome) {
      this.nome = nome;
   }
   //demais métodos e atributos
}
```

Classes abstratas IV

```
public class Gerente extends Empregado {

//Implementação do método abstrato
public double calcularBonificacao() {
  return 0;
}
}

public class Secretario extends Empregado {

//Implementação do método abstrato
public double calcularBonificacao() {
  return 100;
}
}
```

Classes abstratas V

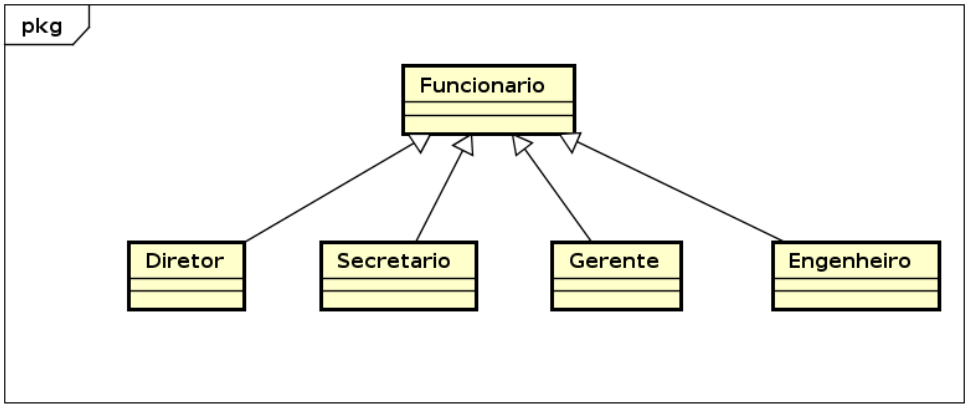
```
public class Main {
  public static void main(String[] args) {
    Gerente g = new Gerente();
    Secretario s = new Secretario();
    Empregado f = new Secretario();
    Empregado e = new Empregado();//Erro! Não é
        possívei instanciar um empregado!
  }
}
```

Importante!

Uma classe abstrata NÃO pode ser instanciada visto que ela é **incompleta**!

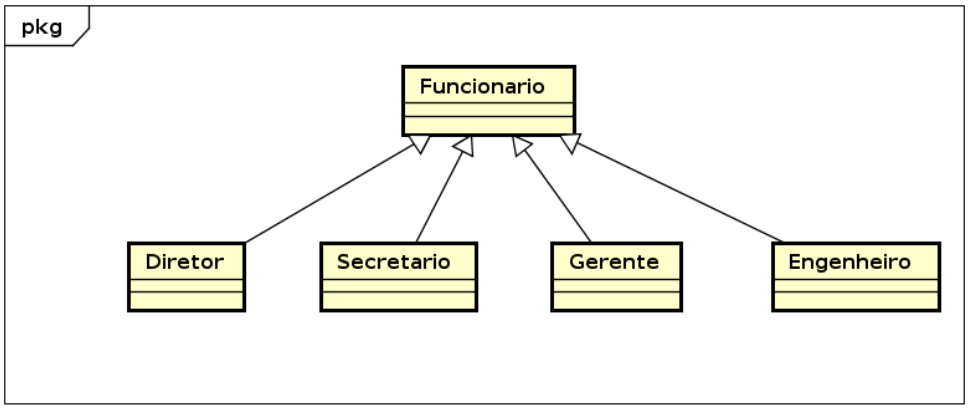
Interfaces

Considere a seguinte hierarquia de classes:



Interfaces

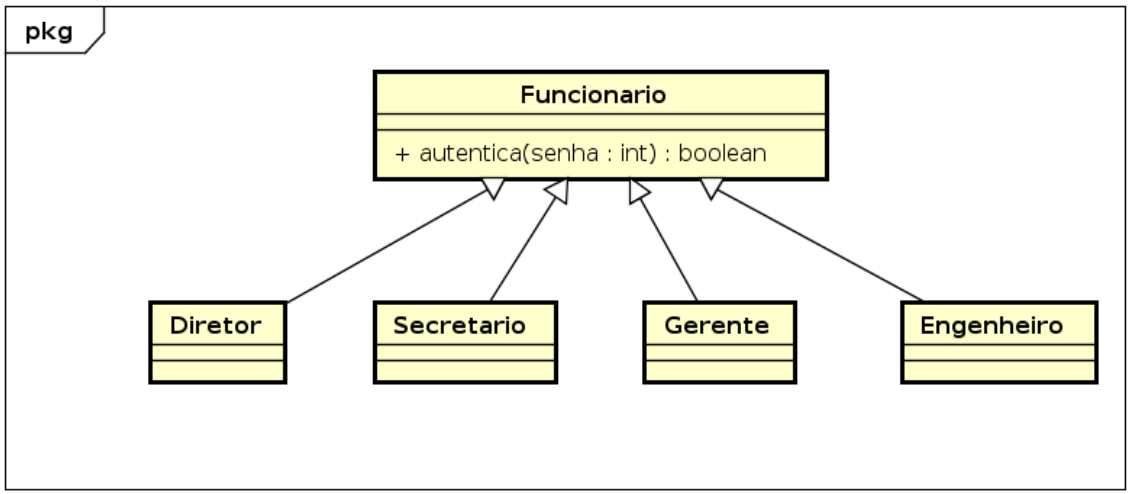
Considere a seguinte hierarquia de classes:



Suponha agora que os tipos **Diretor** e **Gerente** tenham de oferecer suporte a verificação de senha a partir de um método **public boolean autentica(int senha)**. Como modelar da maneira mais genérica possível?

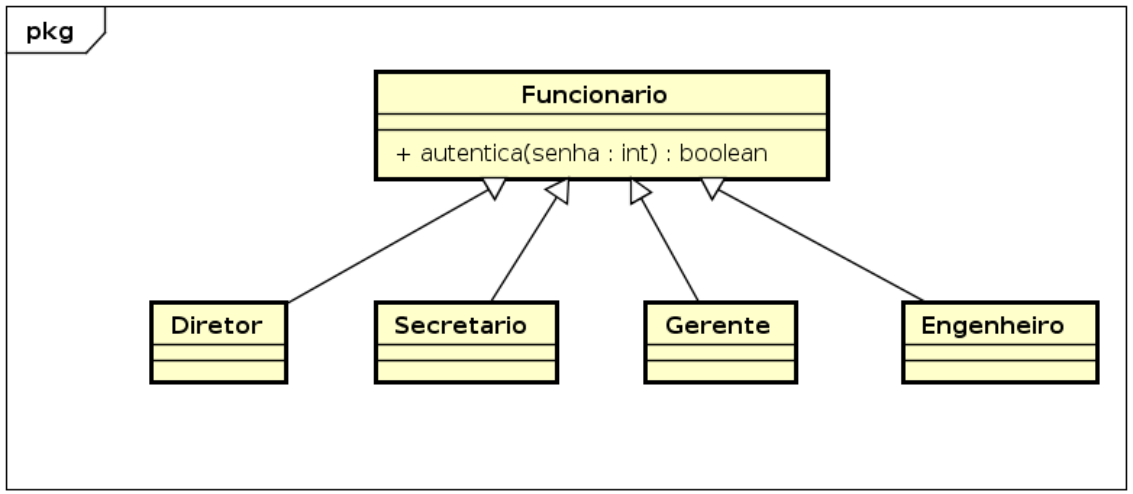
Solução 1

Solução? Qual o principal problema?



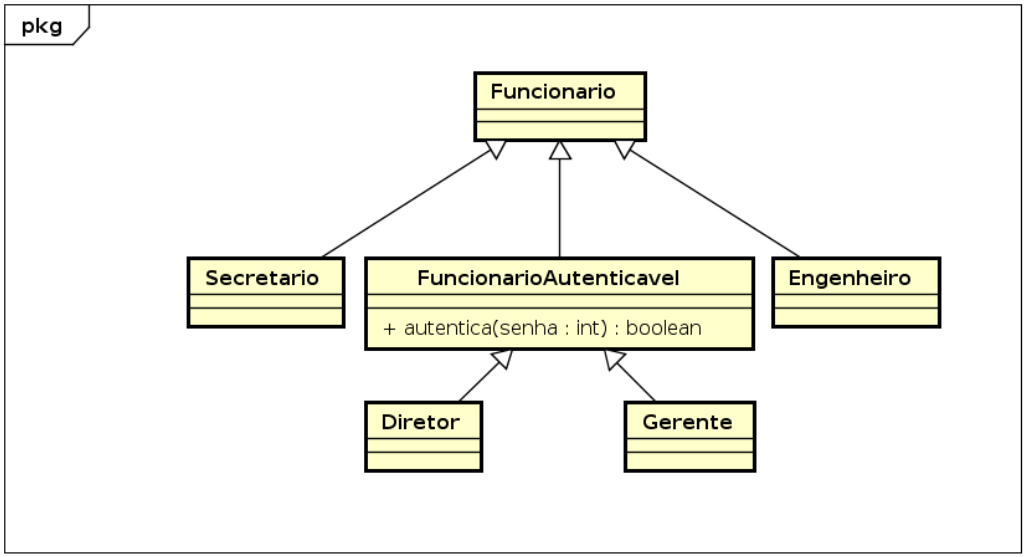
Solução 1

Solução? Qual o principal problema?



- Nem todos os funcionários fazem autenticação

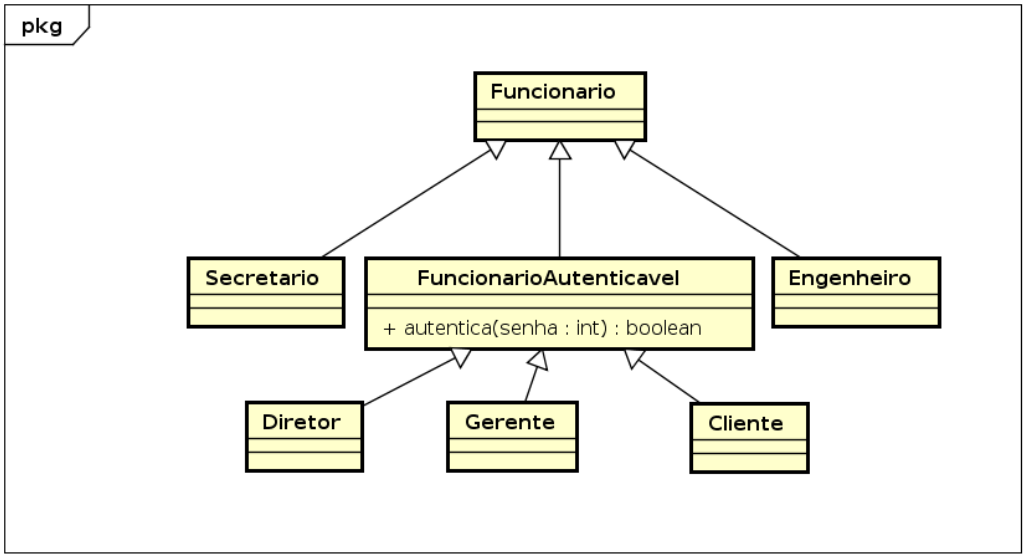
Solução 2



- Conseguimos isolar a implementação em um único ponto
- A classe **FuncionarioAutenticavel** poderia ser definida como **abstrata**

Mais problemas

- E se precisássemos adicionar uma classe Cliente que também tivesse suporte a autenticação. Como seria?



O que há de errado com isso?

Utilizando interfaces para especificar comportamentos

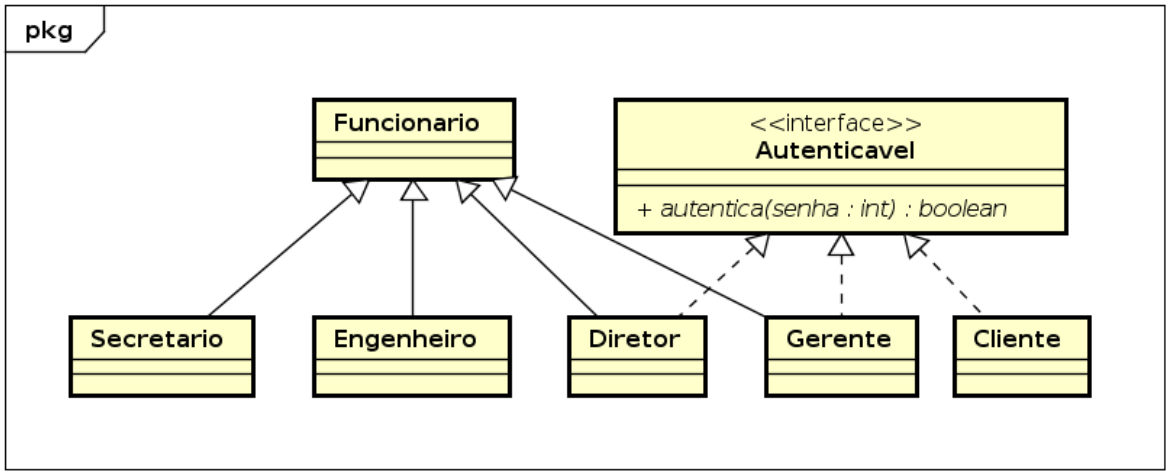
Para o nosso caso, devemos fazer uso de interfaces

Definição

Uma interface pode definir uma série de métodos, mas nunca conter implementação deles. Ela só expõe o que **o objeto deve fazer**, e não **como ele faz**, nem **o que ele tem**. **Como ele faz** vai ser definido em uma **implementação** dessa interface

- O mais próximo do que já vimos seria uma classe sem atributos e com todos os seus métodos abstratos
- Todavia, diferente da herança, uma classe pode implementar várias interfaces de uma vez só

Representação em UML



- A interface especifica os métodos (sem implementação) e as classes implementam tais métodos
- O relacionamento é chamado de **realização**

interfaces na prática I

Código da interface:

```
//Autenticavel.java
public interface Autenticavel {
        public boolean autentica(int senha);
}
```

Implementação:

interfaces na prática II

```
//Gerente.java
public class Gerente extends Empregado implements
    Autenticavel {
        public boolean autentica(int senha) {
                //implementação
        }

        //demais atributos e métodos
}
```

Como seria possível a realização de duas interfaces ao mesmo tempo?

interfaces na prática III

```
//Gerente.java
public class Gerente extends Empregado implements
    Autenticavel, SegundaInterface {
        //atributos e métodos
}
```

Exemplos de aplicação

- Listas em java
- Ordenador de objetos

Herança simples, classe abstrata e interfaces I

Considere os seguintes cenários de um sistema com uma classe **Funcionario**

Cenário 1: customização opcional

O sistema possui uma tipo geral **Funcionario** em que seus atributos e a implementação dos seus métodos servem a maioria dos casos. No caso das exceções, cria-se uma classe especializada herdada de **Funcionario**. Ex.: **Gerente**

- Utiliza-se a herança tradicional
- Pode-se criar objetos tanto da classe **Funcionario** quanto das classes **filhas**

Herança simples, classe abstrata e interfaces II

```
//...
Gerente g = new Gerente(); // Ok, é possível criar um
    gerente!
Funcionario f = new Funcionario(); //Ok, é possível
    criar um funcionário!
Funcionario f2 = new Gerente(); //Ok, é possível criar
    um gerente e atribuir a um funcionário!
//...
```

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA ALAGOAS

Herança simples, classe abstrata e interfaces III

Cenário 2: customização parcial obrigatória

O sistema possui um tipo geral **Funcionario** em que **parte** dos seus atributos e a implementação dos seus métodos servem a maioria dos casos. No entanto, alguns métodos **não possuem** uma implementação padrão devendo sempre ser redefinidas

- Define-se a classe **Funcionario** como **abstrata**
- Pode-se criar objetos apenas das classes **filhas** visto que o tipo **Funcionario** é **incompleto**

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA ALAGOAS

Herança simples, classe abstrata e interfaces IV

```
//...
Gerente g = new Gerente(); // Ok, é possível criar um
    gerente!
Funcionario f = new Funcionario(); //ERRO, NÃO é
    possível criar um funcionário!
Funcionario f2 = new Gerente(); //Ok, é possível criar
    um gerente e atribuir a um funcionário!
//...
```

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA ALAGOAS

Herança simples, classe abstrata e interfaces V

Cenário 3: customização total obrigatória

O sistema possui um tipo geral **Funcionario** em que **todos os métodos** não possuem uma implementação padrão devendo sempre ser redefinidas

- Define-se a classe **Funcionario** como uma **interface**
- Pode-se criar objetos apenas das classes **filhas** visto que o tipo **Funcionario** é **incompleto**

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA ALAGOAS

Herança simples, classe abstrata e interfaces VI

```
//...
Gerente g = new Gerente(); // Ok, é possível criar um
    gerente!
Funcionario f = new Funcionario(); //ERRO, NÃO é
    possível criar um funcionário!
Funcionario f2 = new Gerente(); //Ok, é possível criar
    um gerente e atribuir a um funcionário!
//...
```

Dicas de criação de classes

1. Sempre mantenha os dados privados
2. Sempre inicialize os dados da sua classe
3. Evite muitos tipos básicos na sua classe
4. Nem todos os atributos da sua classe precisam de métodos **get** e **set**
5. Divida as classes que possuem muitas responsabilidades
6. Ao definir nomes para suas classes, métodos e atributos escolha nomes que reflitam suas responsabilidades
   - É importante lembrar que **classes** representam entidades, **métodos** representam ações e **atributos** representam propriedades!