# Reuso por composição e herança

*Programação 2 – Aulas 12, 13 e 14*

## Objetivos da seção

- Apresentar o conceito de reuso de classes
- Apresentar o reuso por composição (solidificar o conceito)
- Apresentar o conceito de reuso por herança
- Como representar herança em UML
- Comparar reuso por composição e reuso por herança
- *Upcasting* e *downcasting*

## Reuso de classes

### Qual é o problema?

- Fazer software é difícil
- Fazer software é lento e caro
  - Time to market complica ainda mais!
  - Não temos tecnologia ainda para fazer software grande do zero, rapidamente e com poucos *bugs*

### Qual é a solução?

- O reuso é “o” caminho mais freqüentemente apontado

- As mesmas idéias básicas devem ter sido projetadas e reprojetadas pela mesma pessoa, ou pessoas diferentes, muitas vezes!
- Será que temos que começar tudo de novo, sempre???

## Por que a solução não é fácil?

- Reuso não acontece automaticamente!
- Para reusar, tem que:
  - Bolar boas abstrações (i.e., abstrações úteis); e
  - Empacotá-las para facilitar o reuso
- Como fazer isso? Que técnicas usar?
- Orientação a Objeto oferece o meio e prometeu muito, mas não cumpriu a promessa
  - No geral, o reuso não é uma realidade fácil de ser alcançada => reusar não é fácil!
  - Pouca gente conseguiu bons resultados
  - Ficou mais no marketing (com exceção de alguns bons programadores)
  - Por que reusar é difícil?
  - Sugestões de William Opdyke em “Refactoring, Reuse, and Reality”:

    i) Técnicos podem não entender o que reusar ou como reusá-lo;

    ii)Técnicos podem não ter a motivação para aplicar técnicas de reuso a não ser que benefícios de curto prazo sejam alcançáveis (porém, o reuso é mais um investimento

em longo prazo);

iii) Overhead, curva de aprendizado e custos de descobrimento devem ser endereçados para que o caminho do reuso seja bem sucedido.

## Tipos de reuso

- Falaremos das técnicas historicamente empregadas para atingir o reuso
  - Do mais simples/antigo para mais sofisticado/novo
  - A *granularidade* de reuso mudou radicalmente ao longo dos anos
  - Vamos parar em uma certa granularidade…

### *Fase pré OO: 1960-1970*

- **O que é reutilizado:** *linhas de código roubadas* de um programa e usadas em outro
  - Nome (moderno) da técnica: Copy-and-Paste
- **O que é reutilizado:** *código comum* de um programa
  - Nome da técnica: Subrotinas
- **O que é reutilizado:** *funções inteiras genéricas*, relacionadas, para servir em várias situações em programas diferentes
  - Nome da técnica: Bibliotecas

### *Fase da revolução OO: após 1980*

- **O que é reutilizado:** *abstrações/conceitos inteiros* através de classes que agregam dados e

seu comportamento (operações que podem ser realizadas sobre os dados)

- Nome da técnica: herança, composição/delegação
- Permite fazer *Programming-by-Difference*

- **O que é reutilizado:** *Contratos de relacionamento* entre objetos
  - Nome da técnica: Interfaces
  - Interface como conceito de "Plug Point"
  - Interface como conceito de "barreira de desacoplamento"
    i) Uma lei fundamental da programação: "*Program to an Interface, not to an Implementation*"
- Existe muito mais sobre reuso a ser dito, mas para a finalidade deste curso, paramos por aqui
  - Moral da história: reusar é difícil, mas fundamental
- Neste módulo do curso estaremos falando apenas de reuso de implementação
- Em um módulo seguinte falaremos mais profundamente sobre as interfaces
- Pensando em um grão maior  e padrões de projeto

# Composição

- Já fizemos reuso por composição várias vezes na nossa disciplina
  - Quem pode dar exemplos?
- Delegar responsabilidades a objetos de outras classes
  - Tipicamente, temos objetos que são atributos de uma classe

- Por exemplo, um objeto da classe Pessoa era o titular de uma conta
- Reusamos todo o código da classe Pessoa, atribuindo responsabilidades a objetos desta classe
- Por exemplo, no método getNomeTitular de ContaSimples3 há uma chamada ao método getNome de Pessoa
- A nossa classe Baralho é um agregado de cartas (objetos da classe Carta)

- Um cuidado ao realizar reuso por composição é se certificar de que o objeto da classe a ser reusada foi corretamente inicializado
  - Quando um objeto é acessado sem que tenha sido antes criado (com **new**), então uma exceção de tempo de execução é lançada, chamada NullPointerException
  - Falaremos mais adiante neste curso de detalhes mais sofisticados do tratamento de erros em Java
- Os objetos usados como atributos podem ser inicializados em diferentes momentos
  i) No construtor da classe que está reusando outra classe por composição
  ii)No momento em que eles são definidos
  iii) Imediatamente antes de usá-los (*lazy initialization*)
  iv) Usando inicialização de instância
- Vejamos um exemplo:

```
package p2.exemplos;


public class ComposicaoEInicializacao {
   // Inicialização no momento da definição
   private String campo1 = "Campo1 - apressadinho!!";

   private String campo2;
   private Integer campo3;
```

```
    private String campo4;

    private int campo5;
    private float campo6;
    private double campo7;
    private static final Object CONST = new Object();

    public ComposicaoEInicializacao() {
        System.out.println("campo1 = " + campo1);
        System.out.println("campo5 = " + campo5);
        System.out.println("campo6 = " + campo6);
        System.out.println("campo7 = " + campo7);
        // Inicialização no construtor
        campo2 = "O correto? Nem sempre... Depende.";
        System.out.println("campo2 = " + campo2);
    }

// Inicialização de instância (Instance initilization)
    {
        campo3 = new Integer(3);
        System.out.println("campo3 = " + campo3);
    }

    @Override
    public String toString() {
        if (campo4 == null) {
            // Inicialização antes de usá-los
            campo4 = "Campo4, atrasadinho e preguiçoso. Mas sempre chega a
tempo.";
            System.out.println("campo4 = " + campo4);
        }
```

```
        return "=> toString de ComposicaoEInicializacao";
    }

    public void printConst() {
        System.out.println(CONST);
    }

    public static void main(String[] args) {
        System.out.println(ComposicaoEInicializacao.CONST);
        ComposicaoEInicializacao ci = new ComposicaoEInicializacao();
        System.out.println(ci);
        ci.printConst();
        System.out.print("\n-----------------------------------------
\n");
        ComposicaoEInicializacao ci2 = new ComposicaoEInicializacao();
        ci2.printConst();
    }
}
```

- A saída deste programa é como segue:

```
java.lang.Object@3e25a5
campo3 = 3
campo1 = Campo1 - apressadinho!!
campo5 = 0
campo6 = 0.0
campo7 = 0.0
campo2 = O correto? Nem sempre... Depende.
campo4 = Campo4, atrasadinho e preguiçoso. Mas sempre chega a tempo.
toString de ComposicaoEInicializacao
java.lang.Object@3e25a5

-----------------------------------------
```

```
campo3 = 3
campo1 = Campo1 - apressadinho!!
campo5 = 0
campo6 = 0.0
campo7 = 0.0
campo2 = O correto? Nem sempre... Depende.
java.lang.Object@3e25a5
```

# Herança

- Antes de falar sobre herança, vamos ver um exemplo
- Suponha que estamos construindo um software para gerenciar os funcionários de uma empresa de software
  - Pessoal do serviço administrativo, programadores e gerentes de projeto
  - O pessoal da limpeza e vigilância é terceirizado
  - Que abstrações devem ser elaboradas? Em outras palavras, quais classes teremos?

```java
package p2.exemplos.feio;

/**
 * Representa um funcionário qualquer, que é uma pessoa que tem uma
 * matrícula, um salário e um tempo de serviço.
 *
 * @author Raquel Lopes
 *
 */
public class FuncionarioAdministrativo {
   private String nome;
   private String cpf;
   private String matricula;
   private double salarioBase;
```

```
    private int tempoDeServico;
    private Funcao funcao;

    public enum Funcao {

        OFFICE_BOY(1), SECRETARIA(3), TELEFONISTA(3), DONO(10);

        private int valor;
        Funcao(int valor) {
            this.valor = valor;
        }
        public int getValor() {
            return valor;
        }

    }

    /**
     * Cria um funcionário.
     *
     * @param nome
     *             O nome do funcionário.
     * @param cpf
     *             O CPF do funcionário.
     * @param matricula
     *             A matrícula do funcionário.
     * @param tempoDeServico
     *             O tempo de serviço (em meses) do funcionário.
     */
    public FuncionarioAdministrativo(String nome, String cpf, String
matricula, int tempoDeServico, Funcao funcao) {
```

```
        this.nome = nome;
        this.cpf = cpf;
        this.matricula = matricula;
        this.tempoDeServico = tempoDeServico;
        this.funcao = funcao;
    }

    /**
     * Recupera o nome do funcionário.
     *
     * @return O nome do funcionário.
     */
    public String getNome() {
        return nome;
    }

    /**
     * Recupera o CPF do funcionário.
     *
     * @return O CPF associado ao funcionário.
     */
    public String getCPF() {
        return cpf;
    }

    /**
     * Recupera a função do funcionário.
     *
     * @return A função atual do funcionário.
     */
    public Funcao getFuncao() {
```

```
    return funcao;
}

/**
 * Atribui uma nova função ao funcionário.
 *
 * @param funcao
 *            Nova função do funcionário.
 */
public void setFuncao(Funcao funcao) {
    this.funcao = funcao;
}

/**
 * Atribui um novo nome ao funcionário.
 *
 * @param nome
 *            Novo nome do funcionário.
 */
public void setNome(String nome) {
    this.nome = nome;
}

/**
 * Atribui um novo CPF ao funcionário.
 *
 * @param cpf
 *            Novo CPF associado ao funcionário.
 */
public void setCPF(String cpf) {
    this.cpf = cpf;
```

```
}

/**
 * Recupera a matrícula do funcionário.
 *
 * @return A matrícula do funcionário.
 */
public String getMatricula() {
   return matricula;
}

/**
 * Atribui uma nova matrícula ao funcionário.
 *
 * @param matricula
 *            O valor da nova matrícula.
 */
public void setMatricula(String matricula) {
   this.matricula = matricula;
}

/**
 * Recupera o salário base do funcionário.
 *
 * @return O salário do funcionário.
 */
public double getSalarioBase() {
   return salarioBase;
}

/**
```

```
     * Atribui um novo salário base ao funcionário.
     *
     * @param salario
     *            O novo salário do funcionário.
     */
    public void setSalarioBase(double salario) {
        salarioBase = salario;
    }

    /**
     * Recupera o tempo de serviço em meses do funcionário.
     *
     * @return O tempo de serviço do funcionário.
     */
    public int getTempoDeServico() {
        return tempoDeServico;
    }

    /**
     * Atribui um novo tempo de serviço ao funcionário que deve ser maior
     * que o tempo de serviço anterior.
     *
     * @param tempoDeServico
     *            Novo valor para tempo de serviço.
     */
    public void setTempoDeServico(int tempoDeServico) {
        if (tempoDeServico > this.tempoDeServico)
            this.tempoDeServico = tempoDeServico;
    }

    /**
```

```
     * Este método computa o salário do funcionário.
     *
     * @return O salário do funcionário;
     */
    public double computaSalario() {
        return salarioBase + gratificacaoPorTempoDeServico();
    }

    private double gratificacaoPorTempoDeServico() {
        double gratificacaoBase = 1.24 * funcao.getValor();
        return getTempoDeServico() * gratificacaoBase;
    }

    /**
     * Representa um funcionário como String.
     *
     * @return A string que representa um funcionário.
     */
    public String toString() {
        // return super.toString() + ", matricula " + getMatricula();
        return "Nome " + getNome() + ", cpf " + getCPF() + ", matricula "
                + getMatricula();
    }

    /**
     * Testa a igualdade de um objeto com este funcionário. Dois objetos da
     * classe FuncionarioAdministrativo são iguais se eles possuem o mesmo
     * nome, mesmo cpf e têm a mesma matrícula.
     *
     * @param objeto
     *            O objeto a comparar com este funcionário.
```

```
     * @return true se o objeto for igual a este funcionario, false caso
     *         contrário.
     */
    public boolean equals(Object objeto) {
        if (!(objeto instanceof FuncionarioAdministrativo)) {
            return false;

        }
        FuncionarioAdministrativo func = (FuncionarioAdministrativo) objeto;

        return getNome().equals(func.getNome())
                && getCPF().equals(func.getCPF())
                && getMatricula().equals(func.getMatricula());
    }

}
```

```
package p2.exemplos.feio;

import java.util.ArrayList;
import java.util.List;

import p2.exemplos.Projeto;

public class Programador {
    private String nome;
    private String cpf;
    private String matricula;
    private double salarioBase;
    private int tempoDeServico;
    private List<String> linguagensEmQuePrograma;
```

```
private String linguagemDePreferencia;
private Projeto projetoAtual;

public Programador(String nome, String cpf, String matricula,
      int tempoDeServico, String preferencia) {
   this.nome = nome;
   this.cpf = cpf;
   this.matricula = matricula;
   this.tempoDeServico = tempoDeServico;
   linguagensEmQuePrograma = new ArrayList<String>();
   linguagemDePreferencia = preferencia;
}

/**
 * Recupera o nome do funcionário.
 *
 * @return O nome do funcionário.
 */
public String getNome() {
   return nome;
}

/**
 * Recupera o CPF do funcionário.
 *
 * @return O CPF associado ao funcionário.
 */
public String getCPF() {
   return cpf;
}
```

```java
    /**
     * Atribui um novo nome ao funcionário.
     *
     * @param nome
     *            Novo nome do funcionário.
     */
    public void setNome(String nome) {
        this.nome = nome;
    }

    /**
     * Atribui um novo CPF ao funcionário.
     *
     * @param cpf
     *            Novo CPF associado ao funcionário.
     */
    public void setCPF(String cpf) {
        this.cpf = cpf;
    }

    /**
     * Recupera a matrícula do funcionário.
     *
     * @return A matrícula do funcionário.
     */
    public String getMatricula() {
        return matricula;
    }

    /**
     * Atribui uma nova matrícula ao funcionário.
```

```
     *
     * @param matricula
     *            O valor da nova matrícula.
     */
    public void setMatricula(String matricula) {
        this.matricula = matricula;
    }

    /**
     * Recupera o salário base do funcionário.
     *
     * @return O salário do funcionário.
     */
    public double getSalarioBase() {
        return salarioBase;
    }

    /**
     * Atribui um novo salário base ao funcionário.
     *
     * @param salario
     *            O novo salário do funcionário.
     */
    public void setSalarioBase(double salario) {
        salarioBase = salario;
    }

    /**
     * Recupera o tempo de serviço em meses do funcionário.
     *
     * @return O tempo de serviço do funcionário.
```

```
     */
    public int getTempoDeServico() {
        return tempoDeServico;
    }

    /**
     * Atribui um novo tempo de serviço ao programador que deve ser maior
     * que o tempo de serviço anterior.
     *
     * @param tempoDeServico
     *            Novo valor para tempo de serviço.
     */
    public void setTempoDeServico(int tempoDeServico) {
        if (tempoDeServico > this.tempoDeServico)
            this.tempoDeServico = tempoDeServico;
    }

    /**
     * Adiciona uma nova linguagem de programação conhecida pelo
     * programador.
     * @param lp
     *            A nova linguagem de programação que o programador conhece.
     */
    public void adicionaLinguagemConhecida(String lp) {
        if (!linguagensEmQuePrograma.contains(lp))
            linguagensEmQuePrograma.add(lp);
    }

    /**
     * Indica a participação do programador em um projeto.
     *
```

```
     * @param projeto
     *            O projeto em que o programador está inserido.
     */
    public void atribuiProjeto(Projeto projeto) {
        projetoAtual = projeto;
    }

    /**
     * Este método computa o salário do programador.
     *
     * @return O salário do funcionário;
     */
    public double computaSalario() {
        return getSalarioBase() * 1.5;
    }

    /**
     * Recupera a linguagem de preferência do programador
     * @return A linguagem de preferência do programador
     */
    public String getLinguagemDePreferencia() {
        return linguagemDePreferencia;
    }

    /**
     * Atribui uma nova linguagem de preferência para o programador
     * @param linguagemDePreferencia
     *            Novoa linguagem de preferência do programador
     */
    public void setLinguagemDePreferencia(String linguagemDePreferencia) {
        this.linguagemDePreferencia = linguagemDePreferencia;
```

```
    }

    /**
     * Representa um programador como String.
     *
     * @return A string que representa um programdor.
     */
    public String toString() {
        return "Nome " + getNome() + ", cpf " + getCPF() + ", matricula "
                + getMatricula() + ", projeto " + projetoAtual.getTitulo();
    }

    /**
     * Testa a igualdade de um objeto com este programador. Dois objetos da
     * classe Programador são iguais se eles são possuem o mesmo nome,
mesmo
     * CPF e têm a mesma matrícula.
     *
     * @param objeto
     *             O objeto a comparar com este programador.
     * @return true se o objeto for igual a este programador, false caso
     *         contrário.
     */
    public boolean equals(Object objeto) {
        if (!(objeto instanceof Programador)) {
            return false;

        }
        Programador func = (Programador) objeto;

        return getNome().equals(func.getNome())
```

```
                && getCPF().equals(func.getCPF())
                && getMatricula().equals(func.getMatricula());
    }
}
```

```
package p2.exemplos.feio;

import java.util.ArrayList;
import java.util.List;

import p2.exemplos.Projeto;

public class Coordenador {
    private String nome;
    private String cpf;
    private String matricula;
    private double salarioBase;
    private int tempoDeServico;
    private List<Projeto> projetosQueGerencia;

    public Coordenador(String nome, String cpf, String matricula,
            int tempoDeServico) {
        this.nome = nome;
        this.cpf = cpf;
        this.matricula = matricula;
        this.tempoDeServico = tempoDeServico;
        projetosQueGerencia = new ArrayList<Projeto>();
    }

    /**
     * Recupera o nome do funcionário.
```

```
     *
     * @return O nome do funcionário.
     */
    public String getNome() {
        return nome;
    }

    /**
     * Recupera o CPF do funcionário.
     *
     * @return O CPF associado ao funcionário.
     */
    public String getCPF() {
        return cpf;
    }

    /**
     * Atribui um novo nome ao funcionário.
     *
     * @param nome
     *             Novo nome do funcionário.
     */
    public void setNome(String nome) {
        this.nome = nome;
    }

    /**
     * Atribui um novo CPF ao funcionário.
     *
     * @param cpf
     *             Novo CPF associado ao funcionário.
```

```
     */
    public void setCPF(String cpf) {
        this.cpf = cpf;
    }

    /**
     * Recupera a matrícula do funcionário.
     *
     * @return A matrícula do funcionário.
     */
    public String getMatricula() {
        return matricula;
    }

    /**
     * Atribui uma nova matrícula ao funcionário.
     *
     * @param matricula
     *            O valor da nova matrícula.
     */
    public void setMatricula(String matricula) {
        this.matricula = matricula;
    }

    /**
     * Recupera o salário base do funcionário.
     *
     * @return O salário do funcionário.
     */
    public double getSalarioBase() {
        return salarioBase;
```

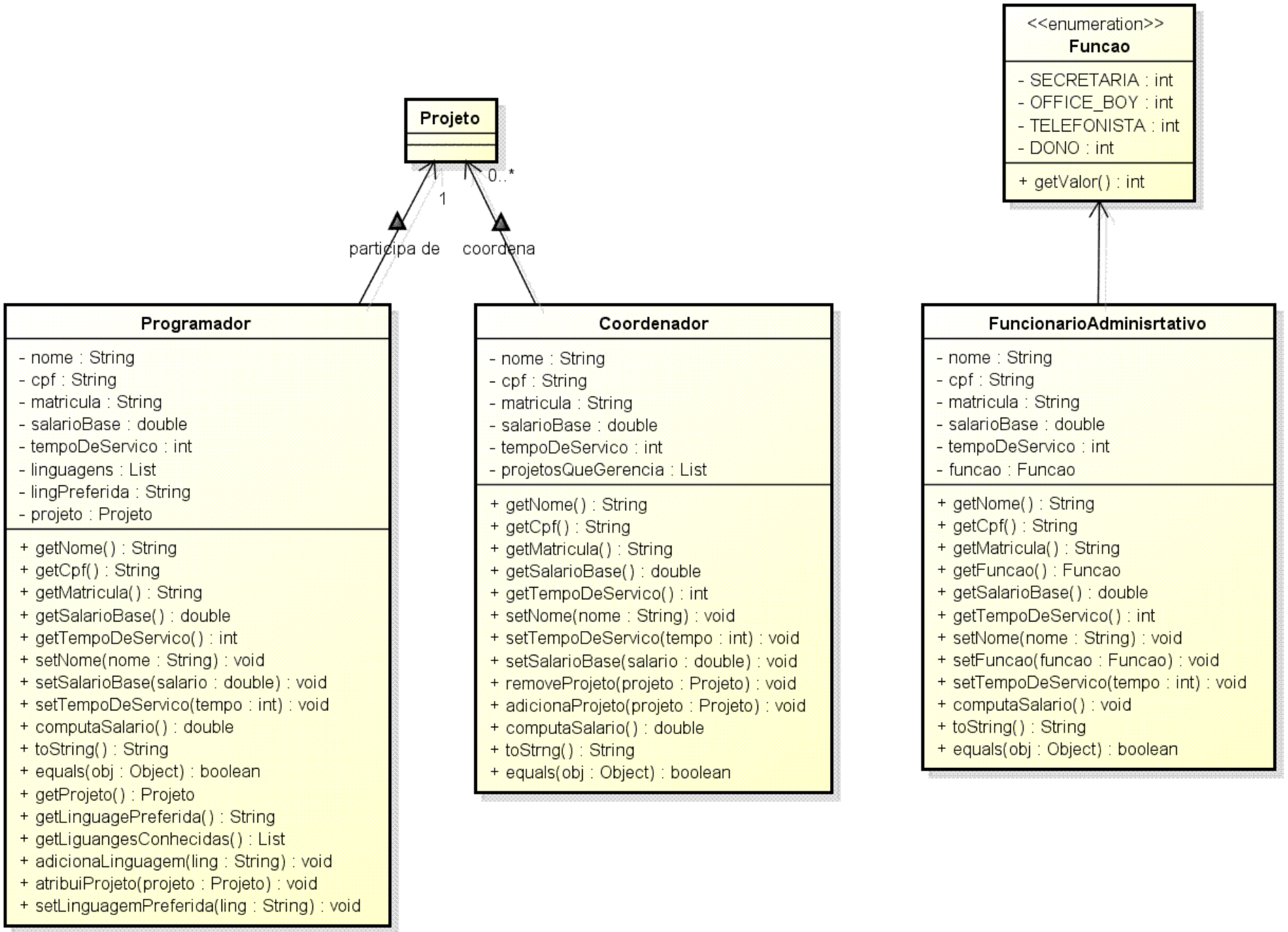
```
    }

    /**
     * Atribui um novo salário base ao funcionário.
     *
     * @param salario
     *            O novo salário do funcionário.
     */
    public void setSalarioBase(double salario) {
        salarioBase = salario;
    }

    /**
     * Recupera o tempo de serviço em meses do funcionário.
     *
     * @return O tempo de serviço do funcionário.
     */
    public int getTempoDeServico() {
        return tempoDeServico;
    }

    /**
     * Atribui um novo tempo de serviço ao funcionário que deve ser maior
     * que o tempo de serviço anterior.
     *
     * @param tempoDeServico
     *            Novo valor para tempo de serviço.
     */
    public void setTempoDeServico(int tempoDeServico) {
        if (tempoDeServico > this.tempoDeServico)
            this.tempoDeServico = tempoDeServico;
```

```
}

/**
 * Este método computa o salário do coordenador.
 *
 * @return O salário do coordenador;
 */
public double computaSalario() {
   return getSalarioBase() * 2.2 + adicionalPorTempoDeServico();
}

private double adicionalPorTempoDeServico() {
   return ((double) getTempoDeServico()) * 2.5;
}

/**
 * Faz este coordenador ser gerente de mais um projeto.
 *
 * @param proj
 *            O novo projeto que este coordenador irá gerenciar.
 * @return true caso o novo projeto não esteja ainda na lista de
 *         projetos e seja adicionado com sucesso; false, caso
 *         contrário.
 */
public boolean adicionaProjeto(Projeto proj) {
   if (projetosQueGerencia.contains(proj))
      return false;
   return projetosQueGerencia.add(proj);
}

/**
```

```
     * Faz este coordenador deixar de ser gerente de um projeto.
     *
     * @param proj
     *            O projeto que este coordenador não irá mais gerenciar.
     * @return true caso o projeto seja removido com sucesso; false, caso
     *         contrário.
     */
    public boolean removeProjeto(Projeto proj) {
        return projetosQueGerencia.remove(proj);
    }

    /**
     * Representa um coordenador como String.
     *
     * @return A string que representa este coordenador.
     */
    public String toString() {
        return "Nome " + getNome() + ", cpf " + getCPF() + ", matricula "
                + getMatricula() + ", projetos que gerencia "
                + projetosQueGerencia;
    }

    /**
     * Testa a igualdade de um objeto com este coordenador. Dois objetos da
     * classe Coordenador são iguais se eles possuem o mesmo nome, mesmo CPF
     * e têm a mesma matrícula.
     *
     * @param objeto
     *            O objeto a comparar com este coordenador.
     * @return true se o objeto for igual a este coordenador, false caso
     *         contrário.
```

```
     */
    public boolean equals(Object objeto) {
        if (!(objeto instanceof Coordenador)) {
            return false;

        }
        Coordenador func = (Coordenador) objeto;

        return getNome().equals(func.getNome())
                && getCPF().equals(func.getCPF())
                && getMatricula().equals(func.getMatricula());
    }

}
```

- E então, como está o cheiro desse programa?
- Vejamos o diagrama de classes (com algumas omissões):



- Nas classes de funcionários da empresa de software vistas (FuncionarioAdministrativo, Programador, Coordenador), há um mau cheiro terrível no código
  - Há muita repetição de código
  - Isso dificulta a manutenção de software pois uma mudança pode implicar alterações em várias partes do código, o que é "prato cheio" para introduzir bugs
  - Suponha que agora queremos também manter o estado civil e a quantidade de filhos de todos eles
- A atividade de limpar código que apresenta mau cheiro chama-se refatoramento
- Vamos refatorar as três classes de funcionários
- Mas antes, vamos aprender um pouco sobre herança…

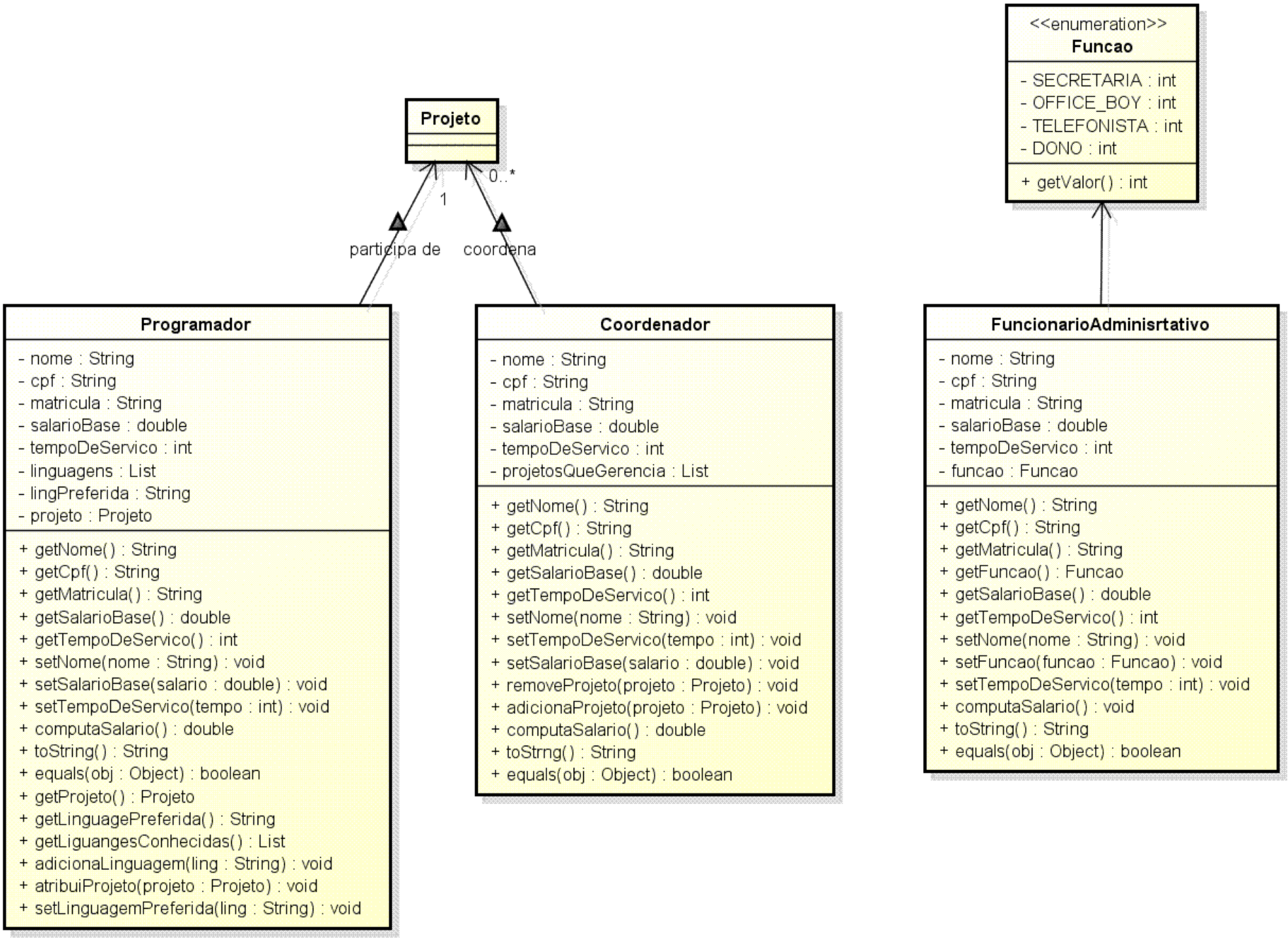
## Algumas palavras sobre herança

- Tipos diferentes de objetos podem ter algo comum entre si. Por exemplo:
  - Mensagem (genérica) e mensagem de áudio, mensagem de vídeo, mensagem multimídia, etc.
  - Avião (genérico) e avião supersônico, avião caça, avião monomotor, etc.
  - Pessoa (genérica) e programador, funcionário, dono de banco, etc.
- Classes podem herdar os campos (atributos) e métodos de outras classes
- A classe que deriva (ou herda) de outra classe é chamada de subclasse, ou classe filha ou classe derivada
- A classe da qual a subclasse é derivada é chamada de superclasse, ou classe base, ou classe mãe
- Relacionamento "**é um**"
- Qual é a idéia?
  - Quando queremos criar uma nova classe e já existe uma classe que já possui parte do

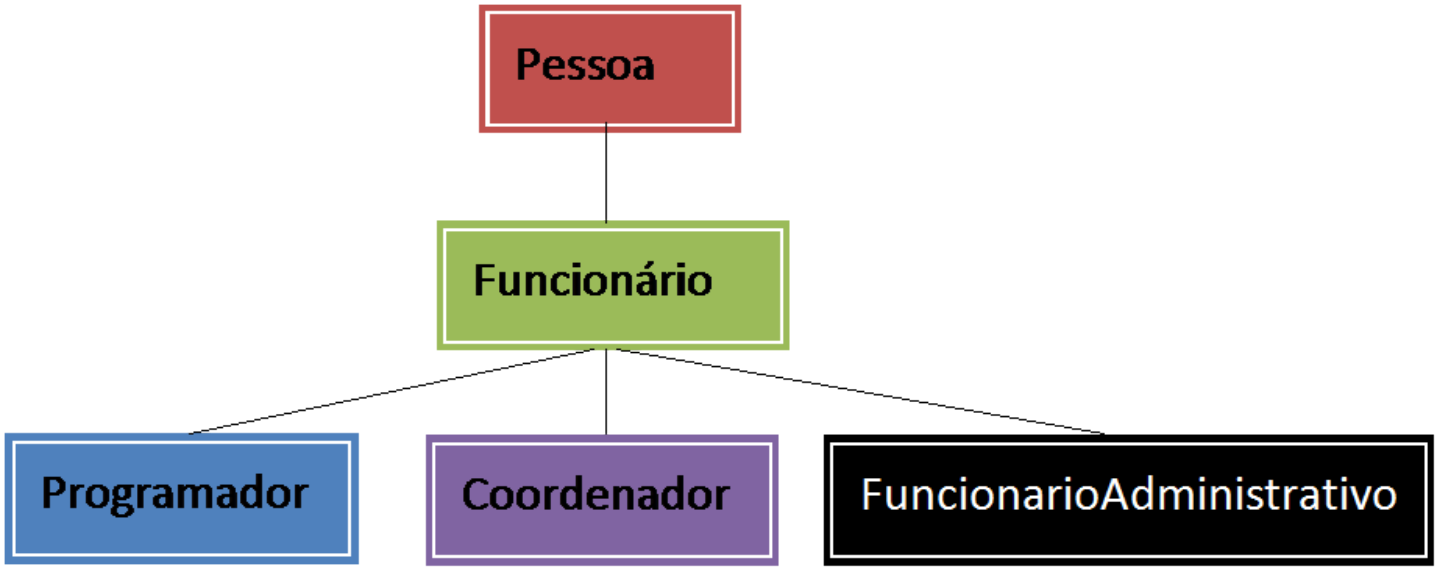
código que queremos, nós podemos derivar a nova classe a partir da classe já existente, o relacionamento "é um tipo especial de" for verdadeiro entre as classes
  - Ao fazer isso, todos os atributos e métodos da classe que já existia podem ser reusados na sua nova classe sem que você precise fazer *copy&paste* e melhor, considerando que a superclasse já foi testada adequadamente, você não precisa testar o código que está sendo reusado!
  - Construtores não são métodos nem atributos de classe, então não são herdados ☹. Mas existe um jeito simples de chamar o construtor da superclasse (veremos isso mais adiante!)
- A idéia é simples, intuitiva e poderosa. Faz parte de toda linguagem orientada a objetos
  - Assim, existe herança em Java também!
  - Em Java, toda classe que não tem uma superclasse definida explicitamente (como todas que criamos até aqui) é uma subclasse da classe Object (raiz da hierarquia)
  - Toda classe, exceto Object, tem uma e apenas uma superclasse. Isto é chamado de herança simples, em oposição à herança múltipla. A maioria das linguagens orientadas a objetos permite apenas herança simples
  - Em Java usamos herança sempre que criamos uma classe, pois todas as classes herdam implicitamente o comportamento da classe Object
    i) Classes podem ser derivadas de classes, que derivam de outras classes e assim sucessivamente, e no ponto mais alto da hierarquia está a classe Object
- A sintaxe para herança deve deixar claro que a nova classe é um tipo especial da classe "mãe"
  - Em Java, usamos a palavra reservada **`extends`** seguida do nome da classe base
  - Todos os campos e métodos são automaticamente herdados da classe base

## Voltando para nosso refatoramento: fatorando o que há de comum

- Tudo que vimos até aqui nos dá uma idéia de como refatorar nosso código?



- Primeiro juntamos tudo que tem de comum entre as três classes de funcionários e criamos uma nova classe que chamaremos Pessoa
- Em seguida criamos uma outra classe, Funcionario, que deve ter tudo que a classe Pessoa tem, e mais alguns atributos e métodos específicos do tipo especial de pessoa que é um funcionário
- Todos os funcionários da empresa de software são funcionários, mas cada funcionário de fato tem sua especialização
- Vamos implementar isso usando herança?
- A seguinte hierarquia representa o que acabamos de falar




- Funcionário herda tudo de Pessoa (um Funcionario é uma Pessoa) e acrescenta novos atributos e comportamento (métodos) a uma Pessoa, transformando-a em uma Pessoa que é um Funcionario
- Programador herda tudo de Funcionario (que havia herdado tudo de Pessoa), acrescentando atributos e métodos específicos de um Programador, transformando o Funcionario em um

Funcionario que é Programador

- O mesmo raciocínio segue para as demais classes (Coordenador e FuncionarioAdministrativo)
- Vamos implementar isso em Java?
- O resultado segue abaixo:

```
package p2.exemplos;


/**
 * Classe representando uma pessoa física.
 *
 * @author   Jacques Philippe Sauvé, jacques@dsc.ufpb.br
 * @version 1.1
 * <br>
 * Copyright (C) 1999 Universidade Federal da Paraíba.
 */

public class Pessoa  {

  private String nome;
  private String cpf;

  // Construtores
  /**
   * Constroi uma pessoa com nome e CPF dados.
   * @param nome O nome da pessoa.
   * @param cpf O CPF da pessoa.
   */
  public Pessoa(String nome, String cpf) {
    this.nome = nome;
```

```
    this.cpf = cpf;
  }

  /**
   * Constroi uma pessoa com nome dado e sem CPF.
   * @param nome O nome da pessoa.
   */
  public Pessoa(String nome) {
    this(nome, "");
  }

  /**
   * Recupera o nome da pessoa.
   * @return O nome da pessoa.
   */
  public String getNome() {
    return nome;
  }

  /**
   * Recupera o CPF da pessoa.
   * @return O CPF associado à pessoa.
   */
  public String getCPF() {
    return cpf;
  }

  /**
   * Ajusta o nome da pessoa.
   * @param nome O nome da pessoa.
   */
```

```java
public void setNome(String nome) {
  this.nome = nome;
}

/**
 * Ajusta o CPF da pessoa.
 * @param cpf O CPF associado à pessoa.
 */
public void setCPF(String cpf) {
  this.cpf = cpf;
}

/**
 * Representa a pessoa como string
 */
@Override
public String toString() {
  return "Nome " + nome + ", cpf " + cpf;
}

/**
 * Testa a igualdade de um objeto com esta pessoa.
 * @param objeto O objeto a comparar com esta pessoa.
 * @return true se o objeto for igual a esta pessoa, false caso contrário.
 */
@Override
public boolean equals(Object objeto) {
  if(! (objeto instanceof Pessoa)) {
    return false;
  }
  Pessoa outra = (Pessoa)objeto;
```

```
        return getNome().equals(outra.getNome())
                && getCPF().equals(outra.getCPF());
    }
}
```

- Note que não foi preciso declarar explicitamente que a classe Pessoa deveria ser uma subclasse da classe Object
- Os métodos toString e equals da classe Pessoa estão sobrescrevendo os métodos toString e equals da classe Object. Veja o código a seguir:

```
Pessoa pessoaRaquel1 = new Pessoa("Raquel V. Lopes", "1234567-88");
System.out.println(pessoaRaquel1.toString());
Pessoa pessoaRaquel2 = new Pessoa("Raquel Lopes", "2345678-99");
System.out.println(pessoaRaquel1.equals(pessoaRaquel2));
```

- Os métodos toString e equals chamados são os métodos da classe Pessoa, que sobrescrevem os métodos toString e equals da classe Object. A anotação `@Override` indica esta "sobrescrita".
- Caso estes métodos não tivessem sido definidos em Pessoa, o que aconteceria quando o método toString ou equals fossem invocados para um objeto Pessoa?
  - Os métodos da classe mãe seriam herdados
  - Não teria como comparar duas pessoas
  - O toString imprimiria o endereço de memória do objeto pessoa (que é o que o toString de Object faz)
- `@Override` é uma anotação que serve para informar ao compilador que este método sobrescreve um método da classe base
  - Ao fazer override de método tenha certeza de que a assinatura do método está idêntica à assinatura do método na classe base; caso contrário você estará apenas adicionando um

novo método a sua nova classe

- Herdamos de classes que têm comportamento bastante parecido com o comportamento da classe que estamos escrevendo, mas podemos modificar este comportamento quando necessário

  i) Fazendo override de métodos

  ii)Acrescentando novo comportamento (novos métodos)

- Veja a seguir a implementação da classe Funcionário

```
package p2.exemplos;

/**
 * Representa um funcionário qualquer, que é uma pessoa que tem uma
 * matrícula, um salário e um tempo de serviço.
 *
 * @author Raquel Lopes
 *
 */
public class Funcionario extends Pessoa {

   private String matricula;

   private double salarioBase;

   private int tempoDeServico;

   /**
    * Cria um funcionário.
    *
    * @param nome
    *            O nome do funcionário.
```

```
     * @param cpf
     *            O CPF do funcionário.
     * @param matricula
     *            A matrícula do funcionário.
     * @param tempoDeServico
     *            O tempo de serviço (em meses) do funcionário.
     */
    public Funcionario(String nome, String cpf, String matricula,
          int tempoDeServico, double salarioBase) {
       super(nome, cpf);
       this.matricula = matricula;
       this.tempoDeServico = tempoDeServico;
       this.salarioBase = salarioBase;
    }

    /**
     * Este método computa o salário do funcionário.
     *
     * @return O salário do funcionário.
     */
    public double computaSalario() {
       return salarioBase;
    }

    /**
     * Recupera a matrícula do funcionário.
     *
     * @return A matrícula do funcionário.
     */
    public String getMatricula() {
       return matricula;
```

```
}

/**
 * Atribui uma nova matrícula ao funcionário.
 *
 * @param matricula
 *            O valor da nova matrícula.
 */
public void setMatricula(String matricula) {
   this.matricula = matricula;
}

/**
 * Recupera o salário base do funcionário.
 *
 * @return O salário do funcionário.
 */
public double getSalarioBase() {
   return salarioBase;
}

/**
 * Atribui um novo salário base ao funcionário.
 *
 * @param salario
 *            O novo salário do funcionário.
 */
public void setSalarioBase(double salario) {
   salarioBase = salario;
}
```

```java
/**
 * Recupera o tempo de serviço em meses do funcionário.
 *
 * @return O tempo de serviço do funcionário.
 */
public int getTempoDeServico() {
   return tempoDeServico;
}

/**
 * Atribui um novo tempo de serviço ao funcionário que deve ser maior que
 * o tempo de serviço anterior.
 *
 * @param tempoDeServico
 *            Novo valor para tempo de serviço.
 */
public void setTempoDeServico(int tempoDeServico) {
   if (tempoDeServico > this.tempoDeServico)
      this.tempoDeServico = tempoDeServico;
}

/**
 * Representa um funcionário como String.
 *
 * @return A string que representa um funcionário.
 */
@Override
public String toString() {
   return super.toString() + ", matricula " + getMatricula();
}
```

```
    /**
     * Testa a igualdade de um objeto com este funcionário. Dois objetos da
     * classe Funcionario são iguais se eles são a mesma pessoa e têm a mesma
     * matrícula.
     *
     * @param objeto
     *            O objeto a comparar com este funcionario.
     * @return true se o objeto for igual a este funcionario, false caso
     *         contrário.
     */
    @Override
    public boolean equals(Object objeto) {
        if (!(objeto instanceof Funcionario)) {
            return false;

        }
        Funcionario func = (Funcionario) objeto;
        Pessoa pessoa = (Pessoa) objeto;
        return super.equals(pessoa)
                && getMatricula().equals(func.getMatricula());
    }
}
```

- Note que aqui tivemos que dizer explicitamente que a classe Funcionario deriva de uma outra classe
  - Isto vai acontecer sempre que desejarmos que a classe mãe seja diferente de Object
- O que está acontecendo?
  - É como se houvesse agora uma Pessoa dentro de Funcionario e como se Funcionario implementasse todos os métodos de Pessoa, delegando a responsabilidade ao método original do objeto Pessoa que está dentro de Funcionario (chamamos a isso de *wrapper*)

- Mas isso tudo é feito automaticamente, com o simples uso de **extends**

- Para que o objeto interno (neste caso, Pessoa) funcione bem, o que é necessário?
  - Em outras palavras: para que possamos usar um atributo que é um objeto, o que temos que fazer antes de usá-lo?
- A palavra reservada super é usada para se referir à superclasse da qual esta classe herda
  - No construtor, usamos super para chamar o construtor da superclasse
    i) Isto é necessário para garantir que o seu objeto da classe base será corretamente instanciado aqui
  - Dentro dos métodos toString e equals (que sobrescrevem métodos da classe Pessoa) fizemos chamadas aos métodos toString e equals da classe Pessoa usando a palavra reservada super
- Todos os métodos e campos da classe Pessoa foram então "herdados" da classe base
  - Mas não podemos mexer nos campos diretamente pois eles são privados (private)
  - Temos que usar os métodos *accessor* e *mutator* (gets e sets) para acessar os atributos
  - É possível trabalhar diretamente com os campos herdados?
    i) Sim, desde que eles tenham visibilidade protected ou public
    ii)Veja o exemplo a seguir

```
//Na classe Pessoa:
...
protected String nome;

protected String cpf;


//Na classe Funcionario:
...
```

```java
/**
 * Representa um funcionário como String.
 *
 * @return A string que representa um funcionário.
 */
@Override
public String toString() {
    return "Nome " + nome +
           ", cpf " + cpf +
           ", matricula " + matricula;
}
```

- É possível usar os métodos diretamente, desde que os atributos da classe mãe tenham visibilidade protected
  - › Na classe Pessoa, os atributos nome e cpf teriam visibilidade protected
  - › Membros protected são acessíveis à classe e às subclasses
  - › Isso é bom?
    i) Causa um acoplamento enorme entre as classes, significando que modificações na classe base podem requerer com freqüência modificações na classe derivada
    ii)Você precisa de uma ótima justificativa para colocar os atributos como protected!
- Note que os métodos equals e toString foram "overrided" (sobrescritos) pela classe Funcionario
  - › Para fazer override a assinatura dos métodos da classe base e da classe derivada devem ser idênticas
  - › Overload é diferente de override
    i) Em overload temos métodos com mesmo nome, mas parâmetros diferentes…
    ii)Em override temos métodos com mesmo nome e mesma entrada
  - › Por isso usar @Override é bom… Porque o compilador avisa quando estivermos enganados,

fazendo overload, em vez de override

- Ainda temos um probleminha… O cliente falou que para cada tipo de funcionário diferente existe uma forma diferente de computar o salário do funcionário
  - › O que a função computaSalario de Funcionario faz?
  - › É um método dummie… Não corresponde a uma realidade.
  - › Uma possibilidade seria retirar este método. Mas gostaríamos que toda classe derivada de Funcionario tivesse este comportamento de computar o salário. Como garantir a presença deste comportamento nas classes derivadas, sem precisar escrever algo dummie, que nem corresponde à realidade?
  - › A solução mais elegante é fazer a classe Funcionario ser uma classe abstrata

    i) Classes abstratas não podem ser instanciadas

    ii)Toda classe que possui, pelo menos, um método abstrato será uma classe abstrata. Porém, nem toda classe abstrata possui métodos abstratos

    iii) Métodos abstratos não têm implementação, mas apenas uma assinatura - *visibilidade tipoDeRetorno nomeMétodo(parâmetros);*

    iv) Qualquer classe que deriva de uma classe abstrata fica obrigada a implementar os métodos abstratos herdados ou a ser abstrata também

    v)Veja abaixo como definir Funcionario como uma classe abstrata

```
package p2.exemplos;

/**
 * Representa um funcionário qualquer, que é uma pessoa que tem uma
 * matrícula, um salario e um tempo de serviço.
 *
 * @author Raquel Lopes
 *
 */
```

```java
public abstract class Funcionario extends Pessoa {

   private String matricula;

   private double salarioBase;

   private int tempoDeServico;

   /**
    * Cria um funcionário.
    *
    * @param nome
    *             O nome do funcionário.
    * @param cpf
    *             O CPF do funcionário.
    * @param matricula
    *             A matrícula do funcionário.
    * @param tempoDeServico
    *             O tempo de serviço (em meses) do funcionário.
    */
   public Funcionario(String nome, String cpf, String matricula,
         int tempoDeServico, double salarioBase) {
      super(nome, cpf);
      this.matricula = matricula;
      this.tempoDeServico = tempoDeServico;
      this.salarioBase = salarioBase;
   }

   /**
    * Este método computa o salário do funcionário.
    *
```

```
     * @return O salário do funcionário.
     */
    public abstract double computaSalario();

    /**
     * Recupera a matrícula do funcionário.
     *
     * @return A matrícula do funcionário.
     */
    public String getMatricula() {
        return matricula;
    }

    /**
     * Atribui uma nova matrícula ao funcionário.
     *
     * @param matricula
     *            O valor da nova matrícula.
     */
    public void setMatricula(String matricula) {
        this.matricula = matricula;
    }

    /**
     * Recupera o salário base do funcionário.
     *
     * @return O salário do funcionário.
     */
    public double getSalarioBase() {
        return salarioBase;
    }
```

```
    /**
     * Atribui um novo salário base ao funcionário.
     *
     * @param salario
     *            O novo salário do funcionário.
     */
    public void setSalarioBase(double salario) {
        salarioBase = salario;
    }

    /**
     * Recupera o tempo de serviço em meses do funcionário.
     *
     * @return O tempo de serviço do funcionário.
     */
    public int getTempoDeServico() {
        return tempoDeServico;
    }

    /**
     * Atribui um novo tempo de serviço ao funcionário que deve ser maior que
o
     * tempo de serviço anterior.
     *
     * @param tempoDeServico
     *            Novo valor para tempo de serviço.
     */
    public void setTempoDeServico(int tempoDeServico) {
        if (tempoDeServico > this.tempoDeServico)
            this.tempoDeServico = tempoDeServico;
```

```
}

/**
 * Representa um funcionário como String.
 *
 * @return A string que representa um funcionário.
 */
@Override
public String toString() {
   return super.toString() + ", matricula " + getMatricula();
}

/**
 * Testa a igualdade de um objeto com este funcionário. Dois objetos da
 * classe Funcionario são iguais se eles possuem o mesmo nome, CPF
 * e têm a mesma matrícula.
 *
 * @param objeto
 *            O objeto a comparar com este funcionario.
 * @return true se o objeto for igual a este funcionario, false caso
 *         contrário.
 */
@Override
public boolean equals(Object objeto) {
   if (!(objeto instanceof Funcionario)) {
      return false;

   }
   Funcionario func = (Funcionario) objeto;
   Pessoa pessoa = (Pessoa) objeto;
   return super.equals(pessoa)
```

```
                && getMatricula().equals(func.getMatricula());
    }
}
```

- Note que o método computaSalário(..) não tem corpo
- Vejamos agora a implementação da classe Programador

```
package p2.exemplos;

import java.util.ArrayList;
import java.util.List;

public class Programador extends Funcionario {

   private List<String> linguagensEmQuePrograma;

   private String linguagemDePreferencia;

   private Projeto projetoAtual;

   public Programador(String nome, String cpf, String matricula,
         int tempoDeServico, String preferencia, int salarioBase) {
      super(nome, cpf, matricula, tempoDeServico, salarioBase);
      linguagensEmQuePrograma = new ArrayList<String>();
      linguagemDePreferencia = preferencia;
   }

   /**
    * Adiciona uma nova linguagem de programação conhecida pelo programador.
    *
    * @param lp
    *             A nova linguagem de programação que o programador conhece.
```

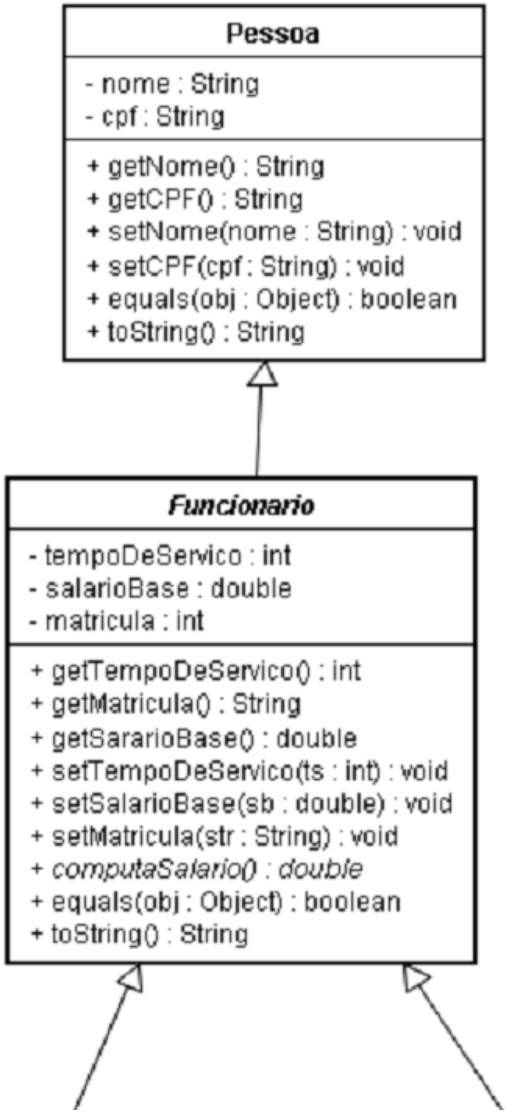
```
     */
    public void adicionaLinguagemConhecida(String lp) {
        if (!linguagensEmQuePrograma.contains(lp))
            linguagensEmQuePrograma.add(lp);
    }

    /**
     * Indica a participação do programador em um projeto.
     *
     * @param projeto
     *            O projeto em que o programador está inserido.
     */
    public void atribuiProjeto(Projeto projeto) {
        projetoAtual = projeto;
    }

    /**
     * Este método computa o salário do programador.
     *
     * @return O salário do funcionário;
     */
    @Override
    public double computaSalario() {
        return getSalarioBase() * 1.5;
    }

    /**
     * @return the linguagemDePreferencia
     */
    public String getLinguagemDePreferencia() {
        return linguagemDePreferencia;
```

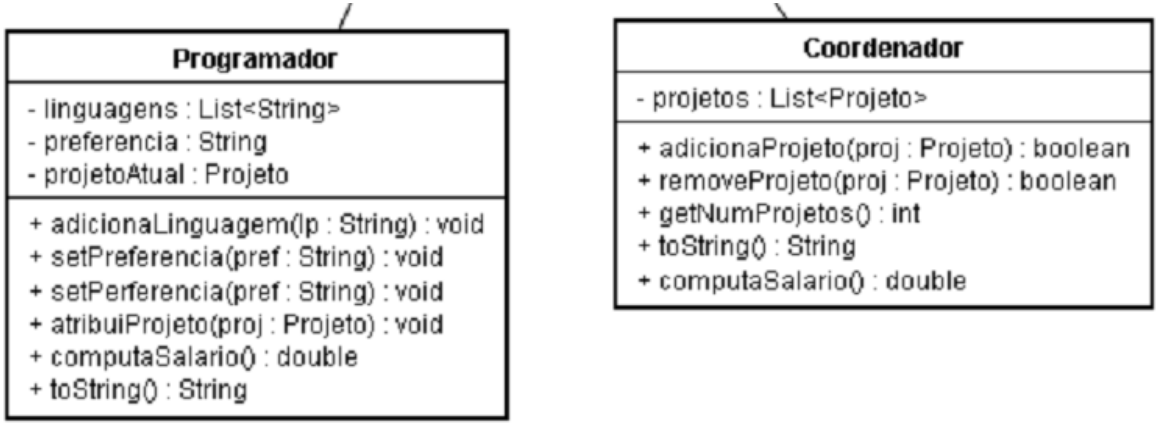
```
    }

    /**
     * @param linguagemDePreferencia
     *            the linguagemDePreferencia to set
     */
    public void setLinguagemDePreferencia(String linguagemDePreferencia) {
        this.linguagemDePreferencia = linguagemDePreferencia;
    }

    /**
     * Retorna a representação deste objeto em String.
     *
     * @return A string que representa este objeto.
     */
    @Override
    public String toString() {
        return "Nome " + getNome() + ", cpf " + getCPF() + ", matricula "
                + getMatricula() + ", projeto " + projetoAtual.getTitulo();
    }
}
```

- Seguindo o mesmo raciocínio podemos escrever a classe Coordenador
  - Fica como exercício para casa
- E então? Parecem mais "cheirosas"?
- Note que adicionamos comportamento novo às classes filhas: **`setLinguagemDePreferencia`**, **`getLinguagemDePreferencia`**, **`atribuiProjeto`** e , **`adicionaLinguagemConhecida`**.
- Note que mudamos alguns comportamentos da classe mãe que não nos convinha: **`toString`** e **`computaSalario`**.

# Como representar herança em UML?

- A relação de herança que existe entre classes pode ser representada como mostramos abaixo usando UML
- Observe que uma classe abstrata tem seu nome em itálico
- Fica óbvio nesta figura que criamos uma hierarquia de classes






# Escolhendo entre herança e composição

**(volte a ler esta seção inteira depois que você entender interfaces e polimorfismo!)**

- Composição e herança são dois mecanismos para reutilizar funcionalidade
- Alguns anos atrás (e na cabeça de alguns programadores ainda!), a herança era considerada a ferramenta básica de extensão e reutilização de funcionalidade
  - A composição estende uma classe pela delegação de trabalho para outro objeto
  - A herança estende atributos e métodos de uma classe
- Hoje, considera-se que a composição é muito superior à herança na maioria dos casos
  - A herança deve ser utilizada em alguns contextos
- Quando usar herança?
  - Quando o efeito que você pretende é criar uma classe que segue a mesma interface (o mesmo contrato) da classe base, porém é uma especialização (um caso especial, com algumas diferenças) da classe base
  - Existe uma relação “é um” entre a classe derivada e a classe base
    i) Por exemplo, Programador é um Funcionario

- Quando usar composição?
  - Dentro de sua classe existe funcionalidade de outra classe, mas a interface que você segue é a interface própria que você criou para a sua nova classe
  - Tipicamente, não existe a relação “é um” que há entre a subclasse e a superclasse
    i) Por exemplo, Conta não é uma Pessoa, mas uma Pessoa é um titular de uma conta, então a Conta embute comportamento de Pessoa dentro dela no que diz respeito ao titular
    ii)É mais usar a relação “tem um”, então Conta tem uma Pessoa (titular)
- As grandes vantagens da composição são:
  - A herança gera um acoplamento muito forte. A composição pode reduzir acoplamento, principalmente se interfaces forem usadas
    i) Voltaremos a falar disso depois! (Cobrem!!!)
  - A composição pode ser mudada em tempo de execução
  - O que ocorre quando uma representação interna de um campo muda na classe base?
    i) Se este campo for visível às subclasses, pode quebrar as subclasses
  - O que acontece se ocorrer uma mudança na forma como um método da classe base é implementada?
    i) As subclasses que esperavam outro comportamento podem quebrar
    ii)Isso também poderia ocorrer com composição
  - O que acontece se a interface da classe base mudar?
    i) Se métodos forem removidos, então as classes filhas podem quebrar
    ii)Se métodos forem adicionados é preciso reavaliar a herança e decidir se deverá ou não haver sobrescrita de alguns métodos
    iii) Se a classe base for abstrata, novos métodos abstratos obrigarão escrita de novos métodos na classe filha
    iv) Se a interface da classe sendo reusada por composição mudar, isto também poderá quebrar o código cliente (se houver remoção de métodos)

v)Métodos marcados como deprecated
(1) Pra dar tempo para a mudança

- A grande vantagem da herança é que ela gera código mais simples de ler e entender
  - A composição usa muitos objetos pequenos e só sabemos quem vai ser composto com quem em tempo de execução
  i) Isso é mais difícil de entender

## Qual é o aspecto mais importante da herança?

- Não é o simples reuso!
  - Isso é bom... Mas existem conseqüências de se usar herança
- O aspecto mais importante é o relacionamento que há entre a nova classe e a classe mãe
  - A nova classe é um tipo da classe existente
- Criamos 5 classes formando uma hierarquia
  - Examinando a hierarquia, podemos dizer que uma subclasse é um tipo de classe mãe
  - Funcionario "é um tipo de" Pessoa
  - Programador "é um tipo de" Funcionario
  - Coordenador "é um tipo de" Funcionario
- A relação "É um tipo de" implica que um objeto da subclasse pode fazer tudo que um objeto da classe mãe faz
  - Portanto, onde um objeto da classe mãe aparece, podemos colocar um objeto de uma subclasse
  - Significa "habilidade de permitir a substituição"
- Exemplo: nas linhas seguintes ...

```
(1)  Funcionario f = new Funcionario(...);
(2)  f.setSalarioBase(novoSalario);
```

(3) f.getSalarioBase();

- ... f é um Funcionario
- Mas tudo funcionaria perfeitamente se fizéssemos:
  - Funcionario f = new Programador(...);
  - f. setSalarioBase(novoSalario);
  - f.getSalarioBase();

## Upcasting e downcasting

- Olhando a hierarquia de classes
  - Conversões que sobem na hierarquia de classes são chamadas de Upcasting

    *i)* Usar um objeto de um tipo mais baixo da hierarquia como se fosse de um tipo mais alto da hierarquia)

    ii)Sempre dá certo, pois a classe de baixo é um tipo da classe de cima (habilidade de permitir a substituição)

    iii) Exemplo: Funcionario f = new Programador(...);
  - Conversões que descem na hierarquia de classes são chamadas de Downcasting

    *i)* Usar um objeto de um tipo mais alto da hierarquia como se fosse de um tipo mais baixo da hierarquia

    ii)Requer o uso de cast

    iii) Nem sempre dá certo, só se o objeto for realmente da classe (tipo) para a qual fizemos o casting

    iv) Exemplo: fazemos downcasting no método equals, depois de termos certeza que o objeto recebido como parâmetro é do tipo que esperamos

```
public boolean equals(Object objeto) {
   if (!(objeto instanceof Funcionario)) {
```

```
            return false;

        }
        Funcionario func = (Funcionario) objeto;
        return super.equals(func)
                && getMatricula().equals(func.getMatricula());
    }
```

*Programa* – *HP da disciplina*