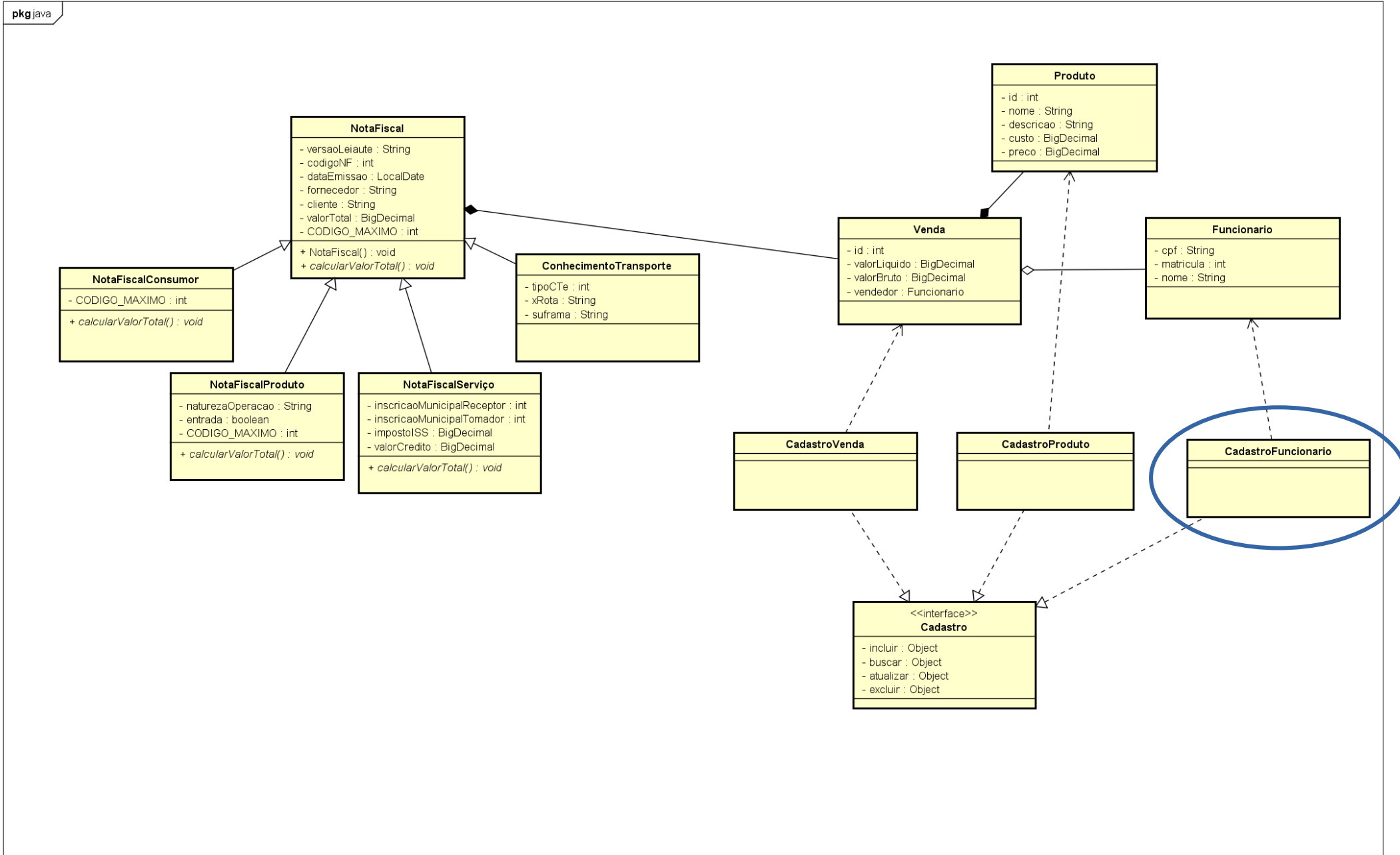


# Coleções em Java

## Capítulo XII

## Um sistema de vendas hipotético



## Um sistema de vendas hipotético

```
public void incluir(Funcionario func){
    int tamanho = funcionarios.length;
    tamanho++;
    Funcionario[] auxiliar = new Funcionario[tamanho];
    for (int i = 0; i < funcionarios.length; i++)
        auxiliar[i] = funcionarios[i];

    funcionarios = auxiliar;
    funcionarios[tamanho] = func;
}
```

```
public void atualizar(Funcionario func){
    String cpf = func.getCpf();
    for (int i = 0; i < funcionarios.length; i++) {
        if(funcionarios[i].getCpf().equals(cpf)){
            funcionarios[i] = func;
        }
    }
}
```

```
public Funcionario buscar(Funcionario func){
    String cpf = func.getCpf();
    for (Funcionario funcionario : funcionarios) {
        if(funcionario.getCpf().equals(cpf))
            return funcionario;
    }
    return null;
}
```

```
public void excluir(Funcionario func){
    String cpf = func.getCpf();
    int cont = 0;
    int tamanho = funcionarios.length;
    tamanho--;
    Funcionario[] auxiliar = new Funcionario[tamanho];
    for (int i = 0; i < funcionarios.length; i++)
        if(!funcionarios[i].getCpf().equals(cpf)){
            auxiliar[cont] = funcionarios[i];
            cont++;
        }
}
```

Esse cadastro segue uma boa prática?  
Como ficaria a manutenção dele? E a performance?

Um sistema de vendas hipotético

Já vimos que trabalhar com arrays pode não ser tão trivial. Uma hora ou outra podemos nos deparar com algumas exceções, como por exemplo a *NullPointerException* ou *ArrayIndexOutOfBoundsException*.

Então, vamos simplificar as coisas!

## Conhecendo o Collections Framework

- Visando nos auxiliar no trabalho com estruturas básicas de dados (Arrays) foi adicionado no pacote `java.util` um conjunto de classes e interfaces
  - Listas ligadas
  - Pilhas
  - Tabelas de espalhamento
  - Árvores
  - Entre outras

## Conhecendo o ArrayList

```
public class CadastroList {  
    private ArrayList funcionarios;  
  
    public void incluir(Funcionario func){  
        funcionarios.add(func);  
    }  
  
    public Funcionario buscar(Funcionario func){  
        int index = funcionarios.indexOf(func);  
        return funcionarios.get(index);  
    }  
  
    public void atualizar(Funcionario func){  
        int index = funcionarios.indexOf(func);  
        funcionarios.set(index, func);  
    }  
  
    public void excluir(Funcionario func){  
        funcionarios.remove(func);  
    }  
}
```

*Ficou bem mais simples.  
Mas por quê o erro?*

## Conhecendo o ArrayList

```
public Funcionario buscar(Funcionario f,
    int index = funcionarios.indexOf(f)) {
    return funcionarios.get(index);
}
```

incompatible types: Object cannot be converted to Funcionario

----  
(Alt-Enter shows hints)

Para ser o mais genérico possível a API definiu que os parâmetros e retornos de métodos serão sempre “Object”

Em Java, todas as classes herdam de Object mesmo que de forma implícita

Então, como resolver esse problema?

## Casting de referências

Precisamos avisar qual o tipo real daquele objeto, por esse motivos utilizamos do mesmo artifício que aplicamos nos tipos primitivos: o casting

```
public Funcionario buscar(Funcionario func){  
    int index = funcionarios.indexOf(func);  
    return (Funcionario) funcionarios.get(index);  
}
```



## Aprimorando com a interface List

Determinamos o “tipo” do nosso ArrayList através do *Generics*. Isso indica que nosso ArrayList só irá armazenar variáveis do tipo *Funcionario*.

```
public List<Funcionario> buscarPorDepartamento(String depto){  
    //consulta no BD  
    List<Funcionario> funcionarios = new ArrayList<>();  
    return funcionarios;  
}
```

O Generics nos ajuda a identificar possíveis erros de casting em tempo de compilação

## Conhecendo a classe Collections

A classe Collections possui métodos estáticos que nos ajudam a trabalhar com o Framework Collection

- Collections.sort → ordena em ordem crescente
- Collections.binarySearch → busca binária
- Collections.max → busca maior elemento
- Collections.min → busca menor elemento
- Collections.reverse → inverte a ordem dos elementos

## Conhecendo a classe Collections


```
public List<Funcionario> buscarPorDepartamento(String depto){  
    //consulta no BD  
    List<Funcionario> funcionarios = new ArrayList<>();  
    Collections.sort(funcionarios);  
    return funcionarios;  
}
```

Qual o motivo do erro?

## Conhecendo a classe Collections

A definição de como a comparação é feita não estava clara

```
public class Funcionario implements Comparable<Funcionario>{  
  
    private String cpf;  
    private int matricula;  
    private String nome;  
}
```



Com o “contrato” assinado, sabemos como todo funcionário tem o método compareTo implementado

```
@Override  
public int compareTo(Funcionario func) {  
    return Integer.compare(this.getMatricula(), func.getMatricula());  
}
```

## Conhecendo a classe Collections

Para a comparação de valores numéricos utilizamos os métodos das classes *Wrappers*. Estas classes de modo geral, podem ser entendidas como **classes de tipos primitivos**, portanto, elas possuem métodos que podem nos ajudar.

## Conhecendo a classe Collections

Iremos utilizar do método *compare*, dependendo do valor a ser comparado.

```
@Override  
public int compareTo(Funcionario func) {  
    return Integer.compare(this.getMatricula(), func.getMatricula());  
}
```



Objeto atual



Objeto a ser comparado

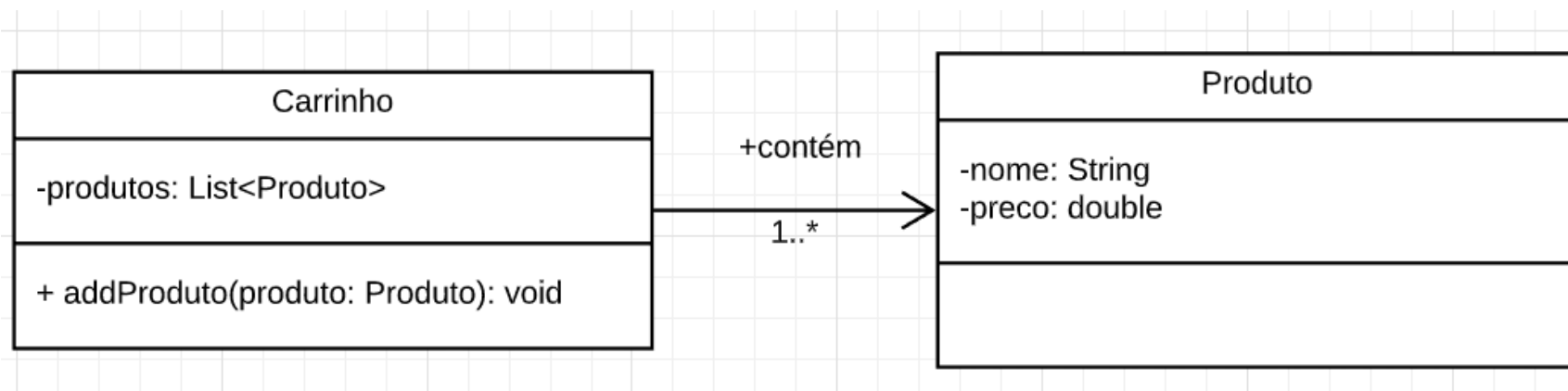
## Conhecendo a classe Collections

E para comparar Strings (Ordem alfabética) ficaria assim

```
@Override  
public int compareTo(Funcionario func) {  
    return this.getNome().compareTo(func.getNome());  
}
```

## Exercícios

1. Implemente um Sistema de *Carrinho de Compras*, onde serão adicionados vários produtos. Cada produto possui um Nome e Preço.



- I. Ordene o ArrayList em ordem crescente de preço;
- II. Ordene o ArrayList em ordem alfabética;
- III. **Desafio:** Ordene o ArrayList em ordem decrescente de preço.



**Obrigado!**