

12. Coleções Parte 1 : Listas e Conjuntos

Prof. Alexandre Krohn



Roteiro

- Introdução a Coleções
- Coleções Java
 - Listas (Lists)
 - ArrayLists
 - LinkedLists
 - Conjuntos (Sets)



Introdução a Coleções

- Uma **coleção** é um objeto que agrupa múltiplos elementos em uma única unidade.
- As coleções são usadas para armazenar, recuperar e manipular dados e transmitir dados de um método para outro.
- As coleções normalmente representam itens de dados que formam um grupo natural, como uma turma, um diretório, uma pasta de correio, uma lista telefônica ...



Coleções vs Vetores

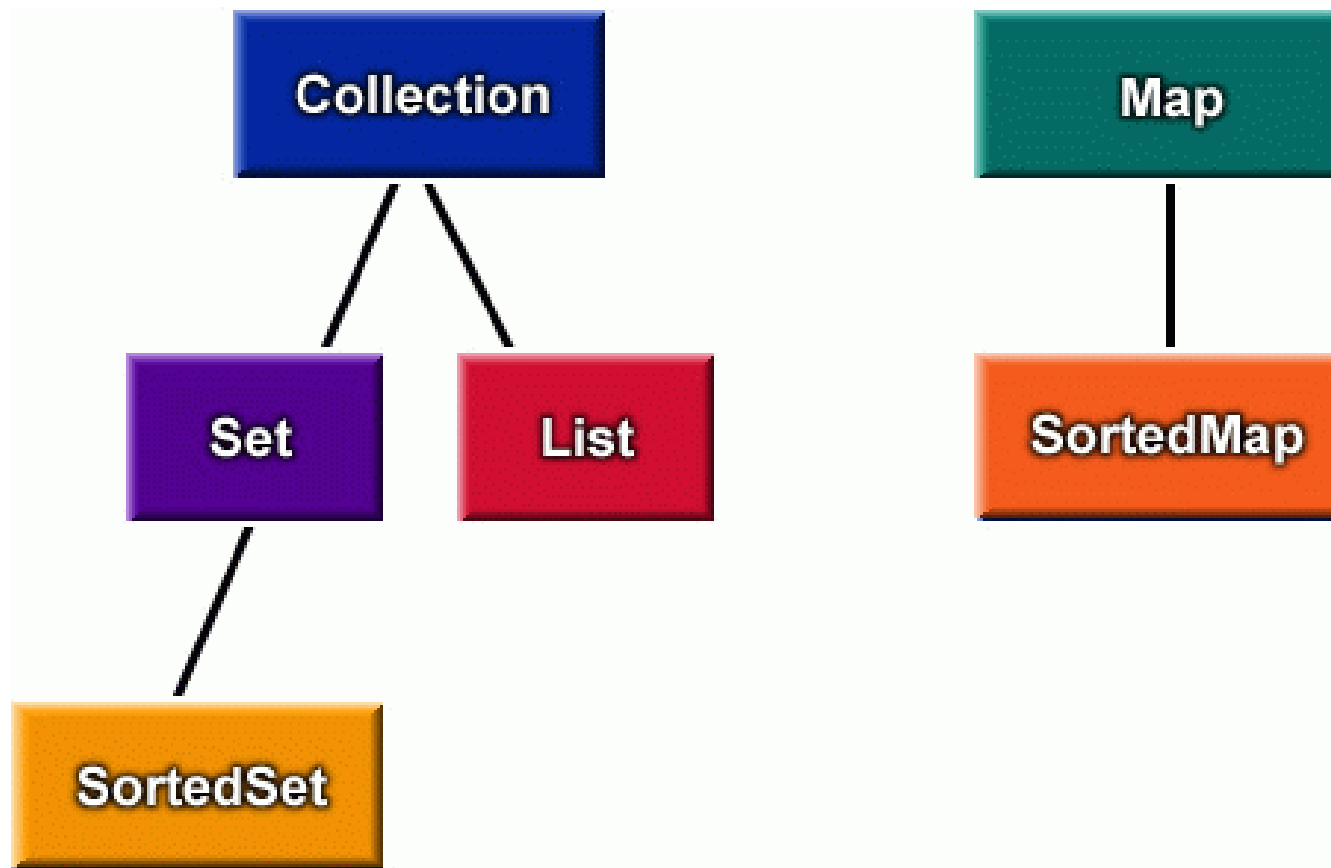
- Ao contrário de vetores, coleções não tem tamanhos pré-determinados
- E seu tamanho não é fixo, elas podem aumentar e diminuir em quantidade de elementos.



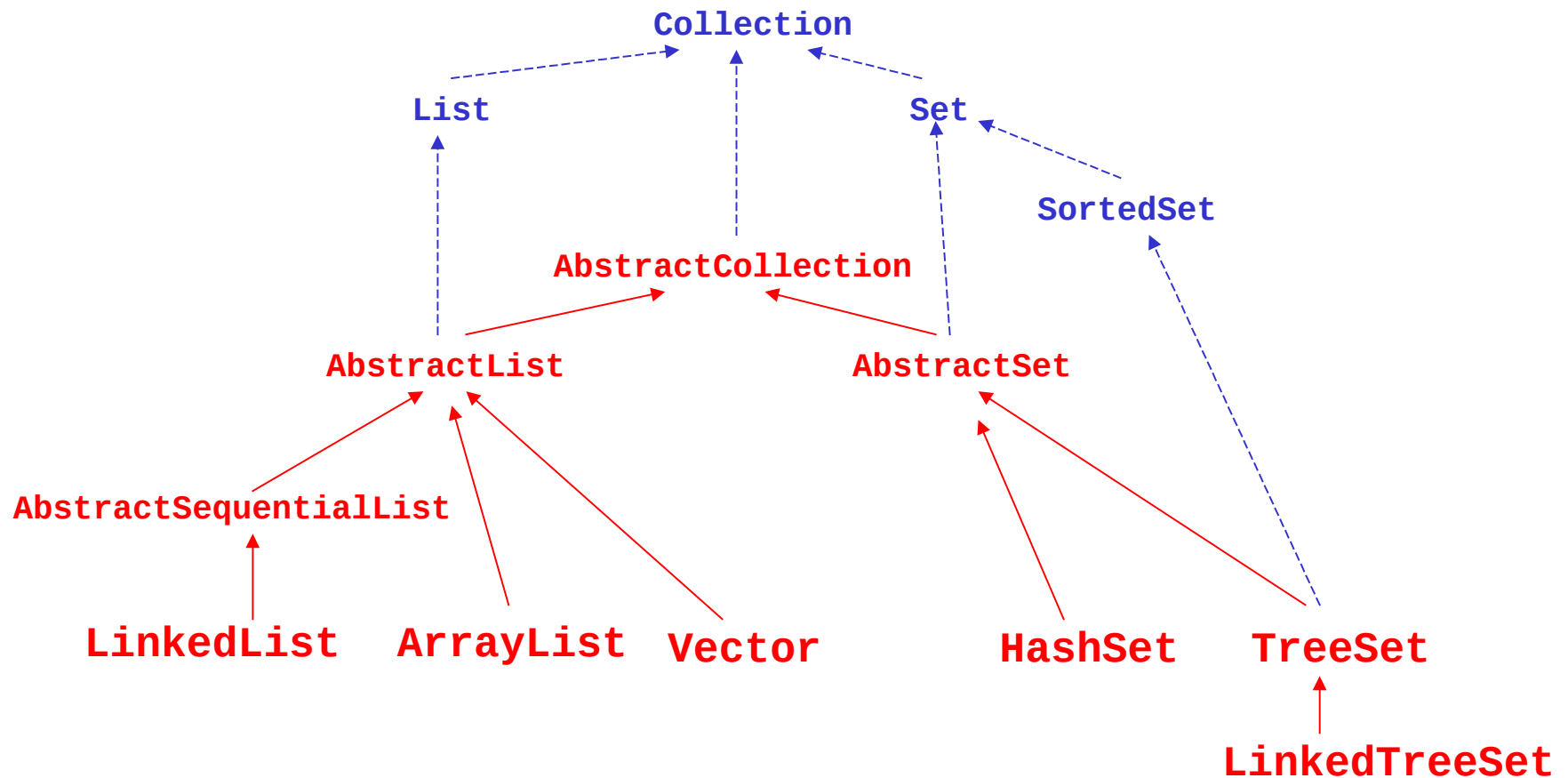
Coleções Java

- O framework de coleções Java é composto de um conjunto de interfaces e classes para trabalhar com grupos de objetos
- O ***Java Collections Framework*** fornece:
 - **Interfaces**: tipos de dados abstratos que representam coleções.
 - **Implementações**: implementações concretas das interfaces de coleções.
 - **Algoritmos**: métodos que executam cálculos úteis, como pesquisa e classificação, em objetos que implementam interfaces de coleções.

Interfaces Java

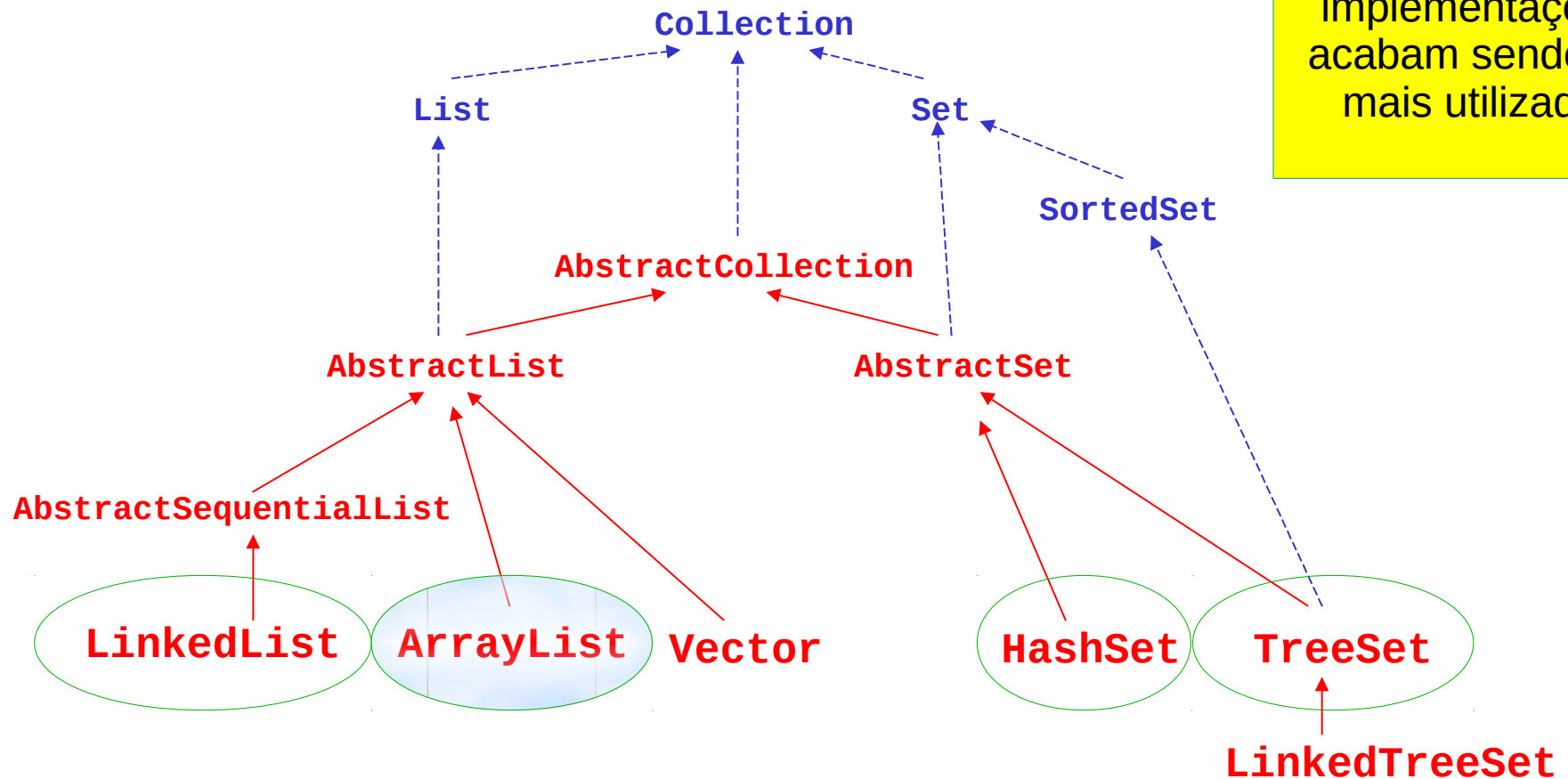


Interfaces e Classes



Interfaces e Classes

Quatro dessas implementações acabam sendo as mais utilizadas



Usando Coleções

- Todas as interfaces e classes do framework Collections estão dentro do pacote `java.util`

```
import java.util.Arrays;
import java.util.Collections;
import java.util.List;
import java.util.Set;
import java.util.TreeSet;

import ...
```

Atenção:

Cuidado para não importar ***java.awt.List*** ao invés de ***java.util.List*** ou nada funcionará...

Usando Coleções

```
String[] vetor = {"Phil", "Mary", "Betty", "Bob"};
List<String> nomes = Arrays.asList(vetor);
Collections.sort(nomes);
System.out.println("Lista Ordenada: " + nomes);
int pos = Collections.binarySearch(nomes, "Bob");
System.out.println("Bob está na posição " + pos);
Collections.shuffle(nomes);
System.out.println("Bagunçada: " + nomes);
```

Executando :

```
Lista Ordenada: [Betty, Bob, Mary, Phil]
Bob está na posição 1
Bagunçada: [Phil, Betty, Mary, Bob]
```



Coleções são interfaces

- Collection é na verdade uma interface
- Cada tipo de coleção tem uma ou mais implementações
- Essas implementações vão adicionando comportamentos diferentes às coleções instanciadas

Collection

- A interface **Collection** define os seguintes métodos:

int **size**(); // Obtém a quantidade de elementos

boolean **isEmpty**(); // Retorna true se a coleção estiver vazia

boolean **contains**(Object element); // Verifica se um elemento está na coleção

boolean **add**(E element); // Adiciona um elemento à coleção

boolean **remove**(Object element); // Retira um elemento da coleção

Iterator<E> **iterator**(); // Obtém um objeto de iteração para a coleção, através do qual pode-se percorrê-la



Listas

- A interface **List** herda de **Collection**, adicionando os seguintes métodos:

boolean **containsAll**(Collection<?> c); // Verifica se uma lista contém todos os elementos de outra

boolean **addAll**(Collection<? extends E> c); // Adiciona uma lista inteira dentro de outra

boolean **removeAll**(Collection<?> c); // Elimina uma lista de dentro de outra

boolean **retainAll**(Collection<?> c); // Elimina todos os elementos de uma lista que **não** estejam contidos na lista informada

void **clear**(); // Elimina todos os elementos da lista (esvazia)

Object[] **toArray**(); // Converte a lista em um vetor de Object

<T> T[] **toArray**(T a[]); // Converte a lista para vetor da classe contida na lista



Listas

- Os elementos em uma lista são ordenados:
 - Por sequência de inserção
 - Por magnitude, ordem alfabética, etc...
- Uma lista pode conter entradas duplicadas.
 - Por exemplo, uma lista de itens comprados pode incluir o mesmo item mais de uma vez.

Criando uma lista

- Cada implementação de coleções deve ter dois construtores :
 - Um sem argumentos para criar uma coleção vazia:

```
List<Integer> primos = new ArrayList<>();
```



Criando uma lista

- Cada implementação de coleções deve ter dois construtores :
 - Um construtor que recebe outra coleção como parâmetro:

```
List<Integer> numeros = new ArrayList<Integer>(primos);
```

Adicionando elementos

- O método `add(...)` é um dos principais métodos utilizados em listas.
- Como o nome diz, ele adiciona elementos às listas:

```
List<Integer> primos = new ArrayList<>();
```

```
primos.add(2);
```

```
primos.add(5);
```

```
primos.add(7);
```

```
primos.add(3);
```

ArrayList e LinkedList

- List é uma interface; **Não** se pode usar: **new List ()**
- Há duas implementações:
 - **LinkedList** : Inserções e exclusões mais rápidas!
 - **ArrayList** : Acesso aleatório mais rápido
- Não é bom expôr a implementação desnecessariamente, portanto:
 - **Bom**: List list = **new** LinkedList ();
 - **Ruim**: LinkedList list = **new** LinkedList ();



ArrayList

- Contém um **vetor** em sua implementação interna, que começa com um tamanho arbitrário
- Cada vez que se chega próximo a quantidade máxima de elementos, o vetor é trocado por um vetor 50% maior, e seus elementos realocados
- Quando os elementos são excluídos, e a lista diminui, o vetor também é trocado por vetores menores
- Isso é feito ao **adicionar e remover elementos**, por isso essa operação é mais **lenta!**



LinkedList

- Contém internamente uma **lista ligada** (Estrutura de dados)
- Incluir e excluir elementos é mais rápido
- Porém, **consultar elementos é feito de forma mais lenta**, pois ao invés de consultar elementos pelos índices, deve-se contar os elementos



ArrayList vs LinkedList

- Ao escolher uma das duas implementações, faça a pergunta:
 - Vou inserir dados com maior frequência ou a maioria das minhas operações será de leitura?
- Se a resposta for **inserir**, escolha ***LinkedList***
- Se for **acessar**, escolha ***ArrayList***

Comportamento externo

- Ambas possuem os mesmos métodos, e pode-se trocar a implementação se julgar-se adequado:

```
List<Pessoa> pessoas = new ArrayList<Pessoa>();  
  
pessoas = new LinkedList<Pessoa>();
```

Métodos

- **Listas : Acesso pelas posições dos elementos:**
 - E **get**(int index); // Obtém um elemento de uma determinada posição
 - E **set**(int index, E element); // Coloca um elemento em uma determinada posição (**substitui**)
 - void **add**(int index, E element); // Adiciona um elemento em uma determinada posição (**insere e empurra os demais para o final da lista**)
 - E **remove**(int index); // Remove um elemento de uma determinada posição
 - boolean **addAll**(int index, Collection<? extends E> c); // Insere uma lista em uma determinada posição de outra

Pro:

Todas estas operações são mais eficientes com a implementação ***ArrayList***

Métodos :

- **Listas : Métodos de pesquisa por elementos:**

int **indexOf**(Object o); // Retorna a posição onde um objeto encontra-se na lista

– int **lastIndexOf**(Object o); // Retorna a **última** posição onde um objeto encontra-se na lista

– Ambos retornam -1 se o objeto não está na lista

– Ambos precisam que o objeto seja de uma classe que implementa os métodos ***equals*** e ***hashCode***



`equals` e `hashCode()`

- Os métodos ***equals*** e ***hashCode*** precisam estar declarados na classe cujos objetos estarão colocados na coleção
- Sua finalidade é “ensinar” a coleção quando dois objetos são considerados iguais
- Sem eles os métodos que “procuram” por objetos nas coleções, como ***contains***, ***remove*** e ***indexOf*** não conseguem encontrar os elementos.



equals e hashCode()

- Define-se estes métodos escolhendo atributos que servem para indicar se dois objetos são iguais.
- Geralmente-se usa-se os atributos id, código, RG, CPF, etc... Qualquer um cujo conteúdo não deva se repetir.

equals e hashCode()

- Exemplo : Considerando-se uma classe Pessoa:

```
public class Pessoa {  
  
    private int codigo;  
    private String nome;  
  
    public Pessoa() {  
    }  
  
    public Pessoa(int codigo, String nome) {  
        this.codigo = codigo;  
        this.nome = nome;  
    }  
  
    public int getCodigo() {  
        return codigo;  
    }  
  
    public void setCodigo(int codigo) {  
        this.codigo = codigo;  
    }  
  
    public String getNome() {  
        return nome;  
    }  
  
    public void setNome(String nome) {  
        this.nome = nome;  
    }  
  
    @Override  
    public String toString() {  
        return "Pessoa [codigo=" + codigo + ", nome=" + nome + "];"  
    }  
}
```

equals e hashCode()

@Override

```
public boolean equals(Object obj) {  
    if (this == obj)  
        return true;  
    if (obj == null)  
        return false;  
    if (getClass() != obj.getClass())  
        return false;  
    Pessoa other = (Pessoa) obj;  
    if (codigo != other.codigo)  
        return false;  
    return true;  
}
```

@Override

```
public int hashCode() {  
    final int prime = 31;  
    int result = 1;  
    result = prime * result + codigo;  
    return result;  
}
```

Os dois métodos podem ser gerados pelo **Eclipse**, no menu **Source** → **Generate hashCode() and equals()...** Seguindo da seleção do atributo.

Generics

- Notem o uso de um tipo de dados entre “<” e “>” ao lado da classe da coleção
- É chamado de Generics, e serve para dizer que aquela coleção em questão só aceitará um tipo de dados:

```
List<Integer> primos = new ArrayList<>();
```



Essa lista só pode conter números inteiros

Generics

- Definir a coleção sem indicar seu tipo é considerado um erro, pois permite que sejam misturados objetos de classes diferentes dentro da mesma
- Quando fazemos isso, somos obrigados a fazer “**cast**” para podermos recuperar os dados:

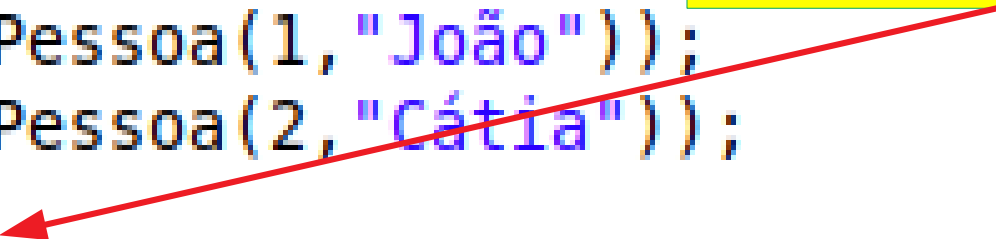
```
List minhaLista = new LinkedList();
```

```
minhaLista.add(new Pessoa(1, "João"));
```

```
minhaLista.add(new Pessoa(2, "Cátia"));
```

```
Pessoa x = (Pessoa) minhaLista.get(0);
```

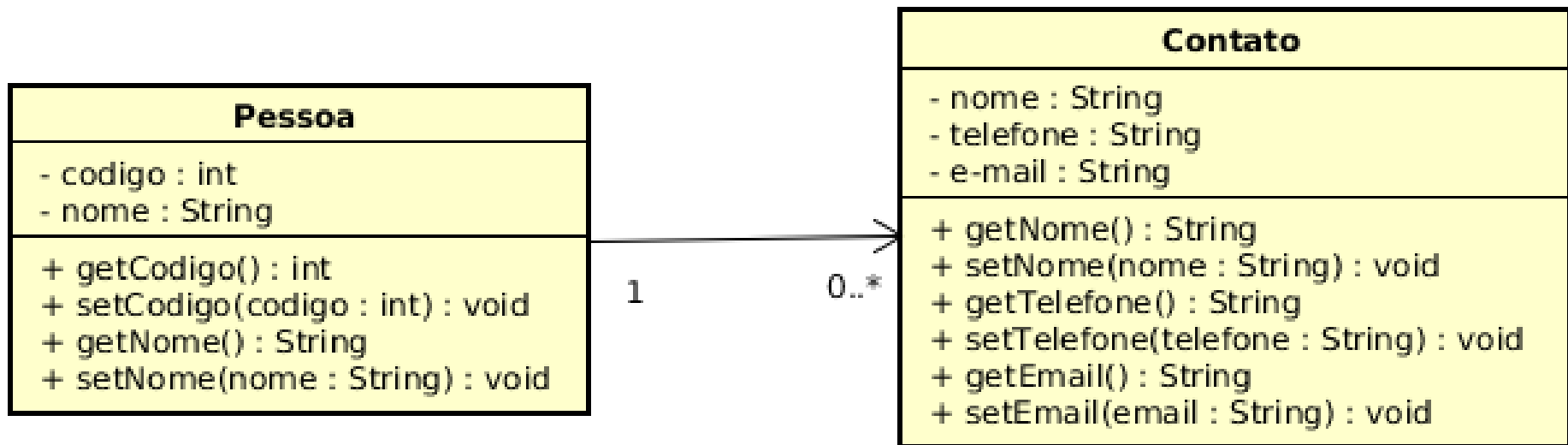
Cast para converter os objetos para o tipo Pessoa



Coleções : Relacionamentos

- Relacionamentos são melhor indicados usando coleções.
- Geralmente listas.

Coleções : Relacionamentos



Essa relação entre duas classes pode ser lida como :
Uma pessoa possui múltiplos contatos (quantidade indefinida)



Coleções : Relacionamentos

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

```
public class Pessoa {
```

```
    private int codigo;
    private String nome;
    private List<Contato> contatos;
```

```
    public Pessoa() {
        contatos = new ArrayList<Contato>();
    }
```

```
    public Pessoa(int codigo, String nome) {
        this();
        this.codigo = codigo;
        this.nome = nome;
    }
```

```
    public void addContato(Contato c) {
        this.contatos.add(c);
    }
```

```
    public void removeContato(Contato c) {
        this.contatos.remove(c);
    }
```

```
    public List<Contato> getContatos() {
        return this.contatos;
    }
```

Programação Ori

```
}
```



Coleções : Relacionamentos

Define-se um atributo
do tipo lista para
representar
o relacionamento



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

```
public class Pessoa {
```

```
    private int codigo;
    private String nome;
    private List<Contato> contatos;
```

```
    public Pessoa() {
        contatos = new ArrayList<Contato>();
    }
```

```
    public Pessoa(int codigo, String nome) {
        this();
        this.codigo = codigo;
        this.nome = nome;
    }
```

```
    public void addContato(Contato c) {
        this.contatos.add(c);
    }
```

```
    public void removeContato(Contato c) {
        this.contatos.remove(c);
    }
```

```
    public List<Contato> getContatos() {
        return this.contatos;
    }
```

Programação Ori

```
}
```



Coleções : Relacionamentos

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

```
public class Pessoa {
```

```
    private int codigo;  
    private String nome;  
    private List<Contato> contatos;
```

```
    public Pessoa() {  
        contatos = new ArrayList<Contato>();  
    }
```

```
    public Pessoa(int codigo, String nome) {  
        this();  
        this.codigo = codigo;  
        this.nome = nome;  
    }
```

```
    public void addContato(Contato c) {  
        this.contatos.add(c);  
    }
```

```
    public void removeContato(Contato c) {  
        this.contatos.remove(c);  
    }
```

```
    public List<Contato> getContatos() {  
        return this.contatos;  
    }
```

Inicializa-se a lista
vazia nos construtores

programação Ori

}



Coleções : Relacionamentos

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

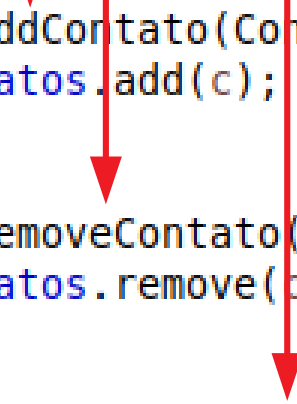
public class Pessoa {

    private int codigo;
    private String nome;
    private List<Contato> contatos;

    public Pessoa() {
        contatos = new ArrayList<Contato>();
    }

    public Pessoa(int codigo, String nome) {
        this();
        this.codigo = codigo;
        this.nome = nome;
    }
}
```

Adiciona-se
métodos para
manipular a lista



```
public void addContato(Contato c) {
    this.contatos.add(c);
}

public void removeContato(Contato c) {
    this.contatos.remove(c);
}

public List<Contato> getContatos() {
    return this.contatos;
}

}
```


Coleções : Relacionamentos

- Por fim, na utilização, é feito:

```
Pessoa p = new Pessoa(23, "Huguinho");
```

```
Contato c1 = new Contato("Zézinho");
```

```
p.addContato(c1);
```

```
// ou
```

```
p.addContato(new Contato("Luizinho"));
```

```
System.out.println("Pessoa : " + p.getNome());
```

```
System.out.println("Contatos :");
```

```
for (Contato c : p.getContatos()) {
```

```
    System.out.println("\t" + c.getNome());
```

```
}
```

Saída:

Pessoa : Huguinho

Contatos :

 Zézinho

 Luizinho



Conjuntos (Set)

- Conjuntos são coleções semelhantes às listas, com duas grandes diferenças:
 - Conjuntos não permitem a inclusão de elementos duplicados (Objetos contidos precisam ***equals*** e ***hashCode***)
 - Para conjuntos, a ordem dos elementos não é relevante



Conjuntos (Set)

- Sets implementam a noção matemática de conjuntos
- Dois conjuntos são iguais quando contém os mesmos elementos
- **Set não possui nenhum método a mais** do que os definidos na interface ***Collection***



Conjuntos (Set)

- As duas implementações mais comuns da interface **Set** são:
 - **HashSet** : Implementa a as operações da interface **Set**
 - **TreeSet** : Implementa as mesmas operações, mas acrescenta a noção de ordem dos elementos. (Que precisam implementar a interface **Comparable**)

HashSet (Exemplo)

```
Set<String> personagens = new HashSet<>();

personagens.add("Mickey");
personagens.add("Donald");
personagens.add("Pateta");
personagens.add("Donald"); // inserindo duplicado
personagens.add("Pluto");

for (String nome : personagens) {
    System.out.println(nome);
}
```

Resultado da execução:

Mickey
Pluto
Donald
Pateta

A ordem de inserção
não é mantida!

TreeSet (Exemplo)

```
Set<String> personagens = new TreeSet<>();

personagens.add("Mickey");
personagens.add("Donald");
personagens.add("Pateta");
personagens.add("Donald"); // inserindo duplicado
personagens.add("Pluto");

for (String nome : personagens) {
    System.out.println(nome);
}
```

Elementos são
Inseridos em ordem
No caso de Strings,
Ordem alfabética

Resultado da execução:

Donald
Mickey
Pateta
Pluto



Dúvidas?





Atividades

- Execute as atividades presentes no documento

12.Lista.de.Exercícios.POO.pdf

Próximos passos



- Mapas