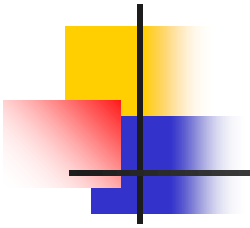
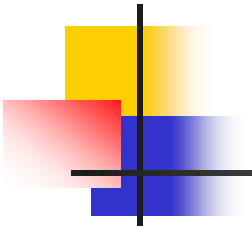
Programação Orientada à Objetos
Coleções

Profs. Rafael Marinho e Leandro Furtado
rafaelmarinho@unipam.edu.br
leandrofurtado@unipam.edu.br



Créditos e Agradecimentos

O material utilizado nessa aula foi gentilmente cedido pela Professora Rachel Carlos Duque Reis (UFPR) e, por esse motivo, o crédito é dela.



Coleções

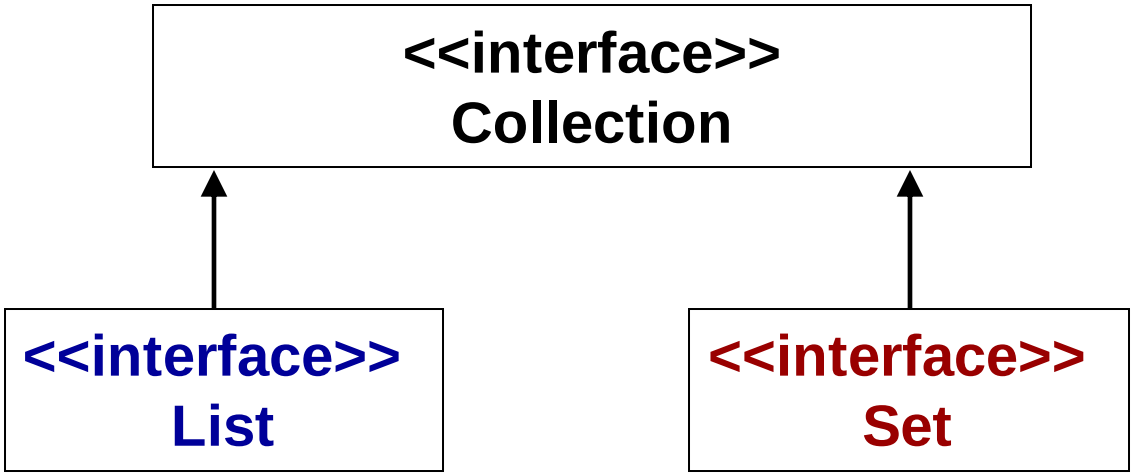
- Implementa uma estrutura de dados que armazena qualquer tipo de objeto.
- Não aceita tipos primitivos como elementos, apenas instâncias de objetos.
- Para guardar tipos primitivos devemos usar as classes *wrapper* (ex.: Integer, Double, Float).

Classes *Wrapper*

- Permite a utilização de tipos primitivos como objetos.

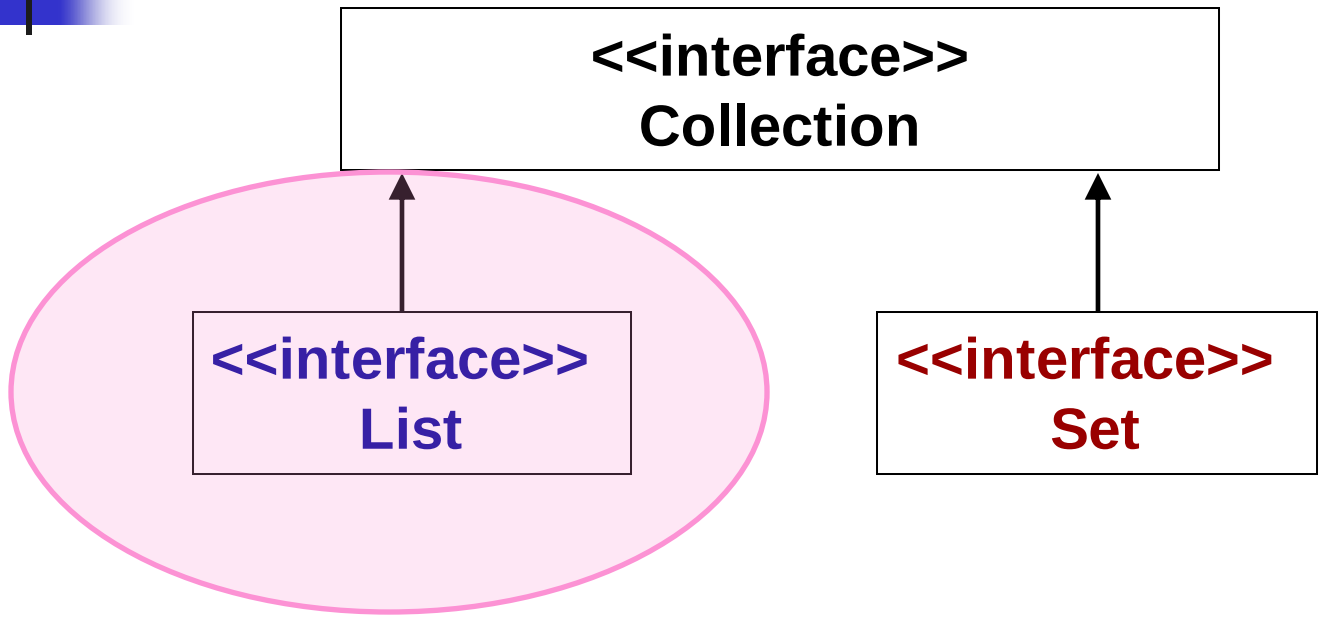
Classe *Wrapper*	**Tipo primitivo**
Boolean	boolean
Byte	byte
Character	char
Short	short
Integer	int
Long	long
Float	float
Double	double

Visão Geral



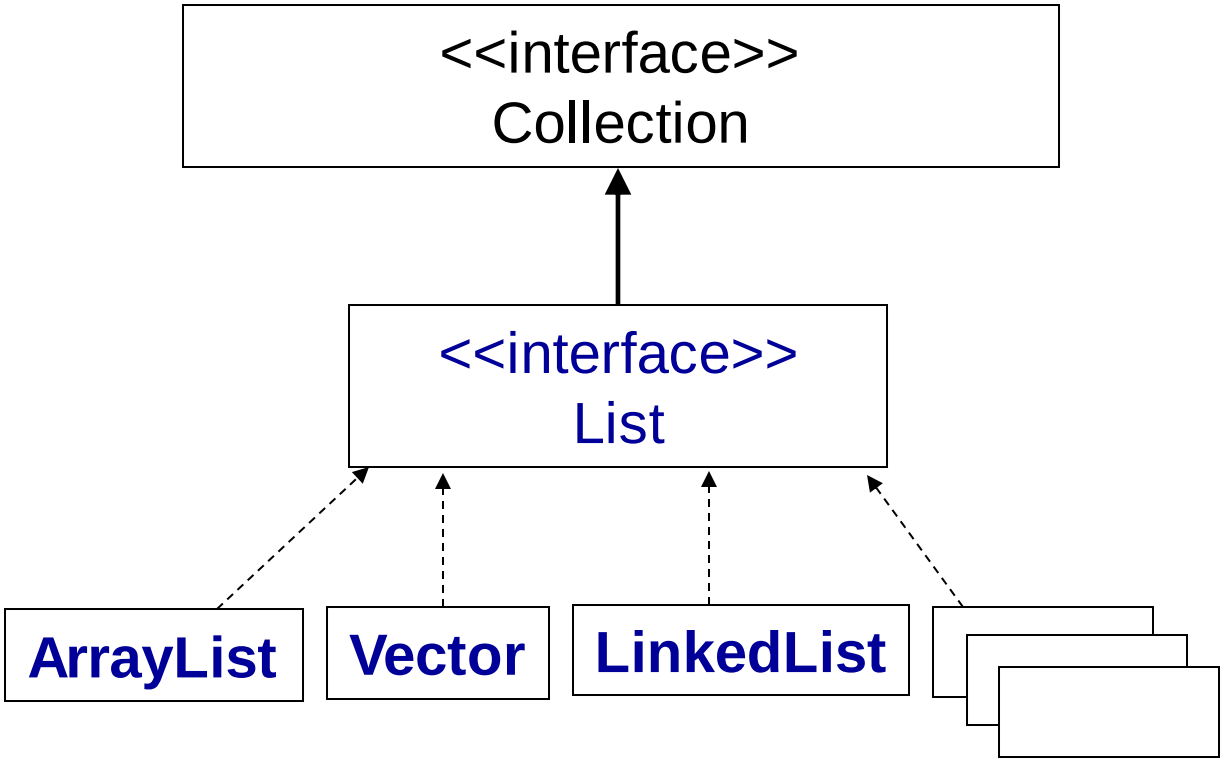
- **Collection**: interface-raiz na hierarquia de coleções a partir da qual as interfaces **List** e **Set** são derivadas.

Visão Geral



- Uma **List** é uma **Collection** que pode conter elementos duplicados.

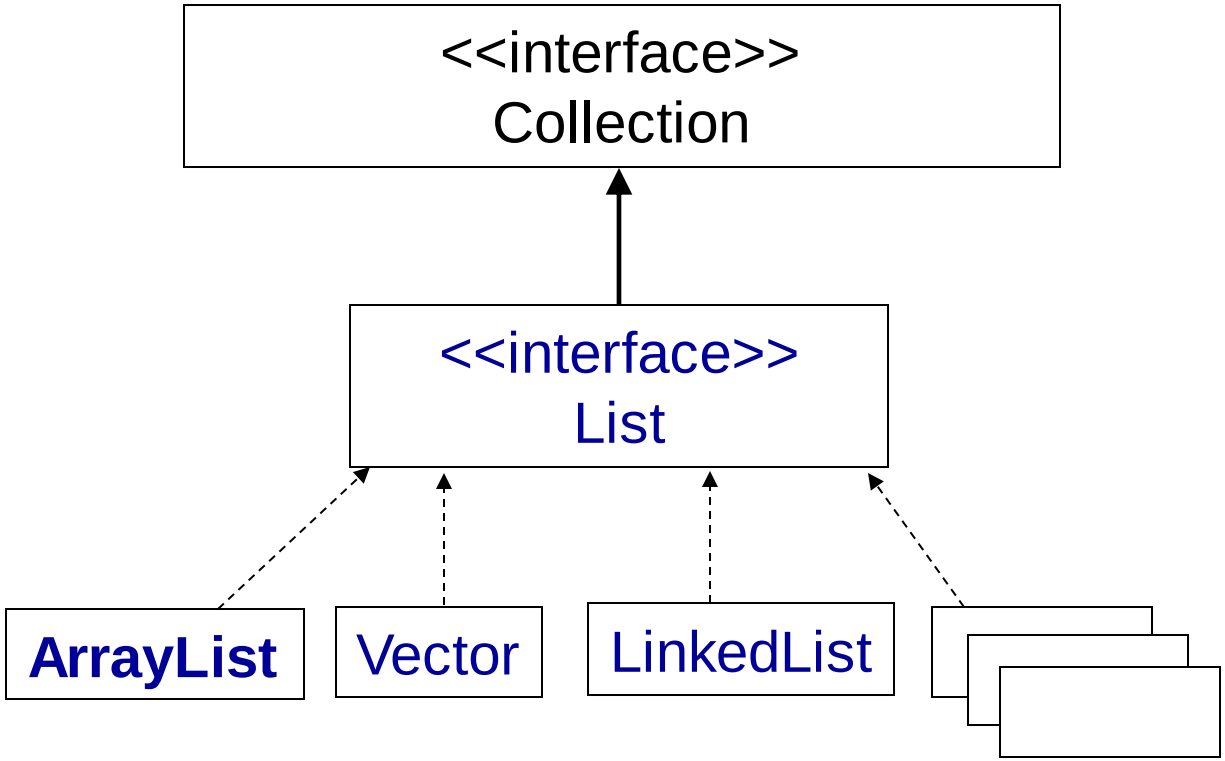
Visão Geral

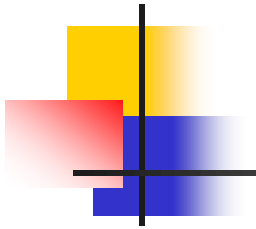


- A interface **List** é implementada por várias classes.

Exemplo: *ArrayList*, *Vector*, *LinkedList*...

Visão Geral


<<interface>>
Collection
<<interface>>
List
ArrayList
Vector
LinkedList



ArrayList

- Principais características:
  - Implementado como um array.
  - Acesso sequencial / aleatório extremamente rápido.
  - Inserção é também extremamente rápida.
  - Pode ser redimensionada dinamicamente, ou seja, aumenta em 50% o tamanho da lista.

ArrayList

A classe ArrayList **não é** uma **lista de arrays**, apesar do nome, é uma **lista de objetos**

ArrayList - operações

- **Criar um objeto ArrayList**
  - Instancie um objeto da classe ArrayList, usando o construtor sem parâmetro ou **com parâmetro**.

```
ArrayList<T> a = new ArrayList<T>();
ArrayList<T> b = new ArrayList<T>(20);
```

  - O valor 20 que foi informado significa a **capacidade inicial** da lista.
  - O **tamanho inicial** das listas **a** e **b** será **zero**, pois nenhum objeto foi adicionado.

ArrayList - operações

- **Criar um objeto ArrayList**
  - Instancie um objeto da classe ArrayList, usando o construtor com ou sem parâmetro

```
ArrayList<T> a = new ArrayList<T>();
ArrayList<T> b = new ArrayList<T>(20);
```

`<T>` representa o **tipo** dos objetos

  - Exemplo:

```
ArrayList<String> a = new ArrayList<String>();
ArrayList<String> b = new ArrayList<String>(20);
```

ArrayList - operações

- **Retornar o tamanho (número de elementos) da lista:**

```
int tamListaA = a.size();
```

- **Adicionar elementos no ArrayList**
  - O método **add()** pode ou não receber como parâmetro a posição na lista que desejamos que ele ocupe.

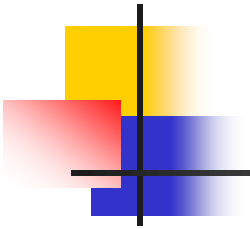
ArrayList - operações

- **Adicionar elementos no ArrayList**
  - Exemplo:

```
Pessoa p = new Pessoa("joao", "joao@email.com");
ArrayList<Pessoa> a = new ArrayList<Pessoa>();

a.add(p);
```

ou

```
a.add(0, p);
```



ArrayList - operações

- **Remover elementos do ArrayList**
  - Basta informar a **posição da lista** que desejamos remover

```
a.remove(int indice);
```

- **Ler os dados do ArrayList**
  - Para **ler dados** da lista podemos usar o método **get()**

```
Pessoa p1 = a.get(int indice);
```

ArrayList - operações

Cuidado !!!

Ao usar os métodos ***add(...)***, ***get()*** ou ***remove()***, o elemento da lista deve existir, caso contrário, será lançada a exceção:

`java.lang.IndexOutOfBoundsException`

ArrayList - operações

- **Percorrer uma lista**
  - O iterator() serve para percorrer e acessar os elementos de uma coleção.
  - Exemplo:

```
Iterator i = a.iterator();
while (i.hasNext())
{
   Pessoa pessoa = i.next();
   System.out.println(pessoa.getNome());
}
```

ArrayList - operações

Atenção!!!

Se uma coleção for modificada **depois** do iterador ser criado, o iterador se torna imediatamente **inválido**.

Qualquer operação realizada com o iterador **depois** desse ponto pode **lançar a exceção**

java.util.ConcurrentModificationException

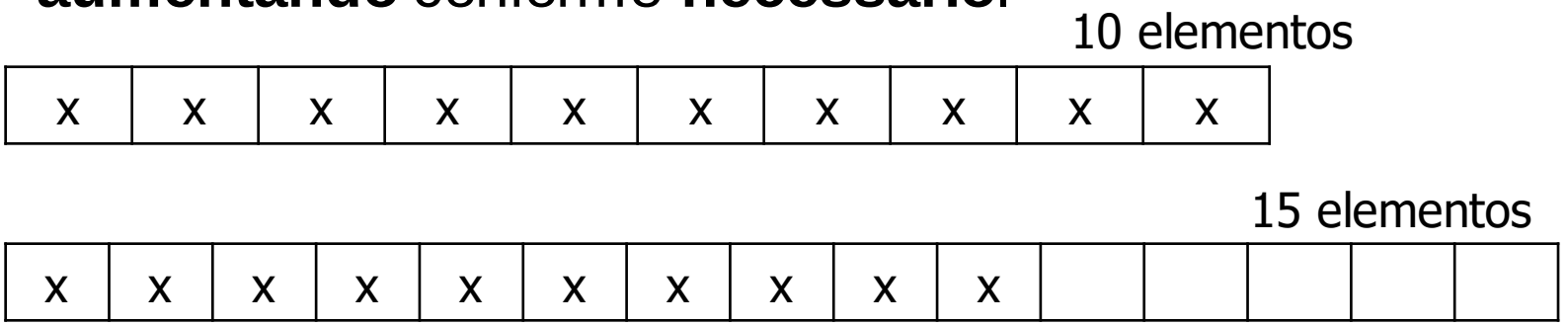
ArrayList - exemplo

```
import java.util.*;
public class Principal {
    public static void main(String[] args){

        ArrayList <String>listaEstados = new ArrayList<String>();

        listaEstados.add("São Paulo");
        listaEstados.add("Rio de Janeiro");
        listaEstados.add("Minas Gerais");
        listaEstados.add("São Paulo");

        Iterator i = listaEstados.iterator();
        while (i.hasNext()) {
            String estado = i.next().toString();
            System.out.println(estado);
        }

        System.out.println("*** " + listaEstados.get(2) + " ***");
    }
}
```

ArrayList - revisão

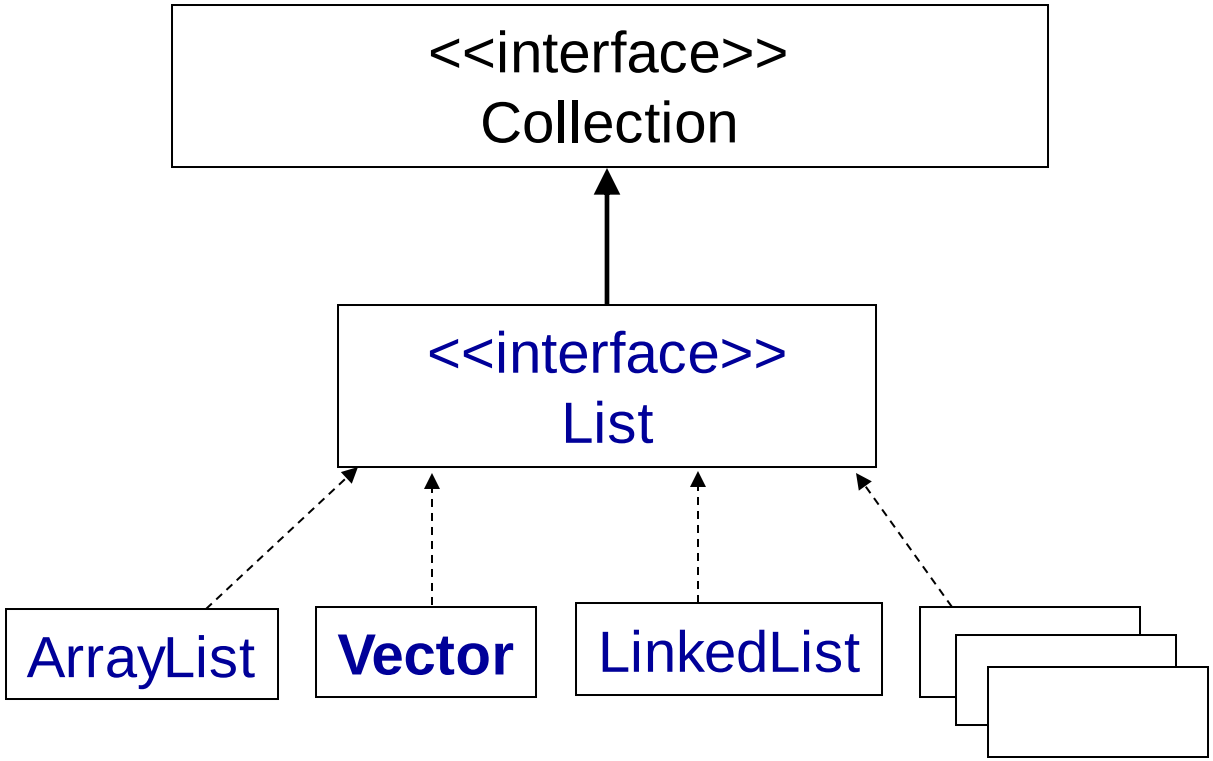
- ArrayList não remove elementos duplicados.

- Todo ArrayList começa com um **tamanho fixo**, que vai **aumentando** conforme **necessário**.



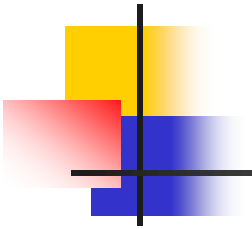
Qual o custo de aumentar a lista de 10 mil para um novo array de 15 mil elementos?

Visão Geral


<<interface>>
Collection
<<interface>>
List
ArrayList
Vector
LinkedList

Vector x ArrayList

```
import java.util.*;
public class Principal {
    public static void main(String[] args){
        System.out.print('\u000C');
        Vector<String> listaEstados = new Vector<String>();

        listaEstados.add("São Paulo");
        listaEstados.add("Rio de Janeiro");
        listaEstados.add("Minas Gerais");
        listaEstados.add("São Paulo");

        Iterator i = listaEstados.iterator();
        while (i.hasNext()) {
            String estado = i.next().toString();
            System.out.println(estado);
        }

        System.out.println("*** " + listaEstados.get(2) + " ***");
    }
}
```

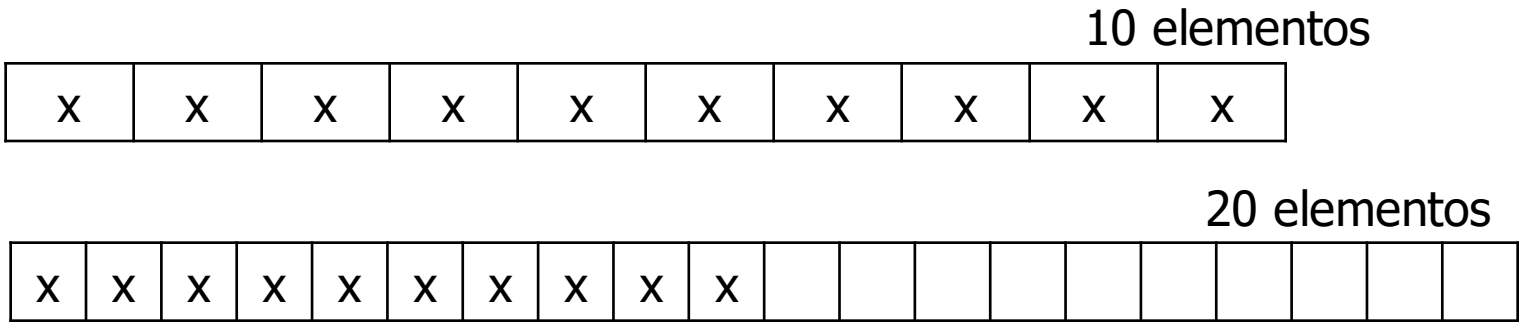


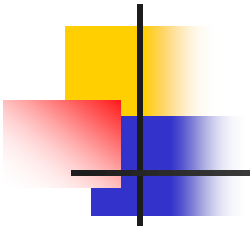
Vector x ArrayList

- Similaridades:
  - Implementado como um array
  - Acesso sequencial / aleatório
  - Possui as operações adicionar, remover, recuperar e percorrer

Vector x ArrayList

- Diferenças:
  - Vetor garante a sincronização dos dados, ou seja, possui suporte nativo a uso por várias threads simultâneas.
  - Alocação do array no Vector aumenta o dobro

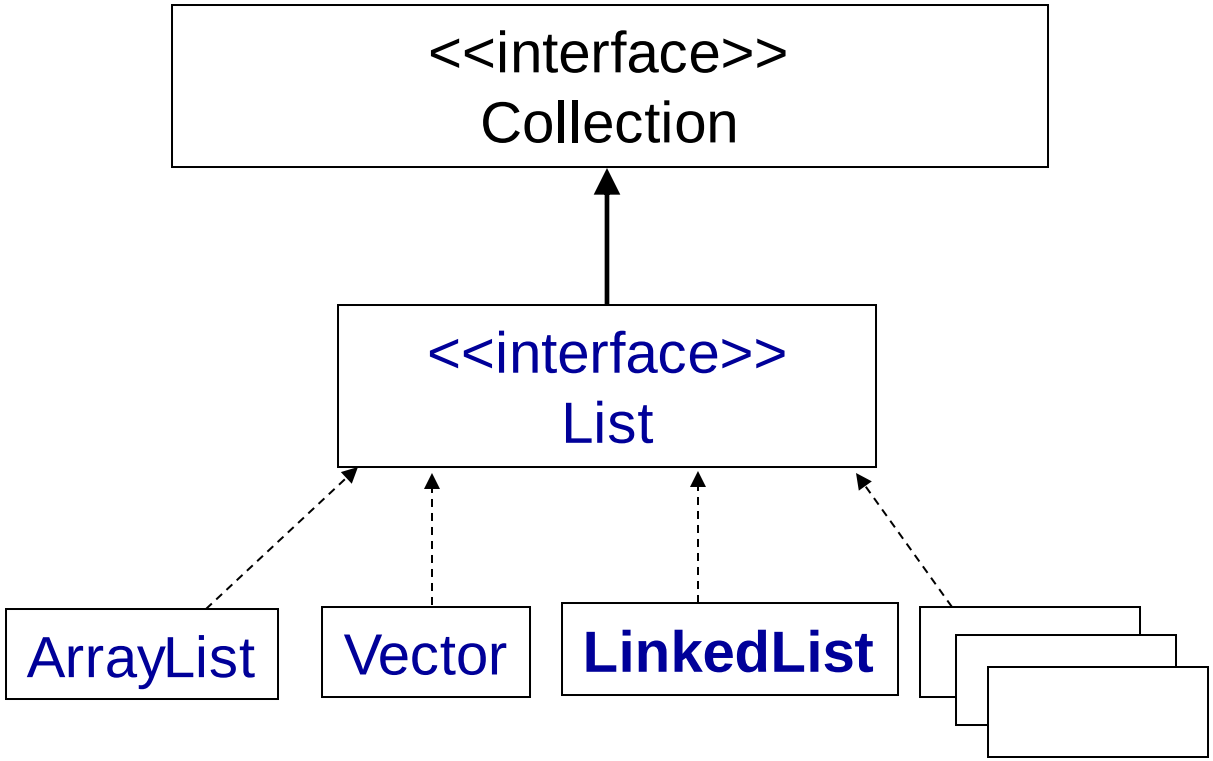




Pesquisar

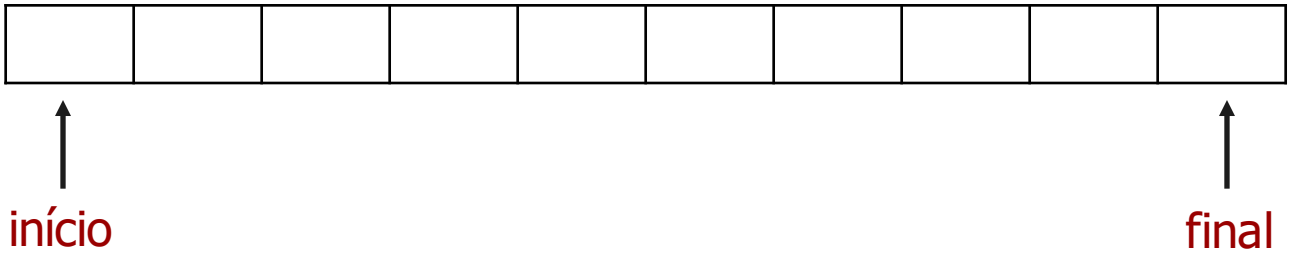
- O que é melhor, usar a classe ArrayList ou a classe Vector?

Visão Geral


<<interface>>
Collection
<<interface>>
List
ArrayList
Vector
LinkedList

LinkedList

- A classe LinkedList trabalha com o conceito de lista encadeada.
- Além de implementar os métodos da interface List (ex.: *add*, *remove*, *get*, etc), a classe LinkedList apresenta outros métodos para acessar os elementos no início ou no final da lista.



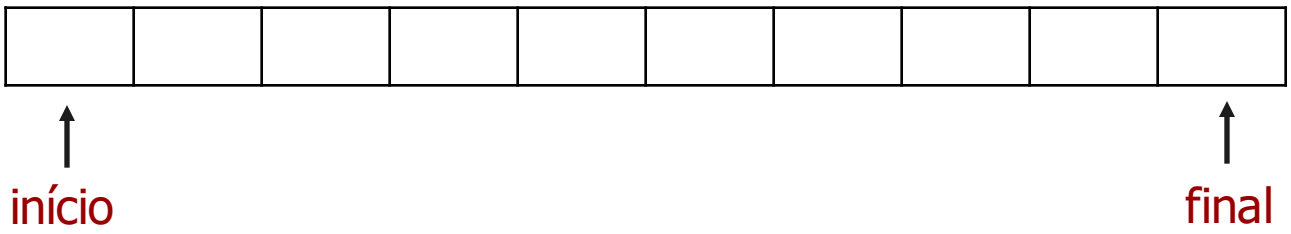
LinkedList - operações

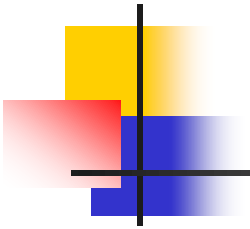
- **Instanciar um objeto**

```
LinkedList<T> lista = new LinkedList<T>();
LinkedList<T> lista = new LinkedList<T>(int i);
```

- **Adicionar um elemento**

```
lista.add(objeto);
lista.addFirst(objeto);
lista.addLast(objeto);
```





LinkedList - operações

- **Retornar o tamanho (número de elementos) da lista:**

```
lista.size();
```

- **Recuperar o elemento (objeto) na posição i:**

```
objeto = lista.get(int i);
```

LinkedList - operações

- **Recuperar o elemento (objeto) na posição i:**

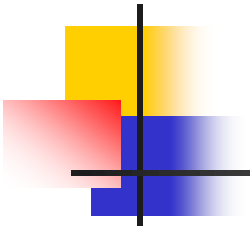
```
objeto = lista.get(int i);
objeto = lista.getFirst();
objeto = lista.getLast();
```

- **Remover o elemento (objeto) da posição i:**

```
lista.remove(int i);
lista.removeFirst();
lista.removeLast();
```

LinkedList - exemplo

```
import java.util.*;
public class Principal {
    public static void main(String[] args){
        System.out.print('\u000C');
        LinkedList <String>listaNomes = new LinkedList<String>();

        listaNomes.add("João");
        listaNomes.addLast("Maria");
        listaNomes.addFirst("Ana");

        Iterator i = listaNomes.iterator();
        while (i.hasNext()) {
            String nome= i.next().toString();
            System.out.println(nome);
        }
    }
}
```



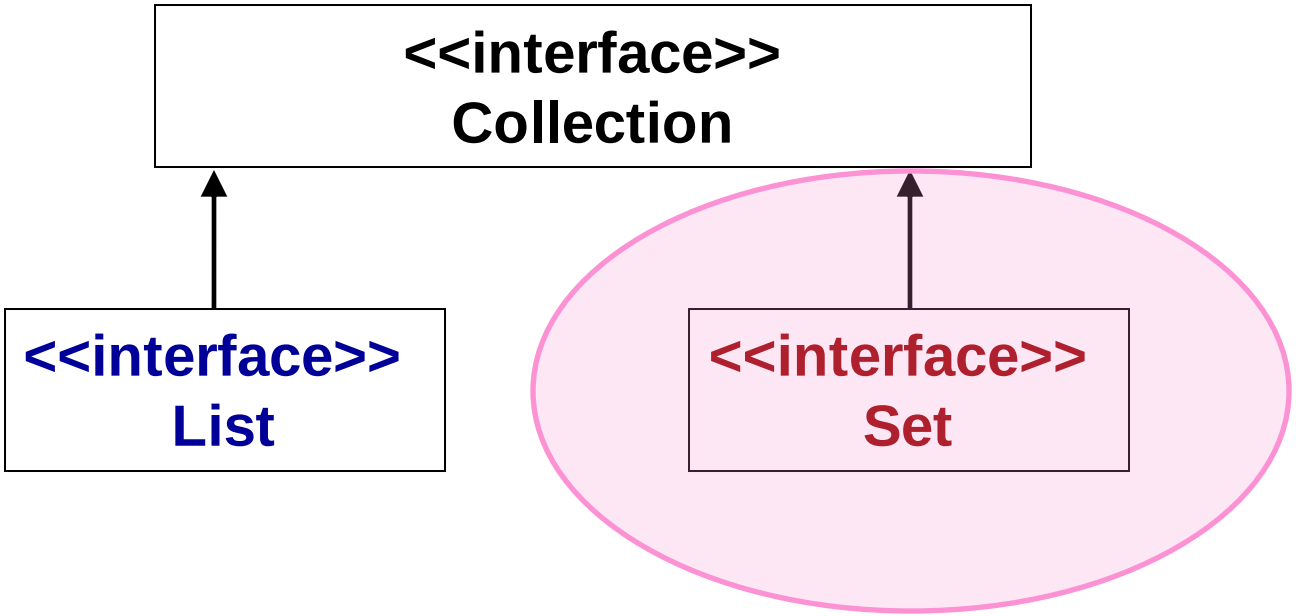
LinkedList x ArrayList

- As diferenças baseiam no custo para inserção, remoção e iteração na lista.

- A LinkedList é a mais rápida para inserção e iteração. Se a lista for apenas para inserir e exibir os elementos (sem remover ou alterar) LinkedList é melhor.

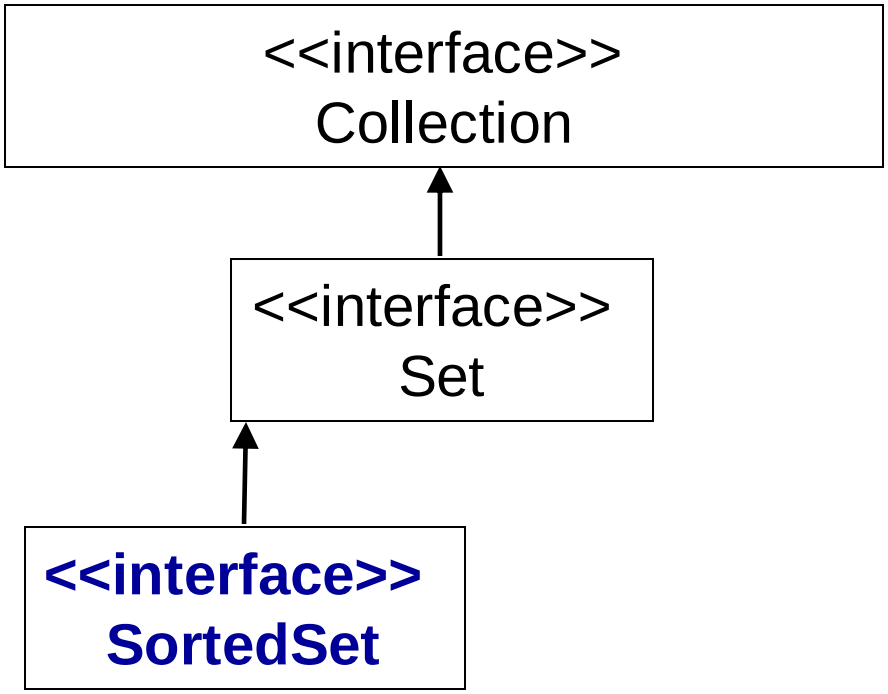
- ArrayList é melhor se você precisa de acesso com índice (acesso aleatório), ou seja , quando você usa o o método get(i).

Visão Geral



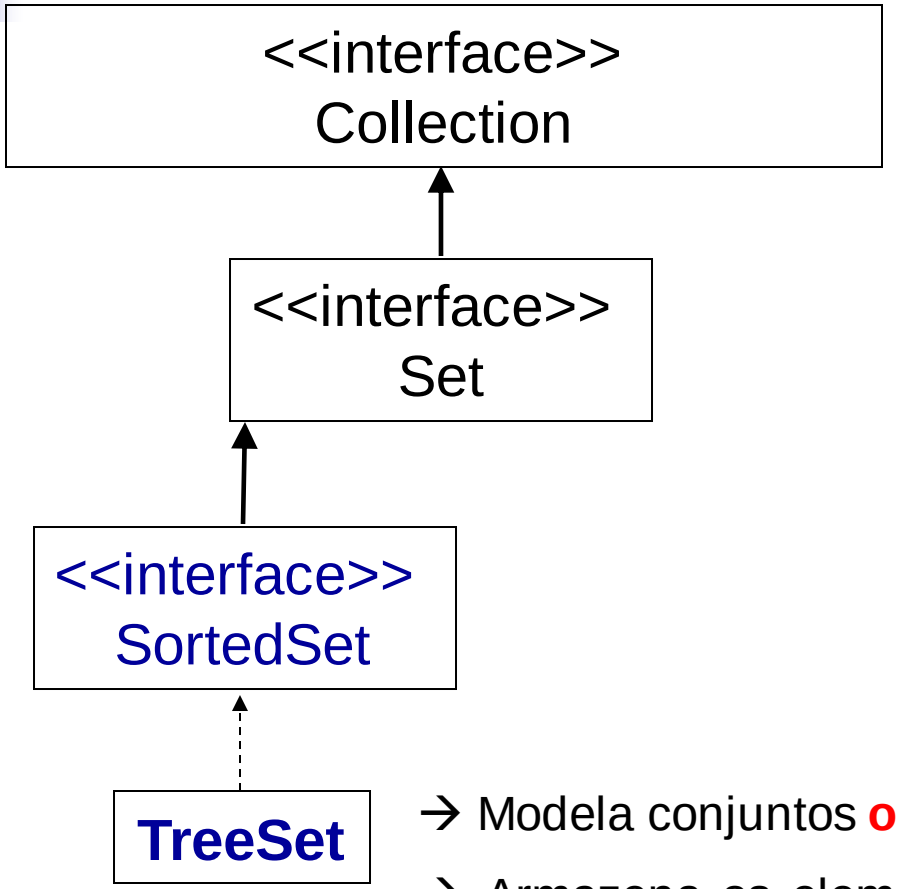
- Um Set é uma Collection que não contém elementos duplicados.
- Não acrescenta nenhum método novo à especificação básica

Visão Geral



- SortedSet é uma **extensão** da interface **Set** que agrega o conceito de **ordenação** ao conjunto.

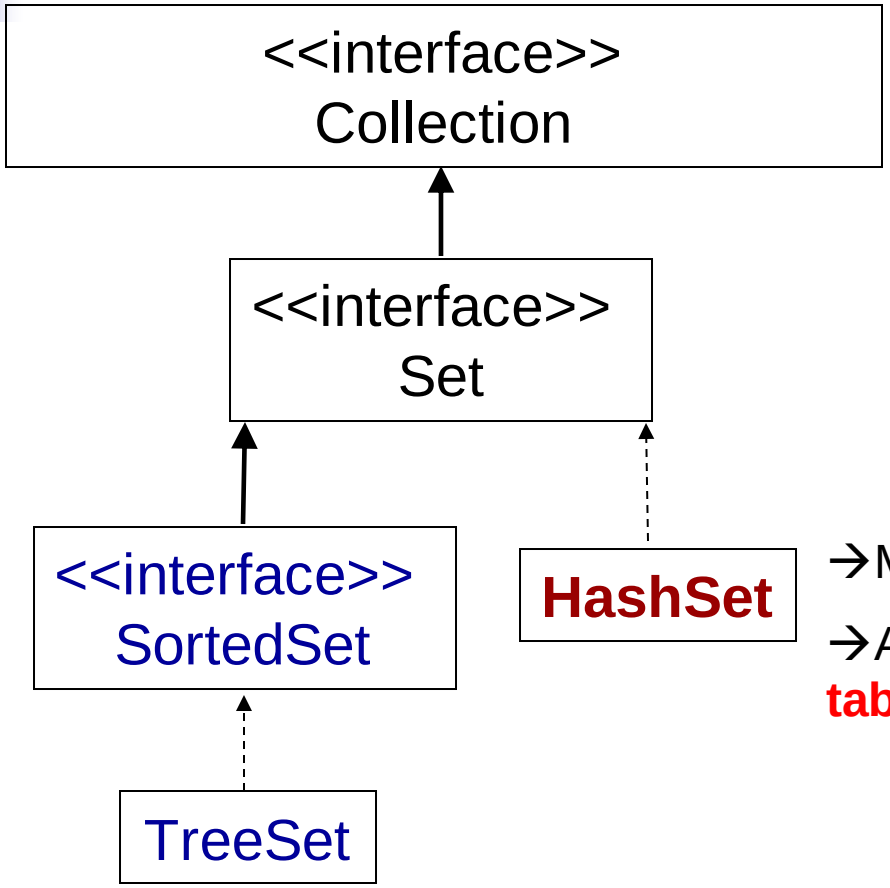
Visão Geral



→ Modela conjuntos **ordenados**.

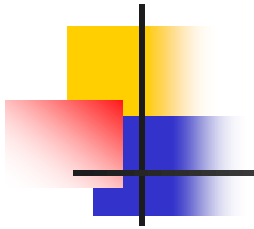
→ Armazena os elementos em uma **árvore**.

Visão Geral



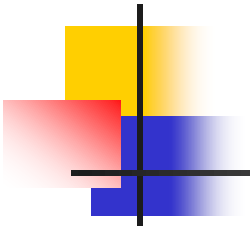
→Modela conjuntos **não ordenados**.

→Armazena os elementos em uma **tabela hash**.



HashSet x TreeSet

- Similaridades:
  - Os elementos de um Set não tem uma ordem como em um array, em que o primeiro elemento está na posição 0
  - Métodos:
    - ✓ add - adicionar um elemento ou um conjunto de elementos.
    - ✓ remove - remover um elemento ou um conjunto de elementos.
    - ✓ contains - retornar *true* se o conjunto possuir algum elemento.
    - ✓ isEmpty - retornar *true* se o conjunto estiver vazio.

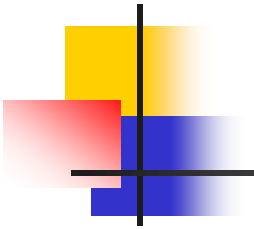


HashSet x TreeSet

- Diferenças:
  - A classe HashSet não possui ordem especifica, enquanto a classe TreeSet define uma ordem.
  - A classe HashSet armazena seus elementos na tabela Hash, enquanto a classe TreeSet armazena em uma árvore.

Exemplo HashSet

```
import java.util.*;
public class PrincipalHashSet {
    public static void main(String[] args) {
        System.out.print('\u000C');

        HashSet<String> c = new HashSet<String>();

        c.add("Maria");
        c.add("Joao");
        c.add("Ana");
        c.add("Joao");
        c.add("Jose");

        Iterator i = c.iterator();
        while(i.hasNext()) {
            String nome = i.next().toString();
            System.out.println(nome);
        }
    }
}
```



Exemplo HashSet

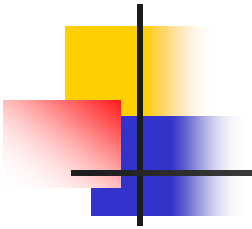
- Resultado: [Joao, Jose, Ana, Maria]

- No exemplo anterior temos um nome que é adicionado duas vezes, mas como a interface Set não permite repetição ele na verdade só é inserido uma vez.

- Observe que não há uma ordem na impressão dos resultados.

Exemplo 1 TreeSet

```
import java.util.*;
public class PrincipalTreeSet {
    public static void main(String[] args) {
        System.out.print('\u000C');

        TreeSet<String> c = new TreeSet<String>();

        c.add("Maria");
        c.add("Joao");
        c.add("Ana");
        c.add("Joao");
        c.add("Jose");

        Iterator i = c.iterator();
        while(i.hasNext()) {
            String nome = i.next().toString();
            System.out.println(nome);
        }
    }
}
```



Exemplo 1 TreeSet

- Resultado é : Ana Joao Jose Maria

- Observe que no TeeSet não é permitido repetições, como em um HashSet, e que neste caso os elementos são automaticamente ordenados.

Exemplo 2 TreeSet

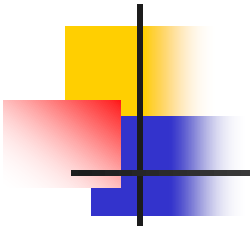
• Vamos agora ver um exemplo com **TreeSet** em que os elementos do conjunto possuem **mais de um campo**.

• Neste exemplo os elementos são **funcionários** de uma empresa que tem **nome** e **salário**.

- Robin
- R$2500,00

Exemplo 2 TreeSet

```
import java.util.*;
public class Principal{

   public static void main(String[] args) {
      Empregado emp1 = new Empregado("Joao", 100);
      Empregado emp2 = new Empregado("Maria", 120);
      Empregado emp3 = new Empregado("Jose", 130);
      Empregado emp4 = new Empregado("Ana", 110);
      Empregado emp5 = new Empregado("Jose", 130);

      Collection <Empregado> c = new TreeSet <Empregado>();
      c.add(emp1);
      c.add(emp2);
      c.add(emp3);
      c.add(emp4);
      c.add(emp5);
      Iterator i = c.iterator();
      while ( i.hasNext() ) {
         Empregado e = (Empregado) i.next();
         System.out.println(e.getNome() + " " + e.getSalario());
      }
   }
}
```



Exemplo 2 TreeSet

- Ao executar o código anterior será lançada a exceção:

java.util.ClassCastException

- O erro ocorre, pois é preciso informar ao TreeSet por qual campo será feita a ordenação dos dados.

- Para isso a classe Empregado deve implementar a interface *Comparable* que nos obriga a implementar o método *compareTo*.

Exemplo 2 TreeSet

```
public class Empregado implements Comparable{
   private String nome;
   private int salario;

   public Empregado(String nome, int salario) {
       this.setNome(nome);
       this.setSalario(salario);
    }

  public String getNome(){
     return this.nome;
  }

  public int getSalario(){
     return this.salario;
  }
  public int compareTo (Object o) {
      Empregado e = (Empregado) o;
      return this.salario - e.salario;
    }
}
```

Método *compareTo*

```
public int compareTo (Object o) {
   Empregado e = (Empregado) o;
   return this.salario - e.salario;
}
```

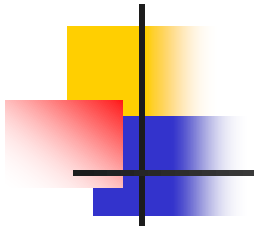
OU

```
public int compareTo (Object o) {
    Empregado e = (Empregado) o;
    if (this.salario < e.salario)
       return -1;
    else if (this.salario > e.salario)
       return 1;
    return 0;
}
```

Resultado – Exemplo 2

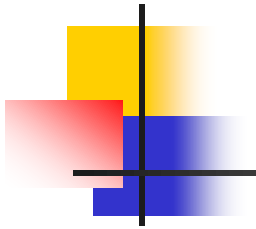
>Joao 100
>Ana 110
>Maria 120
>Jose 130

Bibliografia



- **Biblioteca Virtual – Pearson**
  - Barnes, D. J.; Kolling, M. (2004). Programação Orientada a Objetos com Java: uma introdução prática usando o BlueJ. São Paulo: Pearson Prentice Hall.
  - Barnes, D. J.; Kolling, M. (2009). Programação Orientada a Objetos com Java: uma introdução prática usando o BlueJ. 4ª edição. São Paulo: Pearson Prentice Hall.
  - Deitel, H. M.; Deitel, P. J. (2005). Java como programar. 6ª edição. São Paulo: Pearson Prentice Hall.

Bibliografia



- **Biblioteca Virtual – Pearson (cont.)**

  - Deitel, P. J.; Deitel, H. M. (2010). Java como programar. 8ª edição. São Paulo: Pearson Prentice Hall.

  - Deitel, P. J.; Deitel, H. M. (2017). Java como programar. 10ª edição. São Paulo: Pearson Prentice Hall.
