



Coleções

Prof^a. Rachel Reis
rachel@inf.ufpr.br



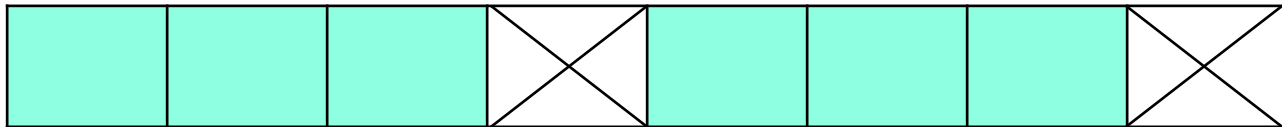
Motivação

- Manipular *arrays* é bastante trabalhoso. Essa dificuldade pode ser observada em diversos momentos:
 - Não é possível redimensionar um *array* em Java
 - É impossível buscar diretamente por um elemento cujo índice não é conhecido
 - Não é possível saber quantas posições do *array* foram ocupadas sem percorrê-lo



Motivação - Exemplo

- Suponha que os dados no *array* abaixo representem contas bancárias. O que acontece quando se deseja inserir uma nova conta?



- Procurar por um espaço vazio? E se não houver?
- Criar um novo *array* maior e copiar os dados antigos para ele?
- Como saber o número de posições ocupadas sem percorrer o *array*?



Solução

- A Sun criou um conjunto de classes e interfaces conhecido como *Collections Framework*.
- A API do *Collections* é robusta e possui diversas classes que representam estruturas de dados avançadas.



Coleção

- Exemplos:

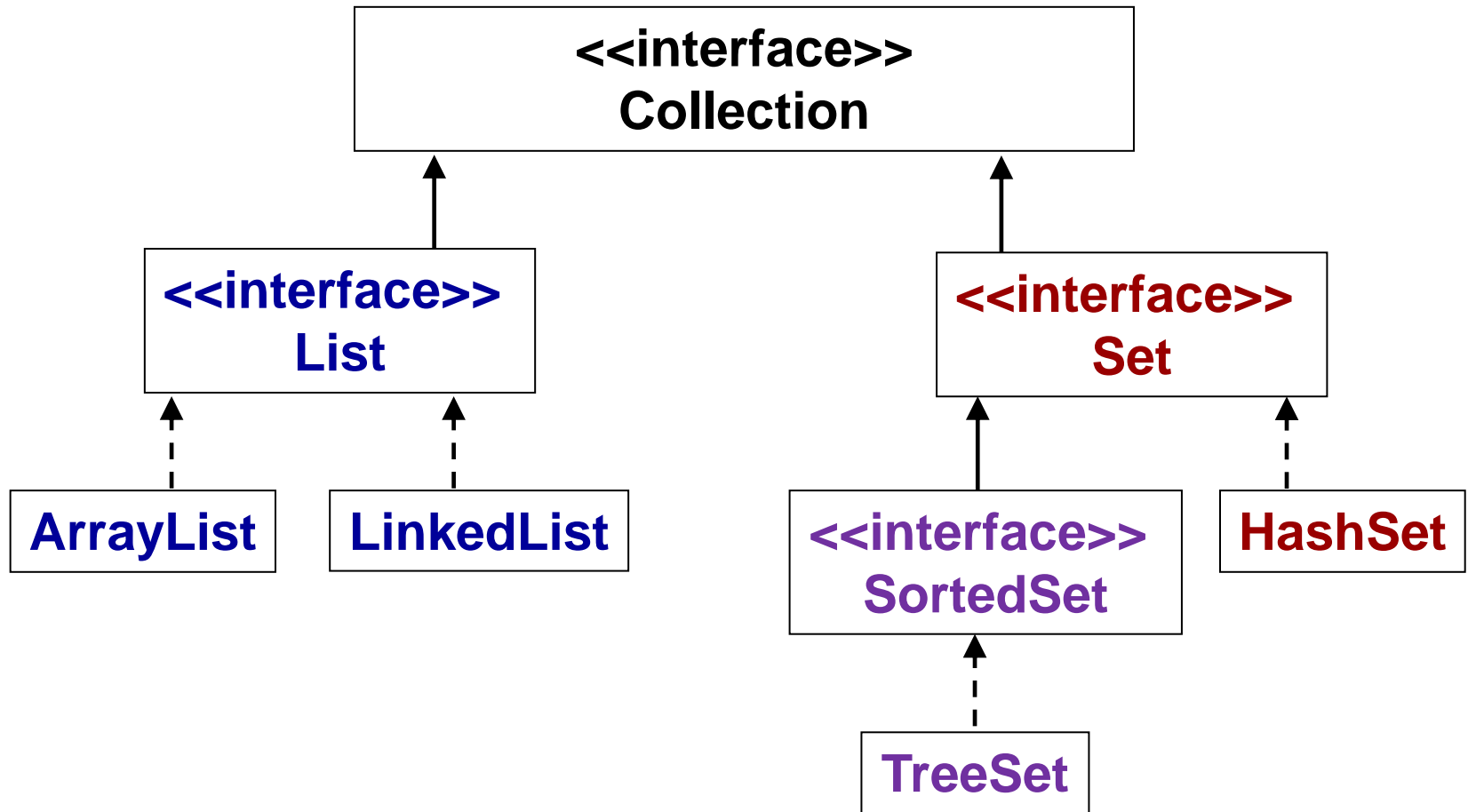
- List

- ArrayList, LinkedList

- Set

- HashSet, TreeSet

Visão Geral





Coleções

- Por que estudá-las?
 - A partir do Java 5 foi adicionado o tipo genérico (*generics*) às *collections*.
 - Logo, são consideradas exemplos de classes que utilizam o polimorfismo paramétrico.



Coleções

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



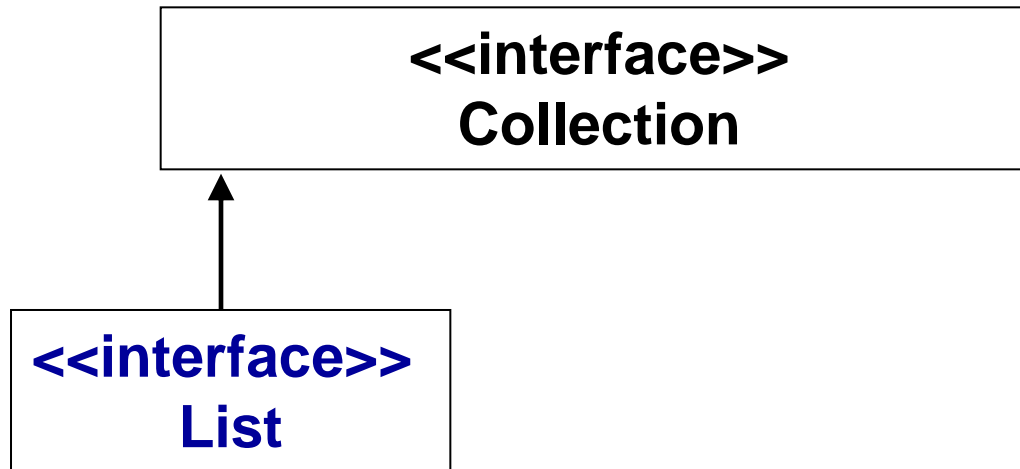
Classes *Wrapper*

- Permite a utilização de tipos primitivos como objetos.

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



Visão Geral



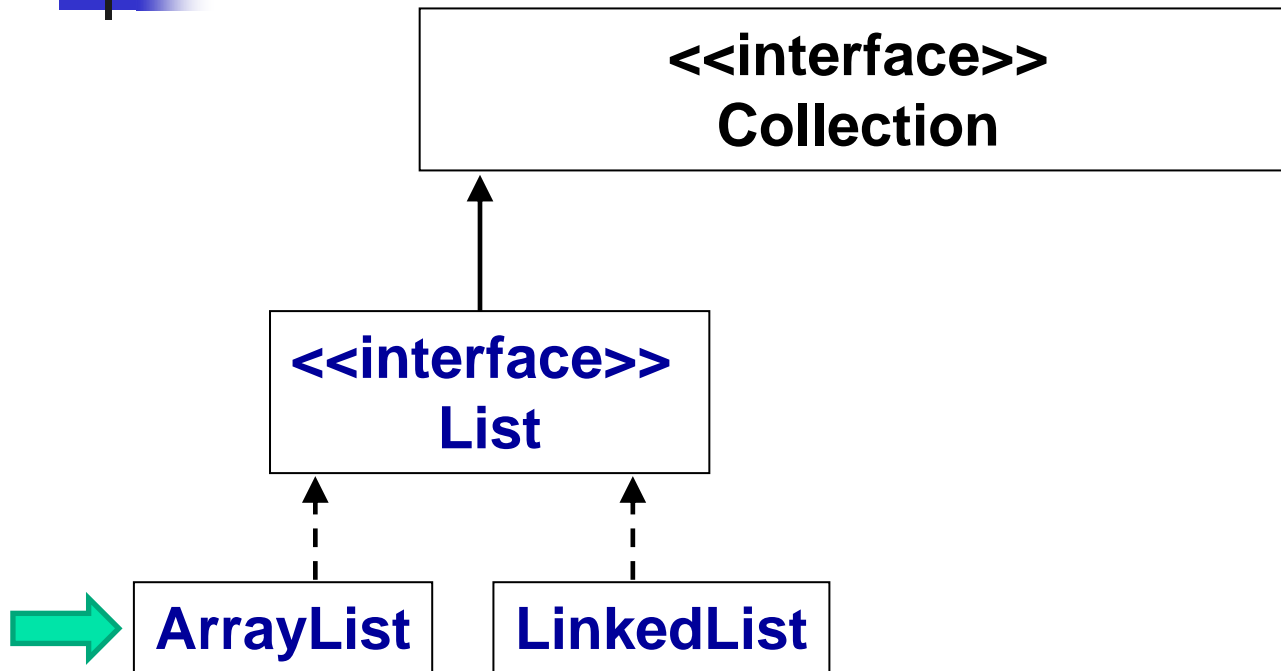


Listas

- É uma *Collection* que pode conter elementos duplicados.
- Mantém uma ordenação específica entre os elementos.
- Resolve os problemas em relação ao *array* (busca, remoção, tamanho, ...).
- A interface *List* é implementada por várias classes.



Visão Geral





ArrayList

- Principais características:
 - Implementado como um array.
 - Acesso sequencial (iniciando no índice 0) e aleatório (acesso a uma posição específica) extremamente rápido.
 - Pode ser redimensionado 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.



- Internamente, essa lista usa um array (encapsulado) como estrutura para armazenar os dados.



Criar um objeto ArrayList

1. Instancia um objeto da classe ArrayList, usando o construtor sem parâmetro.

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

2. Instancia um objeto da classe ArrayList, usando o construtor com parâmetro.

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

→ O valor **20** significa a capacidade inicial da lista.



Criar um objeto ArrayList

- `<T>` representa o tipo dos objetos.

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

- Exemplo:

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

→ O tamanho inicial das listas **a** e **b** será zero, pois nenhum objeto foi adicionado.



Criar um objeto ArrayList

- Lembre-se que é sempre possível abstrair a partir da interface List:

```
List<T> a = new ArrayList<T>();
```



ou

```
Collection<T> a = new ArrayList<T>();
```





Métodos da interface List

- O método **add()** pode ou não receber como parâmetro a posição na lista que desejamos que ele ocupe.
- Exemplo:

```
Pessoa p = new Pessoa("joao", "joao@email.com");  
ArrayList<Pessoa> a = new ArrayList<Pessoa>();  
  
a.add(p);  
OU  
a.add(0, p);
```



Métodos da interface List

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

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

- Para ler dados da lista podemos usar o método get().

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



Métodos da interface List

- Para saber o tamanho (número de elementos) da lista podemos usar o método **size()**.

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



Percorrer um ArrayList

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

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



Percorrer um ArrayList

- O enhanced-for é uma versão simplificada do laço for também usada para percorrer coleções.
- Sintaxe

```
for(<Tipo> <identificador> : <array ou coleção>)  
{  
    <comando>  
}
```



Percorrer um ArrayList

Exemplo:

```
for (Pessoa pessoa: a)
    System.out.println (pessoa.getNome () ) ;
}
```

Sintaxe:

```
for (<Tipo> <identificador> : <coleção>)
{
    <comando>
}
```



Percorrer um ArrayList

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

OU

```
for(Pessoa pessoa: a)  
    System.out.println(pessoa.getNome());  
}
```



```
import java.util.*;
public class Principal{
    public static void main(String[] args)
    {
        ArrayList<String> lista = new ArrayList<String>();
        lista.add("São Paulo");
        lista.add("Paraná");
        lista.add("Santa Catarina");
        lista.add("São Paulo");

        Iterator<String> i = lista.iterator();
        while(i.hasNext()){
            String estado = i.next();
            System.out.println(estado);
        }
    }
}
```





ArrayList - resumo

- ArrayList não remove elementos duplicados.
- Todo ArrayList começa com um tamanho fixo, que vai aumentando conforme necessário.

10 elementos

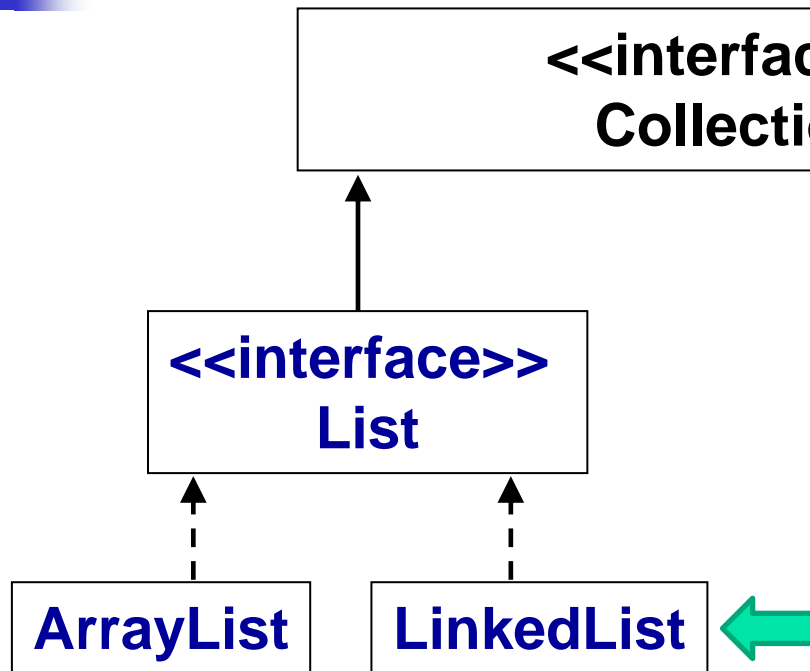
x	x	x	x	x	x	x	x	x	x
---	---	---	---	---	---	---	---	---	---

15 elementos

x	x	x	x	x	x	x	x	x	x					
---	---	---	---	---	---	---	---	---	---	--	--	--	--	--



Visão Geral





LinkedList

- A classe LinkedList trabalha com o conceito de lista encadeada.



- Um objeto da classe LinkedList pode ser instanciado usando o construtor sem e com parâmetro.

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

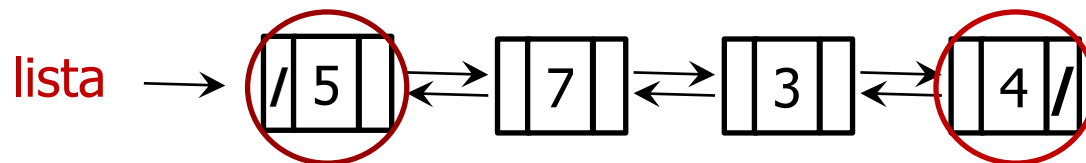
```
LinkedList<T> lista = new LinkedList<T>(int x);
```

→ O valor **x** significa a capacidade inicial da lista.



LinkedList

- 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.





Adicionar elemento no LinkedList

```
// A lista gerencia onde colocar o elemento
```

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

```
// O elemento é adicionado no início da lista
```

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

```
// O elemento é adicionado no final da lista
```

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



Ler os dados do LinkedList

```
// Recupera o elemento de um índice específico  
objeto = lista.get(int i);
```

```
// Recupera o elemento da primeira posição  
objeto = lista.getFirst();
```

```
// Recupera o elemento da última posição  
objeto = lista.getLast();
```



Remover o elemento do LinkedList

```
// Remove o elemento de um índice específico
```

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

```
// Remove o elemento da primeira posição
```

```
lista.removeFirst();
```

```
// Remove o elemento da última posição
```

```
lista.removeLast();
```



```
import java.util.*;
public class Principal{
    public static void main(String[] args)
    {
        LinkedList<String> lista = new LinkedList<String>();
        lista.add("São Paulo");
        lista.add("Paraná");
        lista.add("Santa Catarina");
        lista.add("São Paulo");

        Iterator<String> i = lista.iterator();
        while(i.hasNext()){
            String estado = i.next();
            System.out.println(estado);
        }
    }
}
```



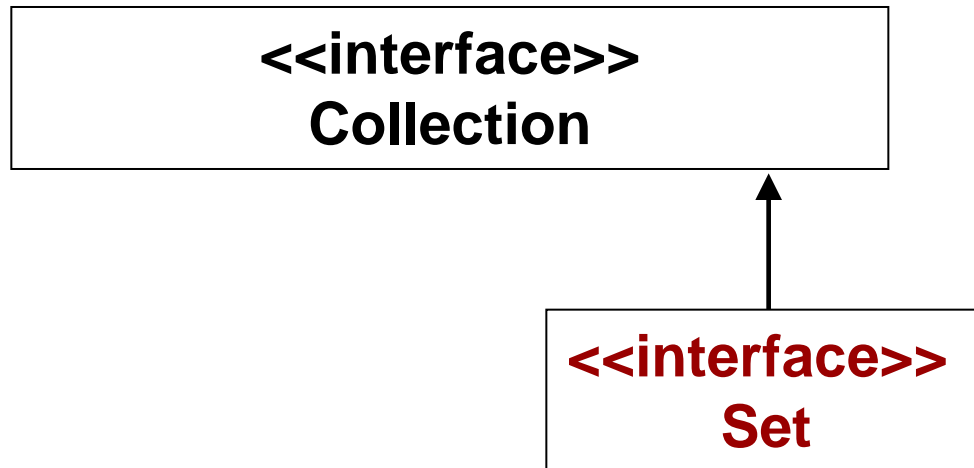


LinkedList x ArrayList

- 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 método `get(i)`.



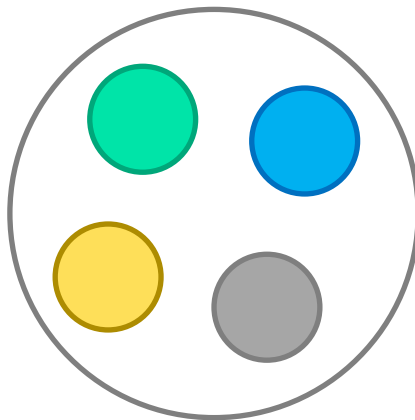
Visão Geral





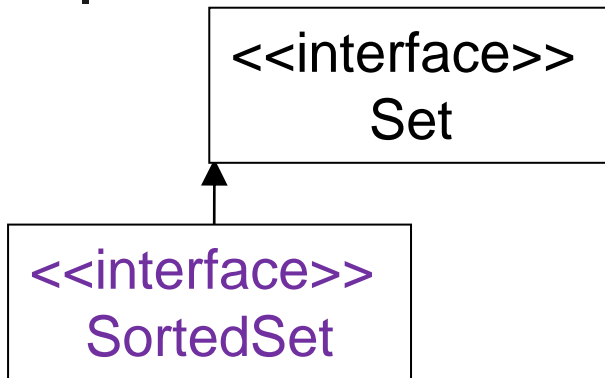
Conjunto

- Um conjunto (*Set*) funciona de forma análoga aos conjuntos da matemática.
- É uma *Collection* que não contém elementos duplicados.
- Os elementos não possuem ordem conhecida.





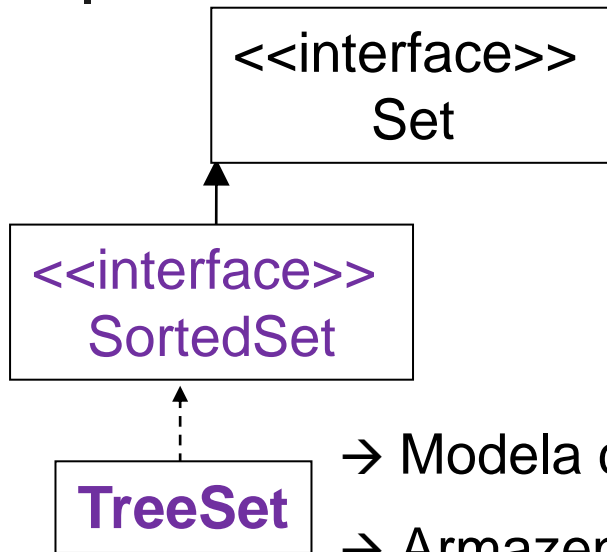
Visão Geral



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



Visão Geral

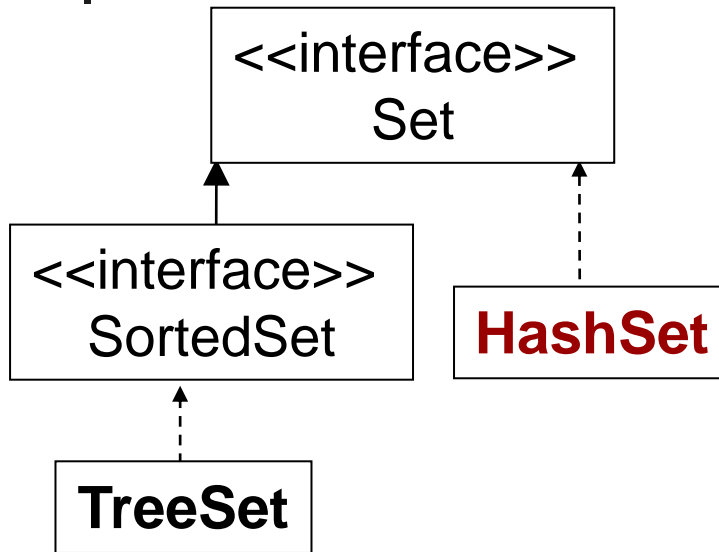


→ Modela conjuntos **ordenados**.

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



Visão Geral



→ Modela conjuntos **não ordenados**.

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





HashSet x TreeSet - Similaridades

- 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 específica, enquanto a classe TreeSet define uma ordem.
- A classe HashSet armazena seus elementos na tabela Hash, enquanto a classe TreeSet armazena em uma árvore.

```
import java.util.*;   
public class Principal{  
    public static void main(String[] args)  
    {  
 HashSet<String> conj = new HashSet<String>();  
        conj.add("São Paulo");  
        conj.add("Paraná");  
        conj.add("Santa Catarina");  
        conj.add("São Paulo");  
  
        Iterator<String> i = conj.iterator();  
        while(i.hasNext()) {  
            String estado = i.next();  
            System.out.println(estado);  
        }  
    }  
}
```



```
import java.util.*; 
public class Principal{
    public static void main(String[] args)
    {
 TreeSet<String> conj = new TreeSet<String>();
        conj.add("São Paulo");
        conj.add("Paraná");
        conj.add("Santa Catarina");
        conj.add("São Paulo");

        Iterator<String> i = conj.iterator();
        while(i.hasNext()){
            String estado = i.next();
            System.out.println(estado);
        }
    }
}
```



Paraná
Santa Catarina
São Paulo



Exemplo TreeSet

- Vamos agora ver um exemplo com TreeSet em que os elementos do conjunto possuem mais de um campo.
- Neste exemplo, os elementos são empregados de uma empresa que possuem nome e salário.



- Robin
- R\$2500,00



Exemplo TreeSet

- Classe Empregado
 - Atributos: nome e salário
 - Construtor completo
 - Métodos get/set
- Classe Principal
 - Método main
 - TreeSet armazenando objetos do tipo Empregado

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

    // Construtor completo
    ...

    // Métodos get/set
    ...

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

```
import java.util.*;

public class Principal{

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

        Collection<Empregado> c = new TreeSet<Empregado>();

        c.add(emp1);
        c.add(emp2);
        c.add(emp3);

        Iterator<Empregado> i = c.iterator();
        while ( i.hasNext() ) {
            Empregado e = i.next();
            S.o.p(e.getNome() + " " + e.getSalario());
        }
    }
}
```



Resultado





Para praticar ...

1. Altere o exemplo da classe Empregado para que os elementos da lista sejam exibidos em ordem decrescente do salário.
2. Altere o exemplo da classe Empregado para que os elementos da lista sejam exibidos em ordem crescente do nome.