Características das Linguagens de Programação I

Conteúdo: Coleções

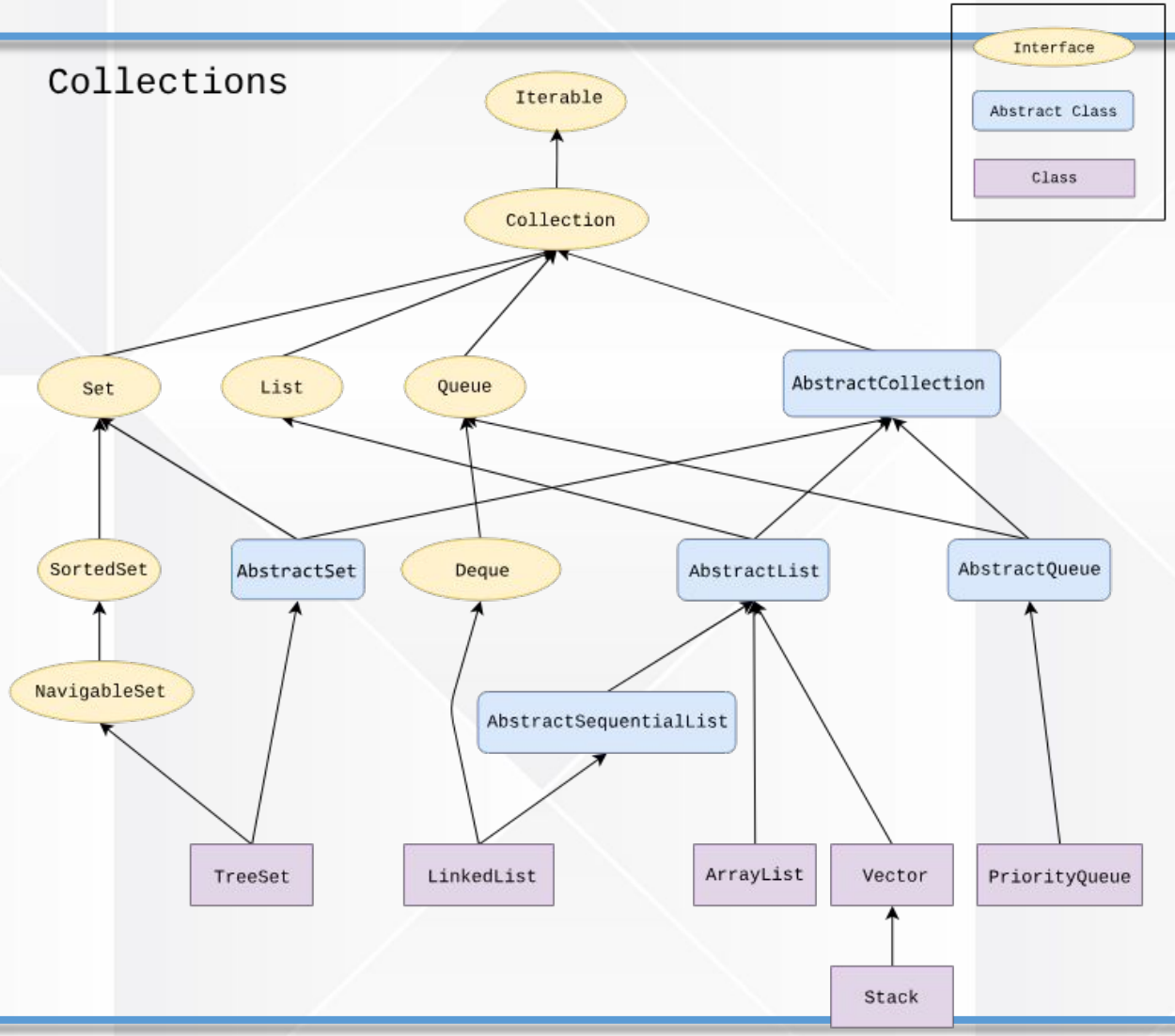
Prof. Dsc. Giomar Sequeiros
giomar@eng.uerj.br

Coleções

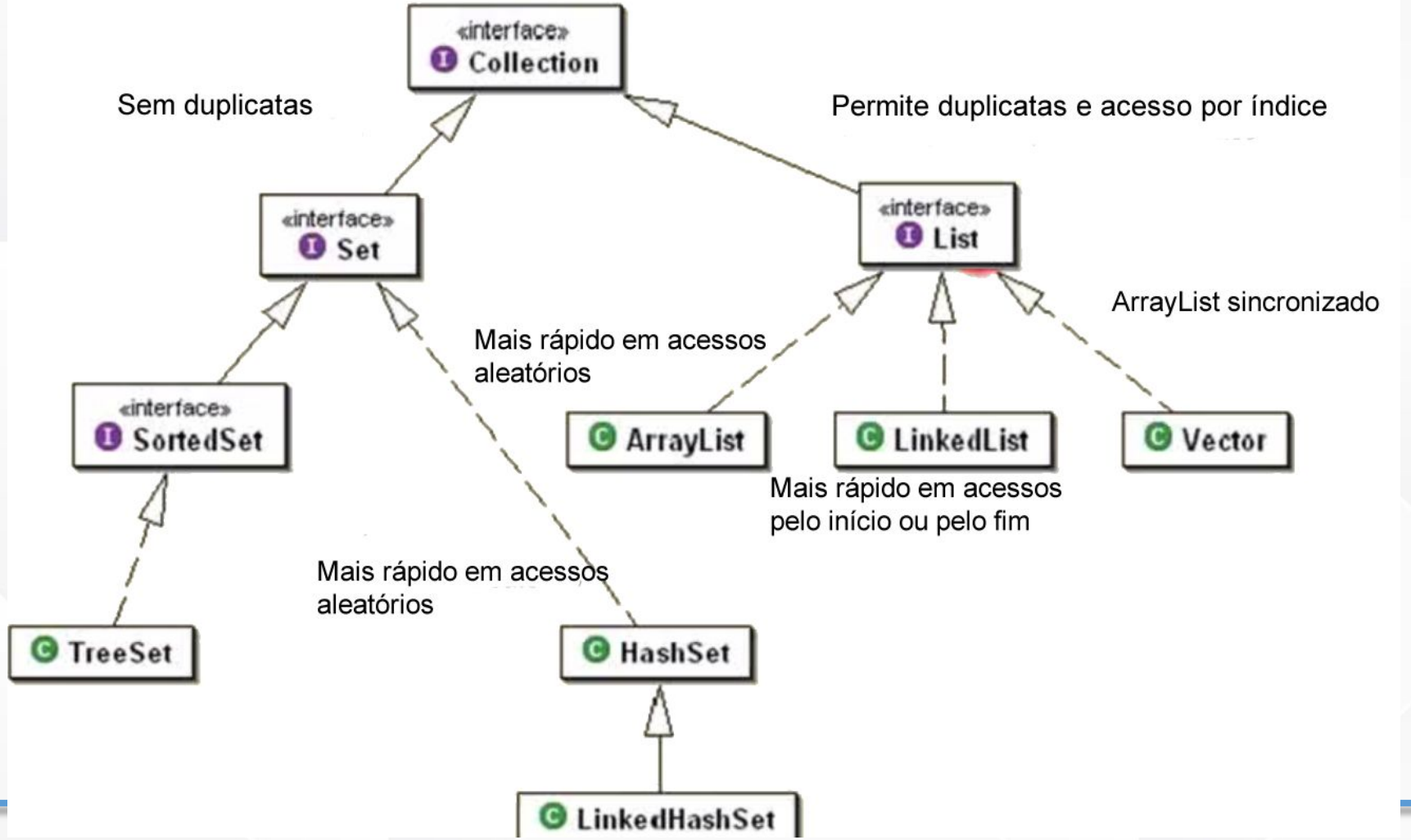
Coleções

- Uma coleção é simplesmente um **objeto** que **agrupa** vários **elementos**.
- É utilizado para **guardar** e **manipular** dados, assim como para transmitir informações entre métodos.
- **Java** possui um **framework** de coleções que está formador por:
  - **Interfaces**: representações abstratas das coleções que permite sua utilização sem conhecer os seus detalhes.
  - **Implementações**: coleções concretas.
  - **Comportamentos**: métodos que permitem realizar operações como busca, ordenações, etc

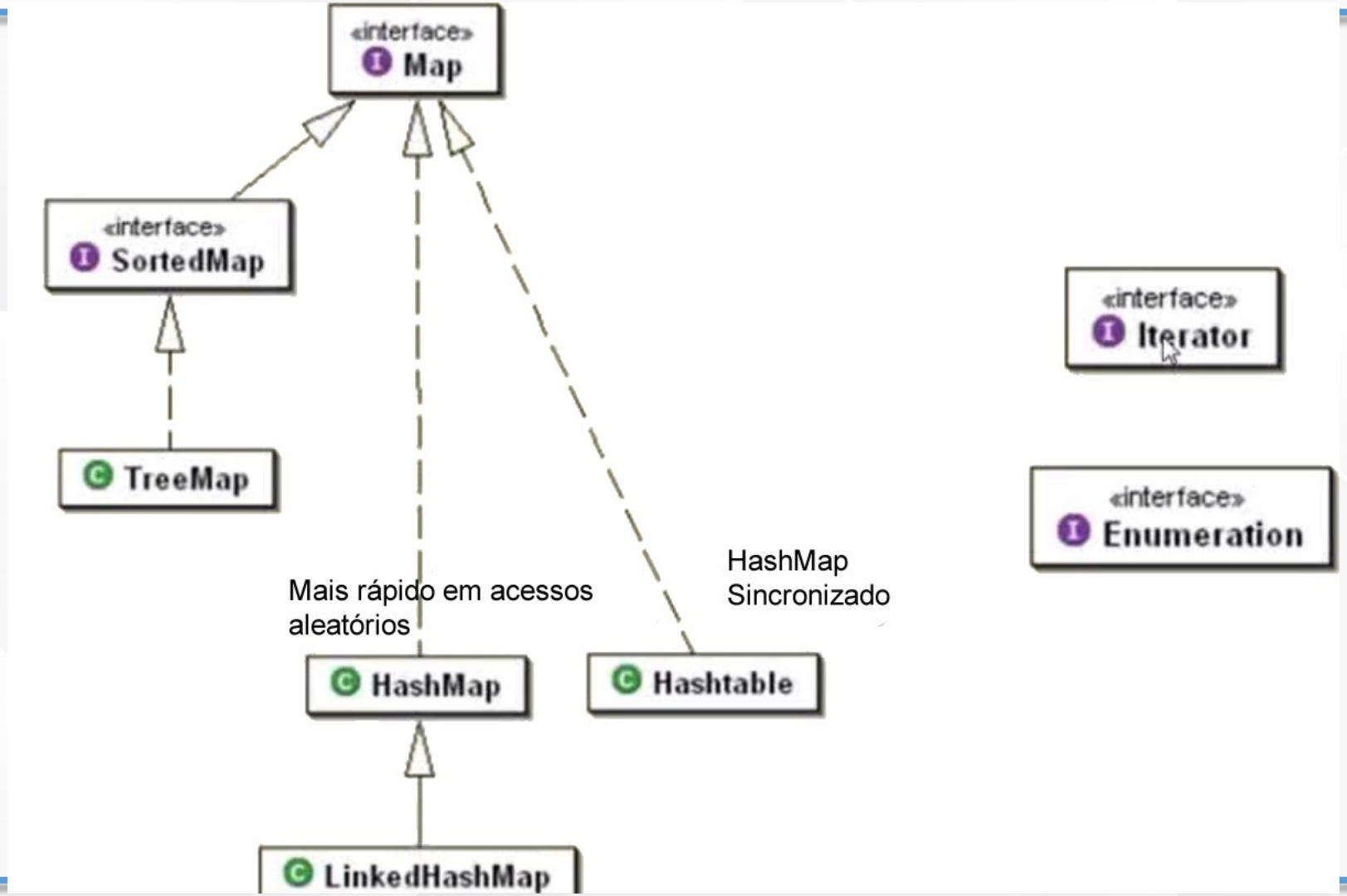
Coleções


Collections
Interface
Abstract Class
Class
Iterable
Collection
Set
List
Queue
AbstractCollection
SortedSet
AbstractSet
Deque
AbstractList
AbstractQueue
NavigableSet
AbstractSequentialList
TreeSet
LinkedList
ArrayList
Vector
PriorityQueue
Stack

Coleções


«interface»
Collection
Sem duplicatas
Permite duplicatas e acesso por índice
«interface»
Set
«interface»
List
ArrayList sincronizado
Mais rápido em acessos
aleatórios
«interface»
SortedSet
ArrayList
LinkedList
Vector
Mais rápido em acessos
pelo início ou pelo fim
Mais rápido em acessos
aleatórios
TreeSet
HashSet
LinkedHashSet

Coleções


«interface»
Map
«interface»
SortedMap
TreeMap
Mais rápido em acessos aleatórios
HashMap
Sincronizado
HashMap
Hashtable
LinkedHashMap
«interface»
Iterator
«interface»
Enumeration

Coleções: java.utils.Collection

- Todas as coleções estão no pacote java.utils.*
- java.utils.Collection é a raiz da hierarquia de coleções.
- Existirão especializações que permitam duplicatas ou não, que permitam ordenação ou não.
- Esta interface contém a definição de todos os métodos genéricos que devem implementar as interfaces.

Coleções: java.utils.Collection

- Os métodos desta interface são:

- Operações básicas:

  - **int** size(); //número de elementos que possui
  - **boolean** isEmpty(); //se não contém algum elemento
  - **boolean** contains(Object element); //se contém esse elemento
  - **boolean** add(Object element); //adicionar um elemento
  - **remove**(Object element); //remover um elemento
  - **Iterator** iterator(); //Retorna uma instância de iterator

Coleções: java.utils.Iterator

- A interface **Iterator** representa um componente que permite iterar sobre os elementos de uma coleção

  public Iterator iterator();

- Todas as coleções oferecem uma implementação de Iterator mediante o método:

- Seus métodos são:
  - **boolean** hasNext(); //Se possui mais elementos
  - Object next(); //Retorna o primeiro elementos e aponta para o próximo

Listas

Interface List

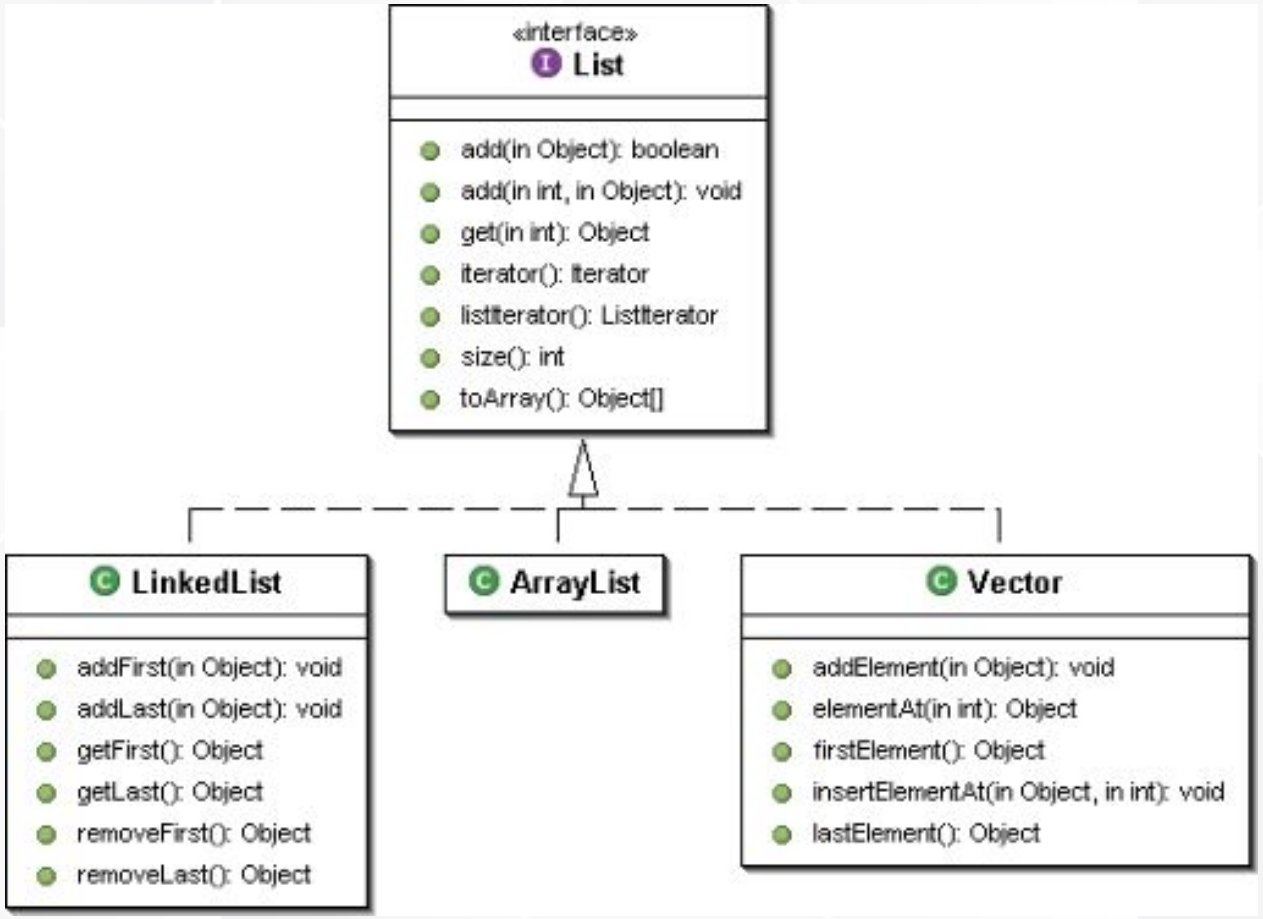
- Uma lista é uma **coleção que permite elementos duplicados** e mantém uma **ordenação** específica entre os elementos.
- Resolve os problemas relacionados ao array (busca, remoção, tamanho,...).

- A API de **Collections** traz a **interface java.util.List**, que especifica o que uma classe deve ser capaz de fazer para ser uma lista. Há diversas implementações disponíveis.
  - A **implementação** mais utilizada da interface List é a **ArrayList**, que trabalha com um array interno para gerar uma lista. Portanto, ela é **mais rápida** na pesquisa do que sua concorrente, a **LinkedList**, que é mais rápida na inserção e remoção de itens nas pontas.

- **ArrayList não é um array!**

Implementações da interface List

- Uma lista é:



Listas

- Para criar um ArrayList, basta chamar o construtor:

```
ArrayList lista = new ArrayList();
```

- É sempre possível abstrair a lista a partir da interface List:

```
List lista = new ArrayList();
```

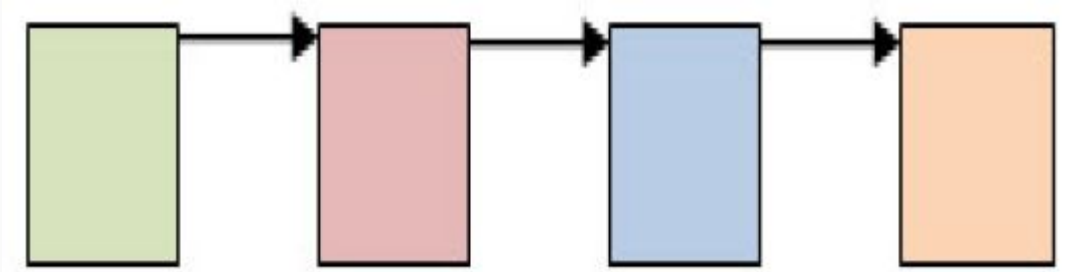
- Para criar uma lista de nomes (String), podemos fazer:

```
List lista = new ArrayList();
lista.add("Manoel");
lista.add("Joaquim");
lista.add("Maria");
```

Listas

- A interface List possui dois métodos **add**, um que recebe o objeto a ser inserido e o coloca no **final da lista**, e um segundo que permite **adicionar** o elemento em **qualquer posição** da mesma. Note que, em momento algum, dizemos qual é o tamanho da lista; podemos acrescentar quantos elementos quisermos, que a lista cresce conforme for necessário.

- Toda lista (na verdade, toda Collection) trabalha do modo mais genérico possível. Isto é, **não há uma ArrayList específica para Strings**, outra para Números, outra para Datas etc. Todos os métodos trabalham com **Object**.



Listas

- Assim, é possível criar, por exemplo, uma lista de Contas Correntes:

```
ContaCorrente c1 = new ContaCorrente();
c1.deposita(100);

ContaCorrente c2 = new ContaCorrente();
c2.deposita(200);

ContaCorrente c3 = new ContaCorrente();
c3.deposita(300);

List contas = new ArrayList();
contas.add(c1);
contas.add(c3);
contas.add(c2);
```

- Para saber quantos elementos há na lista, usamos o método size():

```
System.out.println(contas.size());
```

Listas

- Com o método **get(int)**, que recebe como argumento o índice do elemento que se quer recuperar, podemos fazer um for para iterar na lista de contas, mas como toda lista trabalha sempre com Object, é necessário o **cast** para ContaCorrente se quisermos acessar o getSaldo():

```
for (int i = 0; i < contas.size(); i++) {
  ContaCorrente cc = (ContaCorrente) contas.get(i);
  System.out.println(cc.getSaldo());
}
```

Listas

- Uma lista é uma excelente **alternativa** a um **array** comum, já que temos todos os benefícios de arrays, sem a necessidade de tomar cuidado com remoções, falta de espaço etc.

- A outra implementação muito usada, a **LinkedList**, fornece métodos adicionais para obter e **remover** o primeiro e **último** elemento da lista. Ela também tem o funcionamento interno diferente, o que pode impactar seu desempenho.

Vector

- Outra implementação é a tradicional classe Vector, presente desde o Java 1.0, que foi adaptada para uso com o framework de Collections, com a inclusão de novos métodos.

- Ela deve ser escolhida com cuidado, pois lida de uma maneira diferente com processos correndo em paralelo e terá um custo adicional em relação a ArrayList quando não houver acesso simultâneo aos dados.

Listas com generics

- Em qualquer lista, é possível colocar qualquer Object. Com isso, é possível misturar objetos:

```
ContaCorrente cc = new ContaCorrente();

List lista = new ArrayList();
lista.add("Uma string");
lista.add(cc);
…
```

- Mas e depois, na hora de recuperar esses objetos? Como o método get devolve um Object, precisamos fazer o cast. Mas com uma lista com vários objetos de tipos diferentes, isso pode não ser tão simples...

Listas com generics

- Geralmente,no dia-a-dia, usamos listas como com tipos de dados homogêneos. No Java 5.0 ou superior, podemos usar o recurso de **Generics** para restringir as listas a um determinado tipo de objetos (e não qualquer Object):

```
List<ContaCorrente> contas = new ArrayList<ContaCorrente>();
contas.add(c1);
contas.add(c3);
contas.add(c2);
```

- O uso de um parâmetro ao lado de List e ArrayList indica que nossa lista foi criada para trabalhar exclusivamente com objetos do tipo ContaCorrente. Isso nos traz uma segurança em tempo de compilação:

```
contas.add("uma string"); //erro de compilação
```

Listas com generics

- O uso de Generics também elimina a necessidade de casting, já que, seguramente, todos os objetos inseridos na lista serão do tipo ContaCorrente:

```
for(int i = 0; i < contas.size(); i++) {
  ContaCorrente cc = contas.get(i); // sem casting!
  System.out.println(cc.getSaldo());
}
```

Com generics

```
for (int i = 0; i < contas.size(); i++) {
  ContaCorrente cc = (ContaCorrente) contas.get(i);
  System.out.println(cc.getSaldo());
}
```

Sem generics

Listas com generics

- A partir do Java 7, se você instancia um tipo genérico na mesma linha de sua declaração, não é necessário passar os tipos novamente, basta usar new ArrayList<>(). É conhecido como operador diamante:

```
List<ContaCorrente> contas = new ArrayList<>();
```

Lista: ordenação

- Vimos anteriormente que as listas são percorridas de maneira pré-determinada de acordo com a inclusão dos itens. Mas, muitas vezes, queremos percorrer a nossa lista de maneira ordenada.

- A classe Collections traz um método estático sort que recebe um List como argumento e o ordena por ordem crescente. Por exemplo:

```
List<String> lista = new ArrayList<>();
lista.add("Sérgio");
lista.add("Paulo");
lista.add("Guilherme");

// repare que o toString de ArrayList foi sobrescrito:
System.out.println(lista);
Collections.sort(lista);
System.out.println(lista);
```

Lista: ordenação

- Mas toda lista em Java pode ser de qualquer tipo de objeto, por exemplo, ContaCorrente. E se quisermos ordenar uma lista de ContaCorrente? Em que ordem a classe Collections ordenará? Pelo saldo? Pelo nome do correntista?

```
ContaCorrente c1 = new ContaCorrente();
c1.deposita(500);

ContaCorrente c2 = new ContaCorrente();
c2.deposita(200);

ContaCorrente c3 = new ContaCorrente();
c3.deposita(150);

List<ContaCorrente> contas = new ArrayList<>();
contas.add(c1);
contas.add(c3);
contas.add(c2);

Collections.sort(contas); // qual seria o critério para esta ordenação?
```

Lista: ordenação

- Sempre que falamos em ordenação, precisamos pensar em um **critério** de **ordenação**, uma forma de determinar qual elemento vem antes de qual. É necessário instruir o **sort** sobre como comparar nossas ContaCorrente a fim de determinar uma ordem na lista.

- Para isto, o método sort necessita que todos seus objetos da lista sejam **comparáveis** e possuam um método que se compara com outra ContaCorrente.

- Como é que o método sort terá a garantia de que a sua classe possui esse método? Isso será feito, novamente, através de um **contrato**, de uma interface!

- Vamos fazer com que os elementos da nossa coleção **implementem a interface java.lang.Comparable**, que define o método int compareTo(Object).

- Este método deve retornar zero, se o objeto comparado for igual a este objeto, um número negativo, se este objeto for menor que o objeto dado, e um número positivo, se este objeto for maior que o objeto dado.

Lista: ordenação

Para ordenar as ContaCorrentes por saldo, basta implementar o Comparable:

```
public class ContaCorrente extends Conta
        implements Comparable<ContaCorrente> {

  // ... todo o código anterior fica aqui

  public int compareTo(ContaCorrente outra) {
    if (this.saldo < outra.saldo) {
      return -1;
    }

    if (this.saldo > outra.saldo) {
      return 1;
    }

    return 0;
  }
}
```

Lista: ordenação

- Com o código anterior, nossa classe tornou-se "**comparável**": dados dois objetos da classe, conseguimos dizer se um objeto é maior, menor ou igual ao outro, segundo algum critério por nós definido. No nosso caso, a comparação será feita baseando-se no saldo da conta.

- Repare que o critério de ordenação é totalmente aberto, definido pelo programador. Se quisermos ordenar por outro atributo (ou até por uma combinação de atributos), basta modificar a implementação do método compareTo na classe.

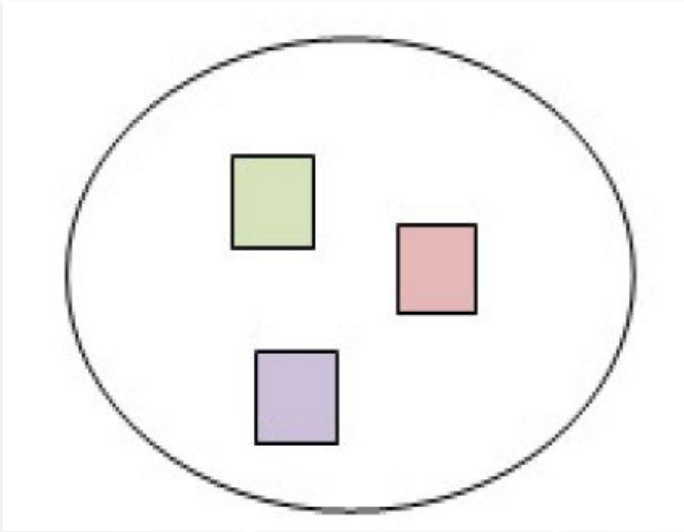
- Quando chamarmos o método **sort** de **Collections**, ele saberá como fazer a ordenação da lista; ele usará o critério que definimos no método compareTo.

Conjuntos

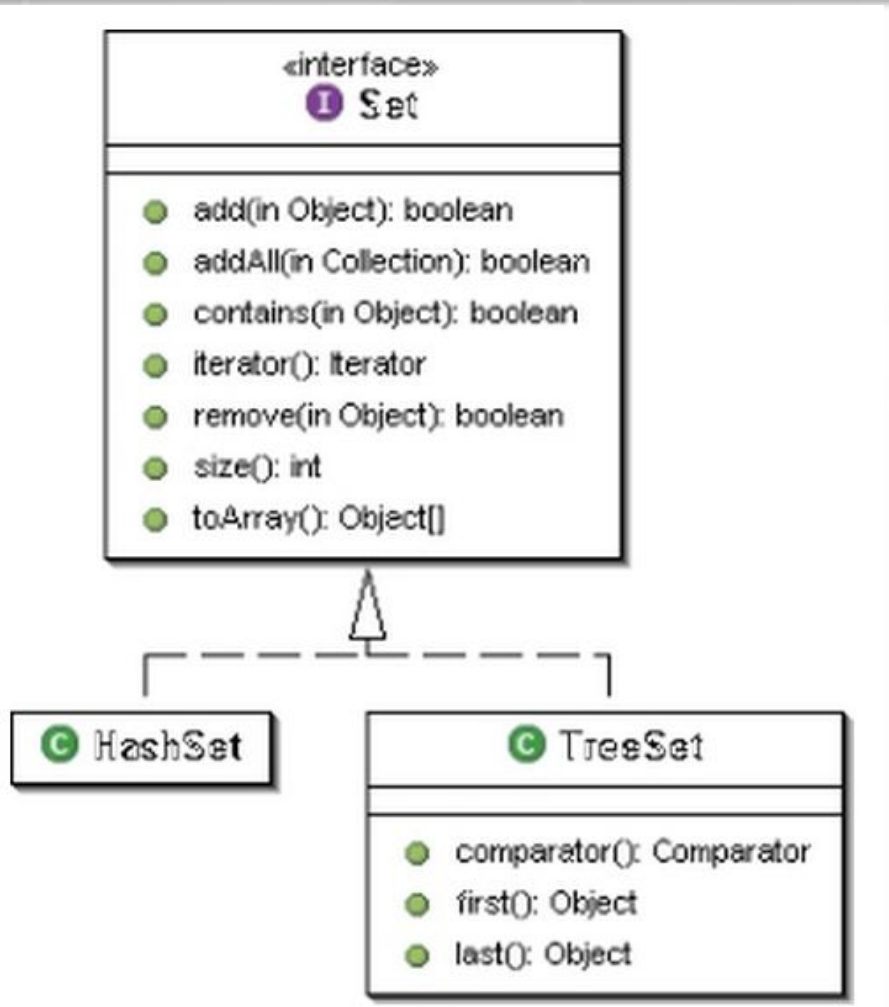
Set

- Um conjunto (Set) funciona de forma análoga aos **conjuntos da matemática**, ele é uma coleção que **não permite elementos duplicados**.

- Outra característica fundamental dele é o fato de que a **ordem em que os elementos** são **armazenados pode não ser a ordem na qual eles foram inserido**s no conjunto. A interface não define como deve ser este comportamento. Tal ordem varia de implementação para implementação.



Set


«interface»
Set
add(in Object): boolean
addAll(in Collection): boolean
contains(in Object): boolean
iterator(): Iterator
remove(in Object): boolean
size(): int
toArray(): Object[]
HashSet
TreeSet
comparator(): Comparator
first(): Object
last(): Object

Set

- Um conjunto é representado pela interface Set e tem como suas principais implementações as classes HashSet, LinkedHashSet e TreeSet.

- O código a seguir cria um conjunto e adiciona diversos elementos, e alguns repetidos:

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

cargos.add("Gerente");
cargos.add("Diretor");
cargos.add("Presidente");
cargos.add("Secretária");
cargos.add("Funcionário");
cargos.add("Diretor"); // repetido!

// imprime na tela todos os elementos
System.out.println(cargos);
```

- Aqui, o segundo Diretor não será adicionado e o método add lhe retornará false.

Set

- O uso de um Set pode parecer desvantajoso, já que ele **não armazena a ordem**, e **não aceita elementos repetidos**. **Não há métodos** que trabalham com **índices**, como o get(int) que as listas possuem.

- A grande vantagem do Set é que existem implementações, como a HashSet, que possui uma performance incomparável com as Lists quando usado para pesquisa (método contains por exemplo).

Set ordenado

- Seria possível usar uma outra implementação de conjuntos, como um **TreeSet**, que insere os elementos de tal forma que, quando forem percorridos, eles apareçam em uma ordem definida pelo método de comparação entre seus elementos. Esse método é definido pela interface java.lang.Comparable. Ou, ainda, pode se passar um Comparator para seu construtor.

- Já o **LinkedHashSet** mantém a ordem de inserção dos elementos.

Percorrendo coleções

- Por exemplo, um Set não possui um método para pegar o primeiro, o segundo ou o quinto elemento do conjunto, já que um conjunto não possui o conceito de "ordem“

- Podemos usar o enhanced-for (o "foreach") do Java 5 para percorrer qualquer Collection sem nos preocupar com isso. Internamente o compilador vai fazer com que seja usado o Iterator da Collection dada para percorrer a coleção.

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

conjunto.add("Rio de Janieiro");
conjunto.add("São gonçalo");
conjunto.add("Niteroi");

for (String palavra : conjunto) {
  System.out.println(palavra);
}
```

- Em que ordem os elementos serão acessados?

Percorrendo coleções

- Para perceber se um item já existe em uma lista, é **muito mais rápido usar algumas implementações de Set do que um List**, e os TreeSets já vêm ordenados de acordo com as características que desejarmos! Sempre considere usar um Set se não houver a necessidade de guardar os elementos em determinada ordem e buscá-los através de um índice.

Percorrendo coleções com Iterator

- Primeiro criamos um Iterator que entra na coleção. A cada chamada do método next, o Iterator retorna o próximo objeto do conjunto. Um iterator pode ser obtido com o método iterator() de Collection, por exemplo numa lista de String:

```
Iterator<String> i = lista.iterator();
```

- A interface Iterator possui dois métodos principais: hasNext() (com retorno booleano), indica se ainda existe um elemento a ser percorrido; next(), retorna o próximo objeto.

Percorrendo coleções com Iterator

- Exemplo:

```
Set<String> conjunto = new HashSet<>();
conjunto.add("item 1");
conjunto.add("item 2");
conjunto.add("item 3");

// retorna o iterator
Iterator<String> i = conjunto.iterator();
while (i.hasNext()) {
  // recebe a palavra
  String palavra = i.next();
  System.out.println(palavra);
}
```

Set: exemplo 1

- Crie uma classe e insira o código a seguir:

```
package colecoes;

import java.util.*;

public class Colecoes {

public static void main(String[] args) {

HashSet  conjunto = new HashSet();

conjunto.add(123);           // insere o núemro 123
System.out.println(conjunto);

conjunto.add(234.7);         // insere o número 234.7
System.out.println(conjunto);

conjunto.add("ABC");         // insere a palavra ABC
System.out.println(conjunto);

conjunto.add("123");         // insere a palavra 123
System.out.println(conjunto);

conjunto.add("ABC");         // insere a palavra 123
System.out.println(conjunto);

}
}
```

Set: exemplo 1

- continua…

```
conjunto.remove("123");   // removendo um elemento
System.out.println(conjunto);

if(conjunto.contains("ABC")){ // buscando um elemento
   System.out.println("Achei o elemento"); }
else{
   System.out.println("Não consegui achar o elemento"); }




if(conjunto.remove("125")){ // tentando remover um elemento
   System.out.println("Consegui remover o elemento"); }
else{
   System.out.println("Não consegui remover o elemento"); }
```

Set: exemplo 1

- continua…

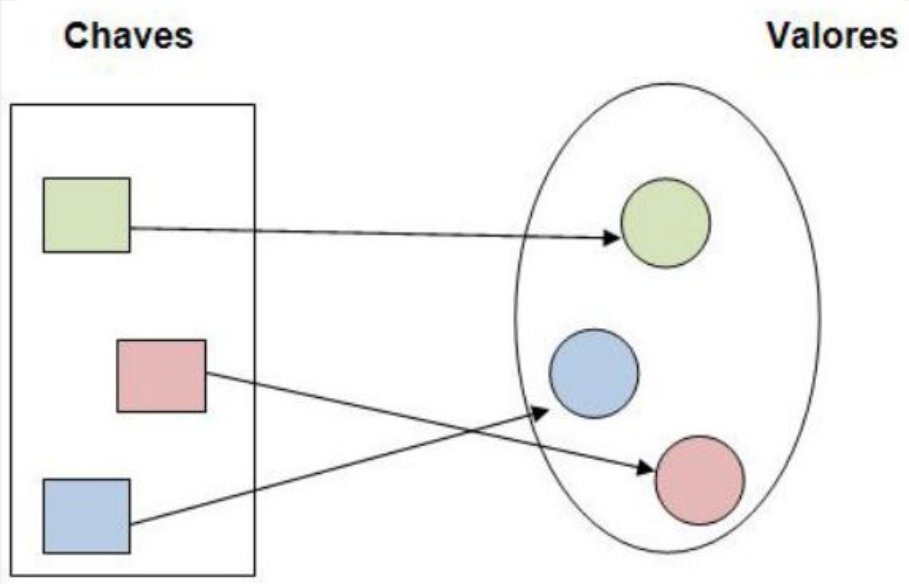
```
Iterator i = conjunto.iterator();
int x;

while(i.hasNext())
{
Object o;
        o = i.next();
if (o instanceof Integer) {
        System.out.println("Achei um Integer:"+ o);
        x=(int) o;
        System.out.println("Achei um Integer:"+ x);
    }
}
```

Mapas

Mapas

- Muitas vezes queremos buscar rapidamente um objeto dado alguma informação sobre ele. Um exemplo seria, dada a placa do carro, obter todos os dados do carro. Poderíamos utilizar uma lista para isso e percorrer todos os seus elementos, mas isso pode ser péssimo para a performance, mesmo para listas não muito grandes. Aqui entra o mapa.

- Um mapa é composto por um conjunto de associações entre um objeto chave a um objeto valor.



Mapas: métodos principais

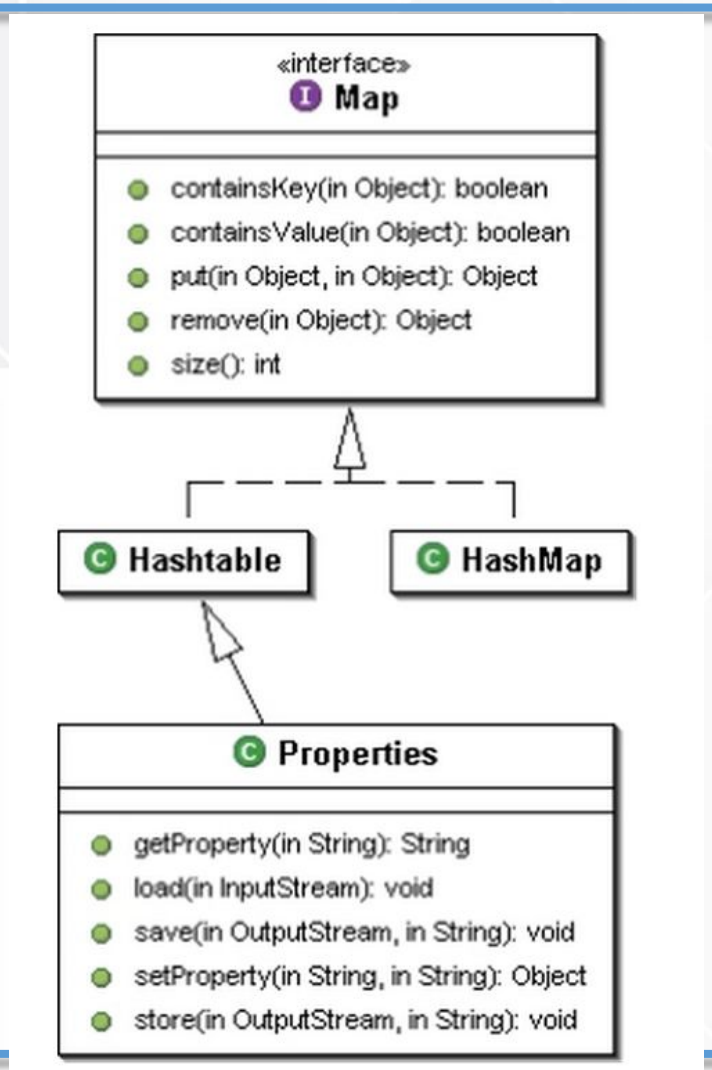
- **put**: adiciona um objeto ao mapa.
- **remove**: remove um objeto do mapa.
- **get**: recupera um objeto do mapa.
- **KeySet**: retorna um set com todas as chaves.
- **values**: retorna uma coleção com todos os valores .

Mapas: exemplo 1

Observe o exemplo: criamos duas contas correntes e as colocamos em um mapa associando-as aos seus donos.

```
ContaCorrente c1 = new ContaCorrente();
c1.deposita(10000);

ContaCorrente c2 = new ContaCorrente();
c2.deposita(3000);

// cria o mapa
Map<String, ContaCorrente> mapaDeContas = new HashMap<>();

// adiciona duas chaves e seus respectivos valores
mapaDeContas.put("diretor", c1);
mapaDeContas.put("gerente", c2);

// qual a conta do diretor? (sem casting!)
ContaCorrente contaDoDiretor = mapaDeContas.get("diretor");
System.out.println(contaDoDiretor.getSaldo());
```

Mapas: implementações



Mapas: exemplo 2

Crie uma classe e implemente o código a seguir:

```
package colecoes4;

import java.util.*;

public class Colecoes4 {

public static void main(String[] args) {

HashMap  mapa = new HashMap();

mapa.put(1,"um");
mapa.put(2,"dois");
mapa.put(4,"quatro");
mapa.put(3,"três");
mapa.put(0,"zero");
System.out.println(mapa);

mapa.remove("dois");
System.out.println(mapa);

mapa.remove(2);
System.out.println(mapa);
  }
}
```

Mapas: exemplo 3

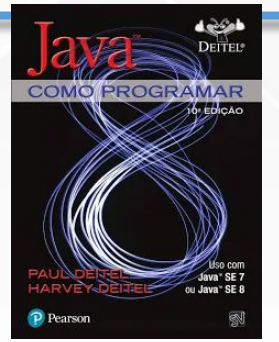
Crie uma classe e implemente o código a seguir:

```
package colecoes5;

import java.util.*;

public class Colecoes5 {

    public static void main(String[] args) {

TreeMap<String,Integer> mapa = new TreeMap<String,Integer>();
mapa.put("um",1);
mapa.put("dois",2);
mapa.put("três",3);
mapa.put("quatro",4);
mapa.put("cinco",5);
System.out.println(mapa);

System.out.println(mapa.get("quatro")+ mapa.get("dois"));
    }
}
```

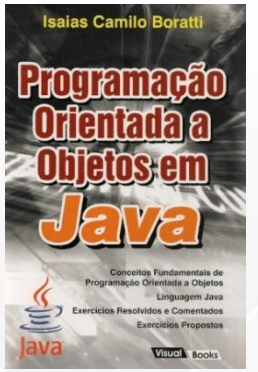
Referências

BIBLIOGRAFIA BÁSICA:

- DEITEL, Harvery M.. Java : como programar. 10ª ed. São Paulo: Pearson - Prentice Hall, 2017.



- BORATTI, Isaías Camilo. Programação Orientada a Objetos em Java : Conceitos Fundamentais de Programação Orientada a Objetos. 1ª ed. Florianópolis: VisualBooks, 2007.



- SIERRA, Kathy; BATES, Bert. Use a Cabeça! Java. 2ª ed. Rio de Janeiro: Alta Books, 2007.