

Collections

Collections

Objetivos:

- Ao final desta apresentação espera-se que você saiba:
 - Distinguir cada tipo de Coleção.
 - Escolher a melhor implementação de Coleção (entre as básicas) para cada tipo de uso.
 - Saber a diferença entre as Interfaces: Collection, Set, Queue, List, SortedSet, NavigableSet, Map, SortedMap e NavigableMap.
 - Conhecer as implementações básicas: ArrayList, Vector, LinkedList, PriorityQueue, TreeSet, HashSet, LinkedHashSet, TreeMap, HashMap e LinkedHashMap.

Collections

- Pre-requisitos:
 - **Igualdade**: Para se trabalhar com coleções é preciso entender a igualdade em Java
- Porque é preciso?
 - Quando chegarmos lá tudo ficará mais claro, mas adiantando, varias funcionalidades de coleções como **contains(Object o)** – que verifica se existe um objeto na coleção – precisa testar a igualdade para funcionar.

Collections

- Como funciona igualdade em Java ?
pessoa1 == pessoa2
- Isso irá testar apenas se as duas variáveis referenciam uma única instância!

Collections

- Como então saber, por exemplo, quando dois objetos **Pessoa** são significativamente iguais?
 - Para esse tipo de teste você deve usar um método que todo objeto Java tem **equals(Object o)**
- Então nosso teste de igualdade seria:
`pessoa1.equals(pessoa2)`

Collections

- Só isso resolve?
 - Não
- Por padrão o método *equals* de **Object** faz o mesmo que o operador ==

```
public class Object {  
    //..  
    public boolean equals(Object obj) {  
        return (this == obj);  
    }  
    //...
```

Collections

- Então se queremos definir em nosso modelo que dois objetos **Pessoa** são iguais quando, por exemplo, seus cpf são iguais?
 - Neste caso nós precisamos ensinar ao nosso programa a igualdade entre nossos objetos
- Como fazer isso??
 - Sobrescrevendo o *equals*

Collections

- Mas não basta simplesmente sobrescrever!
- E existe um **contrato de equals** que **deve ser rigorosamente seguido** para o bom funcionamento do programa e da Java Collections Framework

Collections

- O contrato diz que *equals* é:
 - Reflexível: `a.equals(a) == true`
 - Simétrico: `a.equals(b) == b.equals(a)`
 - Transitivo: `se a.equals(b) && b.equals(c) == true ∴ a.equals(c) == true`
 - Consistente: `a.equals(b)` deve retornar sempre `true` ou sempre `false`, desde que nenhuma propriedade do do Objeto que faz parte do teste *equals* seja alterada.
 - Deve retornar **false** quando testado contra uma variável **null**

Collections

- Portanto para nossa classe **Pessoa** podemos definir um *equals* assim:


```
public class Pessoa {  
    private String cpf;  
    //...  
    @Override // não é necessário mas é bom  
    public boolean equals(Object p) {  
        if (cpf == null)  
            return false;  
        return (p instanceof Pessoa) &&  
            this.cpf.equals(((Pessoa)p).cpf);  
    }  
    //...  
}
```

Collections

- Portanto para nossa classe **Pessoa** podemos definir um *equals* assim:

```
public class Pessoa {  
    private String cpf;  
    //...  
    @Override  
    public boolean equals (Pessoa p) {  
        if (cpf == null)  
            return false;  
        return (p instanceof Pessoa) &&  
            this.cpf.equals(((Pessoa)p).cpf);  
    }  
    //...  
}
```

Eu não podia colocar
A classe **Pessoa**?
No lugar de **Objeto** ?



Collections

- Portanto para nossa classe **Pessoa** podemos definir um *equals* assim:

```
public class Pessoa {  
    private String cpf;  
    //...  
    @Override  
    public boolean equals(Pessoa p) {  
        if (cpf == null)  
            return false;  
        return (p instanceof Pessoa) &&  
            this.cpf.equals(((Pessoa)p).cpf);  
    }  
    //...  
}
```

Eu não podia colocar
A classe **Pessoa**?
No lugar de **Objeto** ?

NÃO!!!!

Pois é preciso
sobrescrever o
método da classe
Object e portanto
manter a mesma
assinatura

Collections

- Então é isso basta sobrescrever o método *equals* corretamente na minha classe e já posso usar:
 pessoa1.equals(pessoa2)
- Não preciso fazer mais nada?
 - **ERRADO!** Precisa sim! E é aí o erro mais comum de quem trabalha com collections
- O contrato de **equals(Object o)** esta fortemente ligado ao de **hashCode()**

Collections

- E pra que serve esse hashCode() ?
 - Calma que assim que chegar em coleções falaremos dele ao estudar **HashSet**, **LinkedHashSet**, **HashMap**, **LinkedHashMap** e **Hashtable**
- Certo mas então o que ele tem a ver com meu teste de igualdade ??
 - As definições do contrato de hashCode() o ligam diretamente ao teste de igualdade, portanto sempre que sobrescrever equals(**Object** o) você terá que sobrescrever também o hashCode()

Collections

- O contrato do **hashCode()** diz:
 - É constante: qualquer chamada a `hashCode()` deve sempre retornar o mesmo inteiro, desde que as propriedade usadas no teste **`equals(Object o)`** não sejam alteradas.
 - É igual para objetos iguais: `a.equals(b) == true` \therefore `a.hashCode() == b.hashCode()`
 - Não é necessário ser diferente para objetos diferentes: ou seja, se `a.equals(b) == false` \therefore `a.hashCode() == b.hashCode() || hashCode() != b.hashCode()`

Collections

- Por padrão o método hashCode() é diferente para cada instancia de **Object**.
- Não é implementado em java e sim com linguagem nativa:

```
public class Object {  
    //..  
    public native int hashCode();  
    //...  
}
```


Collections

- Portanto no nosso modelo dois objetos **Pessoa** com o mesmo cpf, vai retornar *hashCode* diferentes (pois esse é o padrão do *hashCode*)
- Isto fere o contrato de *hashCode*, pois:
 true
 pessoa1.hashCode() != pessoa2.hashCode()

Collections

```
Pessoa p1 = new Pessoa("123.456.789-00"); //igual a p2
String text = "São iguais? %b ... hashCode? %d , %d \n";
for (int i = 0; i < 30; i++) {
    Pessoa p2 = new Pessoa("123.456.789-00"); //igual a p1
    System.out.printf( text, p1.equals(p2), p1.hashCode(),
                      p2.hashCode() );
}
```

- O output disso é algo como:
 - São iguais? true ... hashCode? 11394033 , 4384790
 - São iguais? true ... hashCode? 11394033 , 24355087
 - São iguais? true ... hashCode? 11394033 , 5442986
 - São iguais? true ... hashCode? 11394033 , 10891203
 - São iguais? true ... hashCode? 11394033 , 9023134
 - etc

Collections

- Portanto para nossa classe **Pessoa** um **hashCode()** valido pode ser:

```
public class Pessoa {  
    private String cpf;  
    //...  
    @Override // não é necessário mas é bom  
    public int hashCode() {  
        return (cpf == null) ? 0 : cpf.hashCode();  
    }  
    //...  
}
```

Collections

- Portanto para nossa classe **Pessoa** um **hashCode()** valido pode ser

```
public class Pessoa {  
    private String cpf;  
    //...  
    @Override  
    public int hashCode() {  
        return 12;  
    }  
    //...  
}
```

Eu não poderia retornar sempre o mesmo número? Isso não feriria o contrato!

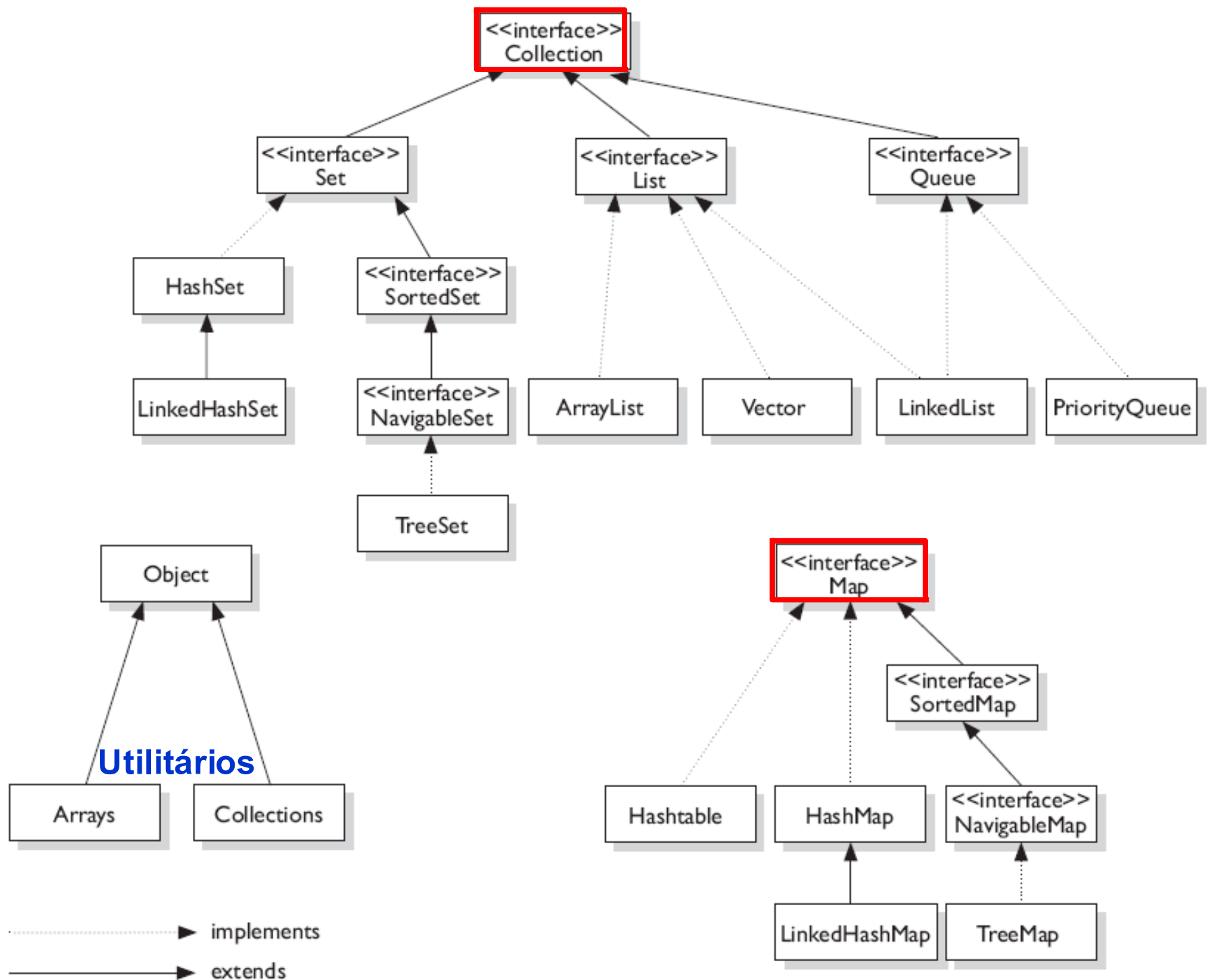
SIM! Poderia...
Porem não é nem um pouco eficiente, como veremos mais adiante quando estudarmos as coleções que usam hash

Collections

- Vamos ao que interesssa...

Collections

- **Java Collections Framework (JCF)**
 - Existem duas interfaces principais são elas:
 - **java.util.Collection**: uma coleção de objetos
 - **java.util.Map**: uma coleção de **chave** **objeto**
 - Toda a estrutura da JCF é baseada e descendem da estrutura destas duas interfaces
- Então como seria esta estrutura ??



Collections

- E eu preciso saber de tudo isso ????
 - SIM!!!! Para a certificação e para a vida de programador java!
- Mas... devagar com andor que o santo é de barro...
- Quais são as interfaces mesmo ??

Collections

- Interfaces da JFC

Collection	Set	SortedSet
List	Map	SortedMap
Queue	NavigableSet	NavigableMap

- Ta bagunçado né ?? Vamos olhar isso direito, começando pela Collection!

Collections

- Métodos de **java.util.Collection<E>**:

boolean add(E e)

boolean addAll(**Collection**<? extends E> c)

boolean contains(**Object** o)

boolean containsAll(**Collection**<?> c)

boolean remove(**Object** o)

boolean removeAll(**Collection**<?> c)

boolean retainAll(**Collection**<?> c)

void clear()

int size()

boolean isEmpty()

Object[] toArray()

<T> T[] toArray(T[] a)

Collections

- Métodos de **java.util.Collection<E>**:

boolean add(E e)

boolean addAll(**Collection**<? extends E> c)

boolean contains(**Object** o)

boolean containsAll(**Collection**<?> c)

boolean remove(**Object** o)

boolean removeAll(**Collection**<?> c)

boolean retainAll(**Collection**<?> c)

void clear()

int size()

boolean isEmpty()

Object[] toArray()

<T> T[] toArray(T[] a)

PERAI PERAI!!

O que é esse <E> ?

e esse add(E e) ?

e esse <? extends E> ?

e esse <?>

e aquele <T> T[] ???

Que classes são essas ?

T e E ???

Collections

- Métodos de **java.util.Collection<E>**:

boolean add(E e)

boolean addAll(Collection<? extends E> c)

boolean contains(Object o)

boolean containsAll(Collection<?> c)

boolean remove(Object o)

boolean removeAll(Collection<?> c)

boolean retainAll(Collection<?> c)

void clear()

int size()

boolean isEmpty()

Object[] toArray()

<T> T[] toArray(T[] a)

Isso se chama genéricos!
Eu vou dar uma breve
explicação apenas
introdutória, e voltamos
para as Collections

PERAI PERAI!!

O que é esse <E> ?

e esse add(E e) ?

e esse <? extends E> ?

e esse <?>

e aquele <T> T[] ???

Que classes são essas ?

T e E ???

Collections

- O que são genéricos ?
 - Os genéricos são amarrações que só existem em tempo de compilação
 - Existem classes genéricas e métodos genéricos
- Para Collection vamos focar em classes genéricas... por exemplo:

```
public class ArrayList<E> ... {  
    boolean add(E e) {  
        ...  
    }  
}
```

Collections

- O que acontece quando declaramos assim?

```
ArrayList<String> strings = new ArrayList<String>();
```

- Para o compilador, e apenas para o compilador
Todos os “T” viram “String”

```
boolean add(String e);
```

- Porém para o bytecode o “T” é sempre **Object**,
ou seja, depois que tá rodando o programa isso
some e tudo vira **Object** novamente

```
boolean add(Object e);
```

Collections

- E podemos criar a classe assim ??
`ArrayList c = new ArrayList();`
SIM!!! podemos!!! E o método gerado fica assim
`boolean add(Object e);`
- E na realidade a classe continua sendo desta forma, os tipos são apenas uma amarração para forçar a *codar* corretamente uma classe genérica.

Collections

- O problema está quando você mistura genérico e não genérico:

```
ArrayList<String> strings = new ArrayList<String>();
```

```
ArrayList c = new ArrayList();
```

- strings.add(a) só aceita **Strings** como argumento.
- c.add(a) aceita qualquer **Object**
- Por exemplo =>

Collections

- O problema esta quando você mistura genérico e não genérico:

```
ArrayList<String> strings = new ArrayList<String>();
```

```
ArrayList c = new ArrayList();
```

```
c.add(new Object()); // ok vai compilar
```

```
strings.add(new Object()); // não compila
```

```
strings.add("abc"); // ok vai compilar
```

Collections

- O problema esta quando você mistura genérico e não genérico:
`ArrayList<String> strings = new ArrayList<String>();`
`ArrayList c = new ArrayList();`
- Mas se você passar **strings** para **c**, você terá um problemão

Collections

- O problema está quando você mistura genérico e não genérico:
`ArrayList<String> strings = new ArrayList<String>();`
`ArrayList c = new ArrayList();`
- Mas se você passar **strings** para **c**, você terá um problemão
`c = strings;`
`c.add(new Object());`

Collections

- O problema esta quando você mistura genérico e não genérico:
`ArrayList<String> strings = new ArrayList<String>();`
`ArrayList c = new ArrayList();`
- Mas se você passar **strings** para **c**, você terá um problemão
`c = strings; // ok compila`
`c.add(new Object());`

Collections

- O problema está quando você mistura genérico e não genérico:
- Mas se você passar **strings** para **c**, você terá um problemão

```
ArrayList<String> strings = new ArrayList<String>();
```

```
ArrayList c = new ArrayList();
```

```
c = strings; // ok compila MAS É PERIGOSO!
```

```
c.add(new Object());
```

Collections

- O problema esta quando você mistura genérico e não genérico:
- Mas se você passar **strings** para **c**, você terá um problemão

```
ArrayList<String> strings = new ArrayList<String>();
```

```
ArrayList c = new ArrayList();
```

```
c = strings;
```

```
c.add(new Object()); // compila
```

Collections

- O problema esta quando você mistura genérico e não genérico:

```
ArrayList<String> strings = new ArrayList<String>();
```

```
ArrayList c = new ArrayList();
```

- Mas se você passar **strings** para **c**, você terá um problemão

```
c = strings;
```

```
c.add(new Object()); // compila NÃO gera Exception
```

Collections

- A Sun aconselha sempre usar os genéricos:
- Os códigos que não usam genérico quando as classes o definem compilam, mas geram *warnings* avisando que pode dar problema.

Collections

- Mas perai e aqueles <?> o que é essa “?”
 - Isso é um operador curinga ela funciona quando você quer falar que aceita uma classe com qualquer *tipagem*.
- Então:
`ArrayList<?> c = new ArrayList<String>();`
É o mesmo que ??
`ArrayList c = new ArrayList<String>();`
NÃO!! São diferentes!

Collections

- Quando você usa o operador `<?>` você não pode mandar objetos para os métodos *tipados*, por exemplo:
 `c.add(a);` // não compila
- Todos os outros métodos, que não tem o **T**, funcionam, portanto deve ser usando quando você quer usar o objeto sem passar objetos *tipados*, por exemplo:
 `c.size();` // funciona normalmente

Collections

- Tá ... e porque não compila ??
 - O motivo é porque você não sabe que tipo de coleção você está trabalhando
- Por exemplo:

```
List<?> lista;  
//eu posso adicionar qualquer lista ai  
lista = new ArrayList<String>();  
lista = new ArrayList<Integer>();  
lista = new ArrayList<Thread>();
```
- Então quando você não sabe o tipo

Collections

- Tá ... e porque não compila ??
 - O motivo é porque você não sabe que tipo de coleção você está trabalhando
- Por exemplo:
`List<?> lista = getList();`
- Você não poderá saber o que essa lista aceita:
`lista.add("Texto");` //e se for uma lista de Thread ?
- Porém você pode usar os métodos não tipados
`lista.contains("Texto");`
`Object o = lista.get(4);` //n sei o tipo da lista ∴ n sei o retorno
`lista.size();`

Collections

- Ta e aquele <? extends T> ?:
 - Ele funciona como uma sub categoria... por exemplo:
`ArrayList<Number> numbers = new ArrayList<Number>();`
- O método addAll nesse caso fica
`boolean addAll(Collection<? extends Number> c);`
- E assim é possível adicionar a *numbers* qualquer coleção tipado como Number ou por uma de suas subclasse.
`ArrayList<Integer> integers = new ArrayList<Integer>();
integers.add(12); integers.add(15);
numbers.addAll(integers); //pois os inteiros também são Number`

Collections

```
ArrayList<Number> list;
```

- Isso quer dizer que list só pode receber o tipo Number... por exemplo:

```
list = new ArrayList<Integer>(); // NÃO compila
```

```
list = new ArrayList<Number>(); // compila
```

- Para ser qualquer sub-tipo de Number é preciso fazer

```
ArrayList<? extends Number> list;
```

```
list = new ArrayList<Integer>(); //agora compila
```

- E qual o retorno para list.get(0) ??

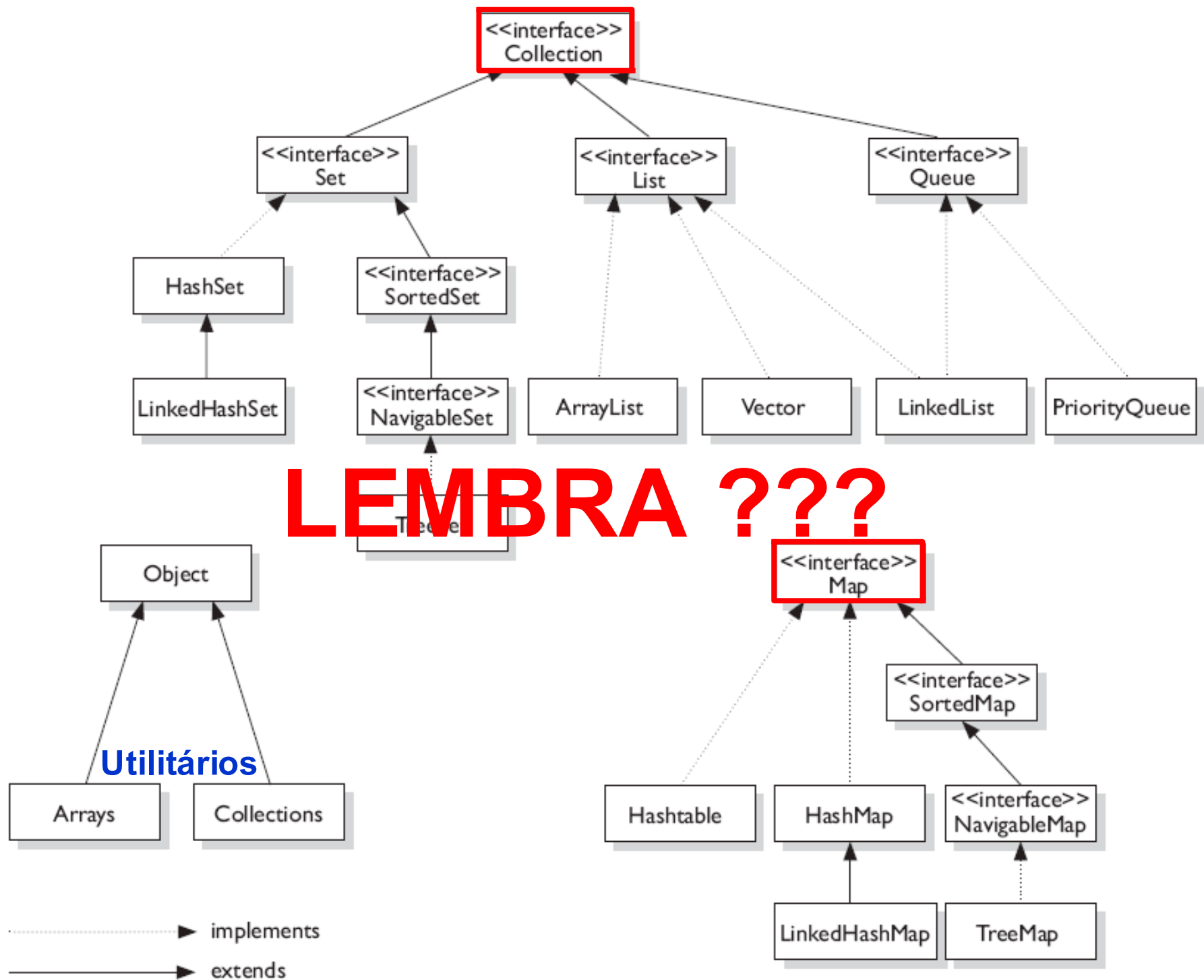
```
Number n = list.get(0); //O retorno é um Number!
```

Collections

- Ta e aquele `<T> T[] toArray(T[] a)?`:
 - Esse é um método generico, e não usa tipagem da classe mas sim do método
- Este método quer dizer que o retorno dele é uma array do tipo “T” que é o mesmo tipo da array enviada no argumento, portanto
 - `nums.toArray(new Number[]) // retorna um Number[]`
 - `nums.toArray(new Integer[])`

Collections

- Ta e aquele `<T> T[] toArray(T[] a)?`:
 - Esse é um método generico, e não usa tipagem da classe mas sim do método
- Este método quer dizer que o retorno dele é uma array do tipo “T” que é o mesmo tipo da array enviada no argumento, portanto
 - `nums.toArray(new Number[])`
 - `nums.toArray(new Integer[]) // retorna um Integer[]`



Collections

- A **java.util.Collection**:
 - Collection é um contrato de interface que representa uma coleção de objetos
 - Pode-se dizer que uma coleção é uma versão turbinada de array, com mais funcionalidades e mais segurança.

Collections

- Métodos **java.util.Collection<E> extends Iterable<E>**:

boolean add(**E** e)

boolean addAll(**Collection**<? **extends E**> c)

boolean remove(**Object** o)

boolean removeAll(**Collection**<?> c)

boolean retainAll(**Collection**<?> c)

void clear()

boolean contains(**Object** o)

boolean containsAll(**Collection**<?> c)

int size()

boolean isEmpty()

Object[] toArray()

<T> T[] toArray(T[] a)

Iterator<E> iterator() //método de Iterable

Collections

- **java.util.Collection<E> extends Iterable<E>:**

boolean add(E e)

boolean addAll(Collection<? extends E> c)

boolean remove(Object o)

boolean removeAll(Collection<?> c)

boolean retainAll(Collection<?> c)

void clear()

Opcional

boolean contains(Object o)

boolean containsAll(Collection<?> c)

int size()

boolean isEmpty()

Object[] toArray()

<T> T[] toArray(T[] a)

Iterator<E> iterator()

COMO ASSIM OPCIONAL?

A interface pede o método, como vocês podem ver, porém na implementação pode-se lançar a exceção `UnsupportedOperationException`. E isso é utilizado em implementações somente leitura, implementações boca de lobo ou outros tipos.

Collections

- **java.util.Collection<E> extends Iterable<E>:**

boolean add(E e)

boolean addAll(Collection<? extends E> c)

boolean remove(Object o)

boolean removeAll(Collection<?> c)

boolean retainAll(Collection<?> c)

void clear()

boolean contains(Object o)

boolean containsAll(Collection<?> c)

int size()

boolean isEmpty()

Object[] toArray()

<T> T[] toArray(T[] a)

Iterator<E> iterator()

Vamos conhecer os métodos!

Collections

- **java.util.Collection<E> extends Iterable<E>:**

boolean add(E e)

boolean addAll(Collection<? extends E> c)

boolean remove(Object o)

boolean removeAll(Collection<?> c)

boolean retainAll(Collection<?> c)

void clear()

boolean contains(Object o)

boolean containsAll(Collection<?> c)

int size()

boolean isEmpty()

Object[] toArray()

<T> T[] toArray(T[] a)

Iterator<E> iterator()

Adiciona um elemento a coleção, onde este elemento deve ser do mesmo tipo <E> da coleção, ou de um sub-tipo. Retorna verdadeiro se houver modificação na coleção, ou seja se o elemento for adicionado.

Implementação é opcional

Collections

- **java.util.Collection<E> extends Iterable<E>:**

boolean add(E e)

boolean addAll(Collection<? extends E> c)

boolean remove(Object o)

boolean removeAll(Collection<?> c)

boolean retainAll(Collection<?> c)

void clear()

boolean contains(Object o)

boolean containsAll(Collection<?> c)

int size()

boolean isEmpty()

Object[] toArray()

<T> T[] toArray(T[] a)

Iterator<E> iterator()

Adiciona uma coleção de elementos do mesmo tipo <E> ou uma coleção de sub-tipos de <E>.

Retorna verdadeiro se pelo menos um elemento for adicionado a coleção, ou seja, se a coleção foi modificada.

Implementação é opcional

Collections

- **java.util.Collection<E> extends Iterable<E>:**

boolean add(E e)

boolean addAll(Collection<? extends E> c)

boolean remove(Object o)

boolean removeAll(Collection<?> c)

boolean retainAll(Collection<?> c)

void clear()

boolean contains(Object o)

boolean containsAll(Collection<?> c)

int size()

boolean isEmpty()

Object[] toArray()

<T> T[] toArray(T[] a)

Iterator<E> iterator()

Remove da coleção a primeira ocorrência de um objeto significativamente igual ao enviado.

Retorna verdadeiro caso o objeto existia na coleção, ou seja, se a coleção foi modificada.

Implementação é opcional.

Collections

- **java.util.Collection<E> extends Iterable<E>:**

boolean add(E e)

boolean addAll(Collection<? extends E> c)

boolean remove(Object o)

boolean removeAll(Collection<?> c)

boolean retainAll(Collection<?> c)

void clear()

boolean contains(Object o)

boolean containsAll(Collection<?> c)

int size()

boolean isEmpty()

Object[] toArray()

<T> T[] toArray(T[] a)

Iterator<E> iterator()

Remove todas da as
ocorrências desta coleção dos
elementos contidos na coleção
c enviada.

Após esta operação nenhum
elemento desta coleção
retornará true para c.contains()

Retorna verdadeiro se pelo
menos um elemento foi
removido, ou seja, se a
coleção foi modificada.

Implementação é opcional.

Collections

- **java.util.Collection<E> extends Iterable<E>:**

boolean add(E e)

boolean addAll(Collection<? extends E> c)

boolean remove(Object o)

boolean removeAll(Collection<?> c)

boolean retainAll(**Collection**<?**>** c)

void clear()

boolean contains(Object o)

boolean containsAll(Collection<?> c)

int size()

boolean isEmpty()

Object[] toArray()

<T> T[] toArray(T[] a)

Iterator<E> iterator()

Faz o processo inverso de
removeAll.

Retém nesta coleção apenas
os elementos que também
estejam contidos na coleção c.
Após esta operação todos os
elementos desta coleção
retornarão true para
c.contains().

Retorna verdadeiro se a
coleção for modificada.

Implementação é opcional

Collections

- **java.util.Collection<E> extends Iterable<E>:**

boolean add(E e)

boolean addAll(Collection<? extends E> c)

boolean remove(Object o)

boolean removeAll(Collection<?> c)

boolean retainAll(Collection<?> c)

void clear()

boolean contains(Object o)

boolean containsAll(Collection<?> c)

int size()

boolean isEmpty()

Object[] toArray()

<T> T[] toArray(T[] a)

Iterator<E> iterator()

Remove todos os elementos
da coleção

Implementação é opcional.

Collections

- **java.util.Collection<E> extends Iterable<E>:**

boolean add(E e)

boolean addAll(Collection<? extends E> c)

boolean remove(Object o)

boolean removeAll(Collection<?> c)

boolean retainAll(Collection<?> c)

void clear()

boolean contains(Object o)

boolean containsAll(Collection<?> c)

int size()

boolean isEmpty()

Object[] toArray()

<T> T[] toArray(T[] a)

Iterator<E> iterator()

Verifica se existe nesta coleção um objeto significativamente igual ao objeto o enviado.

Collections

- **java.util.Collection<E> extends Iterable<E>:**

boolean add(E e)

boolean addAll(Collection<? extends E> c)

boolean remove(Object o)

boolean removeAll(Collection<?> c)

boolean retainAll(Collection<?> c)

void clear()

boolean contains(Object o)

boolean containsAll(Collection<?> c)

int size()

boolean isEmpty()

Object[] toArray()

<T> T[] toArray(T[] a)

Iterator<E> iterator()

Verifica se todos os objetos da coleção c enviada, estão contidos nesta coleção.

Só retorna verdade se this.contains(elementoDeC) for verdade para cada elemento de c.

Collections

- **java.util.Collection<E> extends Iterable<E>:**

boolean add(E e)

boolean addAll(Collection<? extends E> c)

boolean remove(Object o)

boolean removeAll(Collection<?> c)

boolean retainAll(Collection<?> c)

void clear()

boolean contains(Object o)

boolean containsAll(Collection<?> c)

int size()

boolean isEmpty()

Object[] toArray()

<T> T[] toArray(T[] a)

Iterator<E> iterator()

Informa a quantidade de objetos contidos na coleção.
Obs.: Se a coleção contiver objetos nulos eles também serão contabilizados

Sim! Existem coleções que aceitam elementos nulos.

Collections

- **java.util.Collection<E> extends Iterable<E>:**

boolean add(E e)

boolean addAll(Collection<? extends E> c)

boolean remove(Object o)

boolean removeAll(Collection<?> c)

boolean retainAll(Collection<?> c)

void clear()

boolean contains(Object o)

boolean containsAll(Collection<?> c)

int size()

boolean isEmpty()

Object[] toArray()

<T> T[] toArray(T[] a)

Iterator<E> iterator()

Método de comodidade, o mesmo que testar se:

size() == 0

Retorna verdadeiro se não houver elementos nesta coleção.

Collections

- **java.util.Collection<E> extends Iterable<E>:**

boolean add(E e)

boolean addAll(Collection<? extends E> c)

boolean remove(Object o)

boolean removeAll(Collection<?> c)

boolean retainAll(Collection<?> c)

void clear()

boolean contains(Object o)

boolean containsAll(Collection<?> c)

int size()

boolean isEmpty()

Object[] toArray()

<T> T[] toArray(T[] a)

Iterator<E> iterator()

Retorna uma array contendo cada um dos elementos desta coleção, na mesma ordem em que os elementos aparecem no iterator().

Não é a array mantida pela lista (caso assim seja implementada) é uma cópia dela.

Alterações nesta array não são refletidas na coleção.

Collections

- **java.util.Collection<E> extends Iterable<E>:**

```
boolean add(E e)
boolean addAll(Collection<? extends E> c)
boolean remove(Object o)
boolean removeAll(Collection<?> c)
boolean retainAll(Collection<?> c)
void clear()
boolean contains(Object o)
boolean containsAll(Collection<?> c)
int size()
boolean isEmpty()
Object[] toArray()
<T> T[] toArray(T[] a)
Iterator<E> iterator()
```

Retorna todos os elementos da coleção em uma array do mesmo tipo da enviada.

```
if (a.length >= this.size()) {
    Então os elementos serão colocados dentro da própria array enviada e os elementos que sobrarem serão setados null.
} else {
    cria uma nova array do mesmo tipo da enviada e a retorna.
```

Collections

- **java.util.Collection<E> extends Iterable<E>:**

```
boolean add(E e)
boolean addAll(Collection<? extends E> c)
boolean remove(Object o)
boolean removeAll(Collection<?> c)
boolean retainAll(Collection<?> c)
void clear()
boolean contains(Object o)
boolean containsAll(Collection<?> c)
int size()
boolean isEmpty()
Object[] toArray()
<T> T[] toArray(T[] a)
Iterator<E> iterator() /**
```

Cria um **Iterator** que é usado para navegar sobre os elementos desta coleção

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove(); //opcional
}
```

* Este método é parte da interface **Iterable<E>** e toda classe que a implementa pode ser usada em um for-each

Collections

- **java.util.Collection<E>:**
 - Não há restrição, ordem ou classificação definida no contrato de **Collection**
 - A coleção de objetos pode conter qualquer tipo de objeto, em qualquer quantidade, sejam eles repetidos ou não, sem qualquer ordem ou classificação definida.
 - Esta é a forma mais genérica de agrupar objetos

Collections

- Tá! e se eu quiser uma coleção sem deixar que os objetos se repitam ??
 - Então você quer um **java.util.Set** uma sub-interface de Collection

Collections

- `java.util.Set<E>` `extends` `Collection<E>`:
 - Um **Set** tem exatamente a mesma interface de **Collection**, a única mudança é na descrição de seus métodos.
 - O **Set** não aceita elementos repetidos
 - Ao usar `add(obj)` onde o **Set** já contém `obj`, ele simplesmente não o adiciona e retorna `false`.
 - A não implementação de `equals(Object o)` ou a implementação incorreta pode causar efeitos indesejáveis e bug de difícil detecção.

Collections

- `java.util.Set<E>` extends `Collection<E>`:
 - Assim como a `Collection`, não há restrição, ordem ou classificação definida no contrato de `Set`
 - A coleção de objetos pode conter qualquer tipo de objeto, em qualquer quantidade, sem qualquer ordem ou classificação definida, porem **nunca objetos repetidos!**
 - Este é o diferencial de um `Set` para uma `Collection`, **não há objetos significantemente repetidos**, isso quer dizer onde `equals(other)` retorne `true` para outro elemento dentro do mesmo `Set`.

Collections

- É possível também utilizar os conceitos de fila com a Java Collections Framework
 - Através da interface `java.util.Queue<E>`

Collections

- **java.util.Queue<E> extends Collection<E>:**
 - Assim como a **Collection**, não há restrição, ordem ou classificação definida no contrato da fila **Queue**
 - **Há apenas disponibilizado uma interface de fila,** sem especificar quais condições.
 - As suas implementações que vão definir se a **Queue** é uma FIFO, FILO ou o quer que seja.

Collections

- **java.util.Queue<E> extends Collection<E>:**
 - boolean** add(**E** e) //adiciona um elemento a fila se houver capacidade,
// se não houver lança uma IllegalStateException.
 - boolean** offer(**E** e) //adiciona um elemento a fila se houver capacidade,
// se não houver retorna false (sem lança exceção).
 - E** element() //retorna, mas não remove o elemento do topo da fila
//não havendo + elementos lança NoSuchElementException.
 - E** peek() //retorna, mas não remove o elemento do topo da fila
//retorna null se não houver elementos.
 - E** remove() //retorna e remove o elemento do topo da fila
//não havendo + elementos lança NoSuchElementException.
 - E** poll() //retorna e remove o elemento do topo da fila
//retorna null se não houver elementos.

Collections

- Mas e como eu consigo verificar o elemento 4 que esta dentro de uma **Collection**, **Queue** ou **Set**? só da usando o `iterator()` ?
 - Sim infelizmente sim, a única forma de verificar os elementos destas interfaces é percorrendo a coleção toda!
- Putz! E aquele papo de que era uma array turbinada?? Não tem índice ? cadê o `get(3)` ?
 - Para tudo na vida tem uma solução, o que você quer é uma **`java.util.List<E>`**

Collections

- **java.util.List<E> extends Collection<E>:**
 - Uma **List** é uma coleção ordenada (não classificada), muito parecida com arrays, porém com bem mais funcionalidades e sem limite de tamanho.
 - Os métodos `add()` e `addAll()` da interface **Collection** adicionam itens ao final da **List**.
 - Assim como as **Collection** todos os métodos de adição e remoção são opcionais e podem lançar **UnsupportedOperationException**
 - As mudanças principais em relação a **Collection** é que tem seus itens ordenados. Os itens da **List** são ordenados em índice que vão de **0** a `(size() - 1)`, conforme veremos a seguir, e usar índice fora desse range gera **IndexOutOfBoundsException**.

Collections

- **java.util.List<E> extends Collection<E>:**

boolean add(int i, E e)

boolean addAll(int i, Collection<? extends E> c)

E get(int i)

int indexOf(Object o)

int lastIndexOf(Object o)

ListIterator<E> listIterator()

ListIterator<E> listIterator(int i)

E remove(int i)

E set(int index, E element)

List<E> subList(int fromIndex, int toIndex)

Collections

- **java.util.List<E>** extends **Collection<E>**:

boolean add(**int** i, **E** e)

boolean addAll(**int** i, Collection<? extends E> c)

E get(**int** i)

int indexOf(Object o)

int lastIndexOf(Object o)

ListIterator<E> listIterator()

ListIterator<E> listIterator(**int** i)

E remove(**int** i)

E set(**int** index, **E** element)

List<E> subList(**int** fromIndex, **int** toIndex)

Adiciona o elemento **e** na posição de índice **i**. Se houver, o antigo elemento do índice e seus posteriores terão seus índices incrementados

Collections

- **java.util.List<E> extends Collection<E>:**

boolean add(int i, E e)

boolean addAll(int i, Collection<? extends E> c)

E get(int i)

int indexOf(Object o)

int lastIndexOf(Object o)

ListIterator<E> listIterator()

ListIterator<E> listIterator(int i)

E remove(int i)

E set(int index, E element)

List<E> subList(int fromIndex, int toIndex)

Adiciona os elementos de **c** na posição de índice **i**. Se houver, o antigo elemento do índice e seus posteriores terão seus índices incrementados em **c.size()**

Collections

- **java.util.List<E>** extends **Collection<E>**:

boolean add(int i, E e)

boolean addAll(int i, Collection<? extends E> c)

E get(int i)

int indexOf(Object o)

int lastIndexOf(Object o)

ListIterator<E> listIterator()

ListIterator<E> listIterator(int i)

E remove(int i)

E set(int index, E element)

List<E> subList(int fromIndex, int toIndex)

Retorna o elemento do índice **i**

Collections

- **java.util.List<E>** extends **Collection<E>**:

boolean add(int i, E e)

boolean addAll(int i, Collection<? extends E> c)

E get(int i)

int indexOf(Object o)

int lastIndexOf(Object o)

ListIterator<E> listIterator()

ListIterator<E> listIterator(int i)

E remove(int i)

E set(int index, E element)

List<E> subList(int fromIndex, int toIndex)

Retorna o índice do primeiro objeto da list igual ao enviado ou **-1** caso não exista o objeto na coleção.

Collections

- **java.util.List<E> extends Collection<E>:**

boolean add(int i, E e)

boolean addAll(int i, Collection<? extends E> c)

E get(int i)

int indexOf(Object o)

int lastIndexOf(**Object** o)

ListIterator<E> listIterator()

ListIterator<E> listIterator(int i)

E remove(int i)

E set(int index, E element)

List<E> subList(int fromIndex, int toIndex)

Retorna o índice do ultimo objeto da list igual ao enviado ou **-1** caso não exista o objeto na coleção.

Collections

- **java.util.List<E>** extends **Collection<E>**:

boolean add(int i, E e)

boolean addAll(int i, Collection<? extends E> c)

E get(int i)

int indexOf(Object o)

int lastIndexOf(Object o)

ListIterator<E> listIterator()

ListIterator<E> listIterator(int i)

E remove(int i)

E set(int index, E element)

List<E> subList(int fromIndex, int toIndex)

Retorna um **ListIterator** desta **List**. Este objeto é uma extensão do **Iterator**, com a diferença que você pode caminhar p/ frente ou p/ traz na lista, além de adicionar e alterar elementos no índice corrente.

Collections

- **java.util.List<E>** extends **Collection<E>**:

```
public interface ListIterator<E>  
    extends Iterator<E> {
```

```
    boolean hasNext()  
    E next()  
    int nextIndex()  
    boolean hasPrevious()  
    E previous()  
    int previousIndex()  
    void add(E e)  
    void set(E e)  
    void remove()
```

```
}
```

Retorna um **ListIterator** desta **List**.
Este objeto é uma extensão do **Iterator**, com a diferença que você pode caminhar p/ frente ou p/ traz na lista, além de adicionar e alterar elementos no índice corrente.

Collections

- **java.util.List<E>** extends **Collection<E>**:

```
public interface ListIterator<E>  
    extends Iterator<E> {
```

```
    boolean hasNext()  
    E next()  
    int nextIndex()  
    boolean hasPrevious()  
    E previous()  
    int previousIndex()  
    void add(E e)  
    void set(E e)  
    void remove()
```

```
}
```

Verifica se há um próximo elemento na **List**.

Collections

- **java.util.List<E>** extends **Collection<E>**:

```
public interface ListIterator<E>  
    extends Iterator<E> {
```

```
    boolean hasNext()  
    E next()  
    int nextIndex()  
    boolean hasPrevious()  
    E previous()  
    int previousIndex()  
    void add(E e)  
    void set(E e)  
    void remove()
```

```
}
```

Retorna o próximo elemento da **List**.

Se não houver o próximo elemento uma **NoSuchElementException** é lançada.

Collections

- **java.util.List<E>** extends **Collection<E>**:

```
public interface ListIterator<E>  
    extends Iterator<E> {
```

```
    boolean hasNext()  
    E next()  
    int nextIndex()  
    boolean hasPrevious()  
    E previous()  
    int previousIndex()  
    void add(E e)  
    void set(E e)  
    void remove()
```

```
}
```

Retorna o índice do próximo elemento da **List**.

Se não houver um próximo elemento retorna o tamanho da lista, ou seja, o mesmo que `lista.size()`.

Collections

- **java.util.List<E>** extends **Collection<E>**:

```
public interface ListIterator<E>  
    extends Iterator<E> {
```

```
    boolean hasNext()  
    E next()  
    int nextIndex()  
    boolean hasPrevious()  
    E previous()  
    int previousIndex()  
    void add(E e)  
    void set(E e)  
    void remove()
```

```
}
```

Verifica se há um elemento anterior na **List**.

Collections

- **java.util.List<E>** extends **Collection<E>**:

```
public interface ListIterator<E>  
    extends Iterator<E> {
```

```
    boolean hasNext()  
    E next()  
    int nextIndex()  
    boolean hasPrevious()  
    E previous()  
    int previousIndex()  
    void add(E e)  
    void set(E e)  
    void remove()
```

```
}
```

Retorna o elemento anterior da **List**.

Se não houver o elemento anterior uma **NoSuchElementException** é lançada.

Collections

- **java.util.List<E>** extends **Collection<E>**:

```
public interface ListIterator<E>  
    extends Iterator<E> {
```

```
    boolean hasNext()  
    E next()  
    int nextIndex()  
    boolean hasPrevious()  
    E previous()  
    int previousIndex()  
    void add(E e)  
    void set(E e)  
    void remove()
```

```
}
```

Retorna o índice do elemento anterior da **List**.

Se não houver um elemento anterior retorna **-1**.

Collections

- **java.util.List<E>** extends **Collection<E>**:

```
public interface ListIterator<E>  
    extends Iterator<E> {
```

```
    boolean hasNext()  
    E next()  
    int nextIndex()  
    boolean hasPrevious()  
    E previous()  
    int previousIndex()  
    void add(E e)  
    void set(E e)  
    void remove()
```

```
}
```

Adiciona o elemento **e** a **List** no índice atual, entre `nextIndex()` e `previousIndex()`.

Collections

- **java.util.List<E>** extends **Collection<E>**:

```
public interface ListIterator<E>  
    extends Iterator<E> {
```

```
    boolean hasNext()  
    E next()  
    int nextIndex()  
    boolean hasPrevious()  
    E previous()  
    int previousIndex()  
    void add(E e)  
    void set(E e)  
    void remove()
```

```
}
```

Substitui o último elemento
retornado por *next* ou *previous*.

Se anteriormente *next* / *previous*
não foi chamado ou *add* / *remove*
foi invocado será lançada uma
IllegalStateException

Collections

- **java.util.List<E>** extends **Collection<E>**:

```
public interface ListIterator<E>  
    extends Iterator<E> {
```

```
    boolean hasNext()  
    E next()  
    int nextIndex()  
    boolean hasPrevious()  
    E previous()  
    int previousIndex()  
    void add(E e)  
    void set(E e)  
    void remove()
```

```
}
```

Remove o último elemento
retornado por *next* ou *previous*.

Se anteriormente *next* / *previous*
não foi chamado ou *add* / *remove*
foi invocado será lançada uma
IllegalStateException

Collections

- **java.util.List<E> extends Collection<E>:**

boolean add(int i, E e)

boolean addAll(int i, Collection<? extends E> c)

E get(int i)

int indexOf(Object o)

int lastIndexOf(Object o)

ListIterator<E> listIterator()

ListIterator<E> listIterator(int i)

E remove(int i)

E set(int index, E element)

List<E> subList(int fromIndex, int toIndex)

Igual ao método anterior com a diferença que o índice corrente do **ListIterator** será o índice passado.

Obs.: é possível retornar para antes do índice passado.

Collections

- **java.util.List<E>** extends **Collection<E>**:

boolean add(int i, E e)

boolean addAll(int i, Collection<? extends E> c)

E get(int i)

int indexOf(Object o)

int lastIndexOf(Object o)

ListIterator<E> listIterator()

ListIterator<E> listIterator(int i)

E remove(int i)

E set(int index, E element)

List<E> subList(int fromIndex, int toIndex)

Remove o elemento de índice **i** da **List** e o retorna. **Importante!!!**

Integer indexA = **1**; **int** indexB = **1**;

list.remove(indexA)

list.remove(indexB)

são operações totalmente diferentes

O 1º remove um objeto **Integer == 1**

O 2º remove o objeto no índice **1**

Collections

- **java.util.List<E>** extends **Collection<E>**:

boolean add(int i, E e)

boolean addAll(int i, Collection<? extends E> c)

E get(int i)

int indexOf(Object o)

int lastIndexOf(Object o)

ListIterator<E> listIterator()

ListIterator<E> listIterator(int i)

E remove(int i)

E set(int index, **E** element)

List<E> subList(int fromIndex, int toIndex)

Substitui o objeto no índice **index** pelo **element** enviado e retorna o objeto substituído.

Collections

- **java.util.List<E>** extends **Collection<E>**:

boolean add(int i, E e)

boolean addAll(int i, Collection<? extends E> c)

E get(int i)

int indexOf(Object o)

int lastIndexOf(Object o)

ListIterator<E> listIterator()

ListIterator<E> listIterator(int i)

E remove(int i)

E set(int index, E element)

List<E> subList(int fromIndex, int toIndex)

Retorna o elemento do índice **fromIndex** (inclusive) até os elementos de índice **toIndex** (exclusive). Mudança na *subList* são refletidas na **List** original, o contrario impossibilita o uso da *subList*, que, se usada, lançará **ConcurrentModificationException**

Collections

- Ok, mas existe alguma forma de criar Coleções classificadas ?
 - **Sim!!**
- Mas antes de vermos como manter listas classificadas precisamos conhecer duas interfaces **Comparator<T>** e **Comparable<T>**
- Estas duas interfaces indicam a ordem de classificação dos objetos

Collections

- **Comparable<T>** é uma interface que só tem um método:

int compareTo(**T** o)

- Retorna negativo, zero, ou positivo quando este objeto é menor que, igual que ou maior que, respectivamente.

- Objetos que tem ordem natural implementam esta interface... por exemplo:

```
String texto1 = "dado", texto2 = "morta";  
if (texto1.compareTo(texto2) < 0) {  
    System.out.println("texto1 vem 1º"); // imprime este  
} else if (texto1.compareTo(texto2) > 0) {  
    System.out.println("texto1 vem 2º");  
}
```

Collections

- Alguma vez precisamos comparar objetos que não tem ordem natural, ou então comparar objetos por propriedades diferentes da ordem natural
- Neste caso utilizamos **Comparator<T>**
`int compare(T o1, T o2)`
- Um **Comparator** funciona como um objeto ordenador, que recebe dois objetos, e **compara o primeiro com o segundo (nesta ordem)** de acordo com os critérios definidos no **Comparator**.

Collections

- Há! E antes de vermos como funcionam as listas ordenadas precisamos conhecer mais uma forma de usar os genéricos **<? super E>**
- Enquanto que **<? extends E>** não permite o uso dos parâmetros *tipados* com **E**, o **<? super E>** permite o uso, e o parâmetro deve ser da classe **E** ou de uma Sub Classe de **E**, sim! De uma Sub classe... Por exemplo:

```
List<? super Number> numbers;  
numbers = new ArrayList<Number>(); // ok compila!  
numbers = new ArrayList<Integer>(); // NÃO compila!  
numbers = new ArrayList<Object>(); // ok compila!  
numbers.add(1); //numbers aceita qualquer sub classe de Number  
numbers.add(2.33); //ok também aceita
```
- Como a **List** é do tipo **<Number>** ou de uma Super classe de **<Number>** (no caso **Object**), é garantido que dentro da lista vai caber qualquer classe **Number** ou sub-classe de **Number** na lista.

Collections

- Agora vamos lá! Vamos conhecer como funciona as classes classificadas
java.util.SortedSet<E> extends Set<E>
- Esta interface é de uma coleção classificada, onde a classificação do **Comparator** usado por ela, ou a ordem natural é mantida sempre
- Não importa a ordem em que os elementos são adicionados a coleção, os itens dentro da coleção serão sempre mantidas na classificação definida por seu **Comparator** ou pela ordem natural.

Collections

- **java.util.SortedSet<E>** extends **Set<E>**:
 - Comparator**<? **super E**> **comparator()**
 - E** **first()**
 - SortedSet**<**E**> **tailSet**(**E** **fromElement**)
 - E** **last()**
 - SortedSet**<**E**> **headSet**(**E** **toElement**)
 - SortedSet**<**E**> **subSet**(**E** **fromElement**, **E** **toElement**)

Collections

- **java.util.SortedSet<E> extends Set<E>:**

Comparator<? **super E**> comparator()

E first()

SortedSet<E> tailSet(E fromElement)

E last()

SortedSet<E> headSet(E toElement)

SortedSet<E> subSet(E fromElement, E toElement)

Se a ordem usada na **List** classificada for a natural retorna **null**, caso contrario retorna o **Comparator** que define a ordem usada na classificação. O **Comparator** é obrigatoriamente do mesmo tipo de **E** ou de um super tipo de **E**.

Obs.: sendo o **Comparator**<? **super E**> seu método compare(**E** o1, **E** o2) vai aceitar objetos do tipo **E** ou de sub-classe, assim mantemos a integridade.

Collections

- **java.util.SortedSet<E>** extends **Set<E>**:

Comparator<? super E> comparator()

E first()

SortedSet<E> tailSet(E fromElement)

E last()

SortedSet<E> headSet(E toElement)

SortedSet<E> subSet(E fromElement, E toElement)

Retorna o primeiro elemento, de acordo com a classificação desse

SortedSet.

Não importando a ordem em que os elementos são colocados na coleção, será retornado o primeiro elemento de acordo com a classificação da

SortedSet.

Collections

- **java.util.SortedSet<E>** extends **Set<E>**:

Comparator<? super E> comparator()

E first()

SortedSet<E> tailSet(**E** fromElement)

E last()

SortedSet<E> headSet(E toElement)

SortedSet<E> subSet(E fromElement, E toElement)

Define o *tail* – “rabo” – de uma sub **SortedSet** que inicia em *fromElement* (**inclusive**) incluindo todos os elementos maiores que *fromElement*

- A sub coleção criada é um reflexo da coleção principal, e quaisquer operações em uma coleção será refletida na outra, e as duas coleções podem ser alteradas sem problemas.

- A sub coleção não pode receber nenhum elemento menor que *fromElement*, gerando um **IllegalArgumentException**: key out of range

Collections

- **java.util.SortedSet<E>** extends **Set<E>**:

Comparator<? super E> comparator()

E first()

SortedSet<E> tailSet(E fromElement)

E last()

SortedSet<E> headSet(E toElement)

SortedSet<E> subSet(E fromElement, E toElement)

Retorna o ultimo elemento, de acordo com a classificação desse **SortedSet**.

Collections

- **java.util.SortedSet<E>** extends **Set<E>**:

Comparator<? super E> comparator()

E first()

SortedSet<E> tailSet(E fromElement)

E last()

SortedSet<E> headSet(E toElement)

SortedSet<E> subSet(E fromElement, E toElement)

Define a *head* – “cabeça” – de uma sub **SortedSet** que termina em *toElement* (**exclusive**) incluindo todos os elementos menores a *toElement*

- A sub coleção criada é um reflexo da coleção principal, e quaisquer operações em uma coleção será refletida na outra, e as duas coleções podem ser alteradas sem problemas..

- A sub coleção não pode receber nenhum elemento maior ou igual que *toElement*, gerando um **IllegalArgumentException**: key out of range

Collections

- **java.util.SortedSet<E>** extends **Set<E>**:

Comparator<? super E> comparator()

E first()

SortedSet<E> tailSet(E fromElement)

E last()

SortedSet<E> headSet(E toElement)

SortedSet<E> subSet(**E** fromElement, **E** toElement)

Cria uma sub **SortedSet** com os elementos da coleção original iniciando em fromElement (**inclusive**) e terminando em toElement (**exclusive**).

- A sub coleção criada é um reflexo da coleção principal, e quaisquer operações em uma coleção será refletida na outra, e as duas coleções podem ser alteradas sem problemas.

- A nova coleção não pode receber nenhum elemento < *fromElement* ou >= *toElement*, gerando um **IllegalArgumentException**: key out of range

Collections

- Bem legal, mas e se eu quiser encontrar o primeiro objeto da coleção maior a *element*, ou um objeto menor a *element*? Entre outras informações mais detalhas?
 - Nesse caso você precisa de uma `java.util.NavigableSet<E>` **extends** `SortedSet<E>`

Collections

- **java.util.NavigableSet<E>** extends **SortedSet<E>**:
 - E** lower(**E** e) //primeiro elemento menor a **e**
 - E** floor(**E** e) //primeiro elemento menor ou igual a **e**
 - E** higher(**E** e) //primeiro elemento maior a **e**
 - E** ceiling(**E** e) //primeiro elemento maior ou igual a **e**
 - E** pollFirst() //remove e retorna o primeiro elemento ou *null*
 - E** pollLast() //remove e retorna o ultimo elemento ou *null*
 - Iterator<E>** descendingIterator() //Iterator na ordem inversa
 - NavigableSet<E>** descendingSet() //NavigableSet inverso
 - NavigableSet<E>** headSet(**E** toElement, **boolean** inclusive)
 - NavigableSet<E>** tailSet(**E** fromElement, **boolean** inclusive)
 - NavigableSet<E>** subSet(**E** from, **boolean** inc, **E** to, **boolean** inc)

Collections

- **java.util.NavigableSet<E>** extends **SortedSet<E>**:

E lower(**E** e) //primeiro elemento menor a **e**

E floor(**E** e) //primeiro elemento menor ou igual a **e**

E higher(**E** e) //primeiro elemento maior a **e**

E ceiling(**E** e) //primeiro elemento maior ou igual a **e**

E pollFirst() //remove e retorna o primeiro elemento

E pollLast() //remove e retorna o ultimo elemento

Iterator<**E**> descendingIterator() //Iterator na ordem decrescente

NavigableSet<**E**> descendingSet() //NavigableSet na ordem decrescente

NavigableSet<**E**> headSet(**E** toElement, **boolean** inclusive)

NavigableSet<**E**> tailSet(**E** fromElement, **boolean** inclusive)

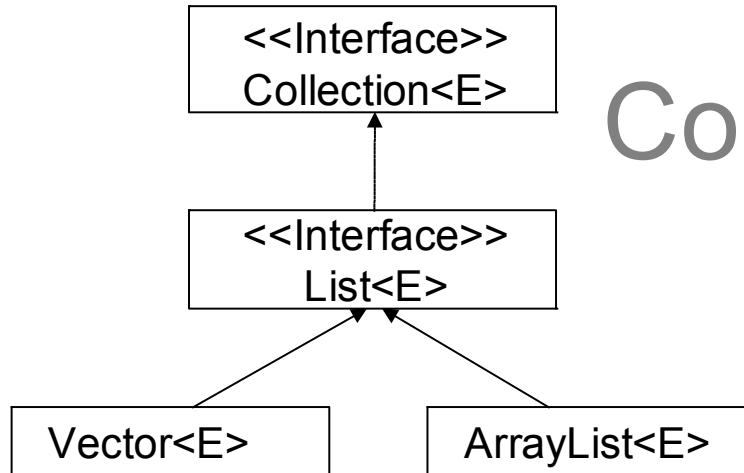
NavigableSet<**E**> subSet(**E** from, **boolean** inc, **E** to, **boolean** inc)

Igual aos métodos da classe **SortedSet** porém aqui você pode escolher se o elemento enviado é inclusive ou exclusive.

Collections

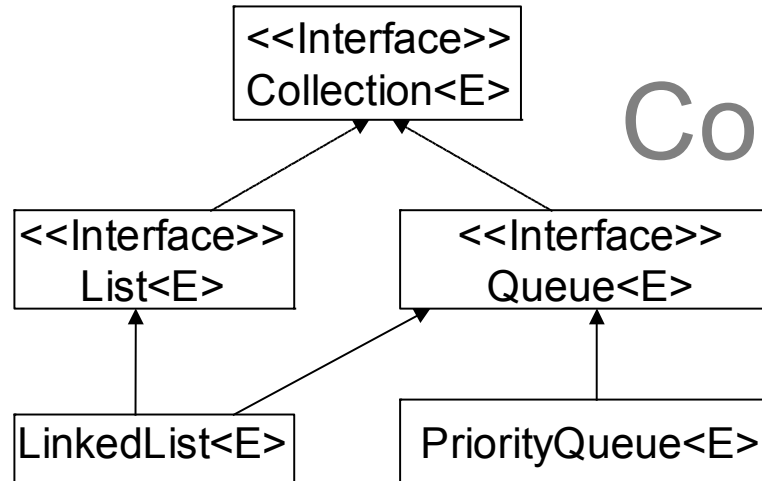
- Existem mais interfaces de Collection ?
 - Que descendam de Collection? nenhuma que vá cair na prova!
Mas é bastante vasta a Java Collection Framework
- Sobre interfaces só falta **Map<E>** e suas sub-interfaces, mas este nós só vamos ver depois das classes Concretas de **Collection**.
- Então vamos as classes concretas

Collections



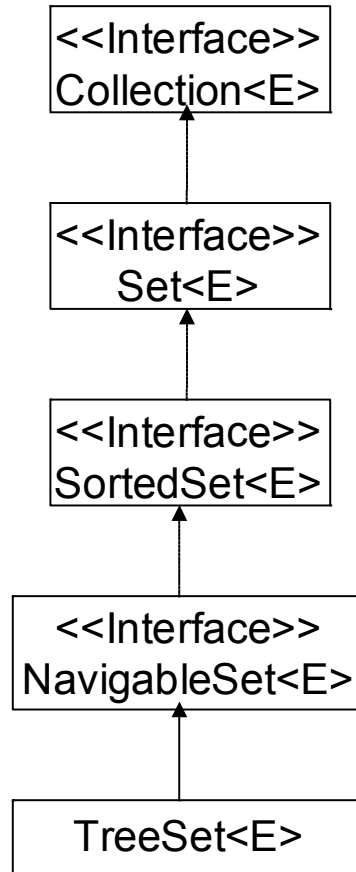
- Para o exame você só precisa saber que as duas classes tem a mesma implementação, exceto pela sincronização:
 - **Vector**: tem todos os seus métodos sincronizados
 - **ArrayList**: NÃO tem os métodos sincronizados.
- Só ter os métodos sincronizados não garante a classe que ela é Thread safe, por isso muito cuidado com as perguntas da prova, e é aconselhável o uso de **ArrayList** pois **Vector** é mais lenta devido a sync

Collections



- **LinkedList**: Implementa **List** e **Queue**, portanto tem todas funcionalidades de **List** além da Fila, onde o modelo de fila implementado é FIFO (first-in-first-out).
- **PriorityQueue**: Essa é uma das poucas que apenas a implementação não dá todas as pistas, **PriorityQueue** é uma fila classificada, ou seja, respeita a classificação natural dos elementos ou de um **Comparator**.
 - `priorityQueue.poll()` retorna sempre o menor elemento (de acordo com o **Comparator** da fila ou a ordem natural).
 - **A PriorityQueue aceita valores duplicados**
 - Não usar **Comparator** com objetos não classificáveis gera exceção ao tentar adicionar elementos.

Collections



- **TreeSet**: uma das classes mais interessante e que mais cai na prova:
 - É uma coleção de **elementos únicos em relação a classificação definida.**
 - Deve-se usar **Comparator**, quando não se quer usar, ou não existe, uma classificação natural.
 - Usar a classe sem **Comparator** com um objeto sem classificação natural gera exceção ao adicionar o elemento.
 - Se a comparação retornar igual para dois elementos, e eles não forem iguais, o **TreeSet** não deixará incluir os dois elementos, será considerado para o **Set** como elementos duplicados, o que não é permitido, e **false** será retornado.
 - Estudem os métodos subSet, tailSet, headSet

Collections

- Bom chegou a hora, vamos para as classes hash, e portanto precisamos explicar pra que serve o código hash e como as classes se comportam.
- Antes de verificar o comportamento de uma coleção hash, precisamos ver como as demais se comportam.
 - Sempre que você dá um `contains(Object o)`, a coleção começa a verificar seus objetos testando um a um se `o.equals(next)` até encontrar o objeto, retornando se encontrou ou não.
 - Quando o método usado é `remove(Object o)`, o processo é o mesmo a coleção testa `equals` um a um até encontrar um objeto igual e o remove.
 - Se a coleção for um Set, a cada `add(Object o)`, antes de adicionar a coleção verifica se `contains(Object o)`, daí você começa a entender o problema e as implicações de ficar percorrendo a coleção tantas vezes testando tantos `equals`.

Collections

- Para resolver esse problema de performance que vai se acumulando as coleções hash se usam de outro conceito para fazer busca de seus elementos.
- Primeiro vamos revisar duas das propriedades do hashCode:
 - Se dois objetos a e b forem iguais e `a.equals(b)` for true então `a.hashCode() == b.hashCode()`
 - O inverso não é verdadeiro, quando `a.equals(b)` for false os `hashCode()` podem ser iguais ou diferentes.
- As coleções hash se utilizam desses dois conceitos para otimizar a pesquisa a seus elementos.

Collections

- Quando você adiciona um elemento a uma coleção hash ela verifica o hashCode do objeto, e o coloca em um recipiente onde só há objetos com o mesmo hashCode, se o recipiente para aquele hashCode ainda não existia ela cria um novo.
 - Assim é mantido o processo sempre que se adiciona um objeto.
- Quando você, por exemplo, precisa verificar se um objeto existe, as coleções hash tem muito menos trabalho para procurar:
 - primeiro elas verificam o código hash do objeto, e olha no recipiente daquele código
 - E então verifica um a um, apenas nesse recipiente, através de equals() para ver se há algum objeto que coincidente.

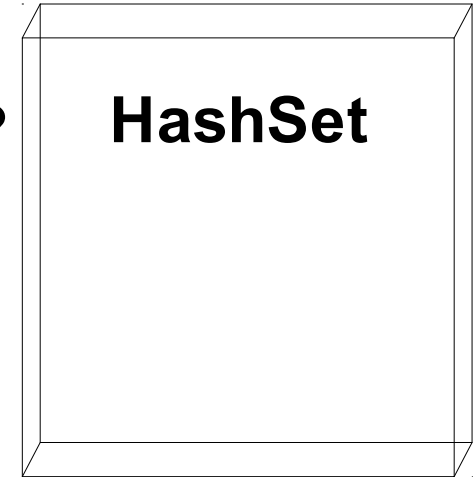
Collections

- Portanto a performance e o funcionamento de uma coleção hash esta intimamente ligada ao quão bem implementado é um código hash.
- Para um melhor entendimento vamos criar uma classe Aluno:

```
public class Aluno {  
    private String nome;  
    public Aluno(String nome) { this.nome = nome;}  
    public String getNome() { return this.nome;}  
    @Override public boolean equals(Object o) {  
        if (nome == null) return false;  
        return (o instanceof Aluno) && nome.equals(((Aluno)o).nome);  
    }  
    @Override public int hashCode() { //hashCode vem da 1º letra do nome  
        return (nome == null) ? 0 : nome.charAt(0);  
    }  
}
```

Collections

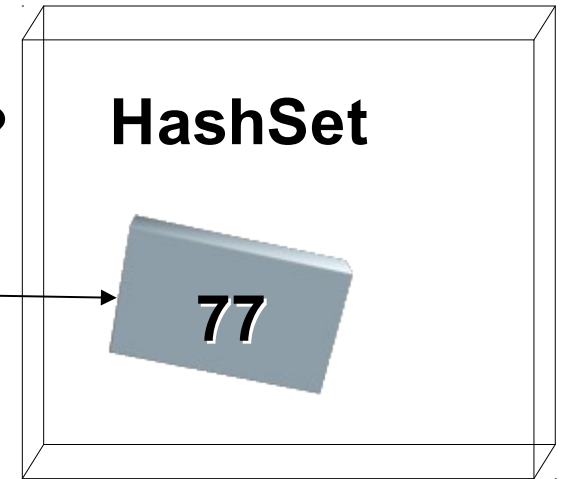
- Como é a divisão em uma coleção Hash?
`Set<Aluno> alunos = new HashSet<Aluno>();`



Collections

- Como é a divisão em uma coleção Hash?

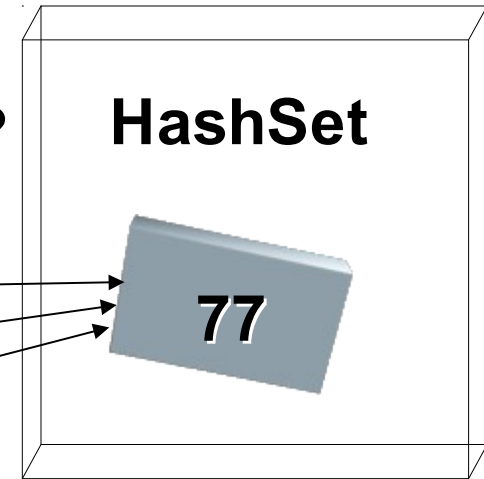
```
Set<Aluno> alunos = new HashSet<Aluno>();  
alunos.add(new Aluno("Marcos"));  
alunos.add(new Aluno("Matheus"));  
alunos.add(new Aluno("Magno"));
```



Collections

- Como é a divisão em uma coleção Hash?

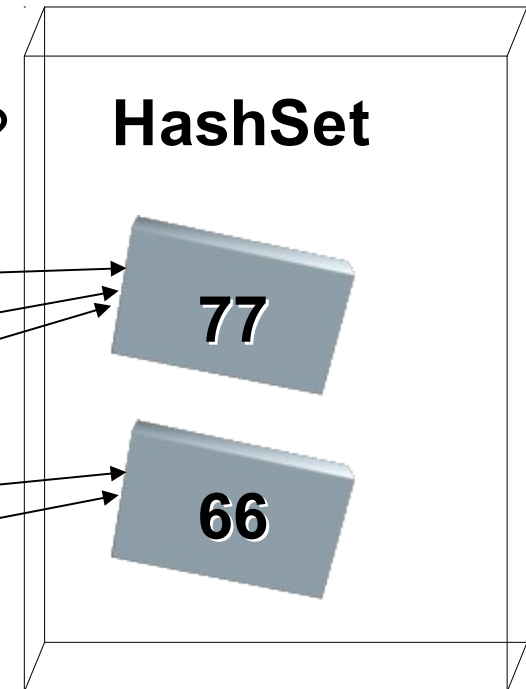
```
Set<Aluno> alunos = new HashSet<Aluno>();  
alunos.add(new Aluno("Marcos"));  
alunos.add(new Aluno("Matheus"));  
alunos.add(new Aluno("Magno"));
```



Collections

- Como é a divisão em uma coleção Hash?

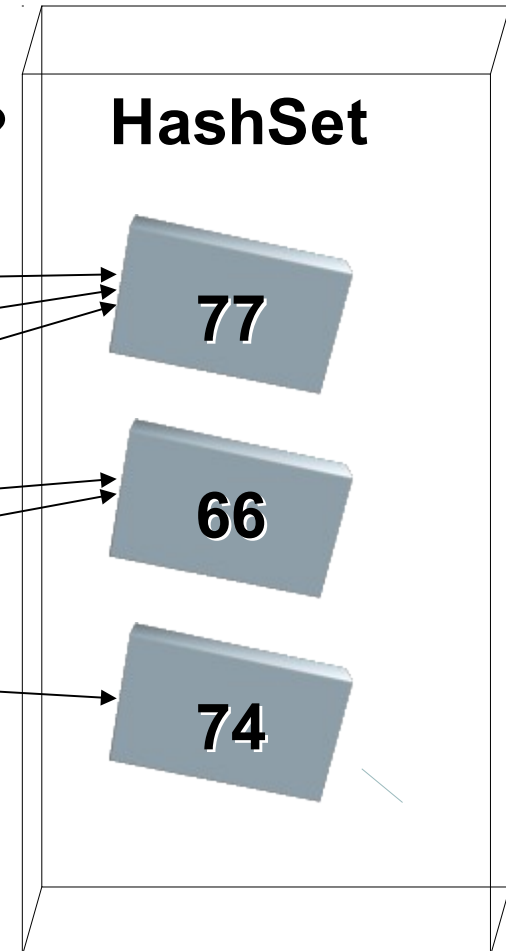
```
Set<Aluno> alunos = new HashSet<Aluno>();  
alunos.add(new Aluno("Marcos"));  
alunos.add(new Aluno("Matheus"));  
alunos.add(new Aluno("Magno"));  
alunos.add(new Aluno("Antonio"));  
alunos.add(new Aluno("Ana"));
```



Collections

- Como é a divisão em uma coleção Hash?

```
Set<Aluno> alunos = new HashSet<Aluno>();  
alunos.add(new Aluno("Marcos"));  
alunos.add(new Aluno("Matheus"));  
alunos.add(new Aluno("Magno"));  
alunos.add(new Aluno("Antonio"));  
alunos.add(new Aluno("Ana"));  
alunos.add(new Aluno("João"));
```



Collections

- Como é a divisão em uma coleção Hash?

```
Set<Aluno> alunos = new HashSet<Aluno>();
```

```
alunos.add(new Aluno("Marcos"));
```

```
alunos.add(new Aluno("Matheus"));
```

```
alunos.add(new Aluno("Magno"));
```

```
alunos.add(new Aluno("Antonio"));
```

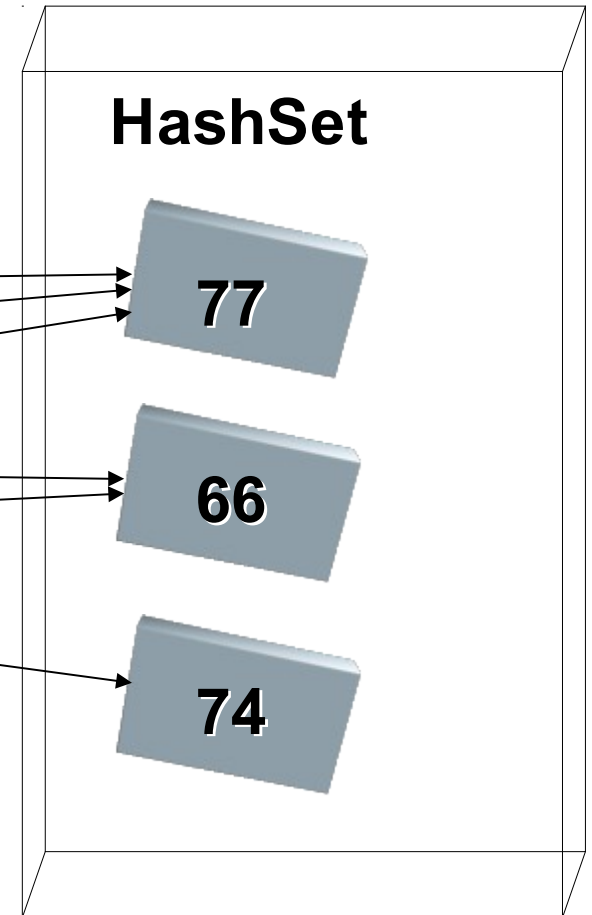
```
alunos.add(new Aluno("Ana"));
```

```
alunos.add(new Aluno("João"));
```

Como então funciona um:

```
alunos.contains(new Aluno("Jô")) ?
```

1. A coleção busca "Jô".hashCode() == 74
2. A coleção busca o container 74
3. Então verifica todos os elementos que neste caso é apenas "João"
testando: "João".equals("Jô"); o que retorna false e acaba o processo



Collections

- Quanto melhor implementado é o hashCode() mais rápida se torna a busca dentro da coleção hash, por exemplo, se mudarmos nossa implementação de hashCode() para:

```
public class Aluno {  
    //...  
    @Override  
    public int hashCode() { //hashCode vem da 1ª letra do nome  
        return (nome == null) ? 0 : nome.charAt(0) + nome.size();  
    }  
}
```

Collections

- Como seria a nova divisão ??

`alunos.add(new Aluno("Matheus"));`

$7 \cdot 77$

`alunos.add(new Aluno("Magno"));`

$5 \cdot 77$

`alunos.add(new Aluno("Marcos"));`

$6 \cdot 77$

`alunos.add(new Aluno("Antonio"));`

$7 \cdot 66$

`alunos.add(new Aluno("Ana"));`

$3 \cdot 66$

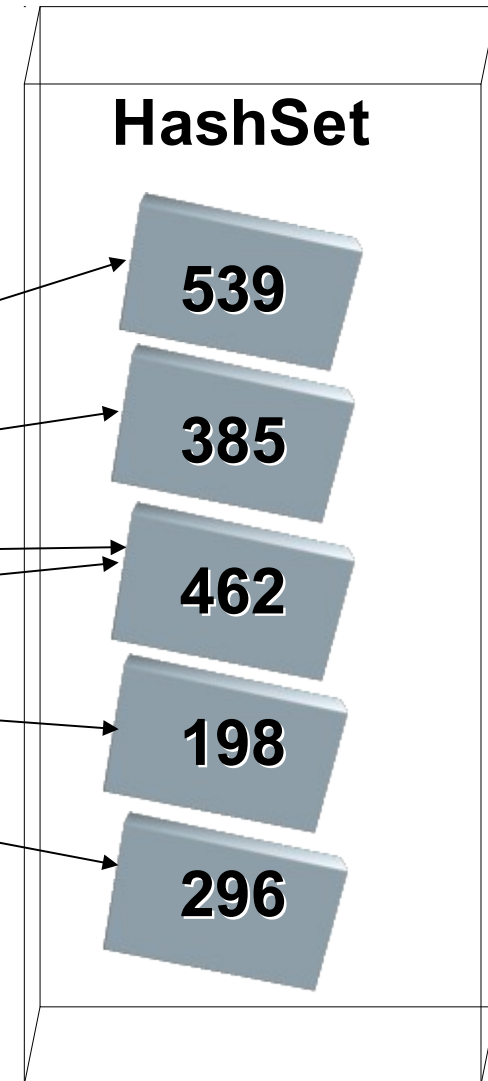
`alunos.add(new Aluno("João"));`

$4 \cdot 74$

Como então funciona um:

`alunos.contains(new Aluno("Jô")) ?`

1. A coleção busca `"Jô".hashCode() == 148`
2. A coleção verifica que não há o container 148
3. o que retorna false e acaba o processo



Collections

- Como coleções não hash realizam a mesma busca?

```
Aluno jo = new Aluno("Jô");
```

```
alunos.contains(jo);
```

```
//A coleção alunos iria testar equals contra cada objeto:
```

```
    "Matheus".equals(jo); //false
```

```
    "Magno".equals(jo); //false
```

```
    "Marcos".equals(jo); //false
```

```
    "Antonio".equals(jo); //false
```

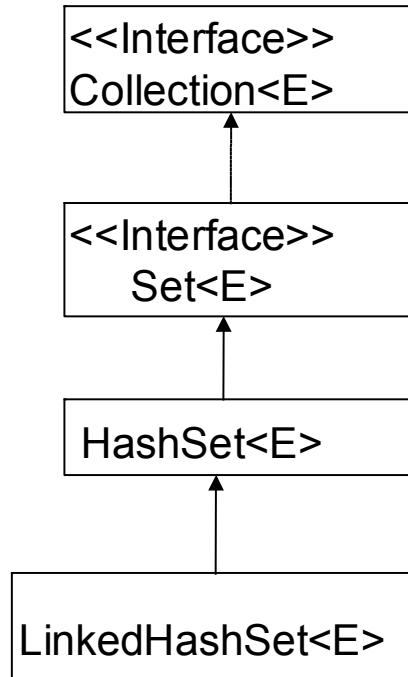
```
    "Ana".equals(jo); //false
```

```
    "João".equals(jo); //false
```

```
    return false;
```

- E essa é diferença de performance de uma coleção hash para uma normal.

Collections



- **HashSet:** uma das classes mais utilizadas (e que pode gerar grandes problemas quando hashCode não é bem implementado):
 - É uma coleção de elementos únicos não ordenada e não classificada.
 - A cada inserção a ordem dos elementos gerados pelo iterator pode alterar totalmente.
 - As buscas são realizadas usando tabelas hash.
- **LinkedHashSet:** A mudança básica para o **HashSet** é que a **LinkedHashSet** mantém armazenada a ordem de inserção.
 - A cada inserção a ordem dos elementos gerados pelo iterator NÃO é alterada.
 - É mantida a ordem de inserção.
 - As buscas são realizadas usando tabelas hash.

Collections

- Finalmente vamos conhecer o funcionamento dos `java.util.Map<K,V>`
- Um Map é uma coleção de pares key-value (chave-valor), porem esta não descende de `Collection<E>` tendo uma interface nova.
- Por exemplo:
 - Em um `Map<String,Thread>` podemos guardar instancias de Thread que são identificadas por Strings, onde é possível gerenciar facilmente as instancias através de suas chaves.
- Vamos conhecer sua interface

Collections

- Métodos de **java.util.Map<K,V>**:
 - V put(K key, V value)
 - void putAll(Map<? extends K,? extends V> m)
 - V remove(Object key)
 - void clear()
 - V get(Object key)
 - boolean containsKey(Object o)
 - boolean containsValue(Object o)
 - Set<Map.Entry<K, V>> entrySet()
 - Set<K> keySet()
 - Collection<V> values()
 - int size()
 - boolean isEmpty()

Collections

- Métodos de **java.util.Map<K,V>**:

V put(K key, V value)

void putAll(Map<? extends K,? extends V> m)

V remove(Object key)

void clear()

V get(Object key)

boolean containsKey(Object o)

boolean containsValue(Object o)

Set<Map.Entry<K, V>> entrySet()

Set<K> keySet()

Collection<V> values()

int size()

boolean isEmpty()

Insere o valor passado na chave indicada, e retorna o antigo valor da chave, ou **null** caso não exista valor anterior.

Implementação é opcional.

Collections

- Métodos de **java.util.Map<K,V>**:

V put(K key, V value)

void putAll(Map<? extends K,? extends V> m)

V remove(Object key)

void clear()

V get(Object key)

boolean containsKey(Object o)

boolean containsValue(Object o)

Set<Map.Entry<K, V>> entrySet()

Set<K> keySet()

Collection<V> values()

int size()

boolean isEmpty()

Copia todos os valores do mapa enviando *m*, para este mapa, nas suas respectivas chaves.

É o mesmo que iterar o `entrySet()` e adicionar as entradas uma a uma.

Implementação é opcional.

Collections

- Métodos de **java.util.Map<K,V>**:

V put(K key, V value)

void putAll(Map<? extends K,? extends V> m)

V remove(**Object** key)

void clear()

V get(Object key)

boolean containsKey(Object o)

boolean containsValue(Object o)

Set<Map.Entry<K, V>> entrySet()

Set<K> keySet()

Collection<V> values()

int size()

boolean isEmpty()

Remove, se estiver presente, o mapeamento de chave enviada, retornando o valor da chave que estava mapeado.

Implementação é opcional.

Collections

- Métodos de **java.util.Map<K,V>**:

V put(K key, V value)

void putAll(Map<? extends K,? extends V> m)

V remove(Object key)

void clear()

V get(Object key)

boolean containsKey(Object o)

boolean containsValue(Object o)

Set<Map.Entry<K, V>> entrySet()

Set<K> keySet()

Collection<V> values()

int size()

boolean isEmpty()

Remove todos as chaves e valores mapeados.

Implementação é opcional.

Collections

- Métodos de **java.util.Map<K,V>**:

V put(K key, V value)

void putAll(Map<? extends K,? extends V> m)

V remove(Object key)

void clear()

V get(**Object** key)

boolean containsKey(Object o)

boolean containsValue(Object o)

Set<Map.Entry<K, V>> entrySet()

Set<K> keySet()

Collection<V> values()

int size()

boolean isEmpty()

Recupera o valor para a chave enviada.

Collections

- Métodos de **java.util.Map<K,V>**:

V put(K key, V value)

void putAll(Map<? extends K,? extends V> m)

V remove(Object key)

void clear()

V get(Object key)

boolean containsKey(**Object** o)

boolean containsValue(Object o)

Set<Map.Entry<K, V>> entrySet()

Set<K> keySet()

Collection<V> values()

int size()

boolean isEmpty()

Verifica se este mapa contém uma chave significantemente igual ao objeto enviado.

Collections

- Métodos de **java.util.Map<K,V>**:

V put(K key, V value)

void putAll(Map<? extends K,? extends V> m)

V remove(Object key)

void clear()

V get(Object key)

boolean containsKey(Object o)

boolean containsValue(**Object** o)

Set<Map.Entry<K, V>> entrySet()

Set<K> keySet()

Collection<V> values()

int size()

boolean isEmpty()

Verifica se este mapa contém
um valor significativamente
igual ao objeto enviado.

Collections

- Métodos de **java.util.Map<K,V>**:

V put(K key, V value)

void putAll(Map<? extends K,? extends V> m)

V remove(Object key)

void clear()

V get(Object key)

boolean containsKey(Object o)

boolean containsValue(Object o)

Set<Map.Entry<K, V>> entrySet()

Set<K> keySet()

Collection<V> values()

int size()

boolean isEmpty()

Recupera a coleção **Set** das **Entry** deste mapa.

Entry são Entradas do mapa é um objeto que guarda o par key-value.

Veremos na pagina seguinte como funciona essa interface **Entry**.

Alterações nesse **Set** são refletidas no **Map** e vice-versa, muito cuidado ao manipular.

Collections

- Métodos de **java.util.Map<K,V>**:

V put(K key, V value)

void putAll(Map<? extends K,? extends V> m)

V remove(Object key)

void clear()

V get(Object key)

boolean containsKey(Object o)

boolean containsValue(Object o)

Set<Map.Entry<K, V>> entrySet()

Set<K> keySet()

Collection<V> values()

int size()

boolean isEmpty()

Recupera um Set contendo todas as chaves do **Map**.

Alterações nesse Set são refletidas no **Map** e vice-versa, muito cuidado ao manipular.

Collections

- Métodos de **java.util.Map<K,V>**:

V put(K key, V value)

void putAll(Map<? extends K,? extends V> m)

V remove(Object key)

void clear()

V get(Object key)

boolean containsKey(Object o)

boolean containsValue(Object o)

Set<Map.Entry<K, V>> entrySet()

Set<K> keySet()

Collection<V> values()

int size()

boolean isEmpty()

Recupera uma **Collection** contendo todos os valores deste **Map**.

Alterações nesse Set são refletidas no **Map** e vice-versa, muito cuidado ao manipular.

Collections

- Métodos de **java.util.Map<K,V>**:

V put(K key, V value)

void putAll(Map<? extends K,? extends V> m)

V remove(Object key)

void clear()

V get(Object key)

boolean containsKey(Object o)

boolean containsValue(Object o)

Set<Map.Entry<K, V>> entrySet()

Set<K> keySet()

Collection<V> values()

int size()

boolean isEmpty()

Informa a quantidade de pares key-value existentes neste mapa.

Collections

- Métodos de **java.util.Map<K,V>**:

V put(K key, V value)

void putAll(Map<? extends K,? extends V> m)

V remove(Object key)

void clear()

V get(Object key)

boolean containsKey(Object o)

boolean containsValue(Object o)

Set<Map.Entry<K, V>> entrySet()

Set<K> keySet()

Collection<V> values()

int size()

boolean isEmpty()

Verifica se o **Map** é vazio
Método de comodidade, o
mesmo que (size() == 0)

Collections

- Métodos de **java.util.Map.Entry<K,V>**:

K getKey() //recupera a chave desta entrada

V getValue() //recupera o valor desta entrada

V setValue(**V** value) //altera o valor para esta chave

Collections

- A uma interface em maps que também é classificada **java.util.SortedMap<K,V>**:
 - Esta interface tem exatamente o mesmo funcionamento da Coleção **SortedSet**, porém se aplica a parte de Keys do **Map**

Collections

- Métodos de **java.util.SortedMap<K,V>**:

Comparator<? **super** **K**> comparator()

K firstKey()

SortedMap<**K**,**V**> tailMap(**K** fromKey)

K lastKey()

SortedMap<**K**,**V**> headMap(**K** toKey)

SortedMap<**K**,**V**> subMap(**K** fromKey, **K** toKey)

Funciona igual
SortedSet para o **Set**
de keys do **Map**

Collections

- Também existe a interface semelhante a **NavigableSet** com mais funcionalidades **java.util.NavigableMap<K,V>**:
 - Esta interface tem exatamente o mesmo funcionamento da Coleção **NavigableSet**, porém se aplica a parte de keys do **Map**

Collections

- Métodos de **java.util.NavigableMap<K,V>**:

K lowerKey(**K** key)

Entry<K,V> lowerEntry(**E** key)

K floorKey(**K** e)

Entry<K,V> floorEntry(**E** key)

E higherKey(**K** key)

Entry<K,V> higherEntry(**K** key)

E ceilingKey(**K** key)

Entry<K,V> ceilingEntry(**K** key)

Entry<K,V> pollFirstEntry()

Entry<K,V> pollLastEntry()

NavigableSet<K> descendingKeySet()

NavigableMap<K,V> descendingMap()

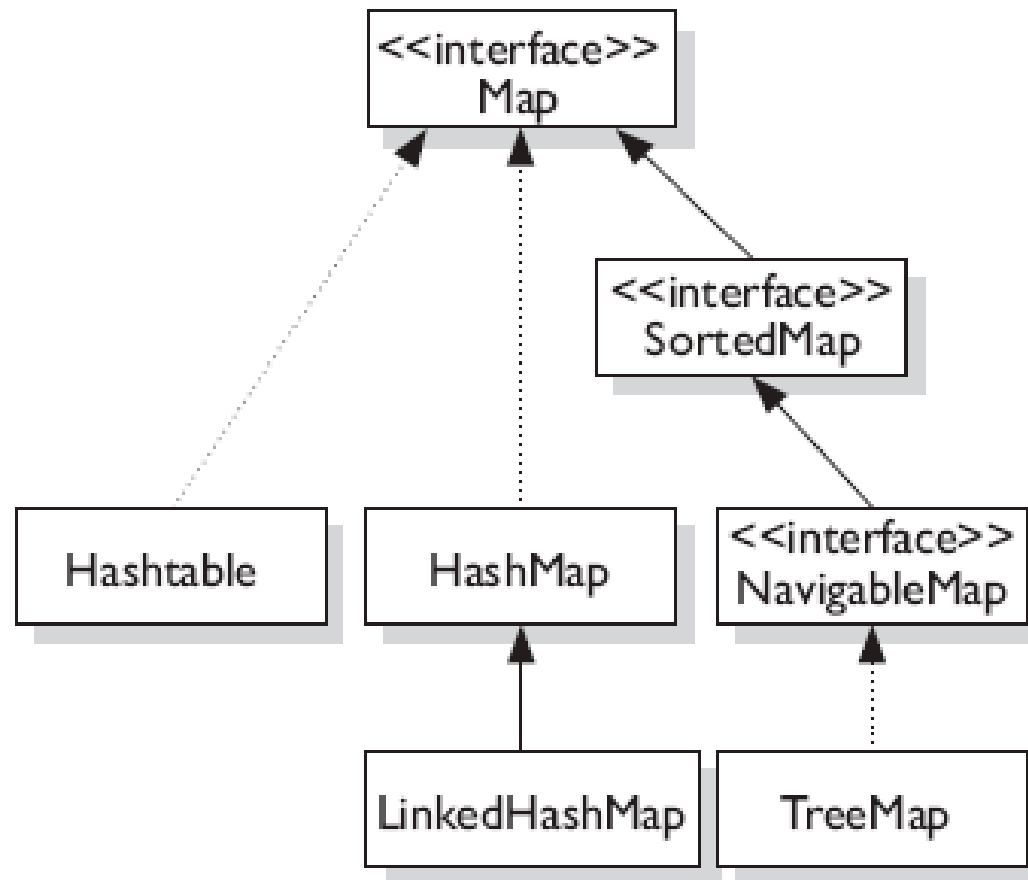
NavigableMap<K,V> headMap(**K** toElement, **boolean** inclusive)

NavigableMap<K,V> tailMap(**K** fromElement, **boolean** inclusive)

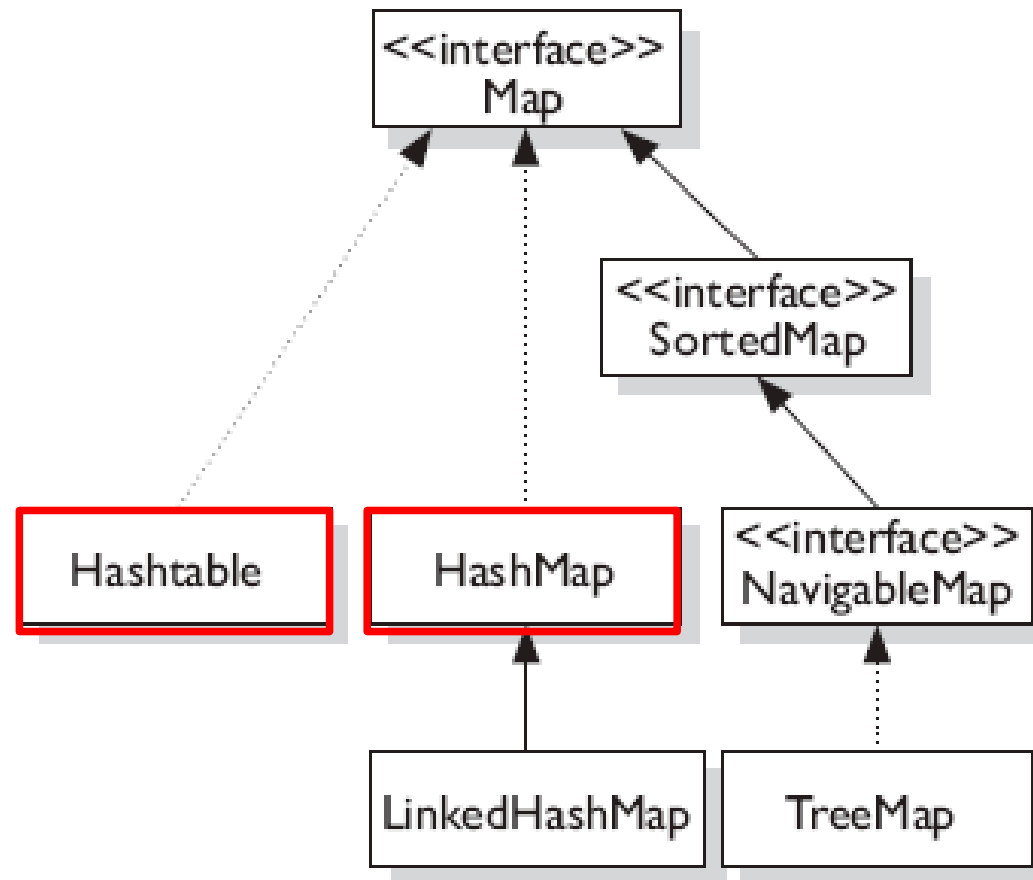
NavigableMap<K,V> subMap(**K** from, **boolean** inc, **K** to, **boolean** inc)

Funciona igual
NavigableSet para o
Set de keys do **Map**

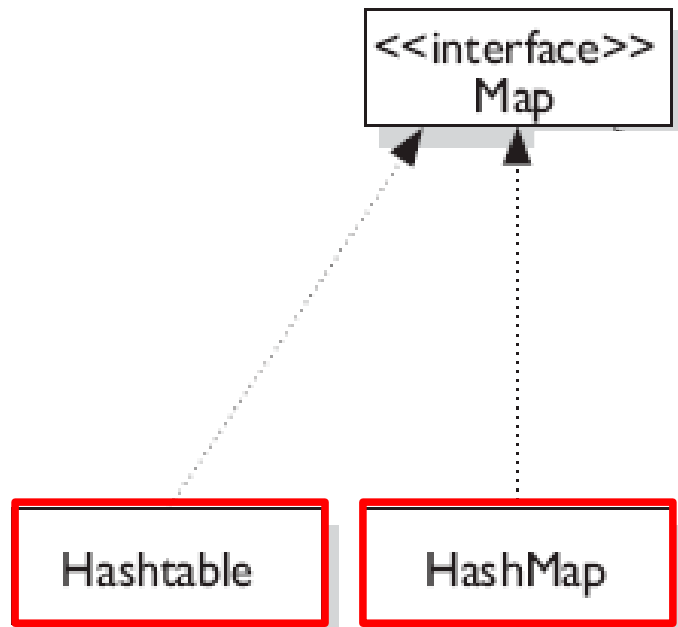
Collections



Collections



Collections

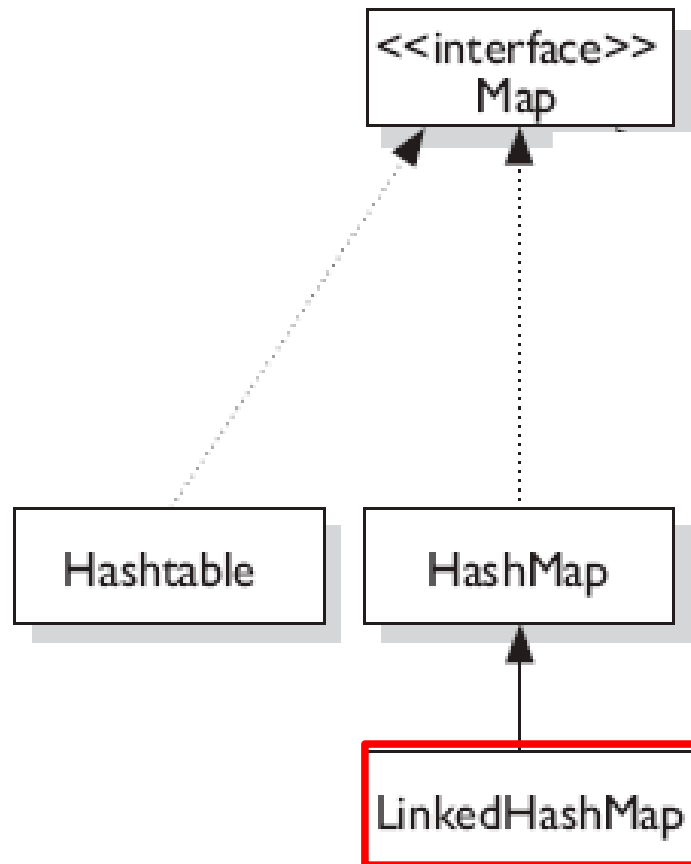


Hashtable e HashMap

Semelhanças: Não ordenadas, não classificadas, Baseiam suas buscas em tabelas hash.

Diferenças: **Hashtable** tem todos os seus métodos sincronizados enquanto **HashMap** não! é aconselhável o uso do **HashMap**

Collections



LinkedHashMap

Semelhanças: não classificadas, baseiam suas buscas em tabelas hash.

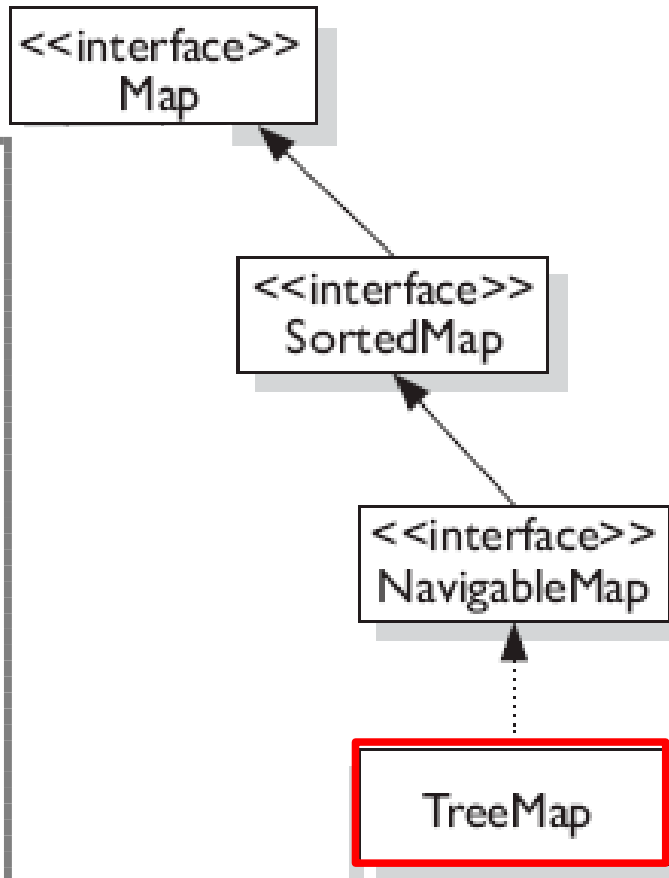
Diferenças: A ordem de inserção é levada em conta, e a iteração de seus `entrySet()`, `keySet()` ou `values()` obedecem a ordem de inserção.

Obs.: não é sincronizada.

Collections

TreeMap

É um Map classificado onde a ordem dos seus elementos segue a ordem de seu Comparator (se houver) ou a natural de seus elementos. Funciona semelhante a TreeSet.



Collections

- Vários métodos utilitários podem ser encontrados na classe **java.util.Collections**, sim Collections com “s” no fim.
- Esta classe esta munida das principais operações que você precisará realizar com Coleções, e todos os seus métodos são estáticos.
- Sua documentação pode ser encontrada neste link:
 - <http://java.sun.com/javase/6/docs/api/java/util/Collections.html>

Collections

- Existe mais alguma interface ou classe concreta? Da **Java Collections Framework** ??
- Onde eu posso encontrar mais informações ?
 - GUJ: <http://www.guj.com.br/>
 - Sergio Taborda's Weblog: <http://sergiotaborda.wordpress.com/java/colecoes-em-java/>
 - Tomaz Lavieri's Blog: <http://java-i9se.blogspot.com/>
 - Tutorial da sun: <http://java.sun.com/docs/books/tutorial/collections/index.html>
 - API Specification: <http://java.sun.com/javase/6/docs/technotes/guides/collections/reference.html>
 - The Java Collection Framework Documentations Contents: <http://java.sun.com/javase/6/docs/technotes/guides/collections/index.html>