Resumo de Collections

Adler Tenório

Collections: Ordenação *(Order)* e Classificação*(Sorted)*

1. Ordenação: *(Order)*

Quando uma coleção é ordenada, significa que você pode iterar em seus elementos em uma ordem específica, ou seja, o primeiro item adicionado na coleção será sempre o primeiro, o segundo será o segundo e o último o último e assim por diante diferentemente de uma coleção não ordenada que você ao iterar na mesma não tem nenhuma ordem, ou seja, a iteração é feira de maneira aleatória. É tanto que as classes que implementam List são ordenadas***(Order)*** e possuem métodos que trabalham com o índice como veremos a seguir.

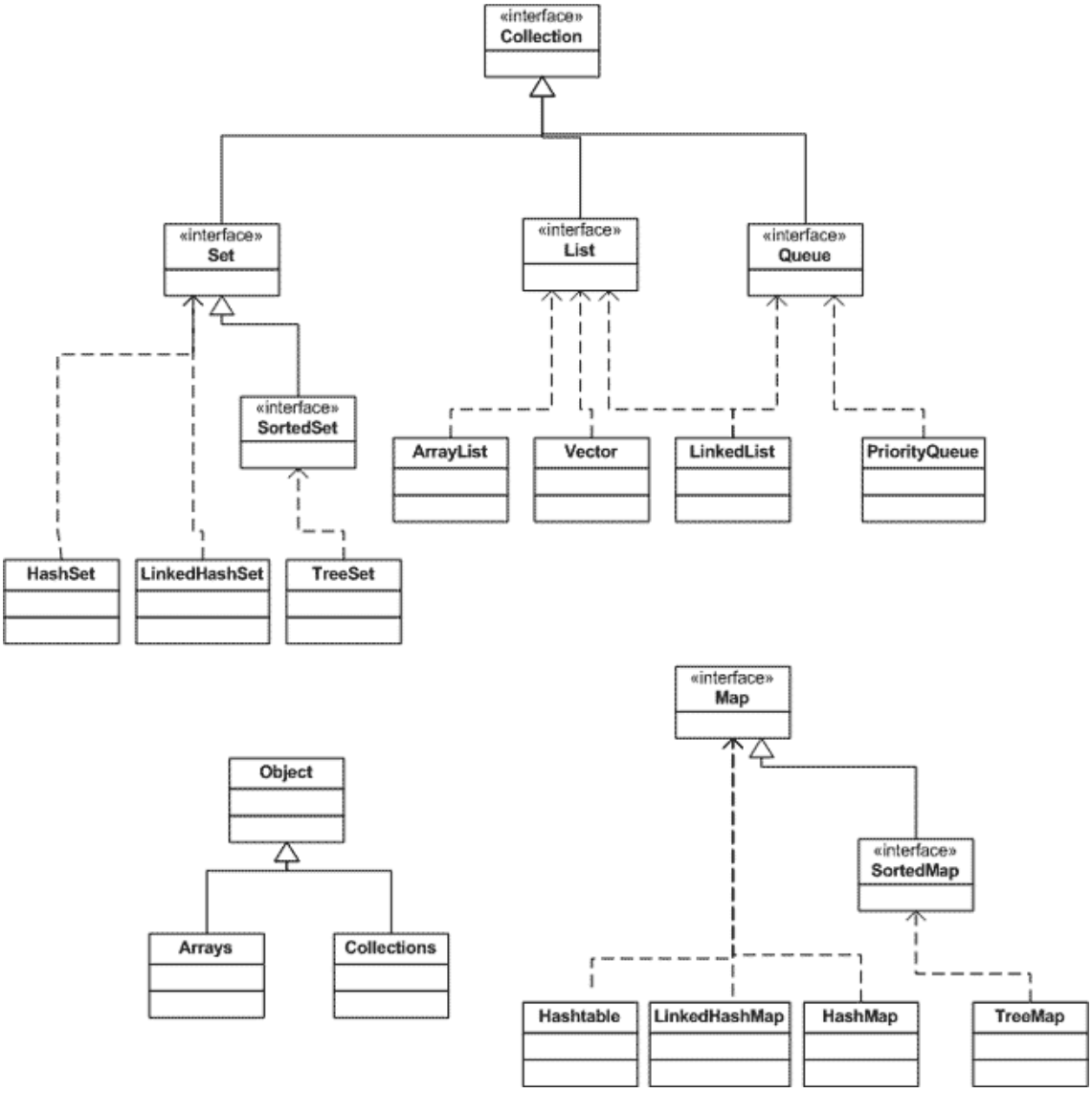
2. Classificação *(Sorted)*

Uma coleção *classificada* significa que a ordem de coleção está determinada por alguma(s) regra(s), conhecida(s) como ordem de classificação. **Uma ordem de classificação não tem nada a ver com o momento em que o objeto foi adicionado na coleção, nem a última vez que o mesmo foi acessado e nem em que posição o mesmo foi adicionado.** *A classificação é feita com base em propriedades do próprio objeto. Você coloca o objeto em uma coleção e a coleção descobrirá em que ordem colocá-los com base na ordem de classificação*.

Uma coleção que mantenha uma ordem como por exemplo ordem de inserção na verdade **não é consideração classificado** a não ser que use algum tipo de classificação, como por exemplo se a o objeto implementar as interfaces ***comparable*** e ***comparator*** e o com isso o programador pode chamar o método `Collections.sort(List<T> lista);` e a classe collections ordenará a coleção de acordo com as regras estabelecidas na implementaçãos dos métodos das interfaces ou até mesmo se a lista tiver tipos da própria api como String e Integer a implementação das Intefaces citadas anteriormente não é necessário pois a classe `Collections` usará a ordem natural que no caso de String é ordem alfabética(a,b,c...) e no caso de Integer é a ordem numérica (1,2,3...).

Pra finalizar lembrem-se que a ordem de classificação *(incluindo a ordem natural)* nada tem a ver com a ordem de inserção, índice ou ordem de acesso.

Hierárquia Geral Collections em Java



::: Interface List – Nessa interface o **índice** tem grande relevância.

Vários métodos relacionados com índices:

– get(int index); indexOf(Object o); add(int index, Objetct o);

- **ArrayList** – **Interação e acesso aleatório rápido;**

  – **Ordenado** por índices;

  – **Não Classificado**;

  – Implementa a interface **RadomAccess** uma interface marcadora e sinalizadora *(isso significa que a mesma não possui métodos)* que indica que essa lista possui acesso aleatório rápido *(geralmente de forma constante)*;

  – Prefira essa lista ao **LinkedList** quando precisar de iteração rápida e **não** pretende executar muitas inserções e exclusões.

- **Vector** – Basicamente a mesmas peculiaridades do **ArrayList** com a diferença que os métodos dessa classe são sincronizados para a segurança em theads;

  – Geralmente usa-se **ArrayList** pois como os métodos de **Vector** são sincronizados há uma perca no desempenho dessa classe;

  – Implementa a interface **RadomAccess.**

- **LinkedList** – Iteração **mais lenta** do que o **ArrayList**;

  – Inserções e remoções mais rápidas do que o **ArrayList**;

  – Ordenado por índices;

  – Elementos duplamente encadeados, esse encadeamento fornece novos métodos além dos obtidos pela interface **List** pra inserção e remoção do início ao final.

  – Implementa a interface Queue *(Fila)* com isso possui métodos próprios para o tratamento de pilhas e filas como*:* ***offer():*** Adiciona elementos na pilha. ***peek():*** Retorna o primeiro elemento adicionado(o elemento da ponta) da pilha e retorna null se a lista estiver vazia. ***poll():***Retonar o primeiro elemento e depois remove, retornando null caso a coleção esteja vazia.

***ArrayList** e **Vector** são as únicas classes que implementão **RadomAccess**.

::: Interface Set

– **Não permite objetos duplicados**, e essa diferença é estabelecida através do método *equals()* do objeto, se tentar adicionar um objeto duplicado nas classes que implementam **Set** **não** dará erro de compilação e **nem** de execução, simplesmente o objeto não é adicionado na coleção.

- **HashSet** – **Não ordenado e não classificado;**
  – Usa o código hashing *(hashcode())* do objeto que é inserido, ou seja, quanto mais eficiente for implementado o *hashcode()* do objeto mais eficiente o desempenho obtido no acesso.

- **LinkedHashSet** – Versão **ordenada** dos sets mas **não classificado**;
  – Permite fazer iterações nos elementos na ordem em que os mesmos foram inseridos.
  – Usa o código hashing *(hashcode())* do objeto que é inserido, ou seja, quanto mais eficiente for implementado o *hashcode()* do objeto mais eficiente o desempenho obtido no acesso.

- **TreeSet** – **Ordenada** e **classificada** *(por default pela ordem natural do objeto).*

  – Além da ordem natural dos objetos a classe **TreeSet** também ordena os objetos de acordo com regras estabelecidas pelos métodos estabelecidos pelas interfaces ***Comparable*** e ***Comparator*** e implementados nas classes que serão inclusas na coleção.

  – Se a classe **não** implementar ***Comparable ou Comparator*** lança erro em tempo de execução ao tentar adicionar o objeto na coleção

```
java.lang.ClassCastException: collections.Dog cannot be cast to
java.lang.Comparable
```

  – Implementa **NavigableSet** ***(Navega começando pelo meio da coleção proporciona um ganho de performace)*** que possui métodos próprios além dos métodos de **Set e SortedSet.**

Exemplo:

```
TreeSet<String> listaNomes = new TreeSet<String>(); //Implementa NabigableSet
listaNomes.add("Adler");     listaNomes.add("Zito");
listaNomes.add("Mauro");     listaNomes.add("Nanda");
```

```
listaNomes.add("Vicente"); listaNomes.add("Barbara");

Set<String> subSet = listaNomes.subSet("Barbara", true ,"Nanda", true);
Set<String> headSet = listaNomes.headSet("Nanda"); // Por defaul é EXCLUSIVE o
elemento passado como parâmetro
Set<String> tailSet = listaNomes.tailSet("Vicente"); // Por defaul é INCLUSIVE o
elemento passado como parâmetro

System.out.println("TreeSet:" + listaNomes); System.out.println("SubSet:" + subSet);
System.out.println("HeadSet:" + headSet);
System.out.println("TailSet:" + tailSet);
```

::: Interface Map – Identificadores únicos (chaves), <Chave, Valor>;

– Não permite chaves duplicadas (não dá erro), se adicionarmos uma chave duplicada no map ele substitui o valor do último item adicionado e preserva a chave.

– Permite buscar um valor com base na chave, obter uma coleção apenas dos valores ou apenas das chaves;

– Usa o primeiro o *equals()* e depois o *hashCode()* para determinar se duas cheves são iguais.

- **HashMap** – Mapa **não ordenado** e **não classificado;**

  – A iteração e o armazenamento das chaves é feita através do *hashCode()* da chave, ou seja, quanto mais eficiente a implementação do *hashCode() melhor* performance ele terá no acesso.

  – Permite apenas um chave **null**, porém muitos valores **null**;

  – Se tentarmos adicionar uma segunda chave **null** no mapa ele simplesmente não adiciona *(não dá erro de compilação e nem erro de execução);*

- **Hashtable** – Mapa **não ordenado** e **não classificado;**

  – Versão sincronizada dos mapas, ou seja, seus métodos são sincronizados para uma maior segurança quanto ao uso de theads;

  – Diferente do **HashMap** o mesmo **não aceita valores nulos** em hipótese alguma causando nullpointer em tempo de execução.

- **LinkedHashMap** – **Ordenado** pela ordem de inserção;

  – **Iteração mais rápida** que o **HashMap**;

– **Inserção** e **remoção** mais lentas que o **HashMap**;

- **TheeMap** – **Ordenado** e **Classificado** pela ordem natural dos elementos ou por uma forma de comparação customizada através das interfaces ***Comparable*** ou ***Comparator;***

  – Implementa **NavigableMap** que possui métodos próprios além dos métodos de **Map** e **SortedMap**.

– Se a classe **não** implementar ***Comparable ou Comparator*** lança erro em tempo de execução ao tentar adicionar o objeto na coleção

```
java.lang.ClassCastException: collections.Dog cannot be cast to
java.lang.Comparable
```

::: Interface Queue *(Fila)* – Ordenados no estilo pilha *(primeiro a entrar é o primeiro a sair)* apesar de admitir outras formas;

– Suportam todos os métodos da interface **Collection,** além de possuir vários métodos em relação a filas.

- **PriorityQueue** *(Prioridade na fila)* – Como o próprio nome já diz dá prioridade aos objetos, ou seja, primeiro a entrar será o primeiro a sair;
  – **Ordenada** e **Classificada** de forma natural por default ou através das interfaces de comparação (***Comparable*** e ***Comparator***) os elementos ordenados primeiro serão classificados primeiro;

Alguns métodos:

offer()*:* Adiciona elementos na pilha.

peek(): Retorna o elemento com maior prioridade da pilha e retorna null se a lista estiver vazia.

poll(): Retonar o elemento com maior prioridade e depois remove, retornando null caso a coleção esteja vazia.

// Possuem também métodos equivalentes só que todos lançam exeception se não conseguir realizar a operação.

Add(): Adiciona elementos na pilha, porém lança exceção se não conseguir adicionar.

element(): Retorna o primeiro elemento adicionado(o elemento da ponta) da pilha e lança NoSuchElementException caso lista esteja vazia.

remove(): Remove o elemento com maior prioridade e lança NoSuchElementException caso lista esteja vazia.