Java Collections

UTN – Facultad Regional Paraná

Titular: Ernesto Zapata Icart

Aux: Mariano Carpio

Java - Collections Framework

Java Collection Framework fue diseñado para alcanzar los siguientes objetivos:

- El framework tiene que ser altamente performante (eficiente).
- El framework tiene que permitir a los diferentes tipos de colecciones trabajar de manera similar y con un alto grado de interoperabilidad.
- El framework tiene que permitir extender o adaptar un tipo de colección fácilmente.

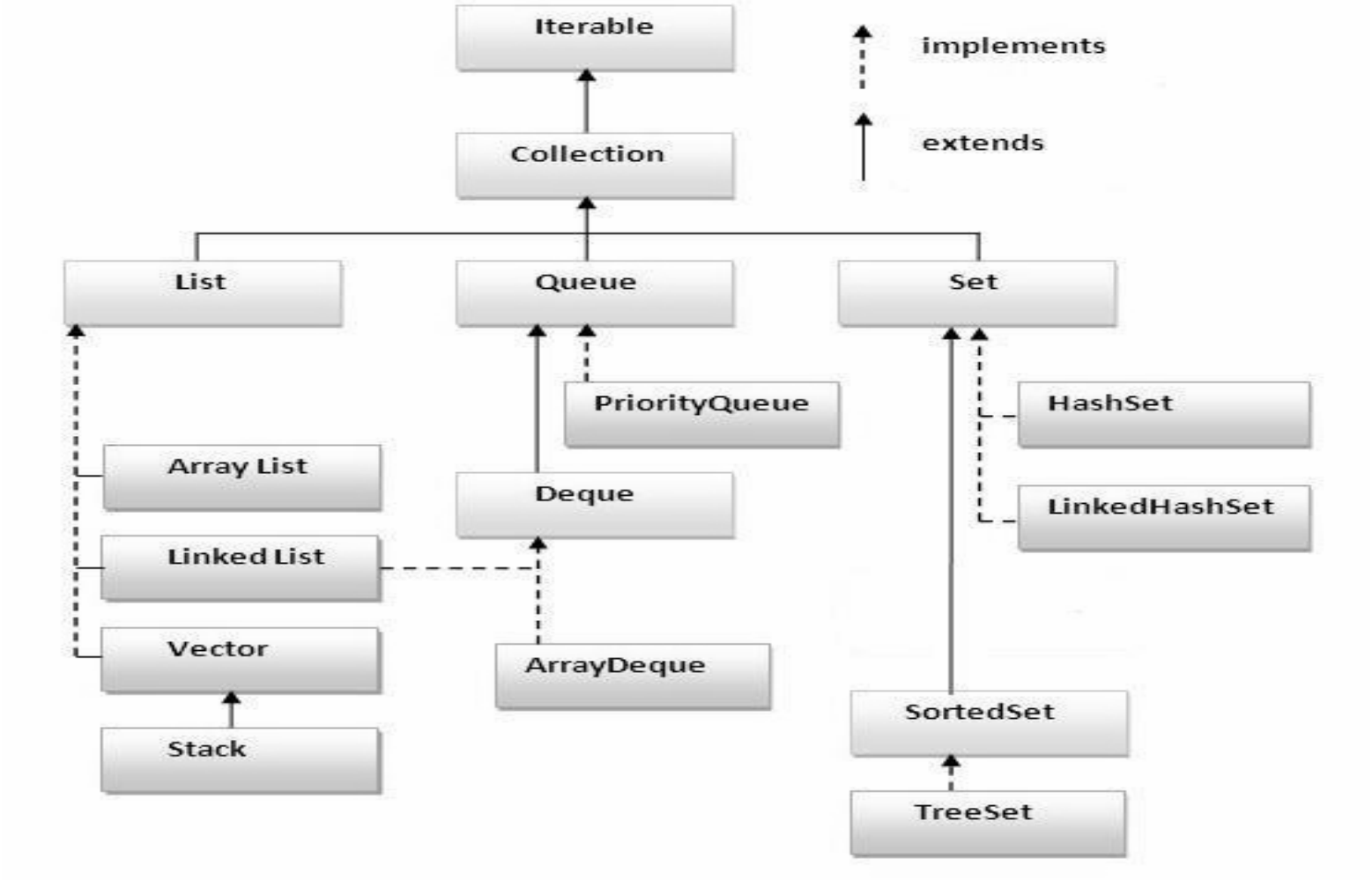
Todo el framework contiene los siguientes elementos:

- **Interfaces**: establecen un protocolo (forma) de manipulación de colecciones independientemente de los detalles de implementación de los distintos tipos de colecciones.
- **Implementaciones** (clases): Son implementaciones concretas de las interfaces de las colecciones. En esencia, son estructuras de datos reusables.
- **Algoritmos**: métodos implementados para llevar a cabo algún tipo de proceso computacional, como búsquedas u ordenamientos, sobre objetos que implementan las interfaces de **Collections**.

Collections

- Las colecciones representan un grupo de elementos (referencias a objetos)
- El paquete **java.util** contiene todas las clases e interfaces que soportan el llamado **Collection Framework**.
- No se especifica si sus elementos están
  - Ordenados
  - Duplicados

Collections



Collections - métodos

Método	Descripción
public boolean **add**(Object elem)	Inserta un elemento en esta colección.
public boolean **addAll**(collection c)	Inserta una colección c dentro de la colección que se invoca el método.
public boolean **remove**(Object elem)	Borra un elemento de la colección.
public boolean **removeAll**(Collection c)	Borra todos los elementos de c de la colección que se invoca el método.
public boolean **retainAll**(Collection c)	Borra todos los elementos que no estén en c de la colección que se invoca el método.
public int **size**()	Retorna el número total de elementos de la colección.
public void **clear**()	Borra todos los elementos de la colección
public boolean **contains**(object elem)	Busca un elemento específico de la colección
public boolean **containsAll**(Collection c)	Retorna true si la colección contiene todos los elementos de la colección c.
public Iterator **iterator**()	Retorna un Iterador
public Object[] **toArray**()	Convierte la colección en un Arreglo
public boolean **isEmpty**()	Verifica si la colección está vacía.

Iterator

- La interfaz **Iterator** provee la facilidad de interactuar con los elementos de una colección en forma secuencial y progresiva, de principio a fin.

Método	Descripción
public boolean **hasNext**()	Retorna true si el iterador contiene más elementos.
public object **next**()	Retorna el elemento actual del iterador y mueve el puntero del cursor a la próxima posición.
public void **remove**()	Borra el ultimo elemento retornado por el iterador.

ArrayList

- La clase ArrayList implementa un arreglo dinámico para almacenar elementos. Extiende AbstractList e implementa la interfaz List.
- Puede contener elementos duplicados.
- Mantiene el orden de inserción.
- No es sincronizado (synchronized).
- Permite acceso aleatorio por medio de un índice.
- La manipulación de elementos en ArrayList es lenta, ya que es necesario reacomodar la lista si un elemento es eliminado.

LinkedList

- La clase LinkedList usa una lista doblemente enlazada para almacenar los elementos. Extiende **AbstractList** e implementa las interfaces **List** y **Deque**.
- Puede contener elementos duplicados.
- Mantiene el orden de inserción.
- No es sincronizado (synchronized).
- La manipulación de elementos en LinkedList es rápida, ya que no es necesario procesamiento extra para reordenar elementos.
- LinkedList puede ser usada como una lista, una pila o una cola.

ArrayList	LinkedList
ArrayList internamente usa arreglos dinámicos para almacenar elementos	**LinkedList** internamente usa una lista doblemente enlazada para almacenar elementos.
La manipulación con **ArrayList** es lenta porque internamente usa arreglos. Si un elemento es removido del arreglo, todos los elementos posteriores deben ser movidos en memoria.	La manipilación con **LinkedList** es más rápida que **ArrayList** porque esto usa una lista doblemente enlazada. No se necesita procesamiento extra en memoria.
ArrayList puede actuar como una **List** porque implementa solamente a la interfaz **List**.	**LinkedList** puede actuar como una lista y como una cola, ya que implementa las interfaces **List** y **Deque**.
ArrayList es mejor para almacenar y accesar elementos.	**LinkedList** es mejor si tenemos que manipular elementos dentro de la estructura.

List

- La interfaz **List** es una subinterface de **Collection**. Contiene métodos para insertar y eliminar elementos basados en índices. Es una fábrica (factory) de la interfaz **ListIterator**.
- Los métodos más usados son:

Método	Descripción
public void **add**(int index,Object element);	Agrega un elemento en la posición *index*.
public boolean **addAll**(int index,Collection c);	Agrega una colección en la posición *index*.
public object **get**(int index position);	Obtiene el elemento de la posición *index*.
public object **set**(int index,Object element);	Sobrescribe un elemento de la lista y retorna el elemento anterior de la posición *index*.
public object **remove**(int index);	Remueve el elemento de la posición *index* y retorna ese elemento.
public ListIterator **listIterator**();	Obtiene un ListIterator.
public ListIterator **listIterator**(int index);	Obtiene un ListIterator a partir de la posición *index*.

ListIterator

- La interfaz ListIterator es usada para recorrer los elementos de una lista en dirección progresiva o regresiva.
- Los métodos más usados son:

Método	Descripción
public boolean **hasNext**();	Indica si tiene o no un sucesor.
public Object **next**();	Mueve el cursor interno a la siguiente posición y retorna el objeto.
public boolean **hasPrevious**();	Indica si tiene o no predecesor.
public Object **previous**();	Mueve el cursos interno a la posición anterior y retorna el objeto.

Diferencia entre List y Set

- **List** puede contener elementos duplicados mientras que Set contiene elementos únicos.
- **HashSet** se utiliza para instanciar **Set**.
- **HashSet** utiliza tablas hash para almacenar elementos. Esto extiende la clase **AbstractSet** e implementa la interfaz **Set**.
- Contiene elementos únicos.

[34, 22, 10, 60, 30] The sorted list is: [10, 22, 30, 34, 60] The First element of the set is: 10 The last element of the set is: 60

Set, HashSet y TreeSet: ejemplo

```
import java.util.*;
public class SetDemo {
        public static void main(String args[]) {
                int count[] = {34, 22,10,60,30,22};
                Set<Integer> set = new HashSet<Integer>();
                try {
                        for(int i = 0; i < 5; i++) {
                                set.add(count[i]);
                        }
                        System.out.println(set);
                        TreeSet sortedSet = new TreeSet<Integer>(set);
                        System.out.println("The sorted list is:");
                        System.out.println(sortedSet);
                        System.out.println("The First element of the set is: "+ (Integer)sortedSet.first());
                        System.out.println("The last element of the set is: "+ (Integer)sortedSet.last());
                } catch(Exception e) {}
        }
}
```

```
[34, 22, 10, 60, 30]
La lista ordenada es:
[10, 22, 30, 34, 60]
El primer elemento de la lista es: 10
El último elemento de la lista es: 60
```

https://www.tutorialspoint.com/java/java_collections.htm

Map

- La interfaz **Map** contiene elementos almacenados como un par (clave, valor). Cada par es conocido como una entrada. **Map** contiene únicamente valores únicos.
- Los métodos más usados son:

Método	Descripción
public Object **put**(object key,Object value)	Inserta una entrada en este **Map**.
public void **putAll**(Map map)	Inserta un mapa dentro de este **Map**.
public Object **remove**(Object key)	Borra una entrada identificada por su clave.
public Object **get**(Object key)	Recupera un objeto identificado por su clave.
public boolean **containsKey**(Object key)	Retorna true si la clave existe en este **Map**.
public boolean **containsValue**(Object value)	Retorna true si el valor existe en este **Map**.
public Set **keySet**()	Retorna un **Set** formado por sus claves.

Algoritmos

- Sorting (Odenamiento): ordena los elementos de una lista en orden ascendente de acuerdo con un criterio de ordenamiento.
- Shuffling (Dispersión/mezcla): El algoritmo aleatorio destruye cualquier rastro de orden que pueda haber estado presente en una Lista.
- Searching (Búsqueda): algoritmo de búsqueda binaria que busca un elemento especificado en una lista ordenada.

Comparator

- La interfaz **Comparator** define dos métodos: *compare()* y *equals()*.
- *int* ***compare****(Object obj1, Object obj2)*. Compara dos elementos por orden. Retorna 0 si son iguales, un valor positivo si obj1 es mayor que obj2, o negativo si obj1 es menor que obj2.
- *boolean* ***equals****(Object obj)*. Verifica si un objeto es igual a otro invocando un Comparator.