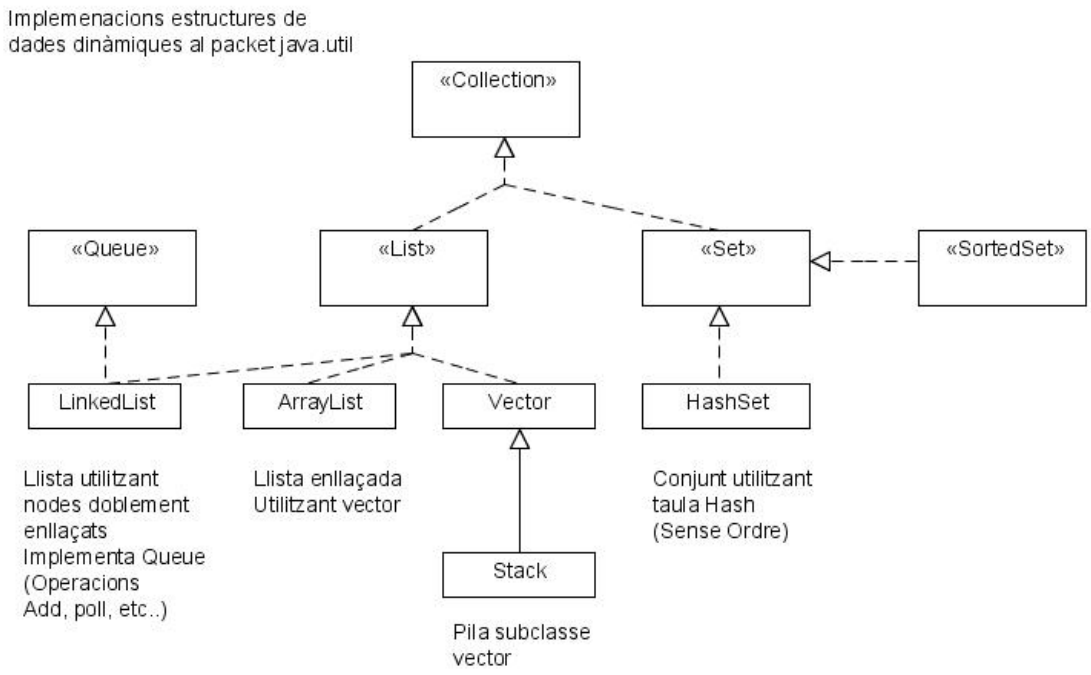


0. Collection

https://docs.oracle.com/javase/6/docs/api/java/util/Collection.html

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

Una colección es la representación de un grupo de objetos que seran conocidos como sus elementos.

El recorrido por los elementos de una colección se puede efectuar gracias al método iterator() que devuelve una referencia a la interficie Iterator. Collection tiene el método iterator() ya que hereda de Iterable.

Iterable: permite que un objeto sea el objeto principal de un bucle for-each.

Iterator: nos permite saber si del elemento actual de un conjunto hay un elemento siguiente, ir al siguiente y eliminar el actual.

ListIterator: lo mismo que iterator y además nos permite obtener y visitar el elemento anterior

1. Interface List

https://docs.oracle.com/javase/6/docs/api/java/util/List.html

Se encarga de definir métodos para trabajar con colecciones (Collection) ordenadas y con elementos repetidos. **Admite elementos repetidos**. Algunos de los métodos de la interface List son los siguientes:

- **add(Object o)**: añade un objeto al final de la lista.
- **add(int indice, Object o)**: añade un objeto a la lista en la posición indicada.
- **get(int indice)**: devuelve el objeto de la lista de la posición indicada.
- **remove(Object o)**: elimina la 1a ocurrencia del objeto en la lista pasado por parámetro.
- **remove(int indice)**: elimina el objeto de la lista que se encuentra en la posición pasada por parámetro.
- **clear()**: elimina todos los elementos de la lista.
- **indexOf(Object o)**: devuelve la posición de la primera vez que un elemento coincida con el objeto pasado por parámetro. Si el elemento no se encuentra devuelve -1.
- **lastIndexOf(Object o)**:devuelve la posición de la última vez que un elemento coincida con el objeto pasado por parámetro. Si el elemento no se encuentra devuelve -1.
- **size()**: devuelve el número de elementos de la lista.
- **contains(Object o)**: devuelve verdadero si en la lista aparece el objeto pasado por parámetro, para lo cual utiliza intrínsecamente el método equals().

Existen implementaciones de la interface List, como son las clases ArrayList, LinkedList, Vector y Stack.

Ejemplo de ArrayList:

```
List listaA = new ArrayList();
listaA.add("elemento 0");
listaA.add("elemento 1");
listaA.add("elemento 2");

//accesso con índice.
String elemento0 = listaA.get(0);
String elemento1 = listaA.get(1);
String elemento3 = listaA.get(2);

//acceso mediante un iterador.
Iterator iterador = listaA.iterator();
while (iterador.hasNext()) {
	String elemento = (String) iterador.next();
}

//acceso mediante un bucle for-each.
for (Object objecto : listaA) {
	String elemento = (String) objecto;
}
```

1.1. ArrayList

https://docs.oracle.com/javase/6/docs/api/java/util/ArrayList.html

Los **ArrayList** se basan en índices, siendo cero la posición inicial e infinito su posición final, o lo que es lo mismo, contiene tantos objetos como necesitemos, almacenando los elementos en un array de objetos. Permiten el acceso aleatorio, por lo que se puede acceder rápidamente a cualquier elemento a un tiempo constante (acceso por posición), pero las inserciones y eliminaciones de cualquier lado (excepto al final) requiere mover todos los elementos, ya sea para hacer un hueco para el nuevo elemento o llenar un espacio dejado por uno que se eliminó (no pueden haber huecos libres). También, si se añaden mas elementos que la capacidad del array (imaginemos que hemos creado un array de longitud 20 y vamos a meter el elemento nº 21), uno nuevo con el doble de tamaño es creado, y el array anterior es copiado al nuevo, con todo el tiempo de cómputo que eso significa. ArrayList es un array de reajuste dinámico.

Esta clase tiene varios constructores, siendo la más importante el ArrayList(), que construye un ArrayList con capacidad cero por defecto pero con infinitos objectos a insertar. Si le queremos dar un tamaño empleamos el constructor ArrayList(int numElementos).

Vector es lo mismo que **ArrayList** pero sus métodos estan syncronized para la programación concurrente.

ArrayList y **Vector** son listas genéricas, por lo que pueden mantener todo tipo de elementos: números, cadenas y otros objetos; mezclados o no.

Ejemplo: crear un ArrayList y un Vector para añadirles varios elementos de diferente tipo

```
import java.util.ArrayList;
import java.util.Vector;
import java.util.Date;

public class EjListArray {
	public static void main(String[] args) {
		ArrayList list1 = new ArrayList();
		list1.add("Hola");
		list1.add(5);
		list1.add(3.14159);
		list1.add(new Date());
		System.out.println("ArrayList");
		for (int i = 0; i < list1.size(); i++) {
			Object valor = list1.get(i);
			System.out.format("Índice: %d; Valor: %s\n", i, valor);
		}

		Vector list2 = new Vector();
		list2.add("Hola");
		list2.add(5);
		list2.add(3.14159);
		list2.add(new Date());
		System.out.println("\nVector");
		for (int i = 0; i < list2.size(); i++) {
			Object valor = list2.get(i);
			System.out.format("Índice: %d; Valor: %s\n", i, valor);
		}
	}
}
```

En list1 y list2 podemos meter datos de cualquier tipo (mezclandolos si queremos) porque al crear las llistas no hemos especificado el tipo. Si queremos especificar un tipo de dato concreto (ex: integer) haríamos

ArrayList<Integer> list1 = new ArrayList<Integer>();

Como list1 y list2 pueden contener datos de varios tipos diferentes, para coger su contenido usamos una variable de tipo Object que se tragará todos los tipos (integers, strings, ...).

Ejemplo: caso en el que todos los objetos que queremos guardar sean integer

```
ArrayList<Integer> lista = new ArrayList<Integer>();
lista.add(2);
lista.add(3);
lista.add(4);
int suma = 0;
for (int i = 0; i < lista.size(); i++) {
    suma = suma + lista.get(i);
}
System.out.println("Suma: " + suma);
```

1.2. LinkedList

https://docs.oracle.com/javase/6/docs/api/java/util/LinkedList.html

La clase **LinkedList** almacena los elementos en una lista vinculada y permiten un acceso a ella de manera secuencial. Cada elemento del conjunto esta formado por el objeto del conjunto y un apuntador al anterior y al siguiente elemento. Te permite realizar inserción y eliminaciones constantemente, pero solo un acceso secuencial a los elementos: solo se puede iterar sobre la lista para adelante y para atrás, eso significa que acceder un elemento en el medio toma tiempo proporcionalmente al tamaño de la lista. No es tan eficiente como los arrays.

LinkedList es una lista genérica, por lo que puede mantener todo tipo de elementos: números, cadenas y otros objetos; mezclados o no.

Ejemplo:

```
import java.util.*;

public class LinkedListDemo {

   public static void main(String args[]) {
      // create a linked list
      LinkedList ll = new LinkedList();
      // add elements to the linked list
      ll.add("F");
      ll.add("B");
      ll.add("D");
      ll.add("E");
      ll.add("C");
      ll.addLast("Z");
      ll.addFirst("A");
      ll.add(1, "A2");
      System.out.println("Original contents of ll: " + ll);

      // remove elements from the linked list
      ll.remove("F");
      ll.remove(2);
      System.out.println("Contents of ll after deletion: "
       + ll);

      // remove first and last elements
      ll.removeFirst();
      ll.removeLast();
      System.out.println("ll after deleting first and last: "
       + ll);

      // get and set a value
      Object val = ll.get(2);
      ll.set(2, (String) val + " Changed");
      System.out.println("ll after change: " + ll);
   }
}
```

Sortida de l'exemple:

```
Original contents of ll: [A, A2, F, B, D, E, C, Z]
Contents of ll after deletion: [A, A2, D, E, C, Z]
ll after deleting first and last: [A2, D, E, C]
ll after change: [A2, D, E Changed, C]
```

Se puede llegar a un elemento de la LinkedList a través de su contenido o de su posición ([0, size() - 1]).

Así que dependiendo de las operaciones que pretendas realizar deberás escoger las implementaciones. Iterar sobre cualquiera de las dos es prácticamente igual de "barato". Técnicamente un ArrayList es más rápido para el acceso (localización de elementos por índice o por valor) y un LinkedList para las operaciones de inserción y eliminación.

2. Interface Queue

https://docs.oracle.com/javase/6/docs/api/java/util/Queue.html

https://docs.oracle.com/javase/6/docs/api/java/util/Deque.html

Queue = colas del tipo FIFO (first in, first out), cola por la que se insertan elementos por un extremo y van saliendo por el otro (como en la cola de la compra).

Deque = double ended queue (cola doblemente terminada, enlazada, que permite insertar y eliminar elementos por ambos extremos a diferencia de una cola tipo FIFO). Es un tipo de Queue.

Generalmente son FIFO (First In First Out). Es una estructura de datos secuencial donde se añaden elementos al final de la cola y se sacan por el principio de la cola.

También existen las colas FIFO (First In First Out) de prioridad donde los elementos se ordenan en función de un valor (comparator).

Las Queue pueden estar basadas en arrays o en listas. **Admite elementos repetidos**.

Algunos de los métodos de la interface Queue son los siguientes:

- **add(Object o)**: añade un objeto al final de la cola, si falla lanza una excepción.
- **addFirst(Object o)**: añade un objeto al principio de la cola
- **offer(Object o)**: añade un objeto a la cola si es posible, si falla retorna false. Para casos en el que el tamaño de la cola es fijo (el nº de elementos que caben está limitado a un nº máximo).
- **remove()**: elimina la cabeza (head) de la cola, si falla (ex: lista vacia) lanza una excepción.
- **removeLast()**: recupera y elimina el último elemento de la cola.
- **poll()**: recupera y elimina la cabeza (head) de la cola, si falla (ex: lista vacia) devuelve null.
- **element()**: devuelve la cabeza (head) de la cola, ¿si falla (ex: lista vacia) lanza una excepción?
- **peek()**: devuelve la cabeza (head) de la cola, ¿si falla (ex: lista vacia) devuelve null?

Existen implementaciones de la interface Queue, como son las clases ArrayDeque y LinkedList.

Métodos de Deque:

	First Element (Head)		Last Element (Tail)	
	Throws exception	*Special value*	*Throws exception*	*Special value*
Insert	addFirst(e)	offerFirst(e)	addLast(e)	offerLast(e)
Remove	removeFirst()	pollFirst()	removeLast()	pollLast()
Examine	getFirst()	peekFirst()	getLast()	peekLast()

Cuando usamos un Deque como una Queue (FIFO):

Queue Method	**Equivalent Deque Method**
add(e)	addLast(e)
offer(e)	offerLast(e)
remove()	removeFirst()
poll()	pollFirst()
element()	getFirst()
peek()	peekFirst()

Cuando usamos una Deque como pila (stack) LIFO:

Stack Method	**Equivalent Deque Method**
push(e)	addFirst(e)
pop()	removeFirst()
peek()	peekFirst()

Esta interfaz se debe de utilizar con preferencia a la clase Stack.

2.1. ArrayDeque y LinkedList

https://docs.oracle.com/javase/6/docs/api/java/util/ArrayDeque.html

https://docs.oracle.com/javase/6/docs/api/java/util/LinkedList.html

El **ArrayDeque** está implementado con un array de tamaño variable, no acepta nulls y es más rápida que la LinkedList. Es una cola doblemente enlazada.

La **LinkedList** es una lista doblemente enlazada que admite nulls.

Algunos de los métodos de la interface **ArrayDeque** y **LinkedList** que no posee **Queue** son los siguientes:

- **addFirst(Object o)**: añade un objeto al principio de la cola.
- **addLast(Object o)**: añade un objeto al final de la cola.
- **clear()**: elimina todos los elementos de la cola.
- **removeFirst()**: recupera y elimina el primer elemento de la cola.
- **removeLast()**: recupera y elimina el último elemento de la cola.
- **pollLast()**: recupera y elimina el último elemento de la cola, si falla (ex: lista vacia) devuelve null.
- **peek()**: devuelve la cabeza (head) de la cola, si falla (ex: lista vacia) devuelve null
- **push()**: coloca un elemento en la cabeza de la pila representada por esta cola. Lanza una excepción si no hay espacio en la pila.
- **pop()**: saca un elemento de la pila (la cabeza) representada por esta cola.
- **contains(Object o)**: devuelve verdadero si en la cola aparece el objeto pasado por parámetro.

Ejemplo de Queue de tipo LinkedList:

```
public class EjColaUso {

	public static void main(String[] args) {
		Queue<String> mensajes = new LinkedList<String>();

		mensajes.offer("Mensaje Uno");
		mensajes.offer("Mensaje Dos");
		mensajes.offer("Mensaje Tres");
		mensajes.offer("Mensaje Cuatro");

		while (mensajes.size() > 0) {
			String msg = mensajes.poll();
			System.out.println(msg);
		}
	}
}
```

El método **offer()** añade el elemento pasado por parámetro a la última posición de la lista.

El método **poll()** recupera y elimina el 1r elemento de la lista.

2.2. Stack

https://docs.oracle.com/javase/6/docs/api/java/util/Stack.html

Stack es una pila de tipo LIFO (Last In First Out). Es una estructura de datos secuencial en la que solo se pueden añadir (**push()**) y sacar (**pop()**) elementos por la cima (top) de la pila. Si la pila tiene 5 elementos y queremos sacar el 3r, deberemos sacar el 1r y el 2n antes de poder llegar al 3r.

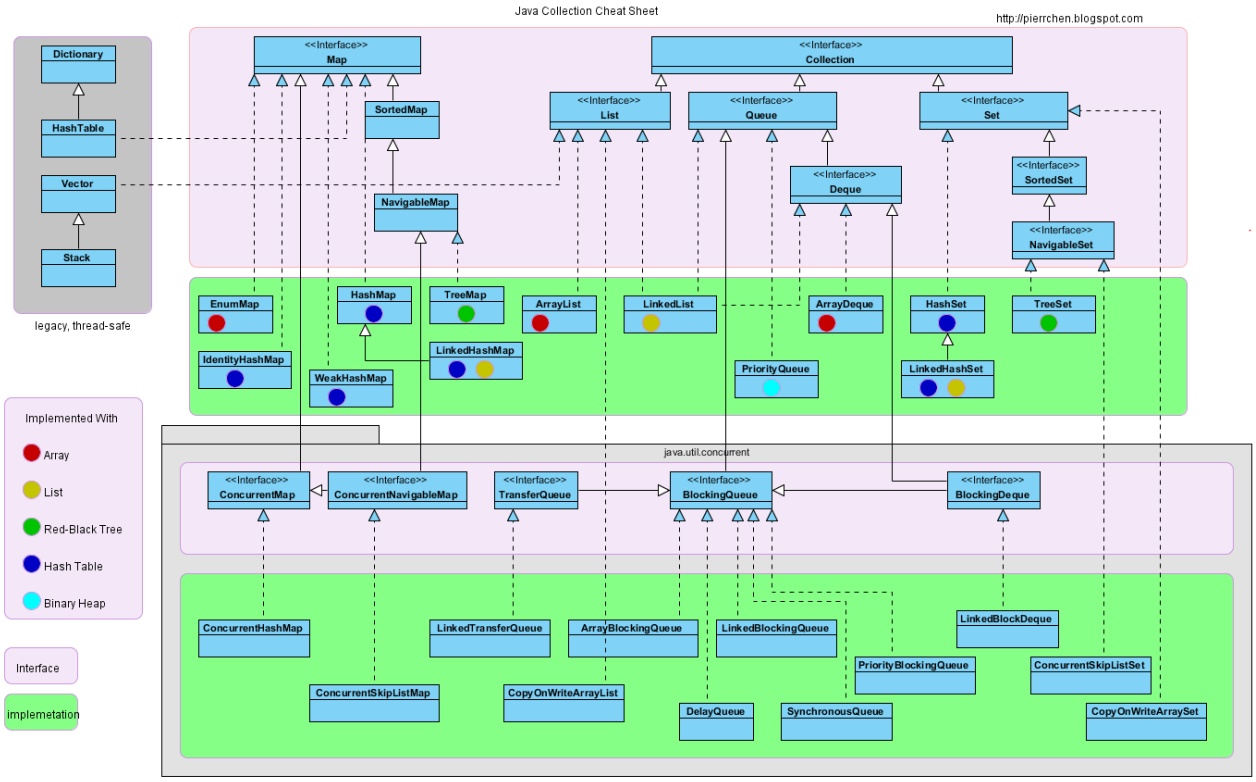
Podemos utilizar la clase Stack que hereda de Vector.

Algunos de los métodos de la interface Queue son los siguientes:

- **push(E item)**: añade un objeto de tipo E en la cima de la pila (top). La pila tiene objetos de tipo E.
- **pop()**: elimina el objeto que está en el top (la cima) de la pila.
- **peek()**: mira al objeto del top de la pila sin borrarlo.
- **empty()**: testea si la pila está vacía.
- **search(Object o)**: devuelve la posición de un objeto en la pila.

Ejemplo de Stack:

```
import java.util.*;

public class StackDemo {

   static void showpush(Stack st, int a) {
      st.push(new Integer(a));
      System.out.println("push(" + a + ")");
      System.out.println("stack: " + st);
   }

   static void showpop(Stack st) {
      System.out.print("pop -> ");
      Integer a = (Integer) st.pop();
      System.out.println(a);
      System.out.println("stack: " + st);
   }

   public static void main(String args[]) {
      Stack st = new Stack();
      System.out.println("stack: " + st);
      showpush(st, 42);
      showpush(st, 66);
      showpush(st, 99);
      showpop(st);
      showpop(st);
      showpop(st);
      try {
         showpop(st);
      } catch (EmptyStackException e) {
         System.out.println("empty stack");
      }
   }
}
```


Java Collection Cheat Sheet
http://pierrchen.blogspot.com
Dictionary
HashTable
Vector
Stack
legacy, thread-safe
<<Interface>>
Map
SortedMap
NavigableMap
<<Interface>>
Collection
<<Interface>>
List
<<Interface>>
Queue
<<Interface>>
Deque
<<Interface>>
Set
<<Interface>>
SortedSet
<<Interface>>
NavigableSet
EnumMap
IdentityHashMap
HashMap
WeakHashMap
TreeMap
LinkedHashMap
ArrayList
LinkedList
PriorityQueue
ArrayDeque
HashSet
LinkedHashSet
TreeSet
Implemented With
Array
List
Red-Black Tree
Hash Table
Binary Heap
Interface
implemetation
java.util.concurrent
<<Interface>>
ConcurrentMap
<<Interface>>
ConcurrentNavigableMap
<<Interface>>
TransferQueue
<<Interface>>
BlockingQueue
<<Interface>>
BlockingDeque
ConcurrentHashMap
ConcurrentSkipListMap
LinkedTransferQueue
CopyOnWriteArrayList
ArrayBlockingQueue
DelayQueue
LinkedBlockingQueue
SynchronousQueue
LinkedBlockDeque
PriorityBlockingQueue
ConcurrentSkipListSet
CopyOnWriteArraySet

3. Interface Set y SortedSet

https://docs.oracle.com/javase/6/docs/api/java/util/Set.html

https://docs.oracle.com/javase/6/docs/api/java/util/SortedSet.html

Sirve para acceder a una conjunto de elementos **sin elementos repetidos** (hay que saber cuándo dos objetos son considerados iguales, para ello se usan **equals()** y **hashcode()**). Puede estar ordenada (**SortedSet = TreeSet**) o no (**Set = HashSet i LinkedHashSet**), y no declara ningún método adicional a los de Collection. Algunos de los métodos de la interface Set son los siguientes:

- **add(Object o)**: añade el objeto pasado por parámetro al Set siempre que éste no exista ya, y devuelve un booleano.
- **clear()**: Elimina a todos los elementos del Set.
- **contains(Object o)**: devuelve true si el Set contiene el objeto pasado por parámetro. Para ello, se compara de forma interna con el método equals (o.equals(x);) o con el método hashCode().
- **isEmpty()**: devuelve true si el Set está vacío.
- **iterator()**: devuelve un iterador
- **remove(Object o)**: elimina el objeto pasado por parámetro si existe y devuelve un booleano.
- **size()**: devuelve un entero que es el número de elementos del Set.

La interface **SortedSet** extiende de la interface **Set** y añade una serie de métodos, entre los que hay que destacar:

- **comparator()**: obtiene el objeto pasado al constructor para establecer el orden; si se emplea el orden natural definido por la interface Comparable, devuelve null;
- **first() / last()**: devuelve el primer o el último elemento del conjunto.

Existen dos implementaciones de conjuntos, como son la clase **HashSet** (proviene de **Set**) y la clase **TreeSet** (proviene de **SortedSet**).

HashSet es la más rápida de las 3, **TreeSet** la más lenta pero es la que se debe usar si necesitas ordenar el set y **LinkedHashSet** si lo que necesitas es un set ordenado por orden de inserción de los elementos en el set.

3.1. HashSet y LinkedHashSet

https://docs.oracle.com/javase/6/docs/api/java/util/HashSet.html

Procede de Set. Un **HashSet** es una estructura de datos que contiene un conjunto de objetos sin orden y sin repetidos. Permite buscar un objeto dentro del conjunto de forma rápida y fácil. Internamente gestiona un array y guarda los objetos utilizando un índice calculado con un código hash del objeto. No se respeta el orden en el que se insertan los elementos y el tamaño del conjunto se adapta dinámicamente a lo que se necesite.

El **LinkedHashSet** internamente gestiona una linked list donde los objetos se van metiendo al final de la lista. Respeta el orden en el que se insertan los elementos.

Hay que saber cuándo dos objetos son considerados iguales, para ello se usan:

- **equals(Object o):boolean**, devuelve true si el objeto que la llama es igual al que pasamos como parámetro (o). Ex: para saber si “y” es igual a “o”: if (y.equals(o) == true){...}

- **hashcode():int**, calcula la clave hash del objeto, 2 objetos con la misma clave indica que son el mismo objeto ya que no pueden haber repetidos.

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){}
  }
}
```

La clase **HashSet** no puede garantizar el orden de iteracción pero la **LinkedHashSet** sí y lo hará en base al orden de inserción.

Ejemplo:

```
HashSet<Dog> dset = new HashSet<Dog>();
dset.add(new Dog(2));
dset.add(new Dog(1));
dset.add(new Dog(3));
dset.add(new Dog(5));
dset.add(new Dog(4));
Iterator<Dog> iterator = dset.iterator();
while (iterator.hasNext()) {
        System.out.print(iterator.next() + " ");
}

//Possible sortida: 5 3 2 1 4 (ningú et pot garantir l'ordre de sortida)


LinkedHashSet<Dog> dset = new LinkedHashSet<Dog>();
dset.add(new Dog(2));
dset.add(new Dog(1));
dset.add(new Dog(3));
dset.add(new Dog(5));
dset.add(new Dog(4));
Iterator<Dog> iterator = dset.iterator();
while (iterator.hasNext()) {
        System.out.print(iterator.next() + " ");
}

//Sortida: 2 1 3 5 4  (l'ordre de sortida és l'ordre d'entrada, FIFO)
```

3.2. TreeSet

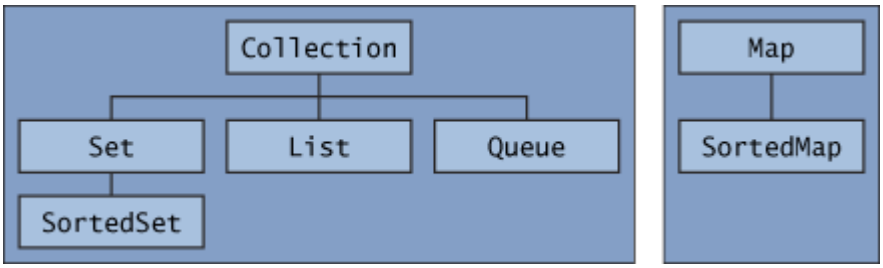
https://docs.oracle.com/javase/6/docs/api/java/util/TreeSet.html

La implementación de **SortedSet** sin duplicados ni objetos NULL pero con orden nos da un **TreeSet**. Guarda los elementos en un árbol.

```java
import java.util.*;

public class SortedSetTest {

   public static void main(String[] args) {

      // Create the sorted set
      SortedSet set = new TreeSet();

      // Add elements to the set
      set.add("b");
      set.add("c");
      set.add("a");

      // Iterating over the elements in the set
      Iterator it = set.iterator();
      while (it.hasNext()) {
         // Get element
         Object element = it.next();
         System.out.println(element.toString());
      }
   }
}
```

4. Interface Map

https://docs.oracle.com/javase/6/docs/api/java/util/Map.html



Un Map es una estructura de datos agrupados en parejas clave/valor; pueden ser considerados como una tabla de dos columnas. La clave debe ser única y se emplea para acceder al valor.

CLAVE	VALOR
ID	número de identificación
actualidad.rt.com	37.48.108.115
google.com	216.58.210.174
MAC	Control de Acceso al Medio
ACS	Aegis Combat System

Map no deriva de Collection, pero sí se pueden ver los Maps como colecciones de claves, de valores o de claves/valores. Algunos de los métodos de la interface Map son los siguientes:

- **clear()**: elimina todos los mapeos del Map.
- **containsKey(Object clave)**: devuelve true si el Map contiene un mapeo igual que el objeto pasado por parámetro (comprueba por la clave del objeto): boolean existe = productos.containsKey(producto);
- **containsValue(Object valor)**: devuelve true si el Map mapea a uno o más claves del objeto valor pasado por parámetro (comprueba por el valor del objeto).
- **get(Object clave)**: devuelve el objeto valor de la clave pasada por parámetro; o null si el Map no encuentra ningún mapeo con esta clave.
- **isEmpty()**: devuelve true si el Map está vacío.
- **put(Clave clave, Valor valor)**: asigna el valor pasado con la clave especificada a otro objeto valor.
- **remove(Object clave)**: elimina el mapeo que tenga la clave pasada por parámetro si existe, devolviendo el valor de dicho mapeo.
- **size()**: devuelve un entero con el número de mapeos del Map.

La interface SortedMap añade métodos similares a los de SortedSet.

Existen tres implementaciones como son las clases **HashMap**, **HashTable** y **TreeMap**.

4.1. HashMap

https://docs.oracle.com/javase/6/docs/api/java/util/HashMap.html

Esta clase mapea claves con valores, pero **no permite claves duplicadas** porque se solaparía el valor. Tanto la clave como el valor pueden ser cualquier tipo de objeto, y ambos **pueden tener el valor null**. Esta clase no garantiza el orden de los elementos ni que no puedan cambiar de orden. HashMap implementa la interfaz Map y extiende de la clase abstracta AbstractMap. Esta clase posee internamente (de la clase AbstractMap) los métodos equals, hashCode y toString. Este tipo de dato no es sincronzado de manera que varias tareas pueden acceder a esta clase en un tiempo determinado (a la vez, ex: una consultando y otra modificando el mapa). Esta es mejor que HashTable.

LinkedHashMap respeta el orden en el que se insertan los elementos.

Ejemplo de HashMap (no hay orden en los elementos del map):

```
import java.util.*;

public class HashMapDemo {

   public static void main(String args[]) {

      // Create a hash map
      HashMap hm = new HashMap();
      // Put elements to the map
      hm.put("Zara", new Double(3434.34));
      hm.put("Mahnaz", new Double(123.22));
      hm.put("Ayan", new Double(1378.00));
      hm.put("Daisy", new Double(99.22));
      hm.put("Qadir", new Double(-19.08));

      // Get a set of the entries
      Set set = hm.entrySet();
      // Get an iterator
      Iterator i = set.iterator();
      // Display elements
      while(i.hasNext()) {
         Map.Entry me = (Map.Entry)i.next();
         System.out.print(me.getKey() + ": ");
         System.out.println(me.getValue());
      }
      System.out.println();
      // Deposit 1000 into Zara's account
      double balance = ((Double)hm.get("Zara")).doubleValue();
      hm.put("Zara", new Double(balance + 1000));
      System.out.println("Zara's new balance: " +
      hm.get("Zara"));
   }
}
```

Salida del programa:

```
Daisy: 99.22
Ayan: 1378.0
Zara: 3434.34
Qadir: -19.08
Mahnaz: 123.22

Zara's new balance: 4434.34
```

Ejemplo de LinkedHashMap (ordenado en función del orden de entrada en el map):

```java
import java.util.*;
public class LinkedHashMapDemo {

   public static void main(String args[]) {
      // Create a hash map
      LinkedHashMap lhm = new LinkedHashMap();

      // Put elements to the map
      lhm.put("Zara", new Double(3434.34));
      lhm.put("Mahnaz", new Double(123.22));
      lhm.put("Ayan", new Double(1378.00));
      lhm.put("Daisy", new Double(99.22));
      lhm.put("Qadir", new Double(-19.08));

      // Get a set of the entries
      Set set = lhm.entrySet();

      // Get an iterator
      Iterator i = set.iterator();

      // Display elements
      while(i.hasNext()) {
         Map.Entry me = (Map.Entry)i.next();
         System.out.print(me.getKey() + ": ");
         System.out.println(me.getValue());
      }
      System.out.println();

      // Deposit 1000 into Zara's account
      double balance = ((Double)lhm.get("Zara")).doubleValue();
      lhm.put("Zara", new Double(balance + 1000));
      System.out.println("Zara's new balance: " + lhm.get("Zara"));
   }
}
```

Salida del programa:

```
Zara: 3434.34
Mahnaz: 123.22
Ayan: 1378.0
Daisy: 99.22
Qadir: -19.08

Zara's new balance: 4434.34
```

4.2. HashTable

https://docs.oracle.com/javase/6/docs/api/java/util/Hashtable.html

Es una clase que implementa Map y que almacena pares del tipo clave/valor en una tabla de dispersión. Al emplearla se proporciona un objeto que sirve como clave y otro como valor (vinculando el valor a la clave). Tanto la clave como el valor **no pueden tener el valor null** Su insercion y búsqueda es rápida. Un ejemplo sería un listín de teléfonos, en el que dado un nombre se proporciona un teléfono. Este tipo de dato es sincronzado de manera que solo una tarea puede acceder a esta clase en un tiempo determinado.

Ejemplo:

```java
import java.util.*;

public class HashTableDemo {

   public static void main(String args[]) {
      // Create a hash map
      Hashtable balance = new Hashtable();
      Enumeration names;
      String str;
      double bal;

      balance.put("Zara", new Double(3434.34));
      balance.put("Mahnaz", new Double(123.22));
      balance.put("Ayan", new Double(1378.00));
      balance.put("Daisy", new Double(99.22));
      balance.put("Qadir", new Double(-19.08));

      // Show all balances in hash table.
      names = balance.keys();
      while(names.hasMoreElements()) {
         str = (String) names.nextElement();
         System.out.println(str + ": " +
         balance.get(str));
      }
      System.out.println();
      // Deposit 1,000 into Zara's account
      bal = ((Double)balance.get("Zara")).doubleValue();
      balance.put("Zara", new Double(bal+1000));
      System.out.println("Zara's new balance: " +
      balance.get("Zara"));
   }
}
```

Salida del programa:

```
Qadir: -19.08
Zara: 3434.34
Mahnaz: 123.22
Daisy: 99.22
Ayan: 1378.0

Zara's new balance: 4434.34
```

4.3. TreeMap

https://docs.oracle.com/javase/6/docs/api/java/util/TreeMap.html

Esta clase implementa SortedMap y se basa en una árbol binario. Ordena sus elementos de forma natural, por ejemplo: si la clave son integers, los ordenara de menor a mayor.

Si implementara la interface SortedMap (ex: SortedMap sm = new TreeMap();) podremos implementar los métodos usados para el ordenamiento de los elementos del árbol.

Ejemplo:

```
import java.util.*;

public class TreeMapDemo {

   public static void main(String args[]) {
      // Create a hash map
      TreeMap tm = new TreeMap();
      // Put elements to the map
      tm.put("Zara", new Double(3434.34));
      tm.put("Mahnaz", new Double(123.22));
      tm.put("Ayan", new Double(1378.00));
      tm.put("Daisy", new Double(99.22));
      tm.put("Qadir", new Double(-19.08));

      // Get a set of the entries
      Set set = tm.entrySet();
      // Get an iterator
      Iterator i = set.iterator();
      // Display elements
      while(i.hasNext()) {
         Map.Entry me = (Map.Entry)i.next();
         System.out.print(me.getKey() + ": ");
         System.out.println(me.getValue());
      }
      System.out.println();
      // Deposit 1000 into Zara's account
      double balance = ((Double)tm.get("Zara")).doubleValue();
      tm.put("Zara", new Double(balance + 1000));
      System.out.println("Zara's new balance: " +
      tm.get("Zara"));
   }
}
```

Map.Entry me = (Map.Entry)i.next(); // **me** es declarado como 1 entrada de tipo Map (por lo que tendrá una clave y un valor) y le asignamos el valor (clave + valor) que a cogido el iterador **i** (del que nos aseguramos que sea del tipo "entrada de tipo Map").

Salida del programa:

```
Ayan: 1378.0
Daisy: 99.22
Mahnaz: 123.22
Qadir: -19.08
Zara: 3434.34

Zara's new balance: 4434.34
```