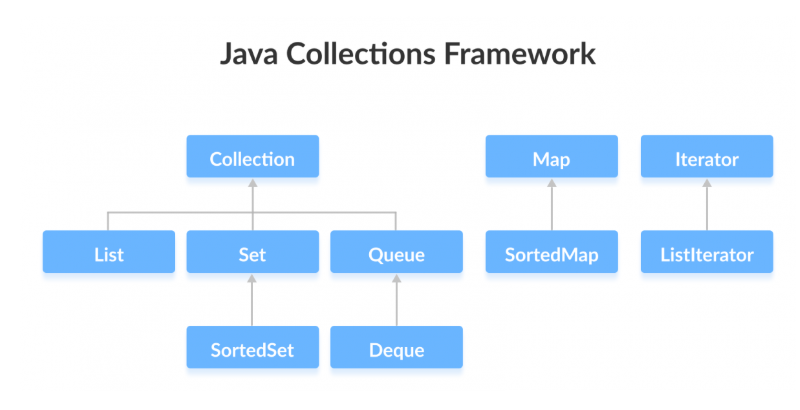
**Introducción a la colecciones**

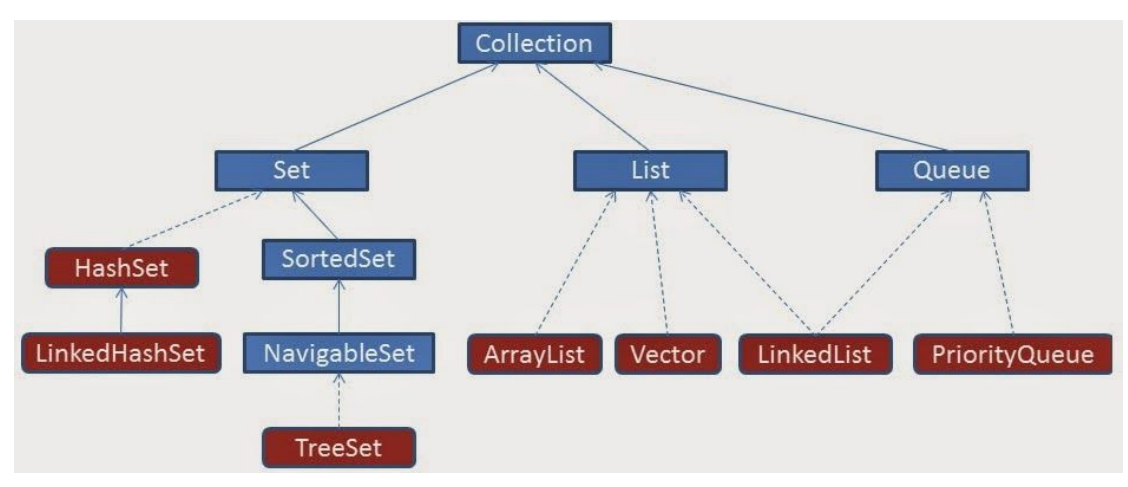
Son el modo de agrupar objetos (otras colecciones, objetos o primitivos), de manera similar al concepto de Array.

La interfaz **collection** es la super interfaz de donde heredan las interfaces utilizadas para el manejo de las colecciones. Es la interfaz raíz de las interfaces del JAVA COLLECTION FRAMEWORK. Java tiene mas de 30 collections. Ej: listas, sets y Queues. Los mapas no forman parte de la interfaz Collection.

Partimos del árbol genérico JFC:



Y luego nos introducimos en Collection



**INTERFAZ SET**

Define una colección que no puede contener elementos duplicados. Contiene únicamente los métodos heredados de Collection añadiendo la restricción de que los elementos duplicados están prohibidos.

Las implementaciones mas conocidas son **HashSet**, **LinkedHashSet** y **TreeSet**.

En el caso de la interfaz **SET** voy a tener que definir una forma por la cual voy a establecer cuando un objeto es duplicado.

Vemos la siguiente url la cual contiene información muy útil relacionada a la interfaz **collection.**

<https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html>

Aquí nos define ciertos métodos que son importantes, por ejemplo el método **add()** (me permite agregar un objeto a una colección)**.**

Vemos estos otros métodos:

**isEmpty(), remove(Object o), size()**

Las **implementaciones** mas conocidas son:

- **HashSet:** Es la que mejor performance tiene dentro de la Interfaz set. Almacena todos los elementos en una tabla **Hash.** En donde el **orden no** está **garantizado en determinadas situaciones**.

Tiene una muy buena performance.

- **TreeSet**: Almacena los elementos en una estructura de árbol **rojo-negro.** Es un poco mas lenta que los Set. Pero me provee orden de los elementos. No admite duplicados

- **LinkedHashSet:** Esta implementada como una HashTable con una LinkedList (un enlace hacia una lista de elementos) asociada.

TreeSet<Producto> p1 = new TreeSet();

Los **generics** me permiten especificar el tipo de dato que le voy a colocar.

**CLASE\_06\_EJEMPLO\_1\_HASHSET**

**Codigo Netbeans:**

****

**Codigo Eclipse (Maven):**

****

En el codigo previo vemos que se comparan dos strings iguales. Y el size sigue siendo 2. Esto es porque el <String> que utilizamos como genérico ya tiene incorporado el método **equals** el cual compara si dos objetos son iguales. En este caso los comparó y detecto que eran iguales. Y deja un size de 2, no de 3.

**HashSet** y **TreeSet** son implementaciones diferentes de la interfaz Set. Tienen los mismos métodos pero implementaciones bien diferentes.

**Pros y contras:**

**TreeSet:** Baja performance a medida que incorporo elementos. Pero me respeta el orden.

**HashSet:** Alta performance pero no garantiza el orden.

En ambos casos no acepta valores duplicados como vimos en el ejercicio.

**CLASE\_06\_EJEMPLO\_2\_HASHSET**

**Codigo Netbeans:**



**Codigo Eclipse:**

****

**CLASE\_06\_EJEMPLO\_3\_HASHSET**

En este ejemplo se adhieren dos objetos de tipo persona a un Set (HashSet). Ambos objetos tienen el mismo valor. En principio vemos como me da el size el valor 2. Siendo que no debería admitir duplicados.

Para eso implementamos el método **equals().**

El método **add()** le pregunta a la colección si ya existe el objeto. Y para averiguar eso Java acude internamente al método *equals* del *genérico*. En este caso de *persona*.

Tambien agregamos el método hashCode() ya que los Set del tipo Hash tienen una doble validación.

**Codigo Netbeans:**

****

**Codigo Eclipse:**

****

**LIST**

<https://docs.oracle.com/javase/8/docs/api/java/util/List.html>

**Interfaz List (hereda también de collection)**

La **interfaz** List, define una sucesión de elementos, los cuales se basan en un **indice**. A diferencia de la interfaz Set, si **admite** elementos **duplicados**.

Las implementaciones mas conocidas son **ArrayList** y **LinkedList.**

Las operaciones mas conocidas con List son:

* *Acceso posicional a elementos*. Ej: Quiero un elemento que este en la posición 3.

Un **set** se recorre secuencialmente. Una **List** se puede traer un valor particular.

* *Operaciones sobre rangos*.

**Metodos mas importantes:**

**add():** esta sobrecargado. Por un lado puedo agregar elementos al final de la lista. Y a su vez, puedo agregar un elemento en una posición.

**get()**: retorna un objeto posicionado en un índice.

**indexOf():** devuelve el índice de un objeto. Si no existe me devuelve un -1.

**remove():** puedo remover a partir de un índice.

**size():** me devuelve el nro de elementos de la lista.

**subList():** Creacion de sublistas a partir de posiciones.

**Implementaciones de List**

**Class ArrayList:** Mejor performance. Utiliza internamente un array. Tamaño expandible.

**Class LinkedList**: Mas lenta que ArrayList. Esta doblemente enlazada. Cada elemento tiene un puntero hacia el próximo y hacia el anterior.

**Pros y Contras:**

**ArrayList** trabaja mas performante. Si tengo que sacar y poner elementos de la mitad de la lista me va a convenir un **LinkedList (puntero)**.

**CLASE\_06\_EJEMPLO\_4\_ARRAYLIST**

**Codigo Netbeans:**

****

**Codigo Eclipse:**



**CLASE\_06\_EJEMPLO\_5\_ARRAYLIST**

1ro: probamos agregar una persona y vemos como se duplica (situación actual).

2do: realizamos la modificación necesaria dentro de agregaPersonas() para evitar la duplicación.



**Codigo Netbeans:**

****

**Codigo Eclipse:**

****

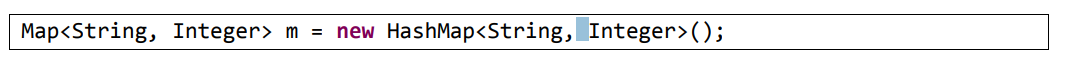
**Continuamos por el ejercicio //D. Comparable => sort() del código**

**Interface Map (k, v) Map (1, “Hola Mundo”) (2, “Chau Mundo”),**

No admite duplicados. Algunas de sus implementaciones son **TreeMap** class (ordenada), **HashMap** class (no ordenada).

Como entre las claves no puede haber elementos duplicados (una clave no puede estar asociada con más de un valor), las claves forman un conjunto (Set). Sin embargo sí que puede haber valores duplicados, por los que estos están en una Collection. Por tanto, un Map está definido por un conjunto de claves, una colección de valores y un conjunto de pares clave-valor (también llamadas **entradas**), que son realmente los elementos de los que está compuesto.

*Sintaxis inicial: (“Hola”, 2) (“Hola1”, 2) (“Hola2”, 2)*

****

<https://docs.oracle.com/javase/7/docs/api/java/util/Map.html>

Algunos métodos utilizados en un Map son:

**clear():** elimina todos los elementos (pares o entradas) del Map.

**containsKey**(): devuelve true si el Map contiene la clave especificada.

**containsValue**(Object value): devuelve true si una o mas claves del Map tienen asociadas el valor especificado.

Como también:

get(Object key), isEmpty(), keySet(), put(K key, V value), remove(Object key), size(), values(), entrySet(), putAll(Map<? Extends K, ¿ extends V> m).



**Abrir con Google Chrome o Acrobat**

<http://www.lsi.us.es/docencia/get.php?id=7245>

Por ejemplo, si tenemos una entrada (par) p que asocie “II” con 2, p.getKey() devuelve la cadena “II” y p.getValue() devuelve el entero 2; si se escribe p.setValue(3), devuelve el entero 2 y modifica el valor asociado con la clave dándole el valor 3. El **toString()** de las entradas es de la forma clave=valor, de modo que la representación como cadena del par resultante sería II=3.

**CLASE\_06\_EJEMPLO\_6\_MAP\_HASHMAP**

**Codigo Netbeans:**



**Codigo Eclipse:**

****

***Ejemplo 1***=> vemos un método en el que le envio un String y me devuelve el Map<character, integer> por el cual me indica cuantas veces aparece cada carácter en una determinada palabra.

***Ejemplo 2*** => vemos un método al cual le envio una Lista de Palabras List<String> palabras, y me devuelve un Map<String, List<Integer>> . Es decir, se quiere obtener un índice de las posiciones que ocupan las cadenas de la lista palabras mediante una lista de enteros.

**SortedMap**

El tipo SortedMap es un subtipo de Map en el que el conjunto de las claves está ordenado. La clase que implementa SortedMap es **TreeMap**.

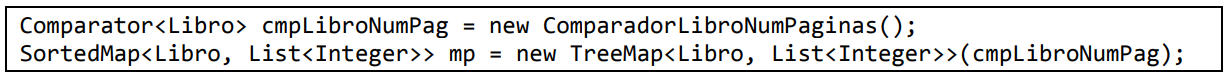
Es necesario que el tipo de las claves tenga un orden natural (es decir, que implemente **Comparable**) o que se proporcione un orden alternativo mediante un **Comparator**.

Por tanto, una inicialización de SortedMap como la siguiente:

“clave 1” = [1, 2, 3, 4, 5 ] , “clave 2” = [10, 20 , 30, 40, 50]

****

Crea un SortedMap donde las claves son de tipo String y están ordenadas por el orden natural. Para crear un SortedMap por un orden alternativo debemos invocar al constructor pasándole como argumento un comparador:

****

Dos claves c1 y c2 son iguales para un SortedMap si compare(c1,c2) == 0.

*Algunos de los métodos mas importantes son:*

**firstKey(), lastkey(), headMap(K toKey), subMap(K fromKey, K toKey), tailMap(K fromKey), comparator():** Devuelve el comparador utilizado para ordenar las claves, o null si se usa el orden natural.

Tambien recibe los métodos de ­Map por herencia: **keySet**()=> devuelve un set; **entrySet**=> devuelve un Set; **values()**=> devuelve una collection.

**CLASE\_06\_EJEMPLO\_7\_SORTEDMAP\_TREEMAP**

**Codigo Netbeans:**

****

**Codigo Eclipse:**

****

**CLASE\_06\_EJEMPLO\_8\_MAP\_HASHMAP**

**Codigo Netbeans:**

****

**Codigo Eclipse:**

****

**INTERFACE COMPARATOR vs COMPARABLE**

**Comparable (Orden Natural):** Nos obliga a implementar el método **compareTo**(Object o). La implementación de este método se hace para indicar el orden natural de los elementos de esa clase.

**Orden natural = comparable = compareTo(Object o)**

**Orden no natural = comparator = compare(Object o1 , Object o2)**

**Comparator:** Nos permite comparar 2 elementos en una colección. Nos obliga a implementar el método **compare**(Object o1, Object o2).

Cuando el **orden natural (edad, nombre)** no nos sirve y queremos ordenar por otro criterio (altura) debemos acudir a la interfaz **Comparator** y aplicar el metodo **compare(Object o1, Object o2).**

**CLASE\_06\_EJEMPLO\_9\_COMPARABLE**

En la clase Persona vamos a definir un orden para poder ordenar colecciones de ese elemento. Utilizaremos ArrayList para contener una colección de objetos.

No hemos detallado los métodos set correspondientes a cada propiedad porque trataremos de centrarnos en el código que resulta de interés para este caso concreto.

Donde vemos cómo efectivamente la lista de Personas ha sido ordenada usando la invocación Collections.sort pasando como parámetro la colección y el orden utilizado en esta invocación es el orden natural definido en su clase, es decir, por el nombre de cada Persona. Definido con el método compareTo (Object o)

**Codigo Netbeans:**



**Codigo Eclipse:**

****

**CLASE\_06\_EJEMPLO\_9\_COMPARATOR**

**Ejercicio para hacer entre todos:**

1. Crear una clase Persona [atributos: id, nombre, altura; Constructor: 3 parametros; toString que devuelva los 3 parametros con estética y salto de línea al inicio.
2. Crear una clase Principal la cual implemente Comparator<Persona>. E implementar el metodo compare(Persona pers1, Persona pers2) el cual genere un ordenamiento por id creciente.
3. En el main de la clase Principal crear un arrayList<Persona> el cual agregue tres personas.
4. Ordenar la collection a través del metodo Collections.sort(lista, criterioDeOrdenamiento).
5. Mostrar la collection ordenada.

**Codigo Eclipse:**



Implementamos Interfaz **Comparator** cuando no queremos ordenar en base al orden natural.

Ver clase **Principal u otra clase que implemente Comparator.** El método sort cuando llama a compare permite comparar todas las alturas, id, etc entre si. Ordenando de esta forma un ordenamiento de menor a mayor altura, id, etc. Si se quiere ordenar de mayor a menor se debe cambiar el signo del método compare (Persona o, Persona o1).