**Las colecciones**

Las colecciones en Java son estructuras de datos dinámicas que permiten almacenar listas o colecciones de datos de un tipo determinado. Además, proporcionan una API muy completa de operaciones para poder realizar sobre toda la colección. Se pueden considerar como una evolución de los *arrays* ya que éstos son estáticos y tienen una operativa más básica. Además, los diferentes tipos de colecciones permiten afinar el comportamiento de cada una de ellas de forma que podremos elegir la que más nos convenga para cada caso. Existen, por ejemplo, colecciones de datos en las que los elementos están automáticamente ordenados en todo momento (TreeSet) u otras que no admiten elementos repetidos (HashSet). Otras estructuras mantienen el orden de los elementos (LinkedList o ArrayList) mientras que en otras esa información no es relevante (HashMap)

Se les conoce con el nombre de genéricos porque son estructuras ya disponibles en la API de Java pero de forma genérica, por lo que se les debe pasar un tipo de datos para convertirlos en una colección de un tipo determinado. Asi, en el momento de la declaración tendremos que indicar el tipo de datos que almacenará dicha colección.

LinkedList<Libro> listaLibros = new LinkedList<Libro>();

*// Desde la verisón 8 de Java podemos utilizar el 'diamante'*

*// y omitir el tipo al instanciar*

LinkedList<Libro> listaLibros = new LinkedList<>();

Es posible no indicar el tipo de dato de la colección, creando lo que se conoce como una colección *raw*, aunque no está recomendado porque evita que el compilador de Java pueda hacer chequeos de tipo en tiempo de compilación, lo que facilita mucho el trabajo con este tipo de estructuras.

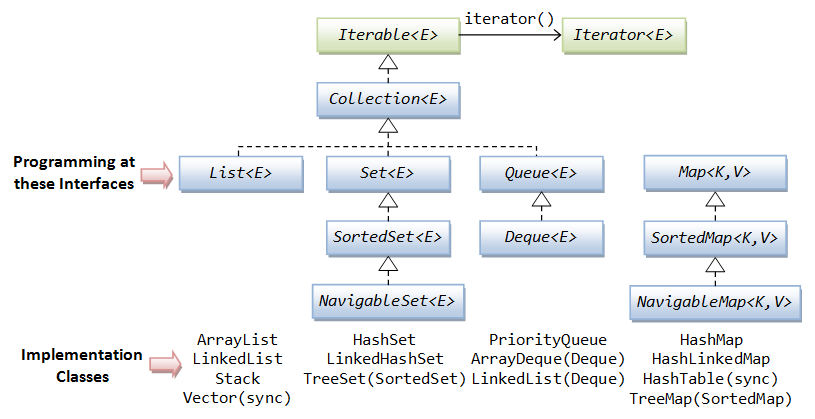
Existen numerosos tipos de datos para almacenar colecciones con diferentes características. En esta sección veremos algunas de las más utilizadas:  LinkedList, ArrayList y HashMap.

También veremos cómo funcionan las pilas y las colas, para las que Java proporciona los tipos Stack y Queue , respectivamente.

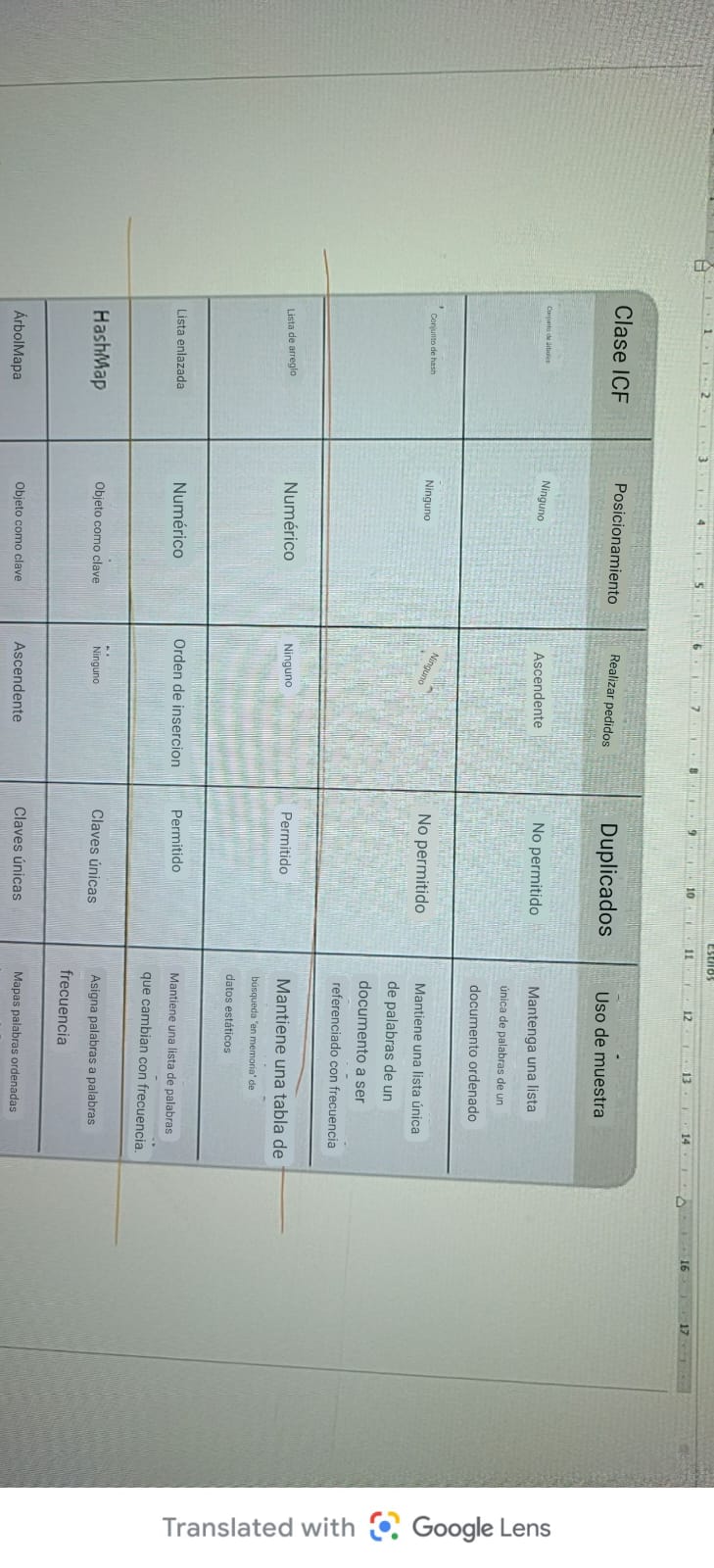
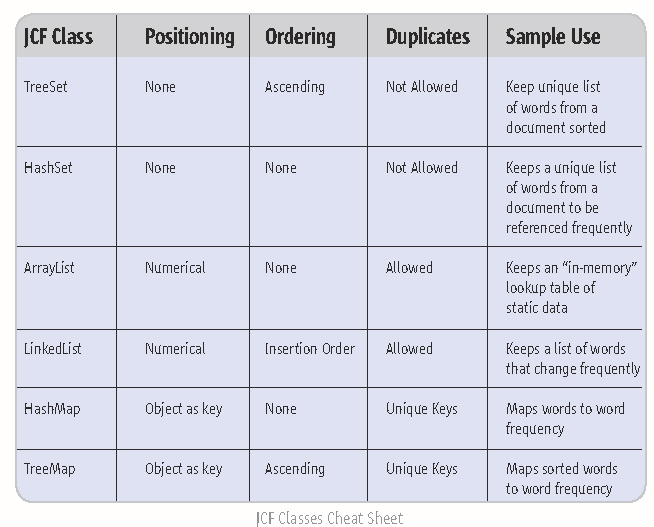
Además, existe una serie de interfaces y clases como List, Collection o Arrays de la que extienden o implementan estas estructuras de colección, lo que permite que además existan operaciones que permiten que se puedan hacer operaciones que combinen distintos tipos de colección y otras estructuras de datos como los arrays.

**A tener en cuenta cuando se trabaja con colecciones**

* Estudiar bien qué tipo de colección interesa más para cada caso (no usar siempre ArrayList porque sí)
* Implementar/Extender los métodos de la clase Object en las clases que diseñe:
  + hashCode()
  + equals()
  + toString()
* Aprender a utilizar algunos interfaces interesantes para usar con colecciones:
  + Comparator
  + Comparable

[](https://java.codeandcoke.com/_detail/apuntes:interfaces_collections.png?id=apuntes%3Acolecciones)**Figure 1:** Tipos de colecciones

**Algunas diferencias**

[](https://java.codeandcoke.com/_detail/apuntes:comparison.jpg?id=apuntes%3Acolecciones)**Figure 2:** Colecciones en Java

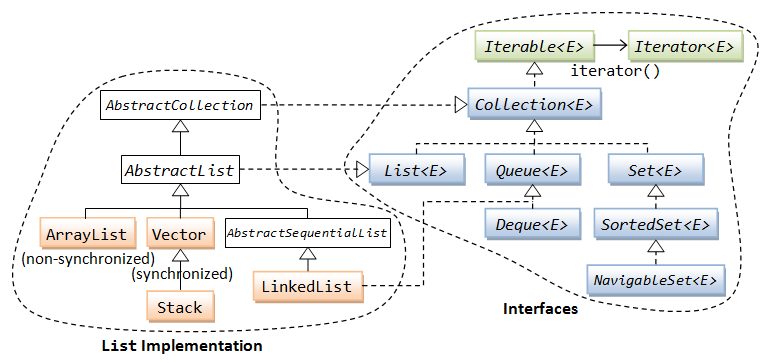
**Interface Collection**

Es la interface padre de todas las colecciones Java

Define una serie de método que cualquier colección tiene que implementar:

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

**Listas**

[](https://java.codeandcoke.com/_detail/apuntes:list.png?id=apuntes%3Acolecciones)**Figure 3:** Colecciones en Java

**List**

Es la interface padre de todas las colecciones de tipo Lista:

* Define una serie de método que cualquier colección tiene que implementar
  + E get(i)
  + int indexOf(e)
  + void sort(Comparator<? super E> c)

**ArrayList**

La clase ArrayList permite definir arrays dinámicos. Son colecciones de datos dinámicas con el acceso y funcionamiento de un array, puesto que sólo es posible recuperar cada elemento a partir de la posición del mismo.

* Implementa la interface List
* Es el tipo de colección más genérica
* Define una colección de elementos sin ninguna característica especial
* Los elementos se pueden obtener a partir de la posición donde fuera añadidos

Entre las operaciones que se pueden realizar con un \verb ArrayList las más comunes son las siguientes:

**Añadir un elemento**

ArrayList<Libro> listaLibros = new ArrayList<>();

. . .

Libro libro = new Libro(. . .);

listaLibros.add(libro);

**Añadir toda una colección al final**

ArrayList<Libro> listaLibros = new ArrayList<>();

ArrayList<Libro> otraListaLibros = new ArrayList<>();

. . .

listaLibros.addAll(otraListLibros);

**Obtener un elemento**

Libro unLibro = listaLibros.get(4);

**Eliminar un libro**

Libro libroEliminado = listaLibros.remove(4);

**Eliminar todos los elementos**

listaLibros.clear();

**Obtener el tamaño de la colección (número de elementos)**

int tamano = listaLibros.size();

**Obtener un array estáticos con los elementos de la colección**

Libro[] libros = listaLibros.toArray();

**LinkedList**

La clase LinkedList es una lista doblemente enlazada, que permite almacenar una colección de objetos, obtenerlos, eliminarlos y algunas operaciones para acceder a los mismos. Acceder al primer y último elemento, entre otros, son operaciones muy directas en este tipo de colección.

* Implementa la interface List
* Se utiliza para definir estructuras *FIFO* (First In First Out), también conocidas como colas
* Define métodos que permiten, directamente, acceder/eliminar el primer elemento de una lista (el primer elemento de la cola)

Entre las operaciones que se pueden realizar con un LinkedList las más comunes son las siguientes:

**Añadir un elemento**

LinkedList<Libro> listaLibros = new LinkedList<>();

. . .

Libro libro = new Libro(. . .);

listaLibros.add(libro);

**Añadir un elemento al principio**

LinkedList<Libro> listaLibros = new LinkedList<>();

. . .

Libro libro = new Libro(. . .);

listaLibros.addFirst(libro);

**Añadir un elemento al final**

LinkedList<Libro> listaLibros = new LinkedList<>();

. . .

Libro libro = new Libro(. . .);

listaLibros.addLast(libro);

**Añadir toda una colección al final**

LinkedList<Libro> listaLibros = new LinkedList<>();

LinkedList<Libro> otraListaLibros = new LinkedList<>();

. . .

listaLibros.addAll(otraListLibros);

**Obtener un elemento**

Libro unLibro = listaLibros.get(4);

**Obtener el primer/último elemento**

Libro primerLibro = listaLibros.getFirst();

Libro ultimoLibro = listaLibros.getLast();

**Eliminar (y obtener) el primer/último elemento**

Libro primerLibro = listaLibros.removeFirst();

Libro ultimoLibro = listaLibros.removeLast();

**Eliminar todos los elementos**

listaLibros.clear();

**Obtener el tamaño de la colección (número de elementos)**

int tamano = listaLibros.size();

**Obtener un array estático con los elementos de la colección**

Libro[] libros = listaLibros.toArray();

**Vector**

* Implementa la interface List
* Se trata de la misma implementación que ArrayList pero sincronizada para su uso en entornos concurrentes.

En el contexto de las colecciones en Java, la ***sincronización*** se refiere a la capacidad de hacer que una operación sea segura para su ejecución en entornos concurrentes, es decir, cuando ***múltiples hilos de ejecución acceden y modifican una estructura de datos compartida al mismo tiempo***. En este caso, cuando se menciona que un Vector está sincronizado, significa que está diseñado para garantizar la coherencia y consistencia de los datos en escenarios donde múltiples hilos pueden acceder y modificar la lista al mismo tiempo.

Cuando se utiliza un Vector en un entorno concurrente, todas las operaciones en el Vector están protegidas por bloqueos internos, lo que significa que solo un hilo puede acceder a la lista en un momento dado. Esto evita que ocurran situaciones de carrera donde múltiples hilos intentan modificar la lista simultáneamente, lo que podría llevar a resultados inesperados o inconsistencias en los datos.

***La sincronización se logra mediante el uso de mecanismos como los monitores de Java (también conocidos como "locks" o bloqueos) para garantizar que solo un hilo pueda ejecutar una sección crítica de código a la vez***. En el caso de un Vector, todas las operaciones, como agregar, eliminar o modificar elementos, están protegidas por estos mecanismos de sincronización.

Sin embargo, es importante tener en cuenta que aunque la sincronización garantiza la seguridad en entornos concurrentes, también puede tener un impacto en el rendimiento, ya que puede introducir cierta sobrecarga debido a la necesidad de adquirir y liberar bloqueos. En aplicaciones donde no se necesite concurrencia, se recomienda utilizar ArrayList, ya que no tiene la sobrecarga asociada con la sincronización.

*¿es posible sincronizar un arrayList?*

* un ArrayList en Java no es sincronizado. Sin embargo, si necesitas hacer que un ArrayList sea sincronizado para su uso en entornos concurrentes, puedes hacerlo mediante la creación de una instancia de ArrayList y luego envolviéndola en una colección sincronizada utilizando Collections.synchronizedList().

**// Crear un ArrayList normal**

ArrayList<String> listaNoSincronizada = new ArrayList<>();

**// Envolver el ArrayList en una lista sincronizada**

List<String> listaSincronizada = Collections.synchronizedList(listaNoSincronizada);

// Ahora la listaSincronizada se puede utilizar de forma segura en entornos concurrentes

**Stack**

Las Pilas son colecciones de datos donde éstos se colocan *apilándolos* y sólo puede ser retirado el elemento que se encuentra encima de la pila. Siguen el principio [LIFO](https://es.wikipedia.org/wiki/Last_in,_first_out) (Last In First Out) donde el último elemento en llegar a la colección es el primero en salir de la misma.

* Implementa la interface List
* Permite manejar una lista como si se tratara de una estructura *LIFO* (Last In First Out), también conocidas como pilas.
* Define métodos (push y pop) para coger y dejar directamente del final de la lista (lo que sería la parte alta de la pila)

**Añadir un elemento**

Stack<Libro> pilaLibros = new Stack<>();

Libro libro = new Libro(. . .);

pilaLibros.push(libro);

**Retirar un elemento de la pila**

Libro libro = pilaLibros.pop();

**Obtener, sin retirar, el elemento de lo alto de la pila**

Libro libro = pilaLibros.peek();

**Obtener la posición de un elemento (posicion >= 1)**

int posicion = pilaLibros.search(libro);

Stack<Libro> pilaLibros = new Stack<>();

// Agregar algunos libros a la pila

pilaLibros.push(new Libro("Don Quijote de la Mancha", "Miguel de Cervantes"));

pilaLibros.push(new Libro("Cien años de soledad", "Gabriel García Márquez"));

pilaLibros.push(new Libro("El Señor de los Anillos", "J.R.R. Tolkien"));

pilaLibros.push(new Libro("1984", "George Orwell"));

// Crear un libro para buscar en la pila

Libro libroBuscado = new Libro("Cien años de soledad", "Gabriel García Márquez");

// Buscar la posición del libro en la pila

int posicion = pilaLibros.search(libroBuscado);

if (posicion != -1) {

System.out.println("El libro se encuentra en la posición " + posicion + " de la pila.");

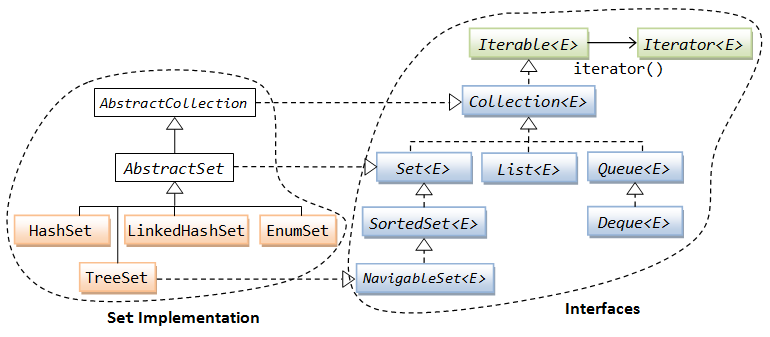
} else {

System.out.println("El libro no se encuentra en la pila.");

}

**Comprobar el tamaño de la pila**

int tamano = pilaLibros.size();

**Sets**[](https://java.codeandcoke.com/_detail/apuntes:set.png?id=apuntes%3Acolecciones)

**Figure 4:** Colecciones en Java

**Set**

* Por definición no acepta elementos duplicados
* No define métodos nuevos, solamente implementa Collection e impide que existan elementos duplicados

**HashSet**

* Rendimiento constante para las operaciones más habituales
* No garantiza el orden de los elementos de la colección

Ejemplo:

// Crear un HashSet

HashSet<String> conjunto = new HashSet<>();

// Agregar elementos al conjunto

conjunto.add("Manzana");

conjunto.add("Banana");

conjunto.add("Fresa");

// Eliminar un elemento del conjunto

conjunto.remove("Banana");

// Actualizar un elemento en el conjunto (eliminar y agregar)

conjunto.remove("Fresa");

conjunto.add("Pera");

// Buscar un elemento en el conjunto

String elementoBuscado = "Manzana";

boolean encontrado = conjunto.contains(elementoBuscado);

// Imprimir el resultado de la búsqueda

if (encontrado) {

System.out.println("El elemento '" + elementoBuscado + "' fue encontrado en el conjunto.");

} else {

System.out.println("El elemento '" + elementoBuscado + "' no fue encontrado en el conjunto.");

}

// Imprimir el conjunto después de eliminar y actualizar elementos

System.out.println("Conjunto después de eliminar y actualizar:");

for (String elemento : conjunto) {

System.out.println(elemento);

}

**LinkedHashSet**

Mantiene una lista doblemente enlazada por lo que puede obtenerse un iterador con el orden en que los elementos fueron añadidos a la colección

**EnumSet**

Es un Set especializado para enumeraciones • El rendimiento es ligeramente inferior al de HashSet

EnumSet<DiasDeLaSemana> conjunto = EnumSet.allOf(DiasDeLaSemana.class);

// Imprimir el conjunto

System.out.println("Conjunto de días de la semana:");

for (DiasDeLaSemana dia : conjunto) {

System.out.println(dia);

}

}

// Enumeración de días de la semana

enum DiasDeLaSemana {

LUNES, MARTES, MIERCOLES, JUEVES, VIERNES, SABADO, DOMINGO

}

En este ejemplo, creamos un **EnumSet** que contiene todos los elementos de la enumeración **DiasDeLaSemana** utilizando el método **allOf()**. Luego, recorremos el conjunto e imprimimos cada día de la semana.

**TreeSet**

* Garantiza el orden de los elementos
* Sus elementos se mantienen siempre ordenados de forma natural o por un Comparator que se proporcione en su constructor
* Su rendimiento es el menor de todos los Sets

TreeSet es una implementación comúnmente utilizada de la interfaz SortedSet en Java. Se utiliza en situaciones donde se requiere mantener un conjunto de elementos ordenados, ya sea de manera natural o utilizando un comparador personalizado. Aquí hay algunas situaciones típicas en las que TreeSet puede ser útil:

Ordenamiento natural de elementos: Cuando se necesita mantener un conjunto de elementos ordenados según su valor natural. Por ejemplo, si tienes un conjunto de números o cadenas y deseas que estén en orden ascendente o descendente.

Ordenamiento personalizado: Cuando se necesita mantener un conjunto de elementos ordenados según un criterio personalizado definido por un comparador. Esto es útil cuando los elementos no son naturalmente comparables o cuando se necesita un orden específico que no está basado en el valor natural de los elementos.

Eliminación eficiente de elementos duplicados: Debido a que TreeSet no permite elementos duplicados y mantiene los elementos ordenados, puede ser útil para eliminar rápidamente duplicados de una colección de elementos mientras se mantiene el orden.

Búsqueda, inserción y eliminación eficientes: TreeSet utiliza un árbol binario de búsqueda balanceado para almacenar los elementos, lo que proporciona un tiempo de búsqueda, inserción y eliminación eficiente (O(log n)) en comparación con otras estructuras de datos que ofrecen operaciones más lentas en conjuntos grandes de elementos.

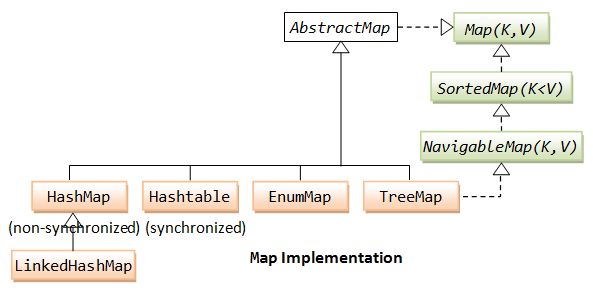
Iteración en orden: TreeSet proporciona iteradores que recorren los elementos en orden ascendente o descendente, lo que puede ser útil en situaciones donde se necesita procesar los elementos en un orden específico.

En resumen, TreeSet es útil en una variedad de situaciones donde se necesita mantener un conjunto de elementos ordenados y realizar operaciones eficientes de búsqueda, inserción y eliminación. Se utiliza en una amplia gama de aplicaciones, como algoritmos de ordenamiento, estructuras de datos para mantener listas ordenadas y en situaciones donde se necesita garantizar un orden específico de los elementos.

**Maps**

Es una colección de elementos que se encuentran mapeados por una clave (parejas clave-valor, key-value, K, V):

* La clave asociada a cada elemento permite recuperarlo
* No se permiten claves duplicadas (se sobrescribe el valor si la clave se vuelve a registrar)

[](https://java.codeandcoke.com/_detail/apuntes:map.png?id=apuntes%3Acolecciones)**Figure 5:** Colecciones Map

**Interface Map**

* Define una serie de método que cualquier colección tiene que implementar
  + void clear()
  + boolean containsKey()
  + boolean containsValue()
  + boolean isEmpty()
  + Set<K> keySet()
  + V put(K, V)
  + V remove(K)
  + V replace(K, V)
  + int size()
  + Collection<V> values()

**HashMap**

Es un mapa que proporciona rendimiento constante para las operaciones put() y get()

**TreeMap**

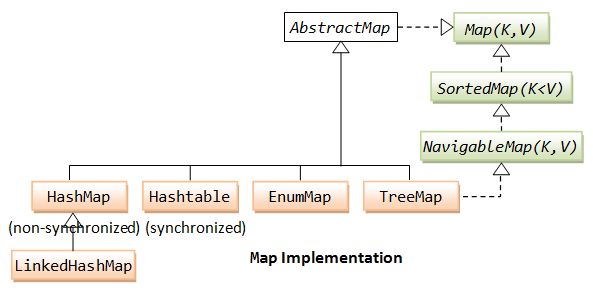
Es un mapa ordenado por el orden natural o Comparator que se indique en el constructor

**LinkedHashMap**

* Mantiene una lista doblemente enlazada
* Es posible obtener un iterador con el orden de inserción de las claves

**Queues**

Define estructuras que pueden ser gestionadas como *FIFO* (First In First Out), lo que se conoce como colas.

[](https://java.codeandcoke.com/_detail/apuntes:map.png?id=apuntes%3Acolecciones)**Figure 6:** Colecciones Map

**Interfaces Queue/Deque**

Define una serie de método que cualquier cola tiene que implementar

* boolean add(e)
* E peek()
* E poll()

Además, la interfaz Deque, que extiende a Queue, y añade elementos para añadir y obtener elementos del inicio y el final:

* void addFirst(e)
* void addLast(e)
* E getFirst()
* E getLast()
* E removeFirst()
* E removeLast()

**ArrayDeque**

Representa una cola

**PriorityQueue**

Representa una cola donde con los elementos ordenador de forma natural o según el Comparator que se le pasa a su constructor

**La clase HashMap**

Es una tabla hash que implementa el interfaz Map de Java, lo que le convierte en una estructura de colección que almacena objetos asociados a una clave. Este tipo de colecciones, a diferencia de los anteriores, no garantiza el orden de los elementos, puesto que éstos se pueden obtener solamente utilizando la clave que se asocio al mismo en el momento de añadirlo.

**Añadir un elemento (pareja clave-valor)**

HashMap<[String](http://www.google.com/search?hl=en&q=allinurl%3Adocs.oracle.com+javase+docs+api+string), Libro> libros = new HashMap<>();

Libro libro = new Libro(. . .);

libros.add(libro.getTitulo(), libro);

**Obtener un elemento**

Libro unLibro = libros.get(tituloLibro);

**Comprobar si existe un elemento (a través de su clave)**

libros.containsKey(tituloLibro)

**Eliminar todos los elementos**

libros.clear();

**Eliminar un elemento**

libros.remove(tituloLibro);

**Obtener una colección con todos los valores**

Collection<Libro> losLibros = libros.values()

**Obtener una colección con todas las claves**

Set<String> losTitulos = libros.keySet();

**Obtener el número de elementos del HashMap**

libros.size();

**Las Colas**

Las colas son colecciones donde los elementos son gestionados según el principio [FIFO](https://es.wikipedia.org/wiki/First_in,_first_out) (First In First Out). Funcionan, básicamente, como la *cola del cine*. El primer elemento que llega a la cola será el primero en salir. Si un elemento se añade a la cola, tendrá que esperar a que salgan todos los que le preceden para salir él.

En Java, la clase LinkedList implementa el interfaz Queue y proporciona una cola FIFO a través de los métodos add(Object) y poll() para retirar el primer elemento.

**El Java Collection Framework** proporciona implementaciones de propósito general de las interfaces del núcleo:

– Para la interface Set, HashSet es la más usada

– Para la interface List, ArrayList es la más usada

– Para la interface Map, HashMap es la más usada

– Para la interface Queue, LinkedList es la más usada

Forma de recorrer las colecciones:

* for-each

– Sintaxis similar a for, sin acceso al elemento recorrido

for (Object o : collection)

System.out.println(o);

* Iterator

– Es un objeto que permite recorrer una colección y eliminar

elementos de la colección de forma selectiva

public interface Iterator

{

boolean hasNext();

Object next();

void remove(); //optional

}

