MANEJO DE COLECCIONES EN JAVA

OpenWebinars

¡BIENVENIDOS!

Soy Luis Miguel López Magaña

Me dedico a ser profesor en ciclos formativos

www.linkedin.com/in/luismi-lopez

AGENDA

1. Introducción
2. Tipos de colecciones y ejemplos de uso
3. Algunos algoritmos para colecciones
4. Librerías de colecciones de terceros

INTRODUCCIÓN

Colecciones en Java

- Una colección es un contenedor para un conjunto de elementos de un tipo en una sola unidad.
- Uso: almacenamiento, recuperación y manipulación de datos.
- Suelen representar elementos que forman grupos naturales
    - Un buzón de correo (colección de mensajes)
    - Un carrito de la compra (colección de items)
    - Una agenda de contactos (colección de parejas de nombre y datos).

Colecciones en Java

- Disponible como framework desde la versión 2
- Paquete *java.util*
- Variaciones y novedades a lo largo de diversas versiones de Java
    - Java 5: uso de genéricos
    - Java 7: operador diamond
    - Java 8: lambdas y streams.
    - Java 9: factorías
    - ...

Framework de Colecciones en Java

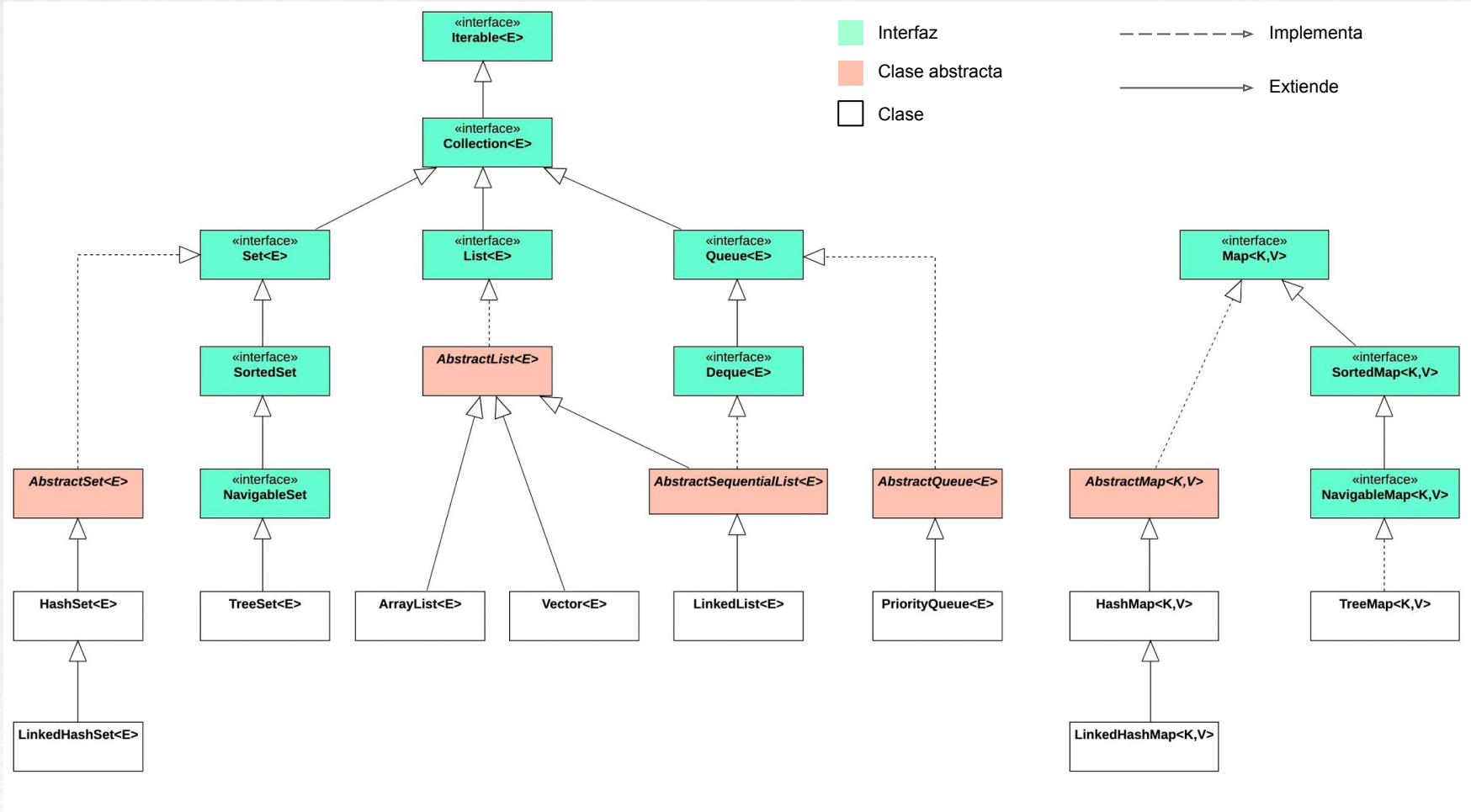
- **Interfaces**: tipos de datos, independientes de su representación
- **Implementaciones**: concreciones de los diferentes tipos
- **Algoritmos**: métodos para buscar, ordenar, clasificar.

  Funcionalidades reutilizables.

Ventajas del framework de colecciones

- Reduce el esfuerzo de programación.
- Aumenta la calidad y velocidad del programa.
- Permite la interoperabilidad con librerías de terceros.
- Reduce el esfuerzo para aprender y usar otras librerías.
- Reduce el esfuerzo para diseñar nuevas librerías.
- Fomenta la reutilización de software.

TIPOS DE COLECCIONES

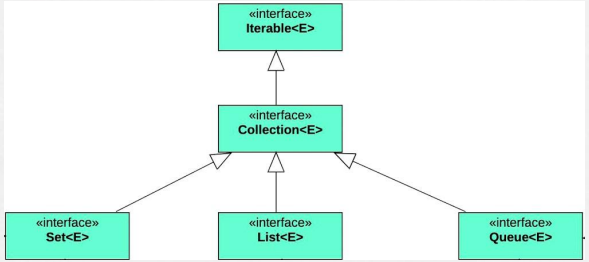

«interface»
Iterable<E>
«interface»
Collection<E>
«interface»
Set<E>
«interface»
List<E>
«interface»
Queue<E>
«interface»
SortedSet
AbstractList<E>
«interface»
Deque<E>
AbstractSet<E>
«interface»
NavigableSet
AbstractSequentialList<E>
AbstractQueue<E>
HashSet<E>
TreeSet<E>
ArrayList<E>
Vector<E>
LinkedList<E>
PriorityQueue<E>
LinkedHashSet<E>
Interfaz
Clase abstracta
Clase
Implementa
Extiende
«interface»
Map<K,V>
«interface»
SortedMap<K,V>
AbstractMap<K,V>
«interface»
NavigableMap<K,V>
HashMap<K,V>
TreeMap<K,V>
LinkedHashMap<K,V>

Iterable<E>

- Concepto de iterador
    - Patrón de diseño
    - Recorrer y eliminar los elementos
    - *Iterator<E>*: *hastNext()*, *next()*, *remove()*
- Permite usar *forEach* y el bucle tipo *for-each*

```java
// Obtenemos el iterable por la vía que corresponda
Iterable<String> unIterable = obtenerIterable();

// Lo podemos recorrer usando un bucle for-each
for (String s: unIterable) {
    System.out.println(s);
}
```

Collection<E>



- Extiende a *Iterable<E>* (hereda su funcionalidad).
- Representa a un grupo de elementos.
- El resto de interfaces heredan de él (salvo *Map* y derivados).
- Permite tener una serie de métodos comunes a (casi) todos los tipos de colecciones.
- JDK no ofrece ninguna implementación directa de esta interfaz.
- Sirve para manipular colecciones de la forma más general posible.

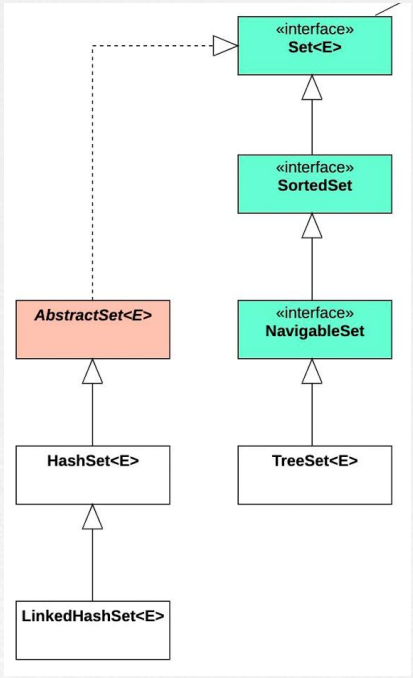
Operaciones de *Collection<E>*

- Tamaño: *size* y *isEmpty*
- Comprobación: *contains*
- Añadir y eliminar: *add* y *remove*
- Iterar: *iterator*
- Operaciones bulk: *containsAll*, *addAll*, *removeAll*, *removeIf*, *retainAll*, *clear*
- Transformar en array: *toArray*
- Streams: *stream*, *parallelStream*

Set<E>

- Se trata de un *Collection<E>* que **no permite duplicados**.
- Abstracción del concepto matemático de conjunto.
- No añade ningún método a los heredados de *Collection<E>*.
- No hay acceso “posicional” (i.e. tercer elemento).
- Mejora la implementación de los métodos *equals* y *hashCode* con respecto a *Collection<E>*
    - Dos instancias de *Set<E>* son iguales si contienen los mismos elementos.

Implementaciones de *Set<E>*



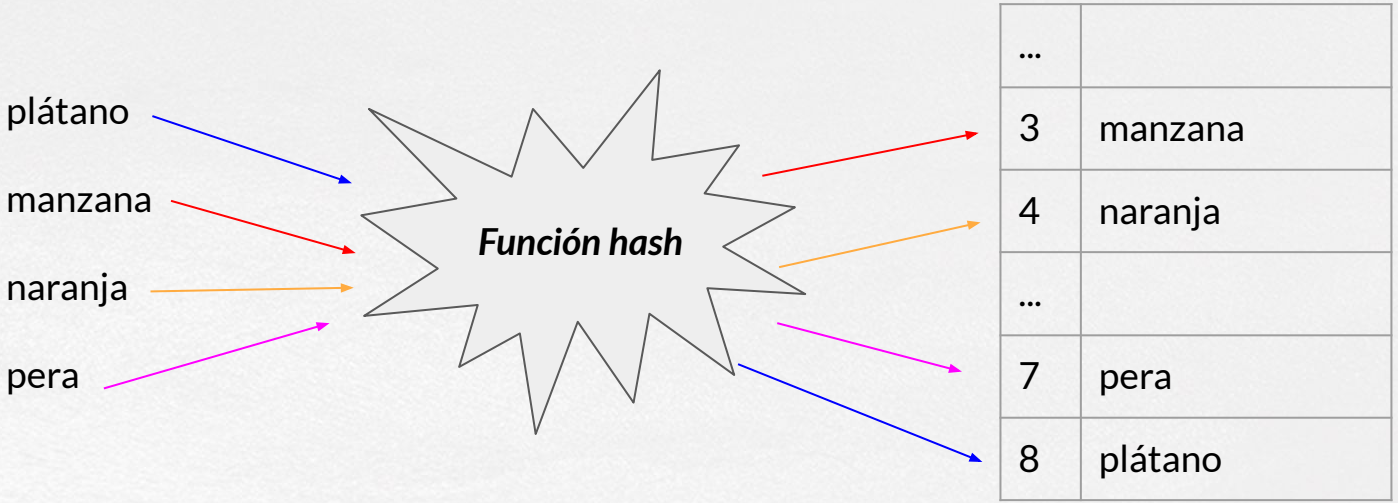
Más rápido

Orden según valor

Orden de inserción

Implementaciones de *Set<E>*

- *HashSet<E>*
  - Almacena sus valores en una tabla hash.



Implementaciones de *Set<E>*

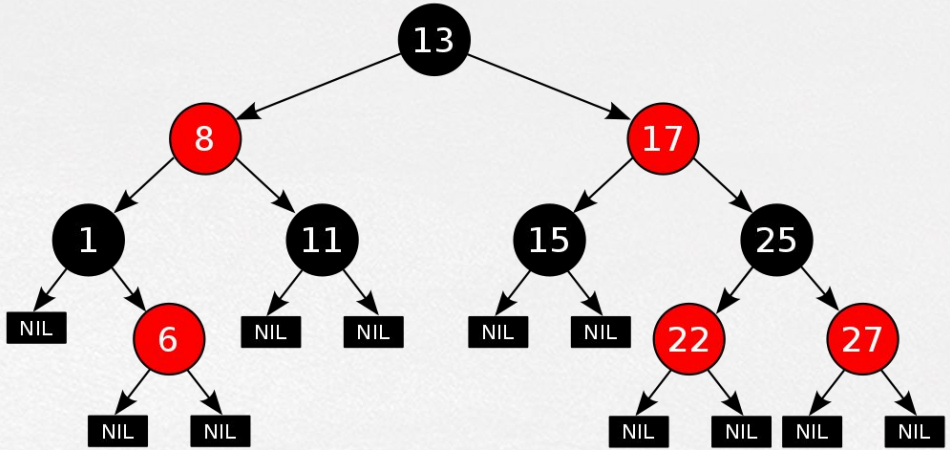
- *HashSet<E>*
  - No podemos predecir nada sobre el orden.
  - Mejor rendimiento de todas.
  - Proporciona tiempo constante (*O(1)*) en las operaciones básicas.
  - Permite insertar valores nulos.
  - No sincronizada.
  - Se mejora el rendimiento si se establece una capacidad inicial no muy elevada.

Implementaciones de *Set<E>*

- *LinkedHashSet<E>*
  - Almacena sus valores en una tabla hash con una lista doblemente enlazada.
  - Mantiene el orden de inserción.
  - Posibilidad de almacenar un valor nulo.
  - No sincronizada.
  - Rendimiento mejor que *TreeSet<E>* pero peor que *HashSet<E>*.

Implementaciones de *Set<E>*

- *TreeSet<E>*
  - Almacena sus valores en un árbol *red-black*.



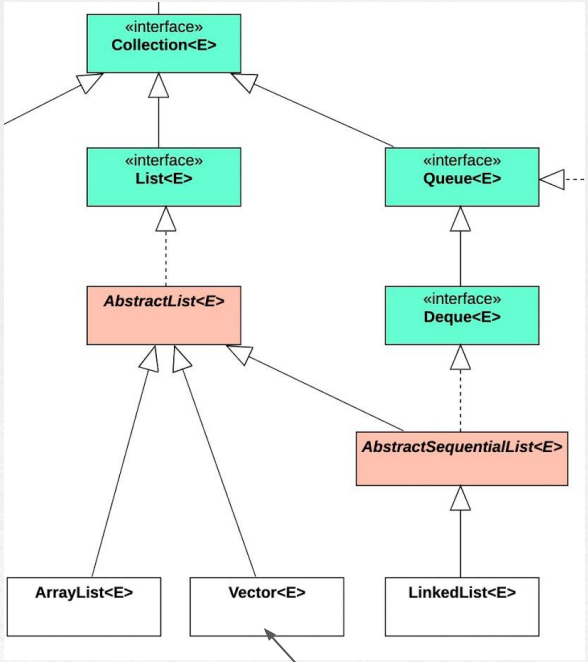
Implementaciones de *Set<E>*

- *TreeSet<E>*
  - Mantiene el orden basado en sus valores.
  - Peor rendimiento que el resto de opciones.
  - Los elementos deben implementar *Comparable*.
  - No permite insertar nulos.
  - No sincronizada.
  - Rendimiento de *O(log(N))* debido a su estructura de árbol.

List<E>

- Se trata de un *Collection<E>* que permite duplicados.
- A las funcionalidades de Collection<E> añade las siguientes:
    - Acceso posicional
    - Búsqueda
    - Iteración extendida
    - Operaciones sobre un rango de elementos de la lista

Implementaciones de *List<E>*



Más usual

Eficiente en algunas situaciones

Sincronizada, pero con métodos *legacy*. Menos recomendable

Operaciones con *List<E>*

- Acceso posicional, parecido al uso de un array

```java
// Una de las formas más sencillas de obtener una lista
List<String> lista = Arrays.asList("Uno", "Dos", "Tres", "cinco", "Seis");

// Uso posicional de la lista
// Los índices comienzan en 0

// Obtener el elemento en la posición 2 (tercer elemento)
String tercerElemento = lista.get(2);
```

Operaciones con *List<E>*

- Búsqueda, para obtener el índice de un elemento

```
// Búsqueda de la primera o la última ocurrencia de un elemento
int posicion1 = lista.indexOf("Dos");
int posicion2 = lista.lastIndexOf( o: "Seis");

System.out.println("\n\nLa primera ocurrencia de Dos está en la posición " + posicion1);
System.out.println("La última ocurrencia de Seis está en la posición " + posicion2);
```

Operaciones con *List<E>*

- Iteración extendida, con *listIterator*
    - Se puede avanzar en cualquier dirección.
    - Se puede añadir, modificar y eliminar

```
// Ejemplo de iteración hacia atrás
for (ListIterator<String> listIterator = lista.listIterator(lista.size());
     listIterator.hasPrevious();) {
    System.out.print(listIterator.previous() + " ");
}
```

Operaciones con *List<E>*

- Operaciones sobre un rango de elementos
    - Se puede obtener un view de una sublista
    - Las operaciones de inserción, actualización y borrado modifican la lista subyacente.

Implementaciones de *List<E>*

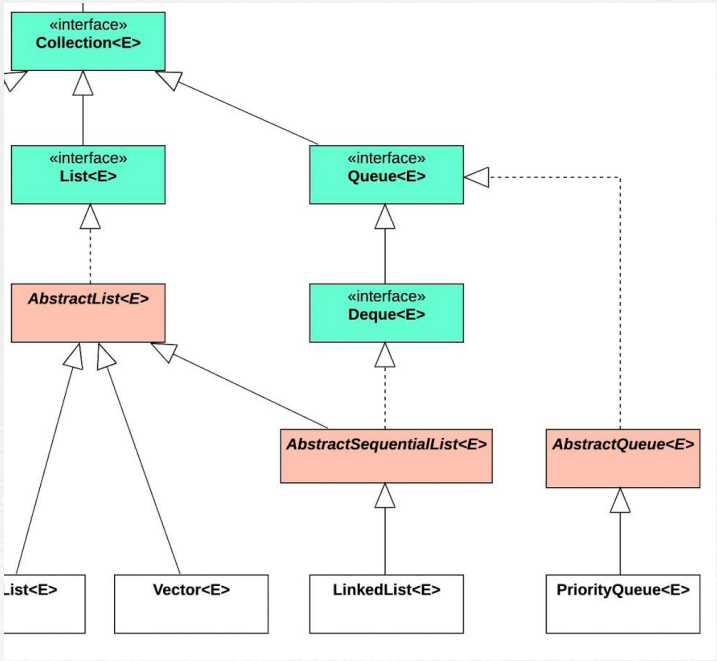
- *ArrayList<E>*
  - Más adecuada en la mayoría de las situaciones
  - Acceso por índice en O(1)
  - Inserción, en media, en O(1)
  - Menos espacio que LinkedList.
- *LinkedList<E>*
  - Suele tener peor rendimiento
  - Acceso por índice en O(n)
  - Inserción/Borrado: O(1) extremos, O(n) por índice, O(1) en iteración.
  - Más espacio (debe incluir dos referencias).

Operaciones con *List<E>*

- Operaciones sobre un rango de elementos

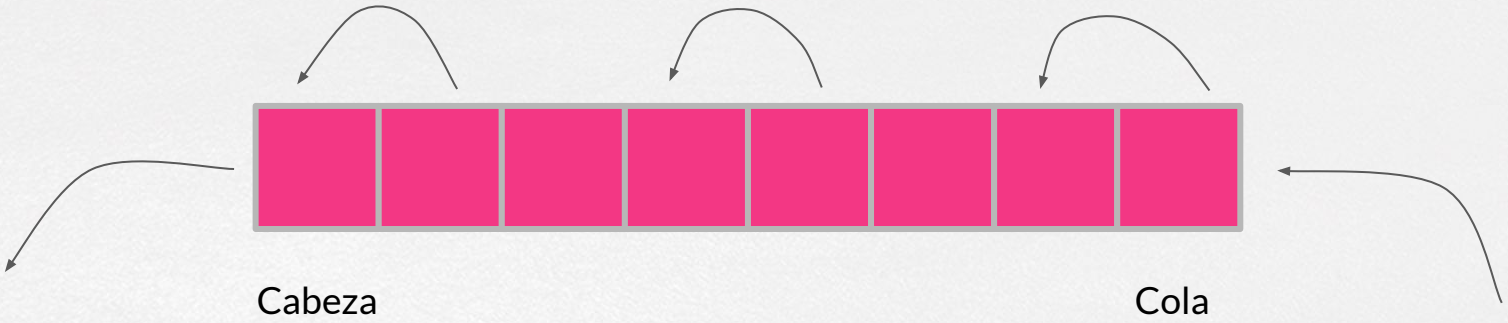
```
List<String> subLista = lista.subList(3,5);
System.out.println("\n\nElementos de la sublista");
for (String s : subLista) {
    System.out.print(s + " ");
}

// Si eliminamos elementos de la sublista
// se eliminan de la lista.
subLista.clear();

System.out.println("\n\nElementos de la lista modificados a través de la sublista");
for (String s : lista) {
    System.out.print(s + " ");
}
```

Otros interfaces relacionados con *List<E>*



Otros interfaces relacionados con *List<E>*

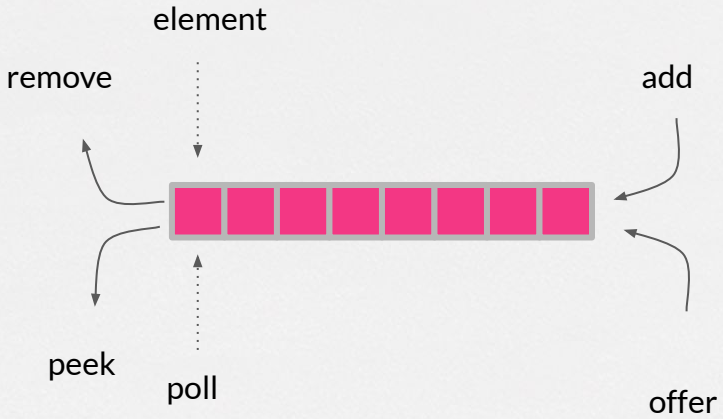
- *Queue<E>*
    - Funciona como una cola FIFO



Otros interfaces relacionados con *List<E>*

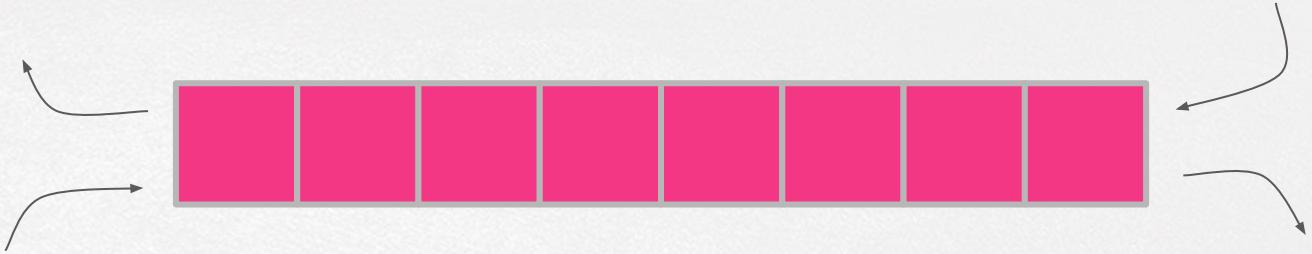
- *Queue<E>*

	Lanzan excepción	Devuelven valor especial
Inserción	*add(e)*	*offer(e)*
Extraer	*remove()*	*poll()*
Examinar	*element()*	*peek()*



Otros interfaces relacionados con *List<E>*

- *Deque<E>*
    - Puede funcionar como una cola (FIFO) o como una pila (LIFO).
    - También puede funcionar como una pila doble o una cola doble.



Otros interfaces relacionados con *List<E>*

- *Deque<E>* como cola

	Método *Queue<E>*	**Equivalente en *Deque<E>***
Inserción	*add(e)*, *offer(e)*	*addLast(e)*, *offerLast(e)*
Extraer	*remove()*, *poll()*	*removeFirst()*, *pollFirst()*
Examinar	*element()*, *peek()*	*getFirst()*, *peekFirst()*

- *Deque<E>* como pila

	Método *Stack<E>*	**Equivalente en *Deque<E>***
Inserción	*push(e)*	*addFirst(e)*, *offerFirst(e)*
Extraer	*pop()*	*removeFirst()*, *pollFirst()*
Examinar	*peek()*	*getFirst()*, *peekFirst()*

Otros interfaces relacionados con *List<E>*

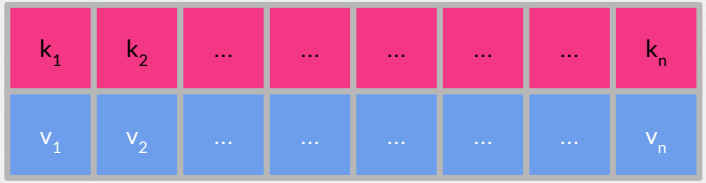
- *Deque<E>* como una cola doble

	Primer elemento (Cabeza)		**Último elemento (Cola)**	
	Lanzan excepción	***Devuelven valor especial***	***Lanzan excepción***	***Devuelven valor especial***
Inserción	*addFirst(e)*	*offerFirst(e)*	*addLast(e)*	*offerLast(e)*
Extraer	*removeFirst()*	*pollFirst()*	*removeLast()*	*pollLast()*
Examinar	*getFirst()*	*peekFirst()*	*getLast()*	*peekLast()*

Implementaciones de Queue<E> y Deque<E>

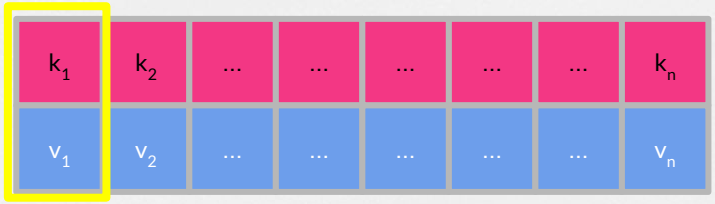
- *Queue<E>*
    - *PriorityQueue<E>*
    - *ArrayDeque<E>*
    - *LinkedList<E>*
- *Deque<E>*
    - *ArrayDeque<E>*
    - *LinkedList<E>*

Map<K,V>



- No hereda de *Collection<E>*.
- Maneja pares *clave,valor*
- Para cada clave, hay un solo valor.
- Cada clave puede existir una sola vez en el map.
- Puede haber una clave nula, y múltiples valores nulos.
- En otros lenguajes de programación se le conoce como diccionario.
- No puede almacenar tipos primitivos; hay que usar los tipos wrapper en su lugar (*int* → *Integer*)

Map.Entry<K,V>



- Clase que permite consultar un par *clave,valor* de un Map.
- No se pueden utilizar para insertar valores.
- Se puede obtener un *Set<Map.Entry<K,V>>* a través del método *Map.entrySet()*.

Operaciones con *Map<K,V>*

- Insertar una clave y su valor
    - *put(key, value)*
- Obtener un valor en base a la clave
    - *get(key)*
- Consultar si una clave o un valor están contenidos
    - *containsKey(key) / containsValue(value)*
- Eliminar el par clave, valor
    - *remove(key)*

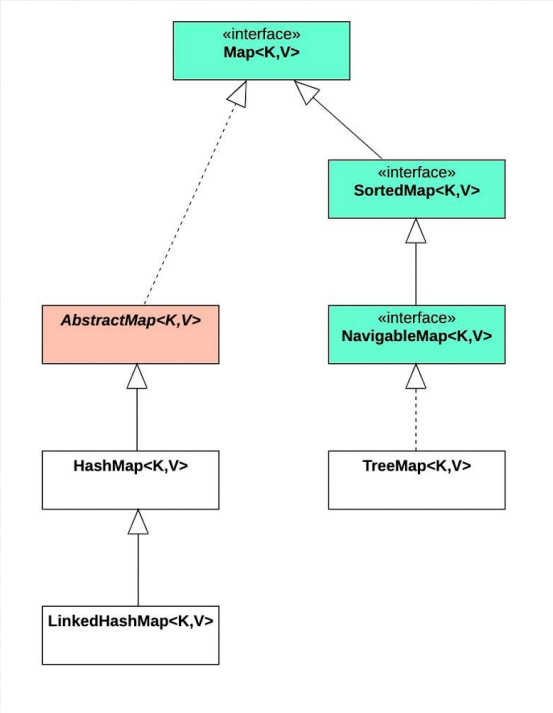
Operaciones con *Map<K,V>*

- Recorrer un Map
- *Opción 1*: forEach
    - Obtener un Set con las claves
    - Para cada clave, obtener los valores
- *Opción 2*: Usando el lambdas
    - Método forEach
    - Expresión lambda (Biconsumer)

Map<K,V>

Más rápido

Orden de inserción



Orden según las claves

Implementaciones de Map<K,V>

- *HashMap<K,V>*
    - Es la más utilizada, por tener el mejor rendimiento.
    - ***No podemos suponer nada sobre el orden de los pares***.
    - El tiempo de ejecución de inserción y consulta es constante *O(1)*.
    - No es sincronizada.

Implementaciones de Map<K,V>

- *LinkedHashMap<K,V>*
    - Rendimiento *un poco peor* que HashMap.
    - ***Ordena los pares* clave,valor *según su inserción***
    - No es sincronizada.

Implementaciones de Map<K,V>

- *TreeMap<K,V>*
    - Peor rendimiento de las 3 implementaciones.
    - ***Mantiene las claves en orden (natural)***
    - No puede tener ninguna clave nula, aunque sí puede almacenar valores nulos.

COLECCIONES PARA
SITUACIONES ESPECIALES

Colecciones no modificables

- Son colecciones que, una vez creadas, no se pueden modificar.
- Si se trata de modificar, se lanza *UnsupportedOperationException*
- Se pueden usar, por ejemplo, como resultado de una operación.
- Hasta Java 8
    - *Collections.unmodifiableXXX(coleccion)*: versión no modificable de la colección.
    - *Collections.emptyXXX*. Colección vacía no modificable.

Colecciones no modificables

- Desde Java 9
    - Métodos factoría *.of(...)*
    - Diferentes versiones: desde 0 elementos hasta un número variable.
    - Disponibles en los diferentes interfaces: *List*, *Set*, *Map*, ...

Colecciones sincronizadas

- Aptas para el uso de diferentes hilos de ejecución.
- Varios procesos/hilos que compiten por el uso de la colección
- Versión sincronizada de una colección no sincronizada
- Métodos *Collections.synchronizedXXX*

Otras implementaciones menos usadas

- *Set<E>*
  - *EnumSet*
    - Set pensado para contener valores de enumeraciones.
  - *CopyOnWriteArraySet*
    - *Thread-safe* pero no sincronizado
    - Cada operación de modificación provoca la creación de un nuevo array subyacente.
    - Adecuado en contextos concurrentes para iterar de una manera segura, pero sin el coste de la sincronización.

Otras implementaciones menos usadas

- *List<E>*
  - *CopyOnWriteArrayList*
    - *Thread-safe* pero no sincronizado
    - Cada operación de modificación provoca la creación de un nuevo array subyacente.
    - Adecuado en contextos concurrentes para iterar de una manera segura, pero sin el coste de la sincronización.

Otras implementaciones menos usadas

- *Map<K,V>*
  - *EnumMap*
    - Implementación de un *Map* de alto rendimiento, donde las claves son valores de una enumeración.
  - *WeakHashMap*
    - Implementación de un Map que tiene referencias débiles a sus claves.
    - Si una clave no se va a usar más, el *garbage collector* puede eliminar el par del Map.

Otras implementaciones menos usadas

- *Map<K,V>*
  - *IdentityHashMap*
    - No es un Map de uso común
    - No cumple el requisito de que las claves se comparen con *equals*.
    - Dos claves son iguales cuando se comparan con ==
- *Deque<E>*
  - *LinkedBlockingDeque*
    - Implementación concurrente.

Otras implementaciones menos usadas

- *Queue<E>*
  - *PriorityQueue*
    - Además del orden de inserción, orden por prioridad.
    - La hemos visto en alguno de los ejemplos
  - Implementaciones concurrentes. *BlockingQueue*
    - *LinkedBlockingQueue*
    - *ArrayBlockingQueue*
    - *PriorityBlockingQueue*
    - *DelayQueue*

ALGORITMOS
PARA COLECCIONES

ALGORITMOS

- Disponibles en la clase *Collections* (plural).
    - Ordenar/Desordenar
    - Búsqueda
    - Operaciones: max/min, frecuencia, ...

Algoritmos de ordenación

- *Collections.sort(list)*
    - Recibe una colección de tipo *List<E>*.
    - Modifica la colección que recibe.
    - Los elementos deben implementar *Comparable*.
- *Collections.sort(list, comparator)*
    - Implementación que ordena según el orden inducido por la instancia de comparator.
- En ambos casos, el algoritmo usado es *Merge Sort*.

Algoritmos de desordenación

- *Collections.shuffle(list)*
    - Realiza la operación opuesta a *sort*
    - Desordena los elementos de una colección
- *Collections.shuffle(list, random)*
    - Versión del anterior, donde el componente aleatorio se le puede proporcionar.

Algoritmos de búsqueda

- Búsqueda binaria.
- La lista debe estar ordenada previamente.
- Si hay varias ocurrencias, no se garantiza cual se va a encontrar.
- Devuelve
    - el índice del elemento si lo encuentra
    - si no, *(-(punto de inserción)-1)*.
        - Esto garantiza que se devuelve un valor mayor que cero sólo si se encuentra

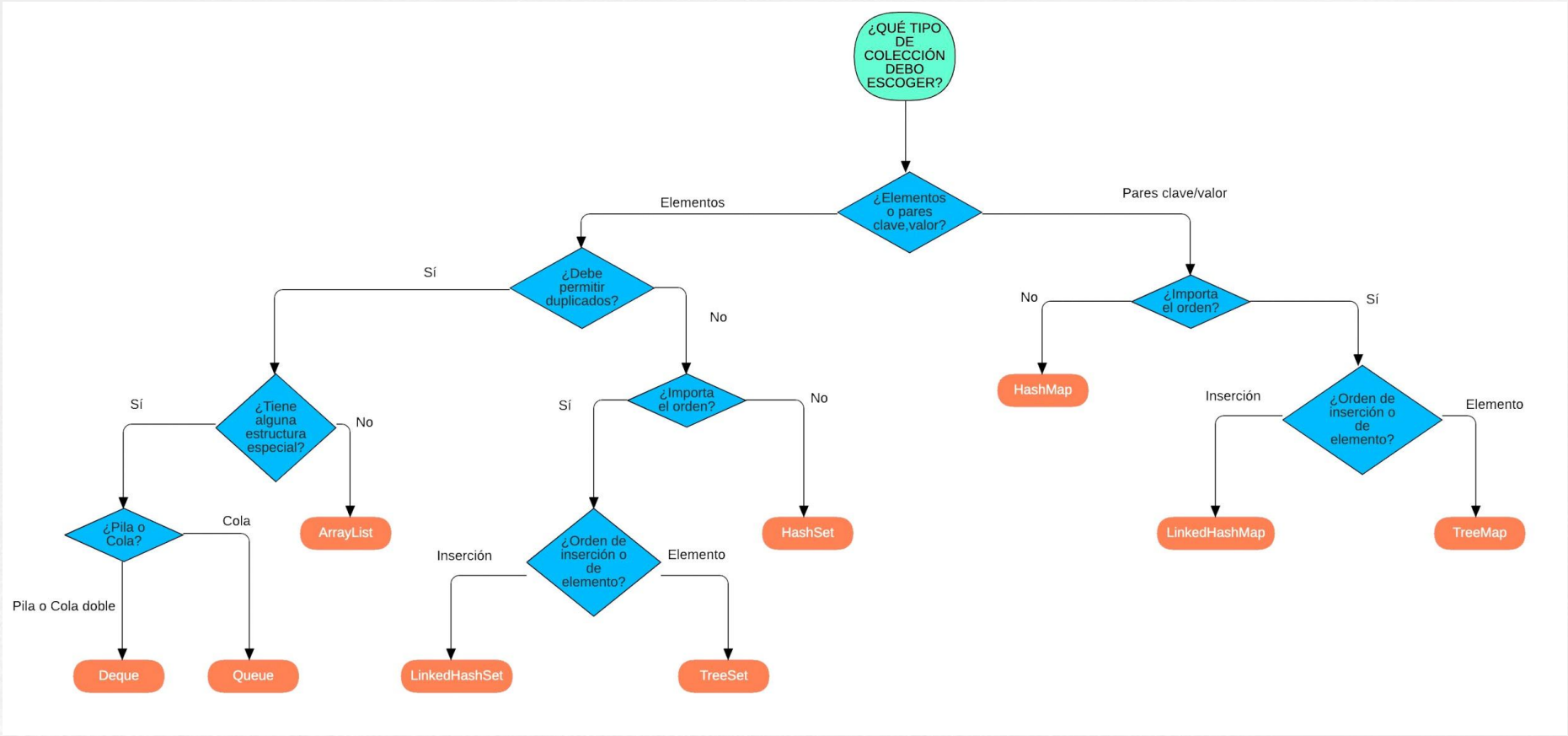
Algoritmos para algunas operaciones

- Máximo y mínimo
    - Acorde al orden natural
    - Otra implementación con un *Comparator*.
    - Aplicable a *Collection*.
- Frecuencia
    - Devuelve el número de ocurrencias de un objeto en una colección.

Algoritmos para algunas operaciones

- Intersección vacía
    - Compara dos colecciones
    - Devuelve true si no hay ningún elemento común entre ambas.

¿CÓMO SABER QUÉ TIPO DE COLECCIÓN ESCOGER?


¿QUÉ TIPO DE COLECCIÓN DEBO ESCOGER?
¿Elementos o pares clave,valor?
Elementos
Pares clave/valor
¿Debe permitir duplicados?
Sí
No
¿Importa el orden?
No
Sí
HashMap
¿Tiene alguna estructura especial?
Sí
No
¿Importa el orden?
Sí
No
Inserción
¿Orden de inserción o de elemento?
Elemento
LinkedHashMap
TreeMap
¿Pila o Cola?
Cola
Pila o Cola doble
ArrayList
¿Orden de inserción o de elemento?
Inserción
Elemento
HashSet
Deque
Queue
LinkedHashSet
TreeSet

LIBRERÍAS DE COLECCIONES DE TERCEROS

Librerías más usuales

- Guava
- Eclipse Collections
- Apache Commons Collections

Guava

- Implementada por Google
- Disponible Java/Android
- Constructores mejorados
- Iterable allá donde sea posible
- Más métodos factoría
- Operaciones de conjuntos (unión, intersección, prod. cartesiano)
- BiMaps, Multisets, Multimaps, Tables

Eclipse Collections

- API más legible (métodos más expresivos)
- Múltiples métodos factoría para colecciones mutables
- Colecciones para tipos primitivos
- Gestión de memoria mejorada
- BiMaps, Multimaps, Bags, ...

Apache Commons Collections

- Colecciones tipo *Bag*
- *BidiMap*: mapas clave → valor y valor → clave.
- *MapIterator* para recorrer un map convenientemente.
- Colecciones compuestas
- Muchas implementaciones de iteradores, comparadores, ...