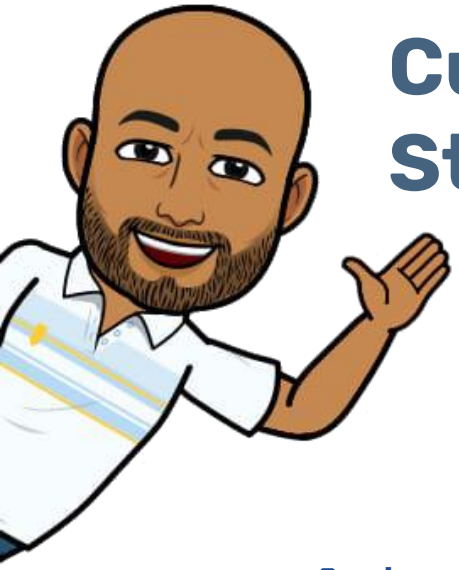


Curso de Java Standard

Ing. Octavio Robleto

octavio.robleto@gmail.com

https://octaviorobleto.com

Introducción

Java nos permite a través de los arreglos almacenar “agrupar” bajo un mismo identificador una cantidad grande de elementos, el problema es que debemos saber de antemano el tamaño del mismo y este no varía; no se pueden añadir elementos nuevos ni eliminar, en el mejor de los casos podemos dejar el valor por defecto del dato primitivo y de ser un Objeto dejarlo en **null**.

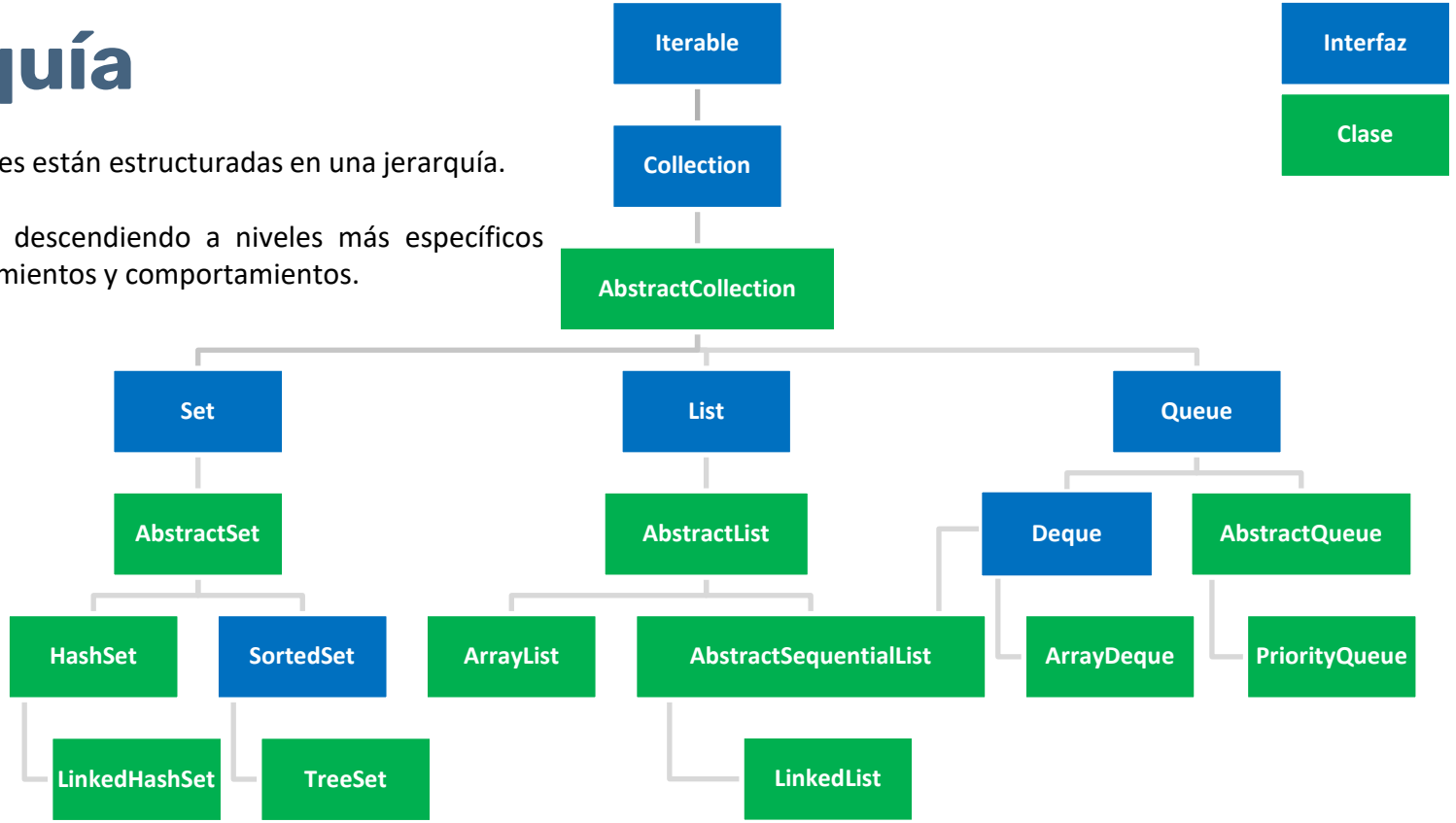
Además de los arreglos convencionales que ya conocemos, Java nos proporciona un conjunto de interfaces y clases (Collections Framework) para agrupar elementos de forma dinámica, por lo que podemos agregar mas Objetos sin indicar un tamaño único o inicial; también reduce el esfuerzo de programación al tiempo que aumenta el rendimiento.

Las colecciones son Interfaces y Clases que tienen parámetros (Genéricas) por lo que debemos indicarles el tipo de Objeto a almacenar.

Jerarquía

Estas clases e interfaces están estructuradas en una jerarquía.

A medida que se va descendiendo a niveles más específicos aumentan los requerimientos y comportamientos.



Es importante aclarar que existen muchas mas colecciones por lo cual tocaremos solo las mas usadas

Métodos Collection

Tipo	Método	Descripción
boolean	add(E e)	Asegura que esta colección contiene el elemento especificado (operación opcional).
boolean	addAll(Collection<? extends E> c)	Agrega todos los elementos de la colección especificada a esta colección.
void	clear()	Elimina todos los elementos de esta colección (operación opcional).
boolean	contains(Object o)	Devuelve verdadero si esta colección contiene el elemento especificado.
boolean	containsAll(Collection<?> c)	Devuelve verdadero si esta colección contiene todos los elementos de la colección especificada.
boolean	equals(Object o)	Compara el objeto especificado con esta colección para la igualdad.
int	hashCode()	Devuelve el valor del código hash para esta colección.
boolean	isEmpty()	Devuelve verdadero si esta colección no contiene elementos.
Iterator<E>	iterator()	Devuelve un iterador sobre los elementos de esta colección.
boolean	remove(Object o)	Elimina una sola instancia del elemento especificado de esta colección, si está existe
boolean	removeAll(Collection<?> c)	Elimina de este conjunto todos sus elementos que están contenidos en la colección especificada
Int	size()	Devuelve el número de elementos en esta colección.
Object[]	toArray()	Devuelve una matriz que contiene todos los elementos de este conjunto
<T> T[]	toArray(T[] a)	Devuelve una matriz que contiene todos los elementos de este conjunto; el tipo de tiempo de ejecución de la matriz devuelta es el de la matriz especificada.

Métodos

```java
E elemento1, elemento2, elemento3, elemento4, elemento5;

Coleccion<E> coleccionPrincipal = new Implementacion<E>();
Coleccion<E> coleccionA = new Implementacion<E>();
Coleccion<E> coleccionB = new Implementacion<E>();

// Agregar elementos
coleccionA.add(elemento1);
coleccionA.add(elemento2);
coleccionB.add(elemento3);
coleccionB.add(elemento4);

// Agregar una Coleccion en Otra
coleccionPrincipal.addAll(coleccionA);
coleccionPrincipal.addAll(coleccionB);

// verificar que contenga el elemento
System.out.println(coleccionPrincipal.contains(elemento1));

// verificar la coleccion contenga todos los elementos de otra coleccion
System.out.println(coleccionPrincipal.containsAll(coleccionA));

// verificar si son iguales dos colecciones
System.out.println(coleccionA.equals(coleccionB));
```

```java
// remover un elemento de la coleccion
coleccionPrincipal.remove(elemento4);

// remover los elementos de una coleccion que existan en otra
coleccionPrincipal.removeAll(coleccionA);

// convertir una coleccion en un arreglo convencional
Object[] objetos = coleccionPrincipal.toArray();

E[] elementos;
coleccionPrincipal.toArray(elementos);

//conocer el tamaño de la coleccion
System.out.println(coleccionPrincipal.size());

// recorrer una coleccion
for (E elemento : coleccionPrincipal) {
    System.out.println(elemento);
}

// limpiar la coleccion
coleccionPrincipal.clear();
```

Iteradores

Algo particular de estas colecciones es que no se pueden recorrer y operar al mismo tiempo (Eliminar o Actualizar la colección), para eso la interfaz **Collection** nos proporciona un método que devuelve un Iterator.

Tipo	Método	Descripción
boolean	hasNext()	Devuelve true si la iteración tiene más elementos.
E	next()	Devuelve el siguiente elemento en la iteración.
void	remove()	Elimina de la colección subyacente el último elemento devuelto por este iterador

```
Iterator<E> iterador = coleccionPrincipal.iterator();

while(iterador.hasNext()) {
    E elementoAuxiliar = iterador.next();
    if (condicion) {
        iterador.remove();
    }
}
```

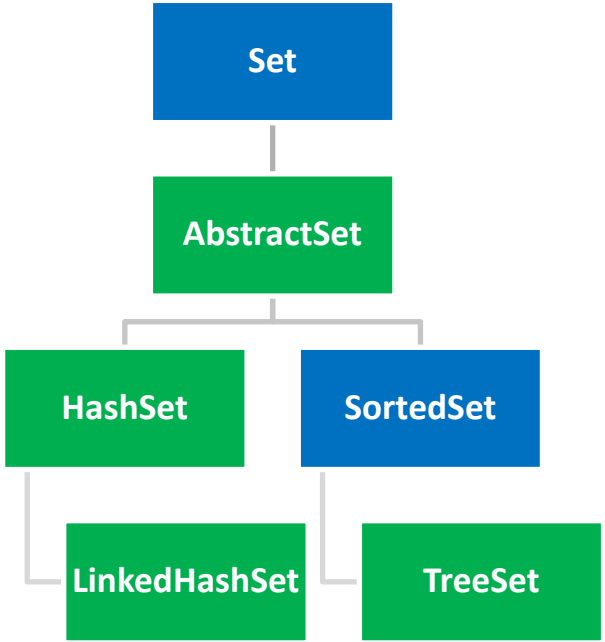
Si solo deseamos acceder al elemento sin manipularlo podemos usar el for-each o for mejorado ya que con el for común no tenemos el índice para poder utilizar dicho ciclo.

Interfaz Set

La interfaz **Set** tiene una peculiaridad, y es que no admite duplicados y es especialmente útil para ir almacenando datos sin la preocupación de que alguno se repita, estos elementos no se mostraran de forma ordenada.

La clase **AbstractSet** Proporciona una implementación esquelética de la interfaz Set y simplemente agrega implementaciones para los métodos **equals** y **hashCode**.

La interfaz **SortedSet** permite que las clases que la implementen tengas los elementos ordenados.



HashSet

Este implementación almacena los elementos en una tabla hash (es un contenedor asociativo “tipo Diccionario” que permite un almacenamiento y posterior recuperación eficiente de elementos); este acceso hace que esta clase sea ideal para búsqueda, inserción y borrado de elementos.

Representa un conjunto de valores únicos (no puede tener valores duplicados) sin ordenar (por ejemplo si hacemos un recorrido de los objetos dentro de un **HashSet** no siempre los obtendremos en igual orden) y tiene una iteración más rápida que otras colecciones.

```
AbstractSet<String> nombres = new HashSet<>();
Set<String> nombresA = new HashSet<>();
HashSet<String> nombresB = new HashSet<>();
```

Equals y hashCosde

- **public boolean equals(Object obj)**
- **public int hashCode()**

Ya hemos visto que podemos comparar con el método **equals** una cadena de caracteres, este método devuelve true o false según sea el caso. Ej: **“Octavio”.equals(“octavio”);** en este caso devuelve false ya que el método es sensible a las mayúsculas y minúsculas.

Pero, ¿Cómo sabe Java que un objeto propio es igual a otro?

Primero que nada debemos saber que este es un método que heredamos de la clase **Object** y que las clases de Java sobreescriben para que los podamos utilizar, así que nosotros debemos hacer lo mismo, indicándole al método cuales son los atributos que debe comparar para que un objeto sea igual a otro.

Pero también debemos sobreescribir otro método el **hashCode**, ya que si dos objetos son iguales se debe generar el mismo numero hash para poder localizar el elementos en la tabla hash.

Por ejemplo nosotros podemos decir que un Auto es Igual a otro si poseen la misma Patente, en otros casos los objetos son iguales si poseen todos los atributos iguales; siempre dependerá del caso de uso.

Equals y hashCode

Los IDE nos proporcionan alguna manera de generar de forma automática estos métodos.

Por ejemplo en Eclipse podemos ir a:
Source –> Generate hashCode() and equals()
En el panel seleccionamos los atributos que queremos que se tengan en cuenta en estos métodos.

Es importante aclarar que si los atributos a comparar también son objetos propios debemos sobreescribir en esos objetos dichos métodos, de lo contrario la interfaz no detectara que son iguales.

Además podemos utilizar el método **equals** a partir de ahora cuando lo creamos conveniente.

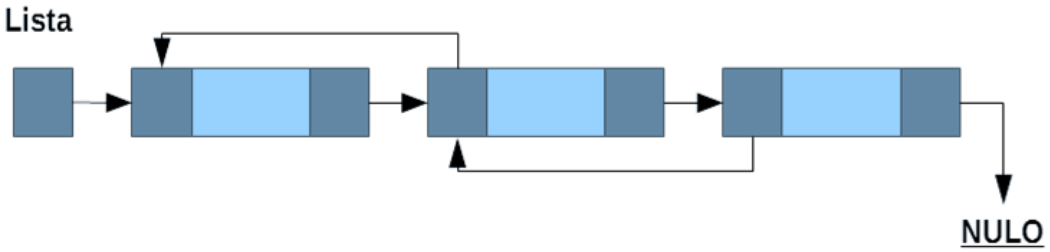
```java
@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + ((patente == null) ? 0 : patente.hashCode());
    return result;
}

@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    Auto other = (Auto) obj;
    if (patente == null) {
        if (other.patente != null)
            return false;
    } else if (!patente.equals(other.patente))
        return false;
    return true;
}
```

LinkedHashSet

Esta implementación almacena los elementos en función del orden de inserción lo que la hace un poco más costosa que HashSet.

```
AbstractSet<String> nombres = new LinkedHashSet<>();
Set<String> nombresA = new LinkedHashSet<>();
HashSet<String> nombresB = new LinkedHashSet<>();
```

Define el concepto de elementos añadiendo una lista doblemente enlazada en la ecuación que nos asegura que los elementos siempre se recorren de la misma forma.



TreeSet

Esta implementación almacena los elementos ordenándolos en función de sus valores (Implementando el algoritmo del árbol rojo – negro: árbol de búsqueda binaria, también llamado árbol binario ordenado o árbol binario ordenado), por lo que es bastante más lento que **HashSet**

Este orden podrá ser Natural o Alternativo.

```
AbstractSet<String> nombres = new TreeSet<>();
Set<String> nombresA = new TreeSet<>();
TreeSet<String> nombresB = new TreeSet<>();
```

```
Set<Auto> autos = new TreeSet<>(); // Orden Natural
Set<Auto> autos = new TreeSet<>(new OrdenAlternativo());
```

Orden

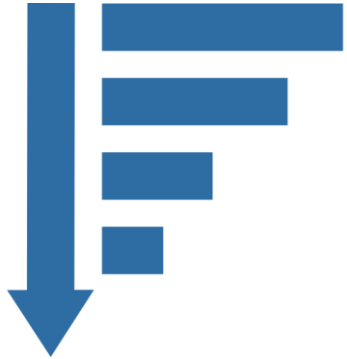
Java proporciona dos interfaces (**Comparable<T>** y **Comparator<T>**) para dar un orden a los objetos de un tipo creado por el usuario, Java ya proporciona un orden natural a los tipos envoltorio como Integer y Double o al tipo inmutable String.

Comparable (Orden Natural) esta interfaz debe ser implementada por la clase y compara el objeto **this** con otro que toma como parámetro el método **compareTo.** PD: El orden natural definido debe ser coherente con la definición de igualdad. Si **equals** devuelve true compareTo debe devolver cero.

Comparator (Orden Alternativo) esta interfaz debe ser implementada en una nueva clase comparando los dos objetos que toma como parámetros el método **compare.**

Los métodos compareTo y compare comparan dos objetos o1 y o2 (en compareTo o1 es **this**) y devuelve un entero que es:

- Negativo si o1 es menor que o2
- Cero si o1 es igual a o2
- Positivo si o1 es mayor que o2



Si el atributo a comparar es un String podemos apoyarnos en el método **compareTo** que posee de lo contrario con una resta entre atributos numéricos basta.

Cuando comparamos o1 contra o2 el orden es ascendente si invertimos la comparación ordenaría de forma descendente.

Orden

Comparable

```
public abstract class Auto implements MantenimientoMecanico, Archivo, Comparable<Auto>{
```

```
@Override
public int compareTo(Auto auto) {
    return this.patente.getNumero().compareTo(auto.getPatente().getNumero());
    // return this.puestos - auto.puestos;
}
```

Comparator

```
public class OrdenAutoMarca implements Comparator<Auto> {

    @Override
    public int compare(Auto auto1, Auto auto2) {
        return auto1.getMarca().compareTo(auto2.getMarca());
        // return auto1.puestos - auto2.puestos;
    }

}
```

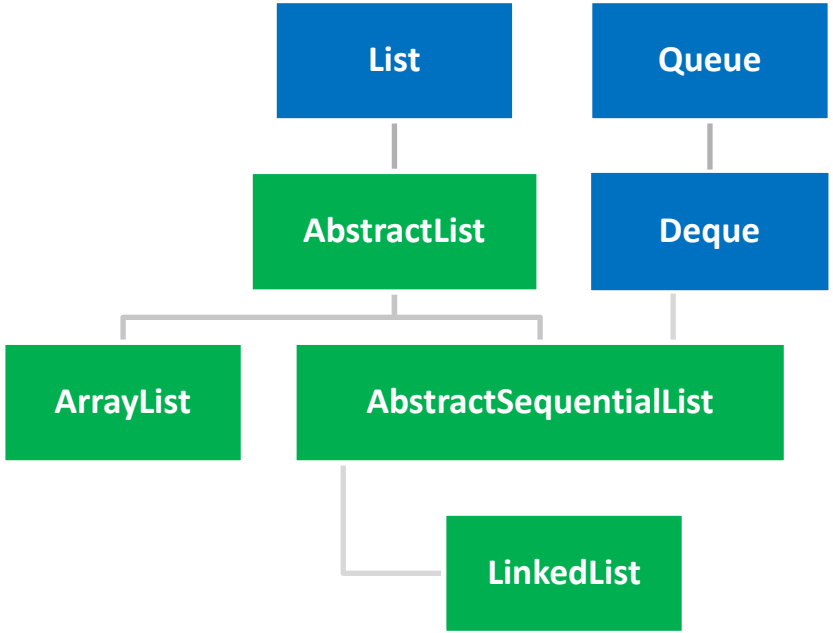
List

La interfaz **List** a diferencia de **Set** si permite elementos duplicados y es la interfaz mas usada por nosotros ya que implementa una serie de métodos que la hacen mas comprensible y fácil de manejar:

- **Acceso posicional a elementos:** manipula elementos en función de su posición en la lista.
- **Búsqueda de elementos:** busca un elemento concreto de la lista y devuelve su posición.
- **Iteración sobre elementos**: mejora el Iterador por defecto.

La clase **AbstractList** Proporciona una implementación esquelética de la interfaz dándole comportamiento a los nuevos métodos que utilizaremos.

La clase **AbstractSequentialList** Proporciona una implementación esquelética de la interfaz dándole acceso secuencial a la colección.



Métodos de las listas

Adicionalmente de los métodos que nos proporciona la interfaz **Collection, List y AbstractList** añaden unos nuevos:

Tipo	Método	Descripción
void	add(int index, E element)	Inserta el elemento especificado en la posición especificada en esta lista
E	get(int index).	Devuelve el elemento en la posición especificada en esta lista
E	set(int index, E element)	Reemplaza el elemento en la posición especificada en esta lista con el elemento especificado
E	remove(int index)	Elimina el elemento en la posición especificada en esta lista
int	indexOf(Object o)	Devuelve el índice de la primera aparición del elemento especificado en esta lista, o -1 si esta lista no contiene el elemento.
ListIterator<E>	listIterator()	Devuelve un iterador de lista sobre los elementos de esta lista
ListIterator<E>	listIterator(int index)	Devuelve un iterador de lista sobre los elementos de esta lista, comenzando en la posición indicada.
List<E>	subList(int fromIndex, int toIndex)	Devuelve una lista de la parte de esta lista entre el especificado fromIndex, el inclusivo y el toIndex exclusivo.
default void	sort(Comparator<? super E> c)	Ordena esta lista según el orden inducido por el especificado Comparator.

Métodos de las listas

```java
// se agrega elemento en los indice
coleccion.add(1, elemento1);

// recorrido de lista con for comun
for (int i = 0; i < coleccion.size(); i++) {
    System.out.println(coleccion.get(i));
}

// se reemplaza elemento de la lista
coleccion.set(0, elemento2);

// se remueve elemento en la posicion indicada, devuelve el elemento eliminado
System.out.println(coleccion.remove(2));

// devuelve el indice de la primera aparicion de Ariel
System.out.println(coleccion.indexOf(elemento4));

// devuelve el indice de la ultima aparicion de Ariel
System.out.println(coleccion.lastIndexOf(elemento5));

// creamos una sublista a partir de un rango
List<E> subLista = coleccion.subList(1, 3);

// ordenamos
coleccion.sort(new Comparador());
```

Iteradores de Lista

Es un iterador Java que nos permite recorrer una lista de elementos en varias direcciones, bien como fueron insertado los elementos o en reversa (hacia adelante o hacia atrás), además de proporcionarnos nuevos métodos sin eliminar los que ya conocemos de los iteradores.

Tipo	Método	Descripción
void	add(E e)	Inserta el elemento especificado en la lista
boolean	hasPrevious()	Devuelve true si este iterador de lista tiene más elementos al recorrer la lista en la dirección inversa.
Int	nextIndex()	Devuelve el índice del elemento que sería devuelto por una llamada posterior a next().
E	previous()	Devuelve el elemento anterior de la lista y mueve la posición del cursor hacia atrás.
Int	previousIndex()	Devuelve el índice del elemento que sería devuelto por una llamada posterior a previous().
void	set(E e)	Reemplaza el último elemento devuelto por next()o previous()con el elemento especificado

```java
// creamos un iterador mejorado
ListIterator<E> iteradorLista = subLista.listIterator();

// recorrer el iterador
while (iteradorLista.hasNext()) {
    E elementoAuxiliar = iteradorLista.next();

    if (elementoAuxiliar.equals(elemento3)) {
        // eliminamos el elemento
        iteradorLista.remove();
    }

    if (elementoAuxiliar.equals(elemento4)) {
        // sustituimos el elemento
        iteradorLista.set(elemento5);
    }
}

// recorrer el iterador en reversa
while (iteradorLista.hasPrevious()) {
    E elementoAuxiliar = iteradorLista.previous();

    if (elementoAuxiliar.equals(elemento5)) {
        // eliminamos el elemento
        elementoAuxiliar.add(elemento4);
    }
}
```

ArrayList

Esta es la implementación mas típica de la interfaz.

Se basa en un array redimensionable que aumenta su tamaño según crece la colección de elementos.

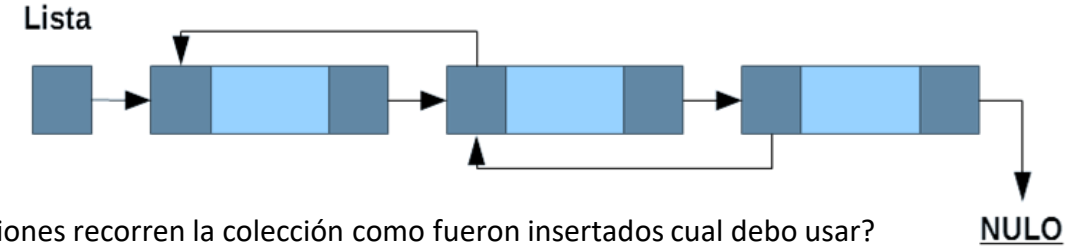
Es la que mejor rendimiento tiene sobre la mayoría de situaciones y como podemos observar se parece mucho a la forma que trabajamos los arreglos comunes con acceso por índice.

```
AbstractList<String> nombres = new ArrayList<>();
List<String> nombresA = new ArrayList<>();
ArrayList<String> nombresB = new ArrayList<>();
```

LinkedList

Esta implementación se basa en una lista doblemente enlazada de los elementos, teniendo cada uno de los elementos un puntero al anterior y al siguiente elemento

```
AbstractList<String> nombres = new LinkedList<>();
List<String> nombresA = new LinkedList<>();
LinkedList<String> nombresB = new LinkedList<>();
```



Si estas dos implementaciones recorren la colección como fueron insertados cual debo usar?

LinkedList permite inserciones o eliminaciones en mejor tiempo utilizando los iteradores, pero solo acceso secuencial de elementos. **ArrayList** por otro lado, permite un acceso de lectura aleatorio rápido, para que puedas obtener cualquier elemento en un tiempo mas optimo. Pero agregar o eliminar desde cualquier lugar menos el final requiere desplazar todos los últimos elementos.