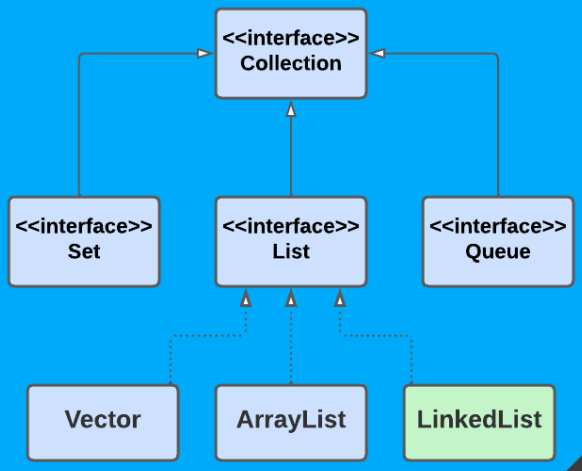
Java

# //Estructura de datos




UTEC
Universidad Tecnológica

LICENCIATURA EN TECNOLOGÍAS
DE LA INFORMACIÓN

# Estructuras de datos en JAVA

## Introducción

Java tiene implementaciones de estructuras de datos o tipo de datos ya definidos que nosotros podemos utilizar en nuestros desarrollos.

## Estructuras de datos

Las estructuras de datos que vamos a ver, siempre implementan una interfaz llamada **Collection**.
¿Qué logramos con esto? Que todo lo que sea una colección, debe implementar esta interfaz, por lo tanto, debe tener una serie de operaciones y servicios mínimos necesarios para poder manejar esta colección. Algunos de estos métodos son:

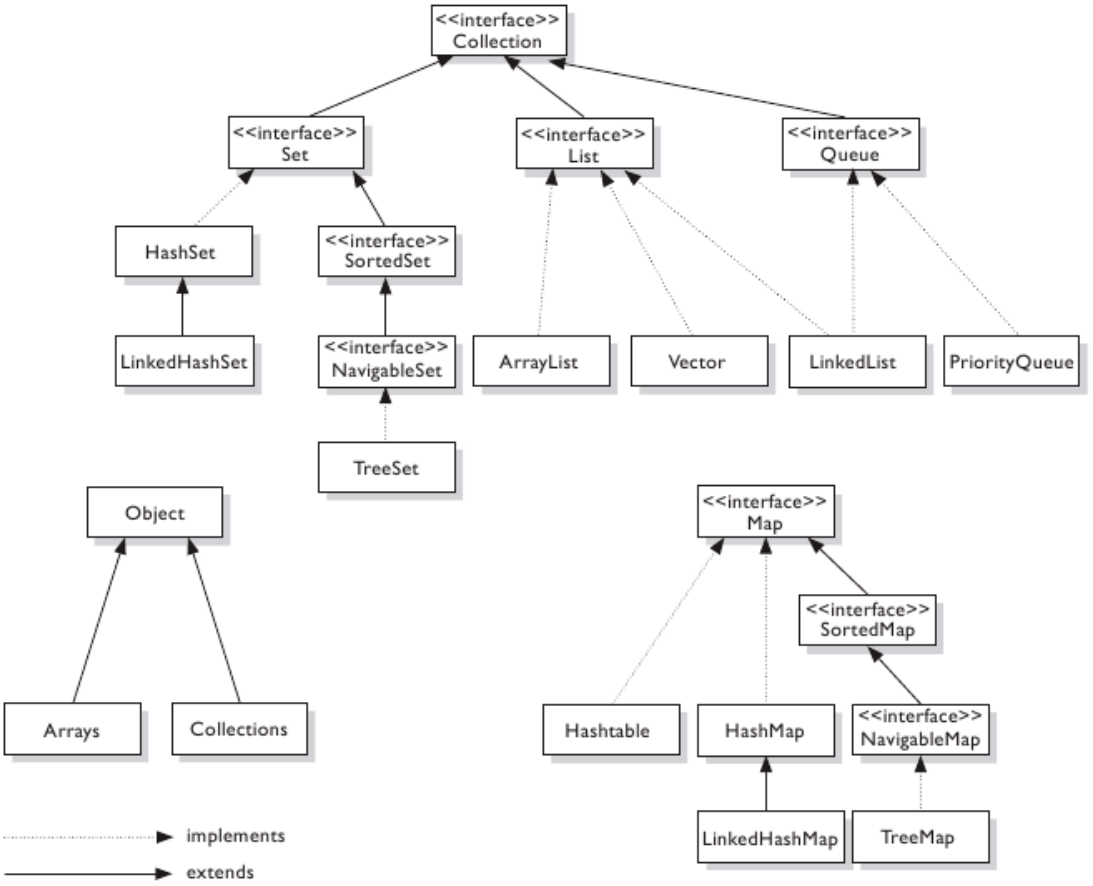
- add
- size
- clear
- contains
- isEmpty
- remove

Existen más (ver en la documentación - https://docs.oracle.com/javase/17/docs/api/ -) pero hemos tomado estos como ejemplo para mostrar ya que son los más utilizados a la hora de programar.

Luego, tenemos diferentes implementaciones de esta interfaz, que java nos provee. Todas estas, implementan las operaciones anteriores y más, y las diferencias entre las distintas clases que implementan esta interfaz, es la manera que se manejan internamente, pero el resultado siempre es el mismo (agregar un elemento, sacar elemento, etc.).

Por ejemplo, la operación add siempre va a agregar un elemento a la colección. Luego, como lo haga internamente, es decisión de la implementación en cuestión.

A continuación, se muestra en forma de diagrama, las diferentes implementaciones que existen de la interfaz Collection:




Como se puede ver en el esquema, existen otras tres sub interfaces que implementan collection, estas son: **Set**, **List** y **Queue**. Las grandes diferencias entre estas tres son conceptuales: cómo se accede a sus elementos y cómo se ordenan los mismos.

- Por un lado, un **Set** (o conjunto en español), es un conjunto donde no puede haber elementos repetidos.
- Por otro lado, las listas (**List**), son implementaciones donde podemos agregar elementos que sí pueden repetirse.
- Y por último tenemos las Colas (“**Queue**”), que son implementaciones donde los elementos se van agregando al final, y sacando del principio.

De estas interfaces, ya tenemos implementaciones provistas, estas son las siguientes:

Implementaciones para “List”:

- ArrayList
- Vector
- LinkedList

Implementaciones para “Set”:

- HashSet (y una subclase de esta que es LinkedHashSet).

Implementaciones para “Queue”:

- LinkedList
- PriorityQueue

Los resultados y la forma de uso de las implementaciones de una misma interfaz es la misma, la principal diferencia entre ambas es la performance (desempeño), esta diferencia de performance a efectos de este curso es imperceptible y no será de importancia por lo que podremos utilizar una u otra indistintamente.

Veamos ahora un programa, utilizando una lista:

```java
import java.util.LinkedList;

public class Main {
	public static void main(String[] args) {

		LinkedList<String> listaStrings = new LinkedList<String>();
		//Agregar elementos a la lista

		listaStrings.add(“Hola”);
		listaStrings.add(“chau”);
		listaStrings.add(“otro texto”);

		//consulta de tamaño

		int tamaño = listaStrings.size();

		//consulta por vacío

		boolean esVacia = listaStrings.isEmpty();

		//contiene elemento
```

```
            boolean contains1 = listaStrings.contains("este texto");
            boolean contains2 = listaStrings.contains("hola");

            //devolver el elemento en la posición 3 de la lista

            String strings = listaStrings.get(3);

        }
}
```

Este programa, muestra cómo crear una lista y como utilizar algunos de sus métodos.

**SINTAXIS**
***ClaseDeLista<TipoDeDatoDeLista> nombreVariable = new ClaseDeLista<TipodeDatoDeLista>()***

Cabe destacar, que cada vez que definamos un objeto de tipo lista, debemos indicarle el tipo de datos que almacena la lista. Esto lo hacemos mediante <TipodeDatoDeLista >.

El tipo de dato de una lista tiene que ser una clase, o sea no puede ser un tipo de datos de los primitivos. Ejemplos de clases que puede ser una lista, String, los wrappers.

Tipos Primitivos y Objetos:

La diferencia entre los tipos primitivos (int, char,double, etc.) que son eficientes pero carecen de las características de los objetos, con atributos y métodos.
Los objetos son instancias de clases que pueden tener atributos y métodos.

Wrappers:
Los wrappers o clases envolventes son una solución que permiten a los programadores utilizar los tipos primitivos como si fueran objetos. Cada tipo primitivo tiene una clase wrapper correspondiente en Java(Ejemplo Integer para int, Double para double,etc.)

También una lista puede tener una clase creada por los programadores, como por ejemplo Persona, Producto, Auto, etc.

Una vez instanciado un objeto de tipo lista, podremos realizar un conjunto de operaciones sobre esta:

**ADD**

Utilizando el add de una lista agregamos el elemento que se pasa como parámetro al final de dicha lista.

```java
listaStrings.add(“Hola”);
listaStrings.add(“chau”);
listaStrings.add(“otro texto”);
```

La lista de nombre listaStrings queda con elementos: “Hola”, “chau”, “otro texto” en ese orden.

**SIZE**

```java
listaStrings.size();
```

Obtiene la cantidad de elementos de una lista. Suponiendo que se realizaron los add anteriores, el tamaño de la lista es 3.

**ISEMPTY**

```java
listaStrings.isEmpty();
```

Retorna true si la lista está vacía. Equivale a: listaStrings.size() == 0.

**CONTAINS**

Retorna true si elemento se encuentra dentro de la lista varLista, false, en caso contrario.

```java
listaStrings.contains(“este texto”);
```

La llamada anterior retorna false, ya que “este texto” no se encuentra en la lista listaStrings.

**GET** varLista.get(índice)

Retorna el elemento que se encuentra en la posición índice de la lista varLista.

```java
listaStrings.get(3);
```

Retorna el elemento de índice 3 de listaStrings. Los índices al igual que en los array comienzan en 0, por lo que al hacer un get de posición 3, el programa lanza una excepción, ocurre un error de ejecución. Ya que no existe el elemento de índice 3.

Pero por ejemplo, al hacer listaStrings.get(1) este devuelve “chau”.

## Recorrer las colecciones

En muchos de nuestros algoritmos y lógica que tengamos, nos será útil recorrer de principio a fin una colección para su procesamiento.

*ITERANDO COLLECTIONS CON FOR*

Una forma de hacerlo es con la sentencia **for**, adaptada para colecciones. Es muy similar a la que hemos visto, pero cambia levemente su sintaxis.

```java
import java.util.LinkedList;

public class Programa {
	public static void main(String[] args) {

		LinkedList<String> listaStrings = new LinkedList<String>();

		//Agregar elementos a la lista

		listaStrings.add(“Hola”);
		listaStrings.add(“chau”);
		listaStrings.add(“otro texto”);

		for (String s : listaStrings){
			System.out.println(s);
		}
	}
}
```

Podemos ver en el ejemplo, que lo que estamos haciendo es “aplicar el for a la colección”. Se puede ver que no se utiliza un índice para decirle a la sentencia for cuantas veces iterara.

Esta variante del for, está pensada exclusivamente para las colecciones, y lo que hace es iterar sobre todos los elementos de la colección en el orden que se encuentren agregados a la misma.

La sintaxis de esta sentencia es la siguiente:

*for (TipoDeDaElemento nombreVariable: coleccionAlterar){*
*//procesamiento del elemento nombreVariable*
*}*

Donde:

- TipoDeDatoElemento: es el tipo de datos que se almacena en la colección en la que estamos iterando.

- nombreVariable: es el nombre que queremos darle al elemento que estamos procesando, para referirnos a él dentro del bloque for, esta sería la variable de control. Esta es la variable utilizada para recorrer los elementos de coleccionAlterar.

- coleccionAlterar: es el nombre de la variable colección sobre la cual queremos iterar.

Luego, dentro de nuestro bloque for, se puede acceder al elemento de la colección a través de nombreVariable.

Por ejemplo:

```java
for (String s : listaStrings){
	Systemout.println(s);
}

for (int i = 0; i < listaStrings.size(); i++){
	System.out.println(listaStrings.get(i));
}
```

Ambas sentencias de for **realizan exactamente lo mismo**, pero la primera forma es más sencilla de utilizar por el programador.

Por ejemplo: en la primera, en la variable “s” ya nos queda almacenado el valor del elemento sin tener que hacer el .get correspondiente, como si se debe hacer en la segunda.

*ITERANDO COLLECTIONS CON OBJETO “ITERATOR”*

Otra manera que tenemos de Iterar en colecciones es a través de un objeto llamado **Iterator**.

Toda colección, tiene un método iterator() que nos devuelve un Iterador de la colección que estamos parados. El iterador, se sitúa por defecto en el primer elemento de la colección, luego,

el objeto iterador, tiene dos métodos claves hasNext() y next(). El primero, nos devuelve true si tenemos un próximo elemento en la colección (false en caso contrario) y el segundo, nos devuelve el elemento en la posición actual, y mueve el iterador a la siguiente posición. Por lo tanto, ahora podemos tener una recorrida de la lista de la siguiente manera:

```java
import java.util.Iterator;
import java.util.LinkedList;

public class Programa {
	public static void main(String[] args) {

		LinkedList<String> listaStrings = new LinkedList<String>();

		//Agregar elementos a la lista

		listaStrings.add("Hola");
		listaStrings.add("chau");
		listaStrings.add("otro texto");

		Iterator<String> it = listaStrings.iterator();

		while (it.hasNext()){
			String s = ir.next();
			System.out.println(s);
		}
	}
}
```

**Uso de Mapas**

Otra estructura muy útil a la hora de programar son los llamados map (mapas).

**Un mapa es una estructura de tupla (dos elementos) donde uno juega el papel de clave y otro de valor almacenado.**

Esto nos permite agregar elementos a esta colección, por una clave en particular, y luego obtenerlos por la misma, sin necesidad de recorrer toda la colección buscando dicho valor. Ya que las claves almacenadas son únicas.

Al igual que en las Collections, una implementación de map debe tener un mínimo de operaciones básicas, ya que implementa la interfaz llamada Map. Donde, al igual que con la List, tenemos varias clases que la implementan y tienen la lógica de las operaciones.

Veamos algunas de estas operaciones:

- clear

- put
- remove
- containsKey
- get
- isEmpty

Estas son algunas de las operaciones básicas que tiene un map.

Por ejemplo, supongamos que tenemos un registro de personas. Sabemos que la CI de cada persona es única, por lo tanto, un mapa podría ser una estructura adecuada para almacenarlas, y poder obtener a las instancias de personas a través de su CI.

Esto sería más rápido que ir iterando sobre una lista por ejemplo.

Veremos la ventaja de colocarlas en un mapa, contra colocarlas en una lista convencional.

Supongamos que tenemos la clase Persona:

Java

```java
public class Persona {

	private String nombre;
	private String ciudadResidencia;
	private String ci;

	public Persona(String nombre, String ciudadResidencia, String ci){
		this.nombre = nombre;
		this.ciudadResidencia = ciudadResidencia;
		this.ci = ci;
	}

	public String getNombre(){
		return nombre;
	}

	public void setNombre(String nombre){
		this.nombre = nombre;
	}

	public String getCiudadResidencia(){
		return ciudadResidencia;
	}

	public void setCiudadResidencia(String ciudadResidencia){
		this.ciudadResidencia = ciudadResidencia;
	}

	public String getCi(){
		return ci;
	}
```

```
        public void setCi(String ci){
                this.ci = ci;
        }
}
```

Y ahora tenemos el siguiente programa que, crea varias personas, las almacena en un mapa y luego devuelve los datos de algunas:

```
Java
import java.util.HashMap;

public class Main {
    public static void main(String[] args) {
        // Creamos el mapa
        HashMap<String, Persona> mapaPersonas = new HashMap<String, Persona>();

        //Creamos instancias de personas
        Persona p1 = new Persona("Andres", "Montevideo", "4.547.166-8");
        Persona p2 = new Persona("Virginia", "Rivera", "4.196.456-1");
        Persona p3 = new Persona("Juan Pablo", "Mercedes", "3.247.186-5");

        // Agregamos las personas con sus respectivas claves, al mapa
        mapaPersonas.put("4.547.166-8", p1);
        mapaPersonas.put("4.196.456-1", p2);
        mapaPersonas.put("3.247.186-5", p3);

        // Obtenemos la persona de CI que deseamos
        Persona personaDelMapa = mapaPersonas.get("4.547.166-8");

        System.out.println("Nombre: " + personaDelMapa.getNombre());
        System.out.println("Residencia: " +
personaDelMapa.getCiudadResidencia());
        System.out.println("CI: " + personaDelMapa.getCi());
    }
}
```

Al igual que con las listas, debemos indicar cuando construimos el mapa, el tipo de datos: de la clave y del valor almacenado. Esto los hacemos colocando <TipoDatoClave,TipoDatoValor> cuando definimos la variable del mapa.

Como se puede ver en el ejemplo, con el get le pedimos al mapa un objeto almacenado por su clave, esto nos devuelve el valor almacenado (que en este ejemplo son objetos de personas). El hecho de haber utilizado mapas para este ejemplo, nos ahorra recorrer toda la colección en busca del elemento que necesitamos.

Ahora veamos cómo lo haríamos con una lista:

```java
import java.util.HashMap;
import java.util.Iterator;
import java.util.LinkedList;

public class Programa {
  public static void main(String[] args){

    //Creamos la lista

    Persona p1 = new Persona("Andres", "Montevideo", "4.547.166-8");
    Persona p2 = new Persona("Virginia", "Rivera", "4.196.456-1");
    Persona p3 = new Persona("Juan Pablo", "Mercedes", "3.247.186-5");

    //Agregamos las personas con sus respectivas claves a la lista

    listaPersonas.add(p1);
    listaPersonas.add(p2);
    listaPersonas.add(p3);

    //Buscamos en la lista la persona con CI que deseamos

    Persona personaDeLista = null;
    for (Persona p: listaPersonas){
      if (p.getCi()=="4.547.166-8"){
        personaDeLista = p;
      }
    }

    System.out.println("Nombre: " + personaDeLista.getNombre());
    System.out.println("Residencia: " +
personaDeLista.getCiudadResidencia());
    System.out.println("CI: " + personaDeLista.getCi());
  }
}
```

Vemos que el código queda más complejo.

En este caso, debemos nosotros buscar el elemento en la lista, iterando de manera “manual” preguntado a cada elemento si tiene la CI que estamos buscando.

Por lo tanto, en este caso es conveniente tener las personas almacenadas en un mapa, donde la clave del mismo sea la CI de ellos. Esto es posible ya que la CI de las instancias persona no se repiten.

Definimos una clase Persona de la siguiente manera:

Java

```
public class Persona {

	public String nombre;
	public int edad;
	public byte cantHijos;

	public Persona(String nombre, int edad, byte hijos){
		this.nombre = nombre;
		this.edad = edad;
		this.cantHijos = hijos;
	}

	public String toString(){
		return "Persona [nombre=" + nombre + ", edad=" + edad + ", cantidad de hijos=" + cantHijos + "]";
	}
}
```

En la clase Principal, con el método main, instancio personas, creo una lista con ellas y luego

Java

```
import java.util.LinkedList;

public class Programa {
  public static void main(String[] args){

    Persona p1 = new Persona("Tito", 40, (byte)2);
    Persona p2 = new Persona("Bety", 30, (byte)0);
    Persona p3 = new Persona("Pablo", 75, (byte)3);
    Persona p4 = new Persona("Maria", 15, (byte)0);
    Persona p5 = new Persona("Claudia", 23, (byte)1);

    LinkedList<Persona> personal = new LinkedList<Persona>();
    personal.add(p1);
    personal.add(p2);
    personal.add(p3);
    personal.add(p4);
    personal.add(p5);

    for (Persona p:personal){
        System.out.println(p);
    }

  }
}
```

Java

```
Persona [nombre=Tito, edad=40, cantidad de hijos=2]
Persona [nombre=Bety, edad=30, cantidad de hijos=0]
Persona [nombre=Pablo, edad=75, cantidad de hijos=3]
Persona [nombre=Maria, edad=15, cantidad de hijos=0]
```

```
Persona [nombre=Claudia, edad=23, cantidad de hijos=1]
```

Este for me obliga a recorrer toda la lista, que en este caso se ajusta a lo que quiero, recorro **toda** la lista para mostrar en consola su contenido.

Si ahora, lo que se necesita es recorrer la lista, pero NO necesariamente toda la lista, vamos a utilizar otra estructura de repetición que sea más eficiente y me permita salir de la recorrida, sin necesidad de llegar al último elemento.

Por ejemplo:

Java

```java
import java.util.LinkedList;

public class Programa {
  public static void main(String[] args){

    Persona p1 = new Persona("Tito", 40, (byte)2);
    Persona p2 = new Persona("Bety", 30, (byte)0);
    Persona p3 = new Persona("Pablo", 75, (byte)3);
    Persona p4 = new Persona("Maria", 15, (byte)0);
    Persona p5 = new Persona("Claudia", 23, (byte)1);

    LinkedList<Persona> personal = new LinkedList<Persona>();
    personal.add(p1);
    personal.add(p2);
    personal.add(p3);
    personal.add(p4);
    personal.add(p5);

  //informar si hay alguna persona que no tiene hijos en la lista
  //usamos un for que nos permita cortar la iteración, porque no sería
necesario recorrer toda la lista
  //alcanza que se encuentre una persona que no tenga hijos para
responder la pregunta

  boolean encontre = false; //definimos una variable que oficie de bandera

    for (int i = 0; i < personal.size() && !encontre; i++){
        if (personal.get(i).cantHijos==0){
               encontre=true;
               System.out.println(personal.get(i));
        }
    }

  }
}
```

```
Java
Persona [nombre=Bety, edad=30, cantidad de hijos=0]
```

Uso un for clásico, donde en la condición de fin le agregamos un control con una bandera para que corte la repetición inmediatamente después de haberlo encontrado.

También podríamos haber utilizando un while, de la siguiente manera:

```
Java
boolean encontre = false; //definimos una variable que oficie de bandera
int i = 0;
while(i < personal.size() && !encontre){
	if (personal.get(i).cantHijos==0){
		encontre = true;
		System.out.println(personal.get(i));
	}
	i++;
}
```

## Clase Array para usar con las listas.

Como mencionamos cuando vimos los arrays, hay una forma a partir de un array pasarlo a una lista. Para ello el array a pasar debe de ser el tipo de dato una clase, para poder cumplir que las listas usan solo clases en el tipo de dato, sino se genera un error.

```
public class Eje1 {
    public static void main(String[] args) {
        int [] numeros = {10,14,32,12,15};
        List<Integer> lista = Arrays.asList(numeros);
    }
}
```

En este caso no nos permite pasar de un array a lista, ya que no cumple con el tipo de dato del array es un int que es un tipo de dato primitivo. Para corregir esto debemos cambiar el int por Integer para que pueda funcionar de acuerdo a los requerimientos.

```java
public class Eje1 {
    public static void main(String[] args) {
        Integer [] numeros = {10,14,32,12,15};
        List<Integer> lista = Arrays.asList(numeros);
        System.out.println(lista);
    }
}
```

```
[10, 14, 32, 12, 15]
```

De esta forma funciona correctamente. Una de las características de pasar un array a una lista, es que la lista que se crea de esta forma no es mutable, eso implica que la lista no se puede modificar su tamaño, por lo que no se puede borrar o agregar elementos de la lista nueva creada.
Para poder tener una lista mutable, deberíamos crear una nueva lista y pasarle todos los elementos a la nueva lista.

## Clase Collections

La clase Collections en Java proporciona una serie de métodos estáticos que nos sirven para trabajar con las listas en general. Veremos algunos métodos con ejemplos explicativos.

### sort(List<T> list):

Ordena la lista proporcionada en orden ascendente, según el orden natural de sus elementos. Todos los elementos en la lista deben implementar la interfaz Comparable.

```java
public class Eje2 {
    public static void main(String[] args) {
        List<String> nombres = new ArrayList<String>();
        nombres.add("Hector");
        nombres.add("Diego");
        nombres.add("Alberto");
        nombres.add("Tatiana");
        nombres.add("Gaston");
        Collections.sort(nombres);
        System.out.println(nombres);
    }
}
```

```
[Alberto, Diego, Gaston, Hector, Tatiana]
```

### shuffle(List<?> list):

Mezcla aleatoriamente la lista especificada. Utiliza este método para desordenar una lista de manera que la posición de cada elemento sea aleatoria.

```java
public class Eje3 {
    public static void main(String[] args) {
        List<String> nombres = new ArrayList<String>();
        nombres.add("Hector");
        nombres.add("Diego");
        nombres.add("Alberto");
        nombres.add("Tatiana");
        nombres.add("Gaston");
        Collections.shuffle(nombres);
        System.out.println(nombres);
    }

}
```

```
[Tatiana, Hector, Alberto, Diego, Gaston]
```

### reverse(List<?> list):

Invierte el orden de los elementos en la lista proporcionada. Esto es útil cuando necesitas invertir el orden actual de los elementos.

```java
public class Eje4 {
    public static void main(String[] args) {
        List<String> nombres = new ArrayList<String>();
        nombres.add("Hector");
        nombres.add("Diego");
        nombres.add("Alberto");
        nombres.add("Tatiana");
        nombres.add("Gaston");
        Collections.sort(nombres);
        Collections.reverse(nombres);
        System.out.println(nombres);
    }
}
```

```
[Tatiana, Hector, Gaston, Diego, Alberto]
```

### binarySearch(List<? extends Comparable<? super T>> list, T key):

Realiza una búsqueda binaria de la clave especificada en la lista ordenada proporcionada. Devuelve el índice del elemento buscado, o un índice negativo si el elemento no está presente.

```java
public class Eje5 {
    public static void main(String[] args) {
        // Crear una lista y añadir elementos
        List<Integer> numeros = new ArrayList<>();
        numeros.add(1);
        numeros.add(3);
        numeros.add(5);
        numeros.add(7);
        numeros.add(9);
        // Es crucial que la lista esté ordenada antes de realizar la búsqueda binaria
        Collections.sort(numeros);
        // Buscar un elemento en la lista
        int indiceBuscado = Collections.binarySearch(numeros, key: 5);

        if (indiceBuscado >= 0) {
            System.out.println("El elemento buscado está en el índice: " + indiceBuscado);
        } else {
            System.out.println("El elemento buscado no está en la lista. Punto de inserción: " + (-indiceBuscado - 1));
        }

        // Intentar buscar un elemento que no existe
        int indiceInexistente = Collections.binarySearch(numeros, key: 6);

        if (indiceInexistente < 0) {
            System.out.println("El elemento no existe. Punto de inserción: " + (-indiceInexistente - 1));
        }
    }
}
```

```
El elemento buscado está en el índice: 2
El elemento no existe. Punto de inserción: 3
```

### fill(List<? super T> list, T obj):

Reemplaza todos los elementos de la lista proporcionada con el objeto especificado. Útil para inicializar o restablecer todos los elementos de una lista.

```
public class Eje6 {
    public static void main(String[] args) {
        // Crear una lista con algunos valores iniciales
        List<String> lista = new ArrayList<>();
        lista.add("Uno");
        lista.add("Dos");
        lista.add("Tres");

        System.out.println("Lista antes de fill: " + lista);

        // Usar Collections.fill para reemplazar todos los elementos
        Collections.fill(lista,  obj: "Nuevo");
        System.out.println("Lista después de fill: " + lista);
    }
}
```

```
Lista antes de fill: [Uno, Dos, Tres]
Lista después de fill: [Nuevo, Nuevo, Nuevo]
```

Este método es particularmente útil cuando necesitas resetear los valores de una lista o establecer todos sus elementos a un valor predeterminado de manera rápida y eficiente.

### max(Collection<? extends T> coll):

Devuelve el valor máximo de la colección dada, según el orden natural de sus elementos. Es especialmente útil para encontrar el elemento más grande en una lista. También está el método min para el elemento más chico de la lista.

```
public class Eje {
    public static void main(String[] args) {
        Integer[] numeros = {23,12,56,98,12,-3};
        List<Integer> lista = Arrays.asList(numeros);
        System.out.println("El mas grande es: "+Collections.max(lista));
        System.out.println("El mas chico es: "+Collections.min(lista));
    }
}
```

```
El mas grande es: 98      El mas grande es: 98
El mas chico es: -3       El mas chico es: -3
```

## replaceAll(List<T> list, T oldVal, T newVal):

Reemplaza todas las ocurrencias del valor especificado en la lista con un nuevo valor.

```java
public class Eje7 {
    public static void main(String[] args) {
        // Crear una lista con algunos valores
        List<String> lista = Arrays.asList("manzana", "banana", "cereza", "manzana", "durazno");
        System.out.println("Lista original: " + lista);

        // Reemplazar todas las ocurrencias de "manzana" con "naranja"
        Collections.replaceAll(lista, oldVal: "manzana", newVal: "naranja");

        System.out.println("Lista después de replaceAll: " + lista);
    }
}
```

```
Lista original: [manzana, banana, cereza, manzana, durazno]
Lista después de replaceAll: [naranja, banana, cereza, naranja, durazno]
```

## swap(List<?> list, int i, int j):

Intercambia los elementos en las posiciones especificadas en la lista.

```java
public class Eje8 {
    public static void main(String[] args) {
        // Crear una lista con algunos elementos
        List<String> lista = Arrays.asList("Rojo", "Verde", "Azul", "Amarillo", "Negro");
        System.out.println("Lista antes de swap: " + lista);
        // Intercambiar elementos: el primero (índice 0) con el tercero (índice 2)
        Collections.swap(lista, i: 0, j: 2);
        System.out.println("Lista después de swap: " + lista);
        // Intercambiar elementos: el segundo (índice 1) con el último (índice 4)
        Collections.swap(lista, i: 1, j: 4);
        System.out.println("Lista después del segundo swap: " + lista);
    }
}
```

```
Lista antes de swap: [Rojo, Verde, Azul, Amarillo, Negro]
Lista después de swap: [Azul, Verde, Rojo, Amarillo, Negro]
Lista después del segundo swap: [Azul, Negro, Rojo, Amarillo, Verde]
```

## Pilas y Colas en Java

Las pilas (Stacks) y las colas (Queues) son estructuras de datos fundamentales en la programación, incluyendo en Java, donde se utilizan ampliamente debido a su eficiencia y versatilidad. Ambas estructuras tienen sus propias características y usos específicos. A continuación, se detallan los conceptos teóricos de ambas:

Pilas (Stacks)

- Concepto: Una pila es una estructura de datos lineal que sigue el principio de Last In, First Out (LIFO), lo que significa que el último elemento añadido es el primero en ser eliminado. Imagina una pila de platos; el último plato que colocas en la cima es el primero que se retirará.
- Operaciones Principales:
    - Push: Agrega un elemento en la cima de la pila.
    - Pop: Elimina y devuelve el elemento de la cima de la pila.
    - Peek: Devuelve el elemento de la cima de la pila sin eliminarlo.
    - isEmpty: Comprueba si la pila está vacía.
- Usos Comunes: Las pilas son útiles en situaciones donde necesitas revertir operaciones, navegar entre estados anteriores (como el botón atrás en los navegadores), implementar algoritmos recursivos, y en la evaluación de expresiones matemáticas (por ejemplo, en conversores de notación infija a postfija).

Colas (Queues)

- Concepto: Una cola es una estructura de datos lineal que sigue el principio de First In, First Out (FIFO), lo que significa que el primer elemento añadido es el primero en ser eliminado. Es similar a una fila de personas esperando su turno; la primera persona en la fila es la primera en ser atendida.
- Operaciones Principales:
    - Enqueue (Offer): Añade un elemento al final de la cola.
    - Dequeue (Poll): Elimina y devuelve el elemento del frente de la cola.
    - Peek: Devuelve el elemento del frente de la cola sin eliminarlo.
    - isEmpty: Comprueba si la cola está vacía.
- Usos Comunes: Las colas son esenciales en la programación de sistemas operativos para la gestión de procesos, en la implementación de buffers, colas de mensajes y en algoritmos de búsqueda como el recorrido por niveles en árboles o grafos (BFS - Breadth-First Search).

Implementación en Java

- Pilas: Aunque Java tiene una clase Stack, se recomienda usar la interfaz Deque (como ArrayDeque) para implementar pilas, ya que proporciona una implementación más completa y consistente.
- Colas: Java proporciona la interfaz Queue, que se implementa en varias clases como LinkedList y PriorityQueue. Para colas concurrentes, existen implementaciones como ArrayBlockingQueue y LinkedBlockingQueue.

Ambas estructuras, pilas y colas, son fundamentales para resolver diversos problemas de programación y son ampliamente utilizadas en el desarrollo de software. Comprender sus características y saber cuándo utilizar cada una es crucial para cualquier desarrollador de Java.

## Ejemplos de Pila

```java
public class EjemploPila {
    public static void main(String[] args) {
        Stack<Integer> pila = new Stack<>();
        // Apilando elementos (push)
        pila.push( item: 10);
        pila.push( item: 20);
        pila.push( item: 30);
        System.out.println("Elementos de la pila: " + pila);
        // Desapilando elementos (pop)
        System.out.println("Elemento desapilado: " + pila.pop());
        System.out.println("Elemento desapilado: " + pila.pop());
        // Observando el elemento superior de la pila sin eliminarlo
        System.out.println("Elemento superior: " + pila.peek());
        // Verificando si la pila está vacía
        System.out.println("La pila está vacía: " + pila.isEmpty());
    }
}
```

```
Elementos de la pila: [10, 20, 30]
Elemento desapilado: 30
Elemento desapilado: 20
Elemento superior: 10
La pila está vacía: false
```

## Ejemplos de Cola:

```java
public class EjemploCola {
    public static void main(String[] args) {
        // Creación de una cola
        Queue<String> cola = new LinkedList<>();

        // Añadiendo elementos a la cola (enqueue)
        cola.offer( e: "Trabajo 1");
        cola.offer( e: "Trabajo 2");
        cola.offer( e: "Trabajo 3");
        cola.offer( e: "Trabajo 4");

        System.out.println("Elementos de la cola: " + cola);

        // Eliminando elementos de la cola (dequeue)
        System.out.println("Elemento removido: " + cola.poll());
        System.out.println("Elemento removido: " + cola.poll());
        // Mostrando los elementos de la cola después de las operaciones de remover
        System.out.println("Elementos de la cola después de remover elementos: " + cola);
        // Observar el frente de la cola sin removerlo
        System.out.println("Frente de la cola: " + cola.peek());
    }
}
```

```
Elementos de la cola: [Trabajo 1, Trabajo 2, Trabajo 3, Trabajo 4]
Elemento removido: Trabajo 1
Elemento removido: Trabajo 2
Elementos de la cola después de remover elementos: [Trabajo 3, Trabajo 4]
Frente de la cola: Trabajo 3
```