PROGRAMACIÓN

UNIDAD 8: **Colecciones de datos**

1. Introducción

- Java nos proporciona un amplio conjunto de clases entre las que podemos encontrar tipos de datos que nos resultarán muy útiles para realizar la programación de aplicaciones en Java.
- Se proporcionan una serie de operadores para acceder a los elementos de estos tipos de datos.
- Dichos operadores son *polimórficos*, ya que un mismo operador se puede emplear para acceder a distintos tipos de datos.
- Este *polimorfismo* se debe a la definición de interfaces que deben implementar los distintos tipos de datos. Siempre que el tipo de datos contenga una colección de elementos, implementará la interfazCollection.
- Esta interfaz proporciona métodos para acceder a la colección de elementos, que podremos utilizar para cualquier tipo de datos que sea una colección de elementos, independientemente de su implementación concreta.

2. Colecciones

Las colecciones representan grupos de objetos, denominados elementos. Podemos encontrar diversos tipos de colecciones, según si sus elementos están ordenados, o si permitimos repetición de elementos o no.

Las colecciones vienen definidas por la interfaz Collection, de la cual heredará cada subtipo específico. En esta interfaz encontramos una serie de métodos que nos servirán para acceder a los elementos de cualquier colección de datos, sea del tipo que sea.

2. Colecciones

Estos métodos generales son:

- **boolean add(Object o)**

  Añade un elemento (objeto) a la colección. Nos devuelve true si tras añadir el elemento la colección ha cambiado, es decir, el elemento se ha añadido correctamente, o false en caso contrario.

- **void clear()**

  Elimina todos los elementos de la colección.

- **boolean contains(Object o)**

  Indica si la colección contiene el elemento (objeto) indicado.

- **boolean isEmpty()**

  Indica si la colección está vacía (no tiene ningún elemento).

- **Iterator iterator()**

  Proporciona un iterador para acceder a los elementos de la colección.

2. Colecciones

- **boolean remove(Object o)**

  Elimina un determinado elemento (objeto) de la colección, devolviendo true si dicho elemento estaba contenido en la colección, y false en caso contrario.

- **int size()**

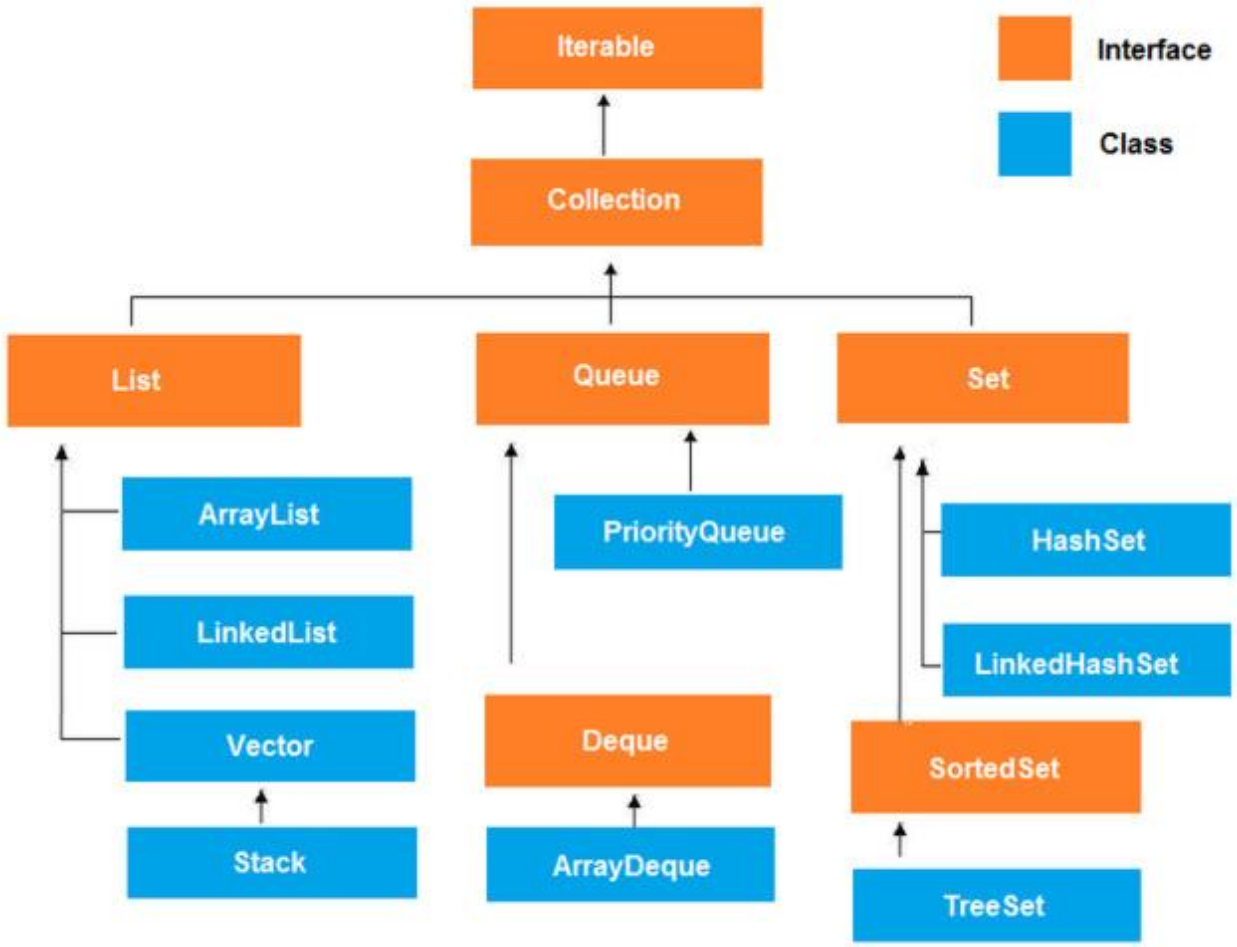
  Nos devuelve el número de elementos que contiene la colección.

- **Object [] toArray()**

  Nos devuelve la colección de elementos como un array de objetos.

Esta interfaz es muy genérica, y por lo tanto no hay ningún tipo de datos que la implemente directamente, sino que implementarán subtipos de ellas. A continuación veremos los subtipos más comunes.

2. Colecciones



2.1. Listas de elementos

Este tipo de colección se refiere a listas en las que los elementos de la colección tienen un orden, existe una secuencia de elementos. En ellas cada elemento estará en una determinada posición (índice) de la lista.

Las listas vienen definidas en la interfaz List, que además de los métodos generales de las colecciones, nos ofrece los siguientes para trabajar con los índices:

- **void add(int indice, Object obj)**

  Inserta un elemento (objeto) en la posición de la lista dada por el índice indicado.

- **Object get(int indice)**

  Obtiene el elemento (objeto) de la posición de la lista dada por el índice indicado.

2.1. Listas de elementos

- **int indexOf(Object obj)**

  Nos dice cual es el índice de dicho elemento (objeto) dentro de la lista. Nos devuelve -1 si el objeto no se encuentra en la lista.

- **Object remove(int indice)**

  Elimina el elemento que se encuentre en la posición de la lista indicada mediante dicho índice, devolviéndonos el objeto eliminado.

- **Object set(int indice, Object obj)**

  Establece el elemento de la lista en la posición dada por el índice al objeto indicado, sobrescribiendo el objeto que hubiera anteriormente en dicha posición. Nos devolverá el elemento que había previamente en dicha posición.

2.1. Listas de elementos

Podemos encontrar diferentes implementaciones de listas de elementos en Java:

- **ArrayList**
- **Vector**
- **LinkedList**

2.1.1. Listas de elementos. ArrayList

- Implementa una lista de elementos mediante un array de tamaño variable.
- Conforme se añaden elementos el tamaño del array irá creciendo si es necesario.
- El array tendrá una capacidad inicial, y en el momento en el que se rebase dicha capacidad, se aumentará el tamaño del array.

Declaración:

```
ArrayList<tipo> nombreArray = new ArrayList<tipo>();
```

2.1.1. Listas de elementos. ArrayList

```java
import java.util.ArrayList;

//Esta clase representa una lista de nombres manejada
//con la clase ArrayList de Java
public class ListaNombres {
    private String nombreDeLaLista; //Establecemos un atributo nombre de la lista
    private ArrayList<String> listadenombres; //Declaramos un ArrayList que contiene objetos String

    public ListaNombres (String nombre) {    //Constructor: crea una lista de nombres vacía
        nombreDeLaLista = nombre;
        listadenombres = new ArrayList<String>(); //Creamos el objeto de tipo ArrayList
    } //Cierre del constructor

    public void addNombre (String valor_nombre) {
        listadenombres.add (valor_nombre);
    }

    public String getNombre (int posicion) {
        if (posicion >= 0 && posicion < listadenombres.size() ) {
            return listadenombres.get(posicion); }
        else { return "No existe nombre para la posición solicitada";}
    } //Cierre del método

    public int getTamaño () {
        return listadenombres.size();
    }

    public void removeNombre (int posicion) {  //Método
        if (posicion >= 0 && posicion < listadenombres.size() ) {
            listadenombres.remove(posicion);
        }
    }
}
```

2.1.2. Listas de elementos. Recorrer colecciones Interfaz Iterable, método iterator() e Iterator

Lo primero es tener claro algunos conceptos que no deben confundirse:

- La **interface Iterable** es una interfaz de uso habitual que se utiliza para recorrer colecciones. Implementar **Iterable** tan sólo obliga a sobreescribir un método que es **iterator().**
- **Método iterator()** con minúsculas es un método igual que puede ser toString() o cualquier otro. Este método devuelve un objeto de tipo **Iterator** (con mayúsculas)
- Los objetos de tipo **Iterator** tienen 3 métodos: **hasNext(), next() y remove().** El primero debe devolver un valor boolean indicando si el iterador tiene un siguiente elemento. El método next(), debe devolver el siguiente elemento del iterador, y remove() debe remover o eliminar el anterior objeto devuelto.

2.1.2. Listas de elementos. Recorrer colecciones Interfaz Iterable, método iterator() e Iterator

Recorrer un ArrayList con un objeto Iterator

Las colecciones en Java implementan la inerfaz Iterable, por lo que podremos recorrer sus elementos haciendo uso de un iterador.

Recorrer una colección con un bucle for tiene sus inconvenientes: Supongamos que vamos recorriendo una lista de 20 objetos y que durante el recorrido borramos 5 de ellos. Probablemente nos saltará un error porque Java no sabrá qué hacer ante esta modificación concurrente.

Un objeto de tipo Iterator funciona a modo de copia para recorrer una colección, es decir, el recorrido no se basa en la colección "real" sino en una copia. De este modo, al mismo tiempo que hacemos el recorrido (sustentado en la copia) podemos manipular la colección real, añadiendo, eliminando o modificando elementos de la misma. Para poder usar objetos de tipo Iterator hemos de declarar en cabecera import java.util.Iterator;. La sintaxis es la siguiente:

```
Iterator <TipoARecorrer> it = nombreDeLaColección.iterator ();
```

2.1.2. Listas de elementos. Recorrer colecciones Interfaz Iterable, método iterator() e Iterator

Ejemplo:

```
// Declaración el ArrayList
ArrayList<String> nombreArrayList = new ArrayList<String>();

// Añadimos 10 Elementos en el ArrayList
for (int i=1; i<=10; i++){
    nombreArrayList.add("Elemento "+i);
}

// Añadimos un nuevo elemento al ArrayList en la posición 2
nombreArrayList.add(2, "Elemento 3");

// Declaramos el Iterador e imprimimos los Elementos del ArrayList
Iterator<String> nombreIterator = nombreArrayList.iterator();
while(nombreIterator.hasNext()){
    String elemento = nombreIterator.next();
    System.out.print(elemento+" / ");
}
```

2.1.3. Clase Vector

- La clase Vector, al igual que ArrayList, también implementa a List, pero de un modo especial. Este modo especial es sincronizado, lo que permite que se pueda usar en entornos concurrentes (es decir, en varios procesos que se ejecutan al mismo tiempo y hacen uso posiblemente de los mismos recursos).
- Utilizaremos la clase Vector en lugar de ArrayList si tenemos previstas circunstancias especiales como procesos concurrentes.
- Las características de esta clase son muy parecidas a las de un ArrayList.

```
Vector<tipo> nombreVector = new Vector<tipo>();
```

2.1.3. Pilas de elementos. Clase Stack

- La clase Stack de Java implementa la interface List.
- Stack se traduce por “pila”
- La clase Stack es una clase de las llamadas de tipo LIFO (Last In - First Out, o último en entrar - primero en salir).
- Esta clase hereda de la clase Vector.
- Las operaciones básicas son :
  - **push** (que introduce un elemento en la pila)
  - **pop** (que saca un elemento de la pila)
  - **peek** (consulta el primer elemento de la cima de la pila)
  - **empty** (que comprueba si la pila está vacía)
  - **search** (que busca un determinado elemento dentro de la pila y devuelve su posición dentro de ella).
- Al crear un objeto de tipo Stack con el constructor básico no contendrá ningún elemento.

```
Stack<tipo> nombreArray = new Stack<tipo>();
```

2.1.3. Pilas de elementos. Clase Stack

Ejemplo:

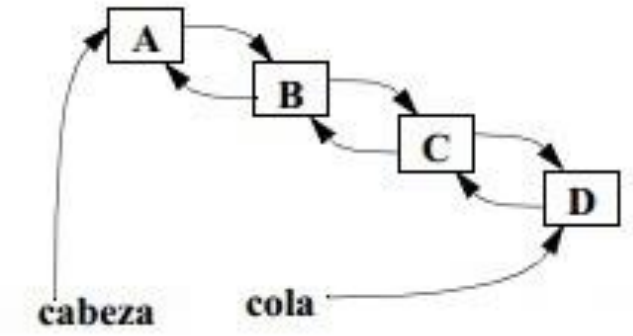
Vamos a ver el uso de la clase Stack con el ejemplo descrito en el siguiente enlace:

Ejemplo de uso de la clase Stack

2.1.5. Clase LinkedList

LinkedList es una clase Java que implementa las interfaces *List* y *Queue* entre otras.

LinkedList implementa métodos para acceder a una lista doblemente enlazada. Acceder a las distintas posiciones de la lista es costoso excepto en la primera y última posición que es inmediato.




```
LinkedList<tipo> lista = new LinkedList<tipo>();
```

2.1.5. Clase LinkedList

Además de los métodos heredados de las interfaces List y Queue, LinkedList implementa métodos específicos:

Descripción	Interfaz
Retorna el primer elemento	`E getFirst()`
Retorna el último elemento	`E getLast()`
Elimina y retorna el primer elemento	`E removeFirst()`
Elimina y retorna el último elemento	`E removeLast()`
Añade o al principio de la lista	`void addFirst(E o)`
Añade o al final de la lista	`void addLast(E o)`

getFirst(), getLast(), removeFirst() y removeLast() lanzan una excepción NoSuchElementException si la lista está vacía.

2.1.5. Clase LinkedList

Como ya hemos comentado, LinkedList implementa la interfaz **Queue** y las operaciones de inserción y extracción son eficientes.

Vamos a ver con un ejemplo como podemos implementar una cola utilizando LinkedList.

Ejemplo:

Escribir una clase para controlar el acceso de clientes a un servicio. Se guardará una cola de espera de clientes y otra cola de clientes ya atendidos. Cada cliente tiene un nombre, un número de móvil. Junto al cliente se guarda su fecha y hora de llegada, y su fecha y hora de atención.

Operaciones

añadir un cliente

atender a un cliente

obtener el tiempo medio de espera de los clientes que aún no han sido atendidos

obtener el tiempo medio de espera de los clientes ya atendidos

mostrar el estado de las colas

Escribir también un programa de prueba. Para la fecha y hora usar la clase predefinida *Calendar*.

2.1.5. Clase LinkedList

Solución:

Vamos a implementar la clase Reloj, que nos dará una medida para calcular el tiempo que tarda en ser atendido un paciente:

```
import java.util.Calendar;
/**
 * Clase que permite obtener la fecha y hora actual,
 * en milisegundos desde la época
 */
public class Reloj
{
	public static long ahora()
	{
		return Calendar.getInstance().getTimeInMillis();
	}
}
```

2.1.5. Clase LinkedList

Ahora vamos a implementar una clase que almacene los datos del cliente:

```
public class DatosCliente {
	String nombre;
	long entrada, salida; // milisegundos
	/** Constructor; pone la hora de entrada*/
	DatosCliente (String c) {
		this.nombre=c;
		entrada=Reloj.ahora();
	}

	void atiende() {
		salida=Reloj.ahora();
	}
}
```

2.1.5. Clase LinkedList

Ahora vamos a implementar la cola de espera, propiamente dicha:

```
public class ColaEspera {

	// colas del servicio
	private Queue<DatosCliente> colaEspera;
	private Queue<DatosCliente> colaAtendidos;

	// Constructor de ColaEspera
	public ColaEspera() {
		colaEspera=new LinkedList<DatosCliente>();
		colaAtendidos=new LinkedList<DatosCliente>();
	}

	// Nuevo cliente; se mete en la cola de espera
	public void nuevoCliente(String c) {
		DatosCliente datos=new DatosCliente(c);
		colaEspera.add(datos);
	}
```

2.1.5. Clase LinkedList

```
// Atender cliente: se saca de la cola de espera y se
mete en la de atendidos;
      public String atenderCliente()throws
NoSuchElementException{
            DatosCliente datos=colaEspera.remove();
            datos.atiende();
            colaAtendidos.add(datos);
            return datos.nombre;
      }
```

2.1.5. Clase LinkedList

```
	public double tiempoEsperaAtendidos()
	{
		long tiempo=0;
		int num=0;
		for (DatosCliente datos: colaAtendidos) {
			tiempo=tiempo+datos.salida-datos.entrada;
			num++;
		}
		if (num==0) {
			return 0.0;
		} else {
			return (((double) tiempo)/num)/1000.0;
	}

}
```