Tema 7:

Estructuras de Almacenamiento Complejas en Java

Indice

1. Introducción

Cuando hemos usado objetos y array de objetos ya hemos usado una estructura de datos. Hay ocasiones en que necesitamos estructuras de datos más complejas.

Si en un problema requerimos que el número de elementos de un array se amplíe o no tenemos un límite máximo de elementos el array no es adecuado.

Para solventar todos estos problemas surgen las agrupaciones de elementos: listas, pilas, colas, tablas hash, conjuntos,…etc. Se suelen llamar **Contenedores**.

Antes de entrar en materia veremos dos conceptos previos que hay que tener claros: **Wrappers** y **Clases y métodos genéricos**.

2. Conceptos previos

2.1 Clases Envoltorio (Wrapper)

En ocasiones es muy conveniente poder tratar los datos primitivos (int, char, boolean) como objetos.

Para ello existen en Java lo que se llama clases envoltorio (wrapper). Sirven para dotar a los datos primitivos con un envoltorio que permita tratarlos como objetos. Las clases envoltorio son:

- `Byte` para `byte`
- `Integer` para `int`
- `Boolean` para `boolean`
- `Float` para `float`, `Double` para `double` ,`Character` para `char`....

```
Integer i = new Integer(5);
int x = i.intValue();
```

2.2. Clases y métodos genéricos

Son clases y métodos que pueden trabajar con diferentes tipos de objetos, facilitando la reutilización de software.

Método génerico

Ejemplo: En una clase **Utilidades** queremos realizar un método estático que vuelque el contenido de un array a otro. Este método será genérico y servirá igual para un array de String que para un array de Persona

```
public class Utilidades {
    public static <T> void volcarArray(T[] origen, T[] destino){

        int tamaño = Math.min(origen.length, destino.length);

        for(int i=0; i < tamaño; i++){

            destino[i]=origen[i];

        }

    }

}
```

Para invocar a un método genérico:

```
public static void main(String[] args) {
      String[] origen = {"Luis", "Pepe", "Lola"};
      String[] destino = new String[7];
      Integer[] array1Int = {10,20,30};
      Integer[] array2Int = new Integer[7];

      Utilidades.volcarArray(origen, destino);
      Utilidades.volcarArray(array1Int, array2Int);

      System.out.println("Array string origen " + Arrays.toString(origen));
      System.out.println("Array sting destino " + Arrays.toString(destino));
      System.out.println("Array Integer origen " + Arrays.toString(array1Int));
      System.out.println("Array Integer destino " + Arrays.toString(array2Int));
}
```

Clases genéricas

- Son clases que pueden trabajar con diferentes tipos de objetos.
- Podríamos crear un clase genérica ArrayConHuecosLibres, donde almacenáramos y borráramos cualquier tipo de elemento, ya sea una Persona, una Figura o un Vehículo.
- Ver el ejemplo de clases genéricas colgado en la plataforma.

3. Contenedores en Java

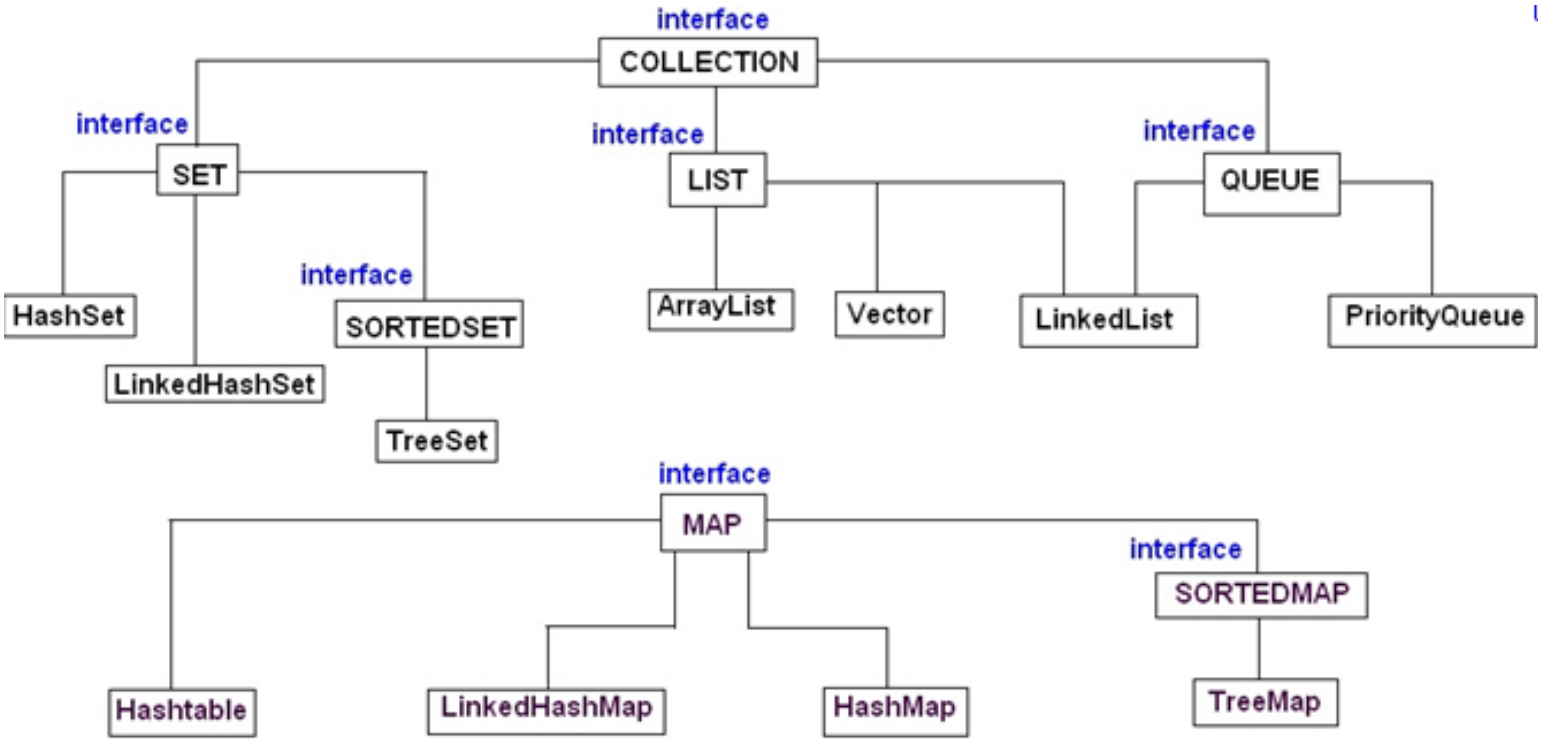
Un contenedor en Java es una agrupación de objetos.

- Tienen su propia asignación de memoria y posibilidad de una nueva asignación para ampliarlas, es decir, tienen gestión de *memoria dinámica*.

- Son estructuras complejas que realizan una gestión interna del almacenamiento y recuperación de los elementos.

- Permiten almacenar objetos, por lo que no se pueden almacenar directamente variables de tipos primitivos. Siempre que tengamos que trabajar con ellos habrá que hacer uso de los Envolventes (Wrappers).

Los contenedores están agrupados en una jerarquía de interfaces y de clases que permiten la utilización de clases ya existentes o heredar de la que mejor se adapte a nuestras necesidades y modificarla.



3.1. Interface *Collection*

- Una *Collection* es una agrupación de elementos que se puede recorrer (o “iterar”) y de lo que se puede saber el tamaño.
- Muchas otras clases implementarán *Collection* imponiendo más restricciones y dando más funcionalidades.
- El interfaz de *Collection* permite añadir, eliminar y recorrer la estructura gracias a un Iterator.
- No se puede construir una Collection, es decir, no se puede hacer “new” de una Collection, ya que es una interface.
- Las clases hijas de la interface Collection tendrán que implementar cada uno de los métodos o ser a su vez Interfaces.

Los métodos de una Collection son:

`int` **`size`**`()`

Obtiene el número de elementos de la colección.

`boolean` **`isEmpty`**`()`

true si la colección está vacía.

`boolean` **`contains`** `(Object element)`

true si la colección contiene un determinado objeto. Ojo: Para comparar objetos las colecciones utilizarán el método equals.

`boolean` **`add`** `(Object element)`

true si se añade el elemento a la colección,
false si el elemento ya existía y no se admiten repetidos, por lo que no se puede añadir.

`boolean` **`remove`** `(Object element)`

Sirve para borrar un determinado objeto de la colección. Devuelve true si se ha encontrado ese objeto y se ha borrado. Ojo porque para encontrar el objeto utiliza equals.

`Iterator` **`iterator`**`()`

Devuelve un objeto “iterador” que permite recorrer los elementos de la colección.

Un *iterador* es un objeto que nos permite recorrer todos los objetos de la colección, al ir invocando progresivamente su método **next**().

`Object[]` **`toArray`**`()`

Devuelve un array con todos los elementos de la colección.

`void` **`clear`**`()`

Elimina los elementos de la colección.

3.2 Interfaz Iterator

- Un Iterador para recorrer / modificar una colección de elementos en Java.
- El método iterator() de Collection devuelve un objeto que implementa esta interfaz.
- Métodos
  - **hasNext**(): Devuelve true si quedan más elementos por tratar.
  - **next**(): Devuelve el siguiente elemento de la colección.
  - **remove**(): Borra el último elemento devuelto por el operador de la colección (con el método next()).

Estructura for para recorrer una Collection

```
for (Object o: Collection)
{
    // Hacer algo con o
}
```

3.3. La interfaz *Set*

Un ***Set*** es una Collection, con la particularidad de que:

- No tiene **elementos repetidos**.
- Los elementos no tienen **ningún orden**.
- La ventaja de utilizar Sets es que los métodos add, remove y contains son muy eficientes.

HashSet:

Es la implementación de Set que más suele usarse. Se basa en una tabla Hash.

```
HashSet<tipo> nombre = new HashSet<tipo>();
```

- Es una clase genérica. En <tipo> se indica el tipo de los elementos del conjunto.
- Es obligatorio que la clase tipo sobre la que se hace el conjunto tenga implementado el método **hashCode**(). Este método retorna un entero que representa el valor hash del objeto. Este valor hash sirve para saber dónde se localiza el objeto dentro de la tabla Hash.
- Si dos objetos son iguales según el método equals entonces ambos deben retornar el mismo valor para el hashCode().

Ejemplo:

```
import java.util.*;
public class TestHashSet{
    public static void main(String[] args) {
        HashSet<String> ciudades = new HashSet<String>();
        ciudades.add("Madrid");
        ciudades.add("Barcelona");
        ciudades.add("Sevilla");
        ciudades.add("Madrid");  //repetido

        Iterator<String> iterador = ciudades.iterator();
        while (iterador.hasNext())
            System.out.println("Ciudad: " + iterador.next());
        // Equivalente a lo anterior
        for (String c: ciudades)
        {
            System.out.println("Ciudad " + c);
        }
    }
}
```

3.4. La interfaz *List*

Un ***List***, es una colección que cumple:

- Puede tener elementos repetidos
- Es relevante el orden de los elementos.
- La interfaz ***List*** declara métodos adicionales, además de los métodos de Collection, que tienen que ver con el orden y acceso a elementos.

Algunos métodos de la Interfaz List:

- **add(Object o):** Añade un objeto al final de la lista.
- **add(int indice, Object o)**: Añade un objeto a la lista en la posición indicada.
- **get(int indice)**: Devuelve el objeto de la lista de la posición indicada (el primero el 0)
- **set(int indice, Object nuevo)**: Reemplaza el objeto que se encuentra en la posición i por el nuevo elemento devolviendo el objeto que ha sido reemplazado.
- **remove(int indice)**: Elimina el objeto de la lista pasado por parámetro.
- **indexOf(Object o)**: Devuelve la posición de la primera vez que un elemento coincida con el objeto pasado por parámetro. Si el elemento no se encuentra devuelve -1.
- **lastIndexOf(Object o)**: Devuelve la posición de la última vez que un elemento coincida con el objeto pasado por parámetro. Si el elemento no se encuentra devuelve -1.

Existen principalmente dos implementaciones de *List*, las dos útiles en distintos casos: ***ArrayList*** y ***LinkedList***.

ArrayList:

La ventaja de *ArrayList* sobre un array normal, es que es expansible, es decir, que crece a medida que se le añaden elementos (mientras que el tamaño de un array es fijo desde su creación). Se define de la siguiente forma:

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

- El tiempo de acceso a un elemento en particular es ínfimo.
- No es la estructura adecuada si queremos eliminar un elemento del principio, o del medio. En este caso debe mover todos los que le siguen a la posición anterior, para “tapar” el agujero que deja el elemento borrado. Esto hace que sacar elementos del medio o del principio sea costoso.

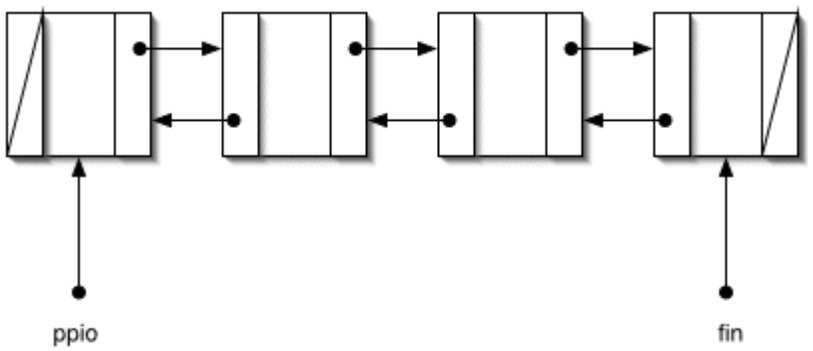
Ejemplo:

Crear un objeto de la clase ArrayList de cadenas y utilizar un iterador para imprimir las cadenas.

```
import java.util.*;

public class TestArrayList {
  public static void main( String args[] ) {
    ArrayList<String> ciudades = new ArrayList<String>();

    ciudades.add("Madrid");
    ciudades.add("Barcelona");
    ciudades.add("Sevilla");
    ciudades.add("Madrid");  //aunque esté repetido se inserta

    //Iteramos sobre el conjunto y nos recorremos la colección
    Iterator<String> iterador = ciudades.iterator();
    while(iterador.hasNext())
       System.out.println(iterador.next());
  }
}
```

LinkedList (lista enlazada):

Los elementos se guardan en una serie de nodos enlazados entre sí. Cada uno de estos nodos tiene una referencia a su antecesor y al elemento que le sigue.




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

Como vemos la lista está parametrizada, indicando el tipo de objeto sobre el que creamos la lista.

La ventaja es que es posible eliminar elementos del principio de la lista y del medio de manera muy eficiente. Para eliminar un elemento

solamente hay que modificar a sus dos “vecinos” para que se “conecten” entre sí ignorando al elemento que se está borrando.

Si utilizamos un LinkedList hay que tener muy en claro sus particularidades en cuanto a rendimiento. Su método get(int) es muy lento porque necesita recorrer para llegar al elemento pedido. Por tanto recorrer una lista de esta forma:

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

ERROR : Esto es ineficiente y lentísimo!!!

Un LinkedList sólo debe recorrerse mediante iteradores.

Ejemplo:

```
import java.util.*;

public class TestLinkedList {
  public static void main( String args[] ) {
    LinkedList<String> ciudades = new LinkedList<String>();

    ciudades.add("Madrid");
    ciudades.add("Barcelona");
    ciudades.add("Sevilla");
    ciudades.add("Madrid");  //repetido

    //Iteramos sobre el conjunto
    Iterator<String> itr = ciudades.iterator();
    while(itr.hasNext())
       System.out.println(itr.next());
  }
}
```

3.5. Listas ordenadas

Crear una lista de forma ordenada

- Una lista puede crearse de forma que los elementos se inserten en un determinado orden.

- Para ello se debe programar el método que inserte en orden. Este método buscará la posición en la que debe insertar y luego llamará a método add(pos, objeto).

- add(pos, objeto): Si la posición es 0 la inserta en el primer lugar de la lista y si es el número de elementos de la lista lo insertará al final.

Ejemplo de lista ordenada

```
public class ListaOrdenada {
	LinkedList <Integer> lista=new LinkedList <Integer> ();

	public void insertarEnOrden( Integer nuevo)
	{
		int pos=buscarSuSito(nuevo);
		lista.add(pos,nuevo);
	}

	private int buscarSuSito(Integer nuevo) {
		// TODO Auto-generated method stub
		boolean encontradoPosicion=false;
		int pos=0;
		Iterator <Integer> it= lista.iterator();
		Integer elemento;


		while ( it.hasNext() && encontradoPosicion== false)
		{
			elemento=it.next();
			if ( nuevo.intValue() < elemento.intValue())
				encontradoPosicion= true;
			else
				pos++;
		}
		return pos;
	}

}
```

Ordenar listas

- Una lista puede ordenarse con el método estático sort de Collection

  Collection.sort(lista)

- Para poder ordenar los elementos de la lista tiene implementar la interfaz **Comparable**, lo que significa programar el método compareTo.

- El método compareTo debe programarse para comparar el objeto actual con otro objeto devolviendo
  - 0 si los objetos son iguales
  - 1 si el objeto es actual es mayor que el otro
  - -1 si el objeto actual es menor que el otro

```java
// Clase Alumno

public class Alumno implements Comparable<Alumno>{
	private String nombre;
	private int nota;
	@Override
	public int compareTo(Alumno otro) {
		// TODO Auto-generated method stub
		int resul;

		if ( nota == otro.getNota())
			resul=0;
		else
		{
			if ( nota > otro.getNota())
				resul=1;
			else
				resul=-1;
		}
		return resul;
	}

	// constructor con parametros..
	// metodos get y set..
	// metodo toString...

}
```

```java
public class TestListaAlumnos {
        public static void main( String args[] ) {
           LinkedList<Alumno> listaAlumnos = new LinkedList<Alumno>();

           listaAlumnos.add(new Alumno("Pepe", 5));
           listaAlumnos.add(new Alumno("Andres", 10));
           listaAlumnos.add(new Alumno("Rosa", 8));
           listaAlumnos.add(new Alumno("Juan", 7));

           // Ordena por el compareTo de Alumnos, es decir por nota
           Collections.sort(listaAlumnos);



           Iterator<Alumno> itr = listaAlumnos.iterator();
           while(itr.hasNext())
              System.out.println(itr.next());
         }


}
```

Ordenar listas por varios criterios

- Una lista también puede ordenarse con el método estático sort de Collection, pero usando un objeto comparador que implemente la interfaz Comparator

  Collection.sort(lista, objetoComparador)

- La interfaz Comparator tiene un método compare que el objeto comparador tendrá que implementar

```java
public class ComparadorPorNombre implements Comparator<Alumno> {

		@Override
		public int compare(Alumno al1, Alumno al2) {
			// TODO Auto-generated method stub
			return  al1.getNombre().compareTo(al2.getNombre());
		}

}
public class TestListaAlumnos {
	public static void main( String args[] ) {
	   LinkedList<Alumno> listaAlumnos = new LinkedList<Alumno>();

	   listaAlumnos.add(new Alumno("Pepe", 5));
	   listaAlumnos.add(new Alumno("Andres", 10));
	   listaAlumnos.add(new Alumno("Rosa", 8));
	   listaAlumnos.add(new Alumno("Juan", 7));



	   //Ordena por el comparador, es decir por el nombre
	   ComparadorPorNombre comp=new ComparadorPorNombre();
	   Collections.sort (listaAlumnos, comp);


	   Iterator itr = listaAlumnos.iterator();
	   while(itr.hasNext())
	     System.out.println(itr.next());
	 }

}
```

4. La interfaz *Map*

- El interfaz Map no hereda del interfaz Collection.
- Representa colecciones con parejas de elementos: clave y valor.
- No permite tener claves duplicadas pero si valores duplicados.

Implementaciones de Map: HashMap y HashTable

Un HashMap o HashTable se puede crear de la siguiente forma:

```
HashMap<tipo,tipo> nombre = new HashMap < tipo,tipo >();
```

Tiene dos parámetros el tipo de la clave y el tipo de los elementos que almacenan.

La principal diferencia entre HashMap y HashTable es que el primero admite claves nulas y además los métodos no están sincronizados, es decir que no está preparada para un acceso concurrente.

Algunos de los métodos más importantes de un *Map* son:

`Object get(Object key)`
Accede al valor de una clave. Devuelve null si no existe esa clave en el map.

`Object put(Object key, Object value)`
Inserta una pareja. Si ya había un valor para esa clave se lo reemplaza.

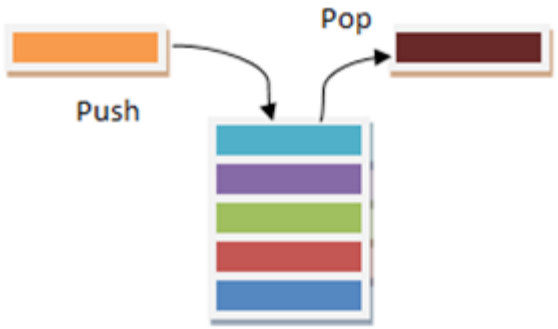
`Object remove(Object key)`
Elimina una pareja.

`Collection<V> values()`
Devuelve una colección con los elementos contenidos en el map

5. Pilas y Colas

La ***Pila*** es una estructura de datos donde los elementos se añaden y suprimen por un único extremo, que es conocido como tope o cabeza de la pila.

Como el último elemento insertado, es el primer en recuperarse/borrarse, nos referimos a estas pilas como pilas **LIFO** ("*Last In - First Out*" - "*último en entrar, primero en salir*").



Stack

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

En este caso la pila se implementa como un vector

Tiene dos operaciones básicas:

- `push()` para insertar un nuevo elemento.
- `pop()` para extraer un elemento, el último que acaba de insertarse.

Colas

Se conoce como ***cola*** y por lo general las colas siguen un patrón que se conoce como **FIFO** (*“First In - First Out”* **-** *“el primero que entra es el primero que sale”*).

Queue

En Java existe la interfaz Queue con los métodos siguientes

E	**element** () Devuelve, pero no la elimina, la cabeza de la cola. Lanza una excepción si la cola está vacía
boolean	**offer** (E o) Inserta el elemento especificado en esta cola, si es posible.
E	**peek** () Devuelve, pero no la elimina, la cabeza de la cola. Devuelve `null` si esta cola está vacía.
E	**poll** () Devuelve y elimina la cabeza de la cola. Devuelve `null` si esta cola está vacía.
E	**remove** () Devuelve y elimina la cabeza de la cola. Lanza una excepción si la cola está vacía.

Importante: La interfaz Queue implementa Collection por lo que tiene todos sus métodos.

6. Interfaces funcionales y expresiones lambda

Una interfaz funcional es aquella que tiene un sólo método abstracto, por ejemplo la interfaz Comparator (un sólo método compare).

Hay 3 formas de ordenar una lista

- Creando una clase que herede de Comparator (la que vimos al ordenar lista).
- Creando una clase anónima.
- Usando expresiones lambda.

Observa los ejemplos de comparación las 3 formas.

Clase anónima

Una clase anónima es una clase sin nombre, definida en la misma línea de código donde se crea el objeto de la clase. Es una clase que hereda de otra existente.

```
LinkedList<Persona> listadoPersonas=new LinkedList<Persona>(  );
listadoPersonas.add( new Persona("22B", "PEPE", "SOL ", 25));
listadoPersonas.add(new Persona("21X", "PEPE", "ADSFFL ", 25));
listadoPersonas.add( new Persona("45X", "LOLA", "ADSFFL ", 55));


Comparator<Persona> comparador = new Comparator<Persona>() {
            Override
            public int compare(Persona persona1, Persona persona2) {
                  return persona1.getDni().compareTo(persona2.getDni());
            }
};
Collections.sort(listadoPersonas, comparador);
```

Expresiones Lambda

Por medio de expresiones lambda podemos referenciar métodos anónimos o métodos sin nombre, lo que nos permite escribir código más claro y conciso que cuando usamos clases anónimas. Las expresiones lambda pueden aparecer como parámetros a métodos.

Una expresión lambda se compone de:

- Listado de parámetros separados por comas y encerrados en paréntesis, por ejemplo: (a,b).
- El símbolo de flecha hacia la derecha: →
- Un cuerpo que puede ser un bloque de código encerrado entre llaves o una sola expresión.

  Ejemplo:

```
(persona1,persona2)->{return persona1.getDni().compareTo(persona2.getDni());}
```

Tener en cuenta

- Si es una única sentencia y no devuelve ningún valor las llaves se pueden omitir
- Si es una única sentencia y devuelve un valor la orden return se puede omitir
- Si hay un solo parámetro de entrada también se pueden omitir los paréntesis

Ordenando listas con lambda

```
LinkedList<Persona> listadoPersonas=new LinkedList<Persona>(  );
listadoPersonas.add( new Persona("22B", "PEPE", "SOL ", 25));
listadoPersonas.add(new Persona("21X", "PEPE", "ADSFFL ", 25));
listadoPersonas.add( new Persona("45X", "LOLA", "ADSFFL ", 55));

Collections.sort(listadoPersonas,
            (persona1,persona2)->{return persona1.getDni().compareTo(persona2.getDni());}
      );
```

7. Interface Stream

Stream y expresiones lambda son la principal novedad de Java 8.Los objetos de las clases que implementan la interface Stream son sucesiones de objetos sobre los que se puede realizar una serie de operaciones hasta dar un resultado final. Las operaciones pueden ser de dos tipos:

- Intermedias: Dan como resultado un nuevo Stream al que seguir aplicando más operaciones
- Terminales: Dan un resultado final (que no es de tipo Stream)

A partir de una colección, o un array, o bien explícitamente se creará un Stream para obtener un resultado final. La ventaja es que podremos realizar las operaciones que realizabamos sobre colecciones o arrays de forma más sencilla.

Formas de crear un Stream

- A partir de una colección

```
Stream<Persona> streamDeArrayList= arrayListPersona.stream();
```

- A partir de un array

```
Persona[] arrayPersonas= { new Persona("22B", "JUANA", "SOL ", 25),
     new Persona("21X", "JUAN", "ADSFFL ", 24)};
Stream<Persona> streamDeArray= Stream.of(arrayPersonas);
```

- Con una lista de objetos que lo inicializan

```
Stream<String>streamCadenas=Stream.of("Sevilla", "Córdoba", "Madrid");
```

Operaciones sobre Stream

- Ejecutar una acción sobre todos los elementos: forEach
- Filtros: Obtener los elementos del stream que cumplen una determinada condición.
  Hay que utilizar la intefaz Predicate<T> tiene un sólo método abstracto que es

```
boolean test(T objeto)
```

- Ordenar: Ordenar el stream por un criterio (sorted)
- Obtener los elementos sin repetir (distinct)

Hay muchos más método muy potentes como:

- Convertir el Stream en Array: toArray( )
- Convertir el Stream en una collección:
  collect(Collectors.toList())
  collect(Collectors.toSet())