

PROGRAMACIÓN

UP7. COLECCIONES DINÁMICAS DE DATOS Y PROGRAMACIÓN FUNCIONAL



1º CFGS DAW

Curso 2024-25

7.1. Colecciones

7.1. Colecciones (interfaz Collection<E>)

En Java, las colecciones de objetos se agrupan utilizando el **Java Collections Framework (JCF)**, un conjunto de interfaces y clases que facilitan el trabajo. El JCF se organiza en torno a varias interfaces clave:

- **Interfaz Collection<E>**: es la base de las colecciones en Java. Define un conjunto básico de operaciones que cualquier colección de objetos debe soportar, como añadir, eliminar y verificar la existencia de elementos.

```
Collection.java
257
258     public interface Collection<E> extends Iterable<E> {
259         // Query Operations
260     }
```

A screenshot of a code editor window titled "Collection.java". The code shown is the definition of the Collection interface, which extends Iterable<E>. It includes a comment for query operations. The code editor has a dark theme with syntax highlighting for Java keywords and comments.

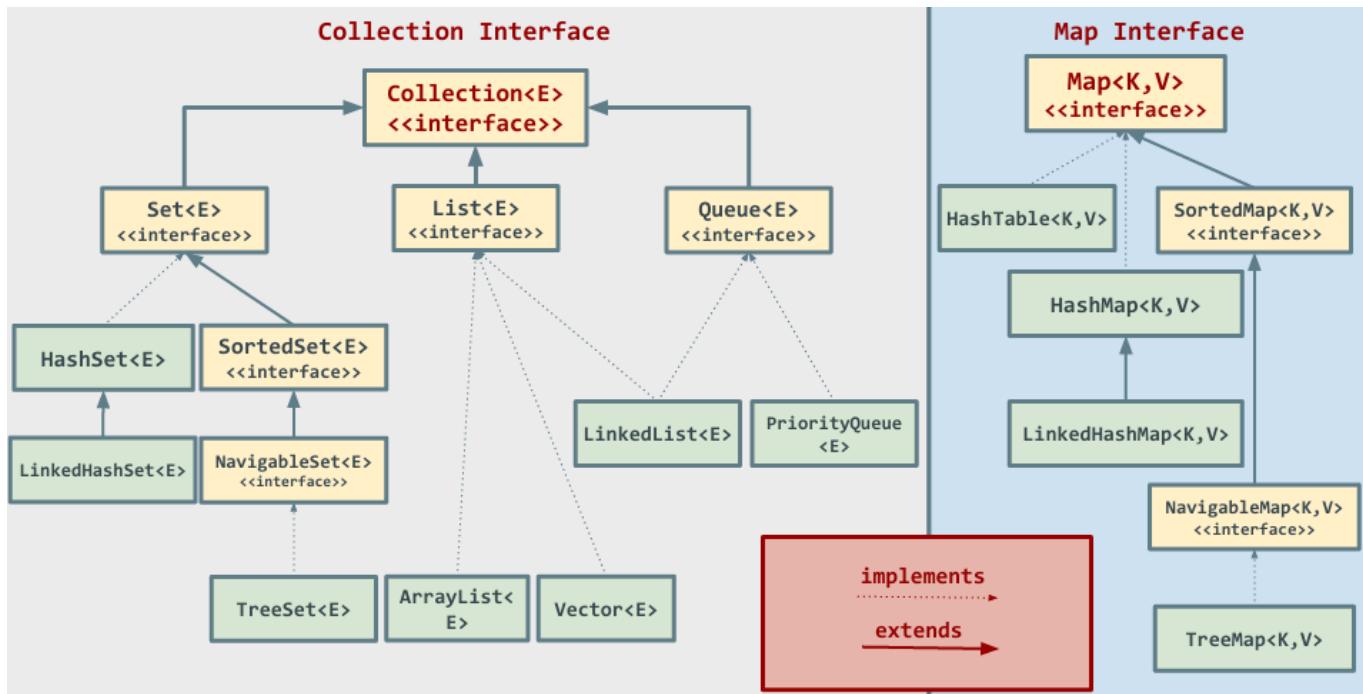
Esta interfaz tiene varias subinterfaces que proporcionan más funcionalidad específica según lo que necesitemos: **List**, **Set** y **Queue**. En los siguientes apartados conoceremos más cada una de estas estructuras, pero como aperitivo, podemos decir que las listas son colecciones de elementos que permiten duplicados (como la clase *ArrayList*). En cuanto a los *Set*, estos presentan una colección de elementos únicos, es decir, no permiten duplicados (clases *HashSet*); y los *Queue* representan una colección de elementos que se procesan siguiendo el principio *FIFO*: los elementos se insertan al final de la cola y se eliminan por el inicio.

- **Interfaz Map<K, V>**: representa una colección de pares *clave-valor* donde cada clave está asociada a un valor. Las claves son únicas, pero los valores pueden duplicarse.

```
Collection.java
Map.java
Author: Josh Bloch
Type parameters: <K> – the type of keys maintained by this map
                  <V> – the type of mapped values
163     public interface Map<K, V> {
164         // Query Operations
165     }
```

A screenshot of a code editor window showing two tabs: "Collection.java" and "Map.java". The "Map.java" tab is active, showing the Map interface definition. It includes author information (Josh Bloch) and type parameter documentation for K and V. The code shows the basic structure of the Map interface. The code editor has a dark theme with syntax highlighting.

HashMap, *TreeMap* y *LinkedHashMap* son implementaciones comunes de *Map*. Este tipo de interfaz la veremos en el próximo apartado (7.2.).



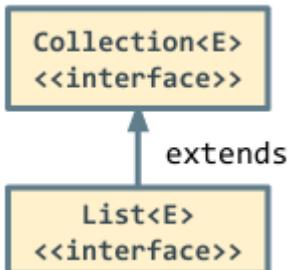
Métodos típicos de la interfaz *Collection*

Como ya conocemos de los *ArrayList*, las principales funcionalidades que comparten todos los objetos que implementan la interfaz *Collection* son las siguientes:

- **boolean add(Object o):** añade un elemento a la colección. Retornará **true** si la colección cambia como resultado de la llamada (es decir, si el elemento se agrega correctamente) y **false** en caso contrario (por ejemplo, si el elemento ya está en un conjunto que no permite duplicados como las clases *Set*).
- **boolean remove(Object o):** elimina una instancia del objeto especificado de la colección, en caso de que exista. Devuelve **true** si la colección cambia y **false** en caso contrario.
- **void clear():** elimina todos los elementos de la colección.
- **boolean contains(Object o):** verifica si la colección contiene una instancia del objeto especificado. Retorna **true** si el objeto existe y **false** en caso contrario.
- **boolean isEmpty():** retorna **true** si la colección está vacía y **false** en caso contrario.
- **int size():** devuelve el número de elementos en la colección.

7.1.1. Listas (List)

La interfaz `List<E>` en Java es una subinterfaz de la interfaz `Collection` y forma parte del JCF mencionado anteriormente.



```
① Collection.java × ② List.java × ③ SequencedCollection.java  
Type parameters: <E> – the type of elements in this list  
140  
141     public interface List<E> extends SequencedCollection<E> {  
142         // Query Operations  
143 }
```

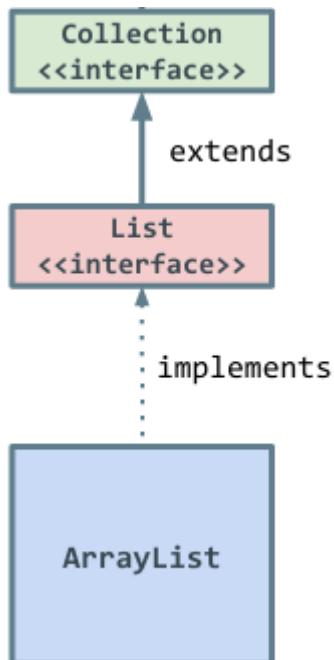
donde `SequencedCollection`:

```
① Collection.java ② List.java ③ SequencedCollection.java ×  
Since: 21  
Type parameters: <E> – the type of elements in this collection  
78 ④ public interface SequencedCollection<E> extends Collection<E> {  
| Returns a reverse-ordered view of this collection. The encounter order of elements in the returned view is the inverse of the encounter order of elements in this collection. The reverse
```

Las listas representan una colección ordenada de elementos, permitiendo duplicados. Como ya sabemos de manejar `ArrayList`, los elementos en una lista están indexados: esto permite acceder, insertar y modificar elementos en posiciones específicas.

--> La letra `<E>` en la interfaz `List` representa un parámetro de tipo genérico. Esto significa que al implementarla se puede especificar el tipo de elementos que se almacenarán en la lista. Por ejemplo, un `List<String>` contendría objetos de tipo `String`.

Implementación de `ArrayList` y métodos principales



Collection.java ArrayList.java AbstractList.java

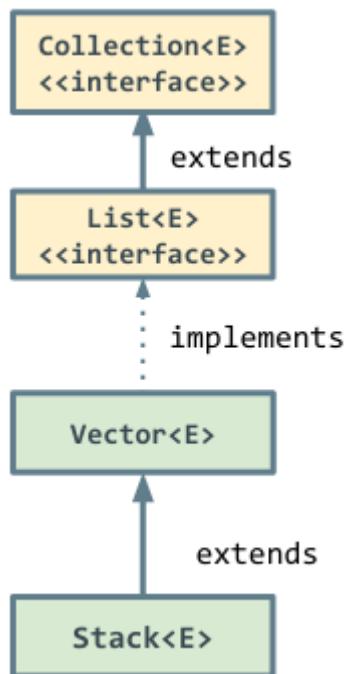
```

Type parameters: <E> - the type of elements in this list

110     @ public class ArrayList<E> extends AbstractList<E>
111         implements List<E>, RandomAccess, Cloneable, java.io.Serializable
112     {
  
```

Ve a AULES
https://aulas.edu.gva.es/fp/pluginfile.php/10055658/mod_resource/content/1/ud5_arraylist/ud5_arraylists/index.html para refrescar los conceptos sobre *ArrayList* vistos en el tema 5.

Pilas (Stack)



```
© Stack.java × © Vector.java
Author: Jonathan Payne
Type parameters: <E> – Type of component elements

50 @ public class Stack<E> extends Vector<E> {
    | Creates an empty Stack.
54     public Stack() {
55 }
```

donde *Vector*.

```
© Stack.java × © Vector.java ×
Author: Lee Boynton, Jonathan Payne
Type parameters: <E> – Type of component elements

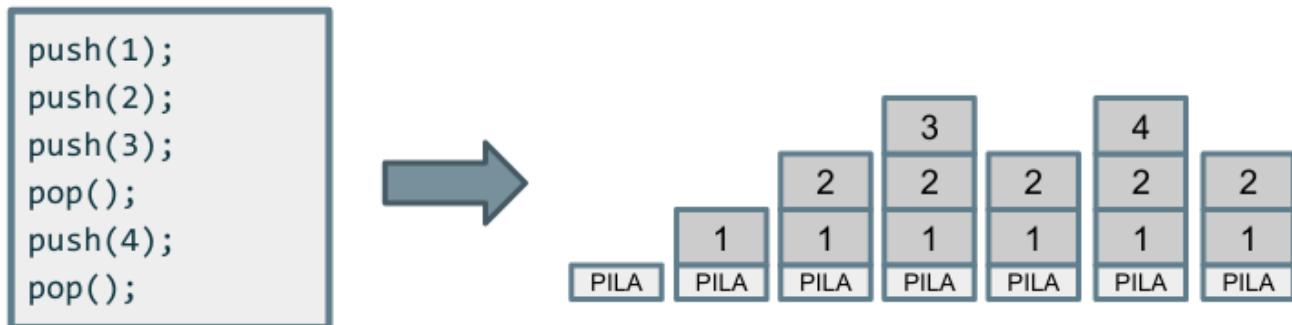
92 @ public class Vector<E>
93     extends AbstractList<E>
94     implements List<E>, RandomAccess, Cloneable, java.io.Serializable
95 {
```

Una pila es una estructura *LIFO (Last In, First Out)*: "último en entrar, primero en salir". Esto significa que el último elemento que se inserta en la pila es el primer elemento que se

elimina de la pila.

Tiene las siguientes operaciones principales:

- **push** (empujar) → inserta un nuevo elemento a la pila.
- **pop** (sacar) → elimina el último elemento insertado.
- **peek()** → devuelve el elemento superior sin eliminarlo.
- **isEmpty()** → indica si la pila está vacía.



Vale... ¿y en Java?

```
1 import java.util.Stack;
2
3 public class EjemploStack {
4     public static void main(String[] args) {
5         Stack<Integer> pila = new Stack<>();
6         pila.push(10);
7         pila.push(20);
8         pila.push(30);
9
10        System.out.println(pila.peek()); // 30 (tope de la pila)
11        System.out.println(pila.pop()); // 30 (último en entrar, prime
12        System.out.println(pila.peek()); // 20 (el nuevo tope de la pil
13    }
14 }
```

Uso de pilas

- Las pilas son útiles en **situaciones en las que se necesitan realizar operaciones en un orden específico**, como en la evaluación de expresiones aritméticas (por ejemplo, primero se multiplica y luego se suma). Las calculadoras y compiladores usan pilas para evaluar expresiones matemáticas sin paréntesis:

Por ejemplo, para evaluar $(3+4)*2$, el compilador lo traducirá a $3\ 4 + 2 *$ (notación postfija, donde el operador aparece después de los operandos), lo almacenará en un vector y usará una pila de la siguiente manera para resolverlo:

1. Insertamos **3** y **4** en la pila. **Pila actual: [3, 4]**
2. Encontramos **+**, sacamos **3** y **4**, sumamos con resultado **7**, y lo metemos de nuevo en la pila. **Pila actual: [7]**
3. Insertamos **2**. **Pila actual: [7, 2]**
4. Encontramos *****, sacamos **7** y **2**, multiplicamos con resultado **14**. **Pila final: [14]**

```
1 import java.util.Stack;
2
3 public class EvaluacionOperadores {
4     public static int evaluar(String expresion) {
5         Stack<Integer> pila = new Stack<>();
6         for (String token : expresion.split(" ")) {
7             if (token.matches("\\d+")) { // si es un número, se apila
8                 pila.push(Integer.parseInt(token));
9             } else { // si es un operador, sacamos dos valores y operan
10                int b = pila.pop();
11                int a = pila.pop();
12                switch (token) {
13                    case "+": pila.push(a + b); break;
14                    case "-": pila.push(a - b); break;
15                    case "*": pila.push(a * b); break;
16                    case "/": pila.push(a / b); break;
17                }
18            }
19        }
20        return pila.pop(); // el resultado final está en lo más alto
21    }
22
23    public static void main(String[] args) {
24        String expresion = "3 4 + 2 *"; // representa  $(3 + 4) * 2$ 
25        System.out.println(evaluar(expresion)); // 14
26    }
27 }
```

- Otro uso común es la implementación de "**Deshacer**" y "**Rehacer**" para gestionar el historial de acciones en editores de texto.
 1. Cada vez que el usuario escribe algo, se guarda en una pila.
 2. Si el usuario presiona "**Deshacer**", se saca el último elemento (`pop()`).

3. Si el usuario quiere "Rehacer", se almacena en otra pila y se restaura (*push()*).

```
1 import java.util.Stack;  
2  
3 public class UndoRedo {  
4     public static void main(String[] args) {  
5         Stack<String> acciones = new Stack<>();  
6         acciones.push("Escribir 'Hola'");  
7         acciones.push("Borrar 'Hola'");  
8         acciones.push("Escribir 'Mundo'");  
9  
10        System.out.println("Última acción: " + acciones.pop()); // "Es  
11    }  
12 }
```

- También se utilizan en la implementación de algoritmos recursivos y en la gestión de llamadas a funciones en una pila de llamadas, ambas vistas en el Tema 4.



Ejercicio 1

-
- a. Crea una pila de letras (*Stack<Character>*).
 - b. Inserta 5 letras a la pila.
 - c. Comprueba si la pila está vacía (*.isEmpty()*).
 - d. Muestra cuántos elementos hay en la pila (*size()*).
 - e. Muestra el último elemento sin eliminarlo (*peek()*).
 - f. Extrae y muestra las letras una por una (*pop()*).
-



Ejercicio 2

Usa una pila para invertir el orden de 3 números introducidos por el usuario.

```
Introduce el primer número: 4
Introduce el segundo número: 8
Introduce el tercer número: 3
Números en orden inverso:
3
8
4
```



Ejercicio 3

Usa una pila para verificar si una expresión matemática tiene paréntesis bien cerrados.

Ejemplo:

- Correcto: " $((2+3)*(5-1))$ "
- Incorrecto: " $(2+3))+((5-1))$ "

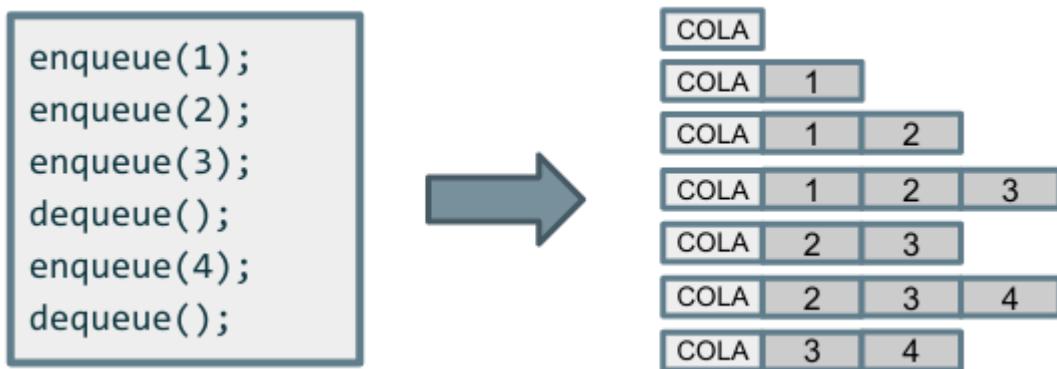
- a. Implementa un método *esBalanceado(String expresion)* que devuelva *true* si los paréntesis están bien cerrados.
- b. Usa una pila de tipo *Stack<Character>* para manejar los paréntesis.

Pistas:

- Recorre la cadena carácter por carácter.
 - Si encuentras un '(', insértalo a la pila.
 - Si encuentras un ')', verifica si hay un '(' en la pila para hacer *pop()*.
 - Al final, la pila debe estar vacía si la expresión es correcta.
-

7.1.2. Colas (Queue)

Una cola es una estructura de datos **FIFO** (First In, First Out), que sigue el principio de "primero en entrar, primero en salir". Esto significa que el primer elemento que se inserta en la cola es el primer elemento que se elimina de la cola.



Son muy útiles en estructuras de datos donde el orden de procesamiento es importante, como colas de impresión, manejo de procesos en sistemas operativos y simulaciones de tráfico.

Los métodos principales de una cola son:

- **add(E e)**: inserta un elemento en la cola y lanza una excepción (**IllegalStateException**) si la **cola está llena***.
- **offer(E e)**: inserta un elemento en la cola y devuelve **false** si la cola está llena en vez de lanzar una excepción.
- **remove()**: elimina y devuelve el primer elemento, y lanza excepción si la cola está vacía.
- **poll()**: elimina y devuelve el primer elemento, y devuelve **null** si la cola está vacía.
- **element()**: obtiene el primer elemento sin eliminarlo y lanza excepción si la cola está vacía.
- **peek()**: obtiene el primer elemento sin eliminarlo y devuelve **null** si la cola está vacía.

***Una cola está llena** cuando ya no puede aceptar más elementos debido a restricciones de memoria o de tamaño predefinido. Esto depende del tipo de implementación de la cola.

Ejemplo en Java usando **ArrayBlockingQueue** y un tamaño fijo de 3:

```
import java.util.concurrent.ArrayBlockingQueue;
import java.util.Queue;

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

```

6
7     Queue<Integer> cola = new ArrayBlockingQueue<>(3); // capacidad
8
9     cola.offer(1);
10    cola.offer(2);
11    cola.offer(3);
12
13    System.out.println("Cola actual: " + cola); // [1, 2, 3]
14
15    // intentamos añadir otro elemento cuando la cola ya está llena
16    boolean exito = cola.offer(4);
17    System.out.println("¿Se ha podido añadir el 4? " + exito); // false
18
}

```

Si usáramos ***add(4)***, lanzaría una excepción ***IllegalStateException***. Pruébalo.

Por otro lado, si una cola no tiene un tamaño máximo, técnicamente nunca está "llena", pero puede quedarse sin memoria si se insertan demasiados elementos.

Diferencia entre ***add()*** vs ***offer()***, ***remove()*** vs ***poll()***, ***element()*** vs ***peek()***

- ✓ Usaremos ***offer()***, ***poll()***, y ***peek()*** si queremos evitar excepciones y manejar la cola de forma segura.
- ✓ Usaremos ***add()***, ***remove()***, y ***element()*** si queremos que el programa falle en caso de error.

En **Java**, la interfaz ***Queue<E>*** puede ser implementada por varias clases, pero en este curso nos centraremos en las ***LinkedList***.

```

import java.util.LinkedList;
import java.util.Queue;

public class EjemploQueue {
    public static void main(String[] args) {
        Queue<String> cola = new LinkedList<>();
        cola.offer("Elemento 1");
        cola.offer("Elemento 2");
        cola.offer("Elemento 3");

        System.out.println(cola.poll()); // "Elemento 1" (eliminado)
        System.out.println(cola.peek()); // "Elemento 2" (siguiente en

```

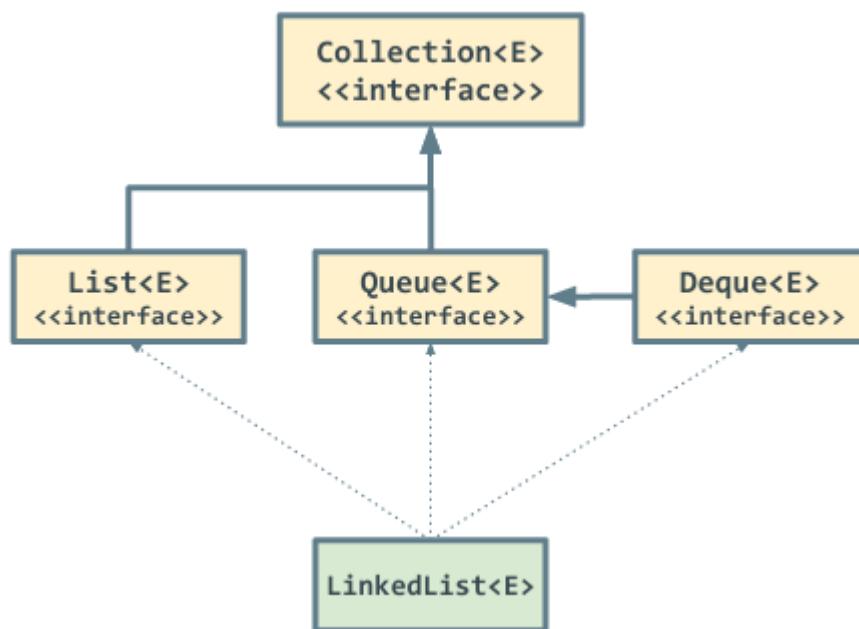
```
    }  
}
```

IMPORTANTE: En una *Queue*, con *offer* no podremos insertar elementos en posiciones arbitrarias. Sólo se añaden elementos al final de la cola, y se eliminan desde el principio (FIFO).

Cuando definimos un *LinkedList* como una cola, los métodos *offer()*, *poll()*, y *peek()* están diseñados para manejar la cola de manera eficiente, mientras que si lo definiéramos como una *List* podríamos manipular elementos en cualquier índice (*add(int index, E element)*).

Clase LinkedList

Como ya sabemos, los *ArrayList* son una implementación de *List* basada en un *array* dinámico. Proporciona acceso rápido y constante en tiempo de ejecución a elementos por índice, pero puede ser ineficiente en operaciones de inserción y eliminación en el medio de la lista. Por ello nacen los *LinkedList*: una estructura de datos dinámica que implementa las interfaces *List<E>* y *Queue<E>*, lo que significa que puede actuar tanto como una lista enlazada y como una cola. También implementa métodos de la interfaz *Deque<E>* (cola doble), la cual solamente usaremos aquí.



Métodos implementados por *LinkedList* a partir de la interface *Deque*:

- *boolean offerFirst(E e)*: inserta el elemento al principio de la lista.
- *boolean offerLast(E e)*: inserta el elemento al final de la lista (similar a *add(E e)*).
- *pollFirst()*: elimina y devuelve el primer elemento de la lista, o *null* si la lista está vacía.
- *pollLast()*: elimina y devuelve el último elemento de la lista, o *null* si la lista está vacía.
- *peekFirst()*: devuelve el primer elemento de la lista sin eliminarlo, o *null* si la lista está vacía.
- *peekLast()*: devuelve el último elemento de la lista sin eliminarlo, o *null* si la lista está vacía.
- *getFirst()*: devuelve el primer elemento de la lista sin eliminarlo.
- *getLast()*: devuelve el último elemento de la lista sin eliminarlo.
- *removeFirst()*: elimina y devuelve el primer elemento de la lista.
- *removeLast()*: elimina y devuelve el último elemento de la lista.

getFirst(), *getLast()*, *removeFirst()* y *removeLast()* lanzan *NoSuchElementException* si está vacía.

```

1 import java.util.Deque;
2 import java.util.LinkedList;
3
4 public class EjemploDequeLinkedList {
5     public static void main(String[] args) {
6
7         Deque<String> deque = new LinkedList<>();
8
9         // insertamos elementos al inicio y al final
10        deque.addFirst("Inicio");
11        deque.addLast("Medio");
12        deque.addLast("Final");
13
14        // mostramos el contenido actual
15        System.out.println("Deque: " + deque); // [Inicio, Medio, Final]
16
17        // obtenemos elementos sin eliminarlos
18        System.out.println("Primer elemento: " + deque.peekFirst()); //
19        System.out.println("Último elemento: " + deque.peekLast()); //
20
21        // eliminamos elementos de los extremos
22        System.out.println("Eliminando primero: " + deque.removeFirst());
23        System.out.println("Eliminando último: " + deque.removeLast());
24
25        // mostramos el estado final del Deque
26        System.out.println("Deque después de eliminaciones: " + deque);
27    }
28 }
29

```

Uso de *LinkedList* con todo tipo de interfaces:

```

1 LinkedList<Alumno> dam = new LinkedList<>();
2
3     //métodos interface Collection
4     dam.add(new Alumno("Pep", "222A", 25));
5     dam.add(new Alumno("Tom", "111A", 20));
6     dam.add(new Alumno("Jon", "444A", 21));
7     dam.add(new Alumno("Tim", "333A", 19));
8     dam.add(new Alumno("Ada", "555A", 18));
9     dam.add(new Alumno("Sam", "666A", 18));
10

```

```

11 //métodos interface List
12 dam.set(2,new Alumno("Ana", "777A",20));
13 dam.add(2,new Alumno("Bil", "777A",20));
14
15 //métodos interfaces Queue y Deque
16 dam.pollFirst();
17 dam.pollLast();
18 dam.offerFirst(new Alumno("Jud", "888A", 24));
19 dam.offerLast(new Alumno("Kim", "999A", 28));
20 System.out.println(dam.removeFirst());
21 System.out.println(dam.removeLast());

```

```

LinkedList<Alumno> dam = new LinkedList<>();

//métodos interface Collection
dam.add(new Alumno("Pep", "222A", 25)); //Pep
dam.add(new Alumno("Tom", "111A", 20)); //Pep, Tom
dam.add(new Alumno("Jon", "444A", 21)); //Pep, Tom, Jon
dam.add(new Alumno("Tim", "333A", 19)); //Pep, Tom, Jon, Tim
dam.add(new Alumno("Ada", "555A", 18)); //Pep, Tom, Jon, Tim, Ada
dam.add(new Alumno("Sam", "666A", 18)); //Pep, Tom, Jon, Tim, Ada, Sam

//métodos interface List
dam.set(2,new Alumno("Ana", "777A",20)); //Pep, Tom, Ana, Tim, Ada, Sam
dam.add(2,new Alumno("Bil", "777A",20)); //Pep, Tom, Bil, Ana, Tim, Ada, Sam

//métodos interfaces Queue y Deque
dam.pollFirst(); //Tom, Bil, Ana, Tim, Ada, Sam
dam.pollLast(); //Tom, Bil, Ana, Tim, Ada
dam.offerFirst(new Alumno("Jud", "888A", 24)); //Jud, Tom, Bil, Ana, Tim, Ada
dam.offerLast(new Alumno("Kim", "999A", 28)); //Jud, Tom, Bil, Ana, Tim, Ada, Kim
System.out.println(dam.removeFirst()); // Jud - dam → Tom, Bil, Ana, Tim, Ada, Kim
System.out.println(dam.removeLast()); // Kim - dam → Tom, Bil, Ana, Tim, Ada

```

Cuándo usar *LinkedList*

- ✓ Cuando necesitemos modificar datos frecuentemente y realizar muchas inserciones y eliminaciones de la lista. Rapidísimo en los extremos.
- ✗ Menos eficiente que *ArrayList* para acceso aleatorio de datos y acceso frecuente a elementos por índice.



Ejercicio 1

- a. Crea una cola de letras (*Queue<Character> cola = new LinkedList<>()*).
- b. Inserta 5 letras a la cola (*offer()*).
- c. Comprueba si la cola está vacía (*.isEmpty()*).
- d. Muestra cuántos elementos hay en la cola (*size()*).

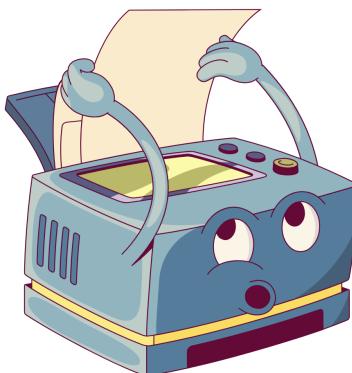
e. Muestra el primer elemento sin eliminarlo (*peek()*).

f. Extrae y muestra las letras una por una (*poll()*).



Ejercicio 2

Simula una cola de trabajos de impresión donde dichos trabajos son procesados en orden FIFO (*First In, First Out*).



Instrucciones:

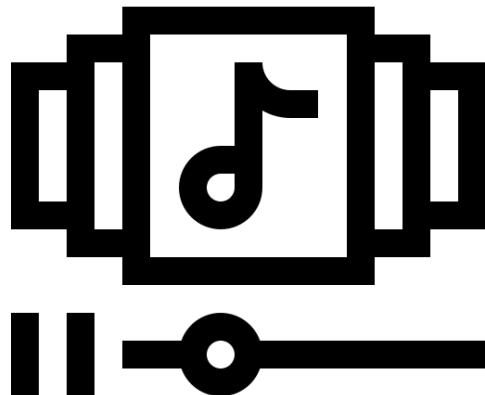
- Los trabajos de impresión son tareas como "*Imprimir documento 1*", "*Imprimir documento 2*", etc, y se procesan en el orden en que fueron insertados a la cola.
- El programa debe mostrar el estado inicial de la impresora, antes de empezar a procesar trabajos.
- El programa debe mostrar el estado de la cola antes y después de procesar cada trabajo.

```
Estado inicial de la cola de impresión: [Imprimir documento 1, Imprimir documento 2, Imprimir documento 3, Imprimir documento 4, Imprimir documento 5]
Procesando: Imprimir documento 1
Estado de la cola después de procesar el trabajo: [Imprimir documento 2, Imprimir documento 3, Imprimir documento 4, Imprimir documento 5]
Procesando: Imprimir documento 2
Estado de la cola después de procesar el trabajo: [Imprimir documento 3, Imprimir documento 4, Imprimir documento 5]
Procesando: Imprimir documento 3
Estado de la cola después de procesar el trabajo: [Imprimir documento 4, Imprimir documento 5]
Procesando: Imprimir documento 4
Estado de la cola después de procesar el trabajo: [Imprimir documento 5]
Procesando: Imprimir documento 5
Estado de la cola después de procesar el trabajo: []
No hay más trabajos en la cola de impresión.
```



Ejercicio 3

Simula la reproducción de una *playlist* de Spotify musical usando una cola. Los temas se reproducen en el orden en que se insertan a la cola (*FIFO*).



Instrucciones:

- a. Crea una cola de canciones (`Queue<String> playlist = new LinkedList<>()`).
- b. Añade varias canciones a la cola como si fueran canciones en una lista de reproducción.
- c. Simula la reproducción de canciones: la primera canción en la cola debe ser la que se reproduce.
- d. Despues de reproducir una canción, se elimina de la cola y se reproduce la siguiente canción.
- e. Muestra la canción que se está reproduciendo en cada momento y el estado de la lista de reproducción después de cada canción.

```
Estado inicial de la playlist: [Let it be - The Beatles, Bohemian Rhapsody - Queen, Shape of You - Ed Sheeran, Blinding Lights - The Weeknd, Stay - Justin Bieber]
Reproduciendo: Let it be - The Beatles
Estado de la playlist después de reproducir una canción: [Bohemian Rhapsody - Queen, Shape of You - Ed Sheeran, Blinding Lights - The Weeknd, Stay - Justin Bieber]
Reproduciendo: Bohemian Rhapsody - Queen
Estado de la playlist después de reproducir una canción: [Shape of You - Ed Sheeran, Blinding Lights - The Weeknd, Stay - Justin Bieber]
Reproduciendo: Shape of You - Ed Sheeran
Estado de la playlist después de reproducir una canción: [Blinding Lights - The Weeknd, Stay - Justin Bieber]
Reproduciendo: Blinding Lights - The Weeknd
Estado de la playlist después de reproducir una canción: [Stay - Justin Bieber]
Reproduciendo: Stay - Justin Bieber
Estado de la playlist después de reproducir una canción: []
La playlist ha terminado.
```



Ejercicio 4: la lista de la compra

Como en cualquier lista de la compra, las acciones que más vamos a realizar sobre la lista es añadir o borrar elementos, por lo que deberemos implementar una *LinkedList* para que todo sea más eficiente.

La aplicación debe tener un menú interactivo donde el usuario pueda realizar diferentes operaciones sobre la lista, como añadir productos al final, eliminar productos, buscar productos, mostrar la lista, etc.

Requisitos:

1. El usuario podrá añadir productos al final de la lista utilizando la opción "Añadir producto al final" (*offerLast(String producto)*)
2. El usuario podrá eliminar el primer producto de la lista (*pollFirst()*).
3. El usuario podrá eliminar un producto específico de la lista (por su nombre) utilizando la opción "Eliminar producto" (*remove(Object producto)*). Si el producto a eliminar no se encuentra en la lista, se debe informar al usuario (*contains(Object producto)*).
4. El usuario podrá ver el primer producto de la lista utilizando la opción "Ver primer producto" (*peekFirst()*).
5. El usuario podrá buscar si un producto específico está en la lista utilizando la opción "Buscar producto". Si el producto a buscar no se encuentra en la lista, se debe informar al usuario.
6. El usuario podrá ver la lista completa de compras en cualquier momento utilizando la opción "Mostrar lista actual".
7. Si la lista está vacía, se deben manejar los intentos de eliminar o acceder a elementos con mensajes adecuados (*isEmpty()*).

```
--- LISTA DE LA COMPRA ---
1. Añadir producto al final
2. Eliminar primer producto
3. Eliminar producto por nombre
4. Ver primer producto
5. Buscar producto
6. Mostrar lista actual
X. Salir

Seleccione una opción: 1

-----
Producto a añadir: Leche
```

Para mostrar la lista actual (opción 5), se debe obtener la posición que ocupa cada elemento de la lista (*indexOf(String producto)+1*) y mostrarlo junto a su descripción.

```
--- LISTA DE LA COMPRA ---  
1. Añadir producto al final  
2. Eliminar primer producto  
3. Eliminar producto por nombre  
4. Ver primer producto  
5. Buscar producto  
6. Mostrar lista actual  
X. Salir
```

Seleccione una opción: 1

Producto a añadir: Leche



Ejercicio 5: palíndromos

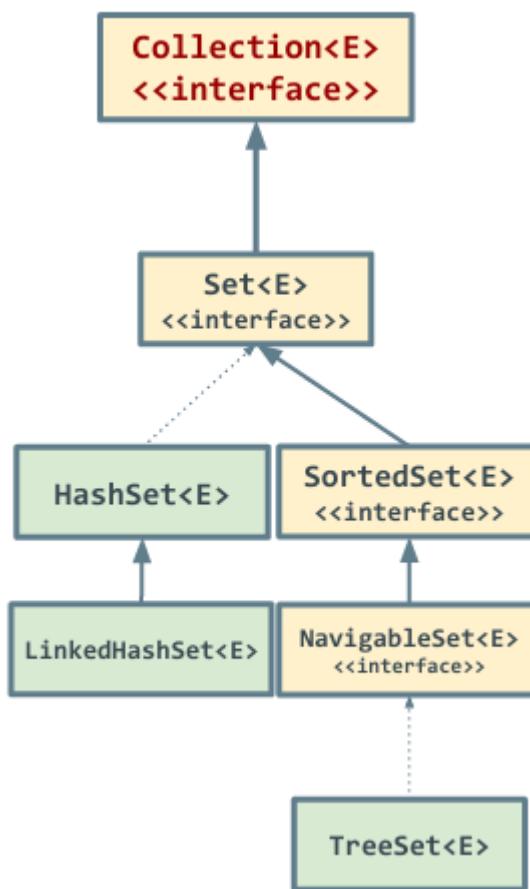
Un palíndromo es una palabra que se lee igual en ambos sentidos, como por ejemplo *radar*. Diseña una función en Java que determine si una palabra definida como una lista doblemente enlazada (*Deque*) de caracteres (*Character*) es palíndroma o no.

RADAR
Oso ALA
RECONOCER

7.1.3. Conjuntos (Set) - sin duplicados

Como ya hemos comentado durante la introducción del tema, la interfaz **Set** representa una colección de elementos únicos, es decir, no permite elementos duplicados. De esta manera, **su implementación del método `add()` garantiza que no se inserten elementos duplicados**. Si se intenta añadir un elemento que ya está presente en el conjunto, el método `add()` simplemente retornará **false** y no modificará el conjunto.

Al igual que *List*, la interfaz *Set* extiende *Collection*, por ello, hereda todos los métodos definidos en la interfaz *Collection*. Sin embargo, **la interfaz Set no introduce métodos propios**. Es decir, seguiremos usando los típicos: `add()`, `remove()`, `contains()`, `size()`, `isEmpty()`,...



Es importante tener en cuenta que el comportamiento de los métodos heredados de *Collection* está adaptado a las propiedades de un conjunto, es decir, no permiten elementos duplicados. Para que se comporte como tal, **es necesario una correcta implementación de los métodos `equals()` y `hashCode()`**.

¿Y eso qué es!? Ahora lo vemos...

Clase HashSet

HashSet es muy rápida, pero no garantiza el orden de los elementos. Es decir, si en una lista tenemos guardados *A*, *B* y *C*, cada vez que la imprimamos podría mostrarse en cualquier orden.

```
1 import java.util.HashSet;
2 import java.util.Set;
3
4 public class EjemploHashSet {
5
6     public static void main(String[] args) {
7
8         Set<String> conjunto = new HashSet<>();
9
10        conjunto.add("B");
11        conjunto.add("A");
12        conjunto.add("C");
13        conjunto.add("A"); // no se añadirá, porque ya existe
14
15        System.out.println("Conjunto: " + conjunto); // [A, B, C] (pueden aparecer en orden diferente)
16
17        System.out.println("¿Contiene 'B'??: " + conjunto.contains("B"))
18
19        conjunto.remove("A");
20
21        System.out.println("Tamaño del conjunto: " + conjunto.size());
22
23    }
24
25 }
```

```
"C:\Program Files\Java\jdk-23\bin\java.exe" "-Djava.awt.headless=true"
Conjunto: [A, B, C]
¿Contiene 'B'??: true
Tamaño del conjunto: 2

Process finished with exit code 0
```

Si necesitamos que se mantenga el orden de inserción, en lugar de *HashSet* deberemos implementar una lista de tipo *LinkedHashSet*:

```
1 import java.util.LinkedHashSet;
2 import java.util.Set;
3
4 public class EjemploLinkedHashSet {
5
6     public static void main(String[] args) {
7
8         Set<Integer> numeros = new LinkedHashSet<>();
9
10        numeros.add(3);
11        numeros.add(1);
12        numeros.add(2);
13
14        System.out.println("Conjunto ordenado: " + numeros); // [3, 1,
15
16    }
17
18 }
```

Y si queremos que además los elementos se ordenen automáticamente (ascendente) conforme se van añadiendo, entonces necesitaremos una estructura de tipo árbol: *TreeSet*.

```
import java.util.TreeSet;
import java.util.Set;

public class EjemploTreeSet {

    public static void main(String[] args) {

        Set<Integer> treeSet = new TreeSet<>();

        treeSet.add(50);
        treeSet.add(10);
        treeSet.add(30);

        System.out.println("Conjunto ordenado: " + treeSet); // [10, 30

    }
}
```

Comparación de *HashSet*, *TreeSet* y *LinkedHashSet*

Vamos a añadir los siguientes valores en tres estructuras distintas: *HashSet*, *TreeSet* y *LinkedHashSet*.

```
1 | String[] palabras = {"banana", "manzana", "pera", "uva", "manzana", "ce
```

```
1 | import java.util.HashSet;
2 | import java.util.LinkedHashSet;
3 | import java.util.Set;
4 | import java.util.TreeSet;
5 |
6 | public class ComparacionSets {
7 |     public static void main(String[] args) {
8 |
9 |         String[] palabras = {"banana", "manzana", "pera", "uva", "manza
10 |
11 |         Set<String> hashSet = new HashSet<>();
12 |         Set<String> treeSet = new TreeSet<>();
13 |         Set<String> linkedHashSet = new LinkedHashSet<>();
14 |
15 |         for (String palabra : palabras) {
16 |             hashSet.add(palabra);
17 |             treeSet.add(palabra);
18 |             linkedHashSet.add(palabra);
19 |         }
20 |
21 |         System.out.println("HashSet (orden impredecible): " + hashSet);
22 |         System.out.println("TreeSet (ordenado alfabéticamente): " + tre
23 |         System.out.println("LinkedHashSet (orden de inserción): " + lin
24 |
25 |     }
26 | }
```

Resultado:

```
"C:\Program Files\Java\jdk-23\bin\java.exe" "-javaagent:C:\Program Files\Java\VisualVM\lib\visualvm-agent.jar" -Djava.awt.headless=true -Djava.net.preferIPv4Stack=true
HashSet (orden impredecible): [banana, manzana, pera, uva, cereza]
TreeSet (ordenado alfabéticamente): [banana, cereza, manzana, pera, uva]
LinkedHashSet (orden de inserción): [banana, manzana, pera, uva, cereza]

Process finished with exit code 0
```



Ejercicio 1: Filtrar palabras únicas

Dado un texto, almacenar sólo las palabras únicas en un *HashSet* (sin repeticiones).

Pistas:

1. Pide al usuario que ingrese una frase.
2. Divide la frase en palabras.
3. Guarda las palabras en un *HashSet* para eliminar duplicados. **Cuidado con las mayúsculas y minúsculas**, ya que no se deben tener en cuenta.
4. Muestra el conjunto de palabras únicas.

```
Escribe una frase: hola mundo hola programadores java java mundo
Palabras únicas: [hola, mundo, programadores, java]
```



Ejercicio 2: Agenda de contactos (ordenados alfabéticamente)

Crear una agenda donde los nombres se almacenen de forma ordenada usando *TreeSet*.

Implementa un menú, donde el usuario pueda:

- Añadir un contacto.
- Mostrar todos los contactos ordenados.
- Buscar si un contacto existe.
- Eliminar un contacto.

```
----- AGENDA -----  
1. Añadir contacto  
2. Mostrar contactos  
3. Buscar contacto  
4. Eliminar contacto  
X. Salir
```

```
Elige una opción: 1
```

```
-----  
Nombre del nuevo contacto: Carlos
```

```
----- AGENDA -----  
1. Añadir contacto  
2. Mostrar contactos  
3. Buscar contacto  
4. Eliminar contacto  
X. Salir
```

```
Elige una opción: 1
```

```
-----  
Nombre del nuevo contacto: Ana
```

```
----- AGENDA -----  
1. Añadir contacto  
2. Mostrar contactos  
3. Buscar contacto  
4. Eliminar contacto  
X. Salir
```

```
Elige una opción: 2
```

```
-----  
Contactos: [Ana, Carlos]
```



Ejercicio 3: Historial de navegación (mantener orden de visitas)

Simular un historial de navegación donde se guarden las páginas visitadas sin repetir, pero manteniendo el orden de acceso con *LinkedHashSet*.

```
Escribe una URL visitada (o 'salir' para terminar): google.com
Escribe una URL visitada (o 'salir' para terminar): openai.com
Escribe una URL visitada (o 'salir' para terminar): google.com
Escribe una URL visitada (o 'salir' para terminar): wikipedia.org
Escribe una URL visitada (o 'salir' para terminar): salir
```

Historial de navegación:

1. google.com
2. openai.com
3. wikipedia.org



Ejercicio 4

Implementa una función *mezclados()* que tenga como parámetros dos listas de enteros ordenados de menor a mayor, y que devuelva una nueva lista como unión de ambas con sus elementos ordenados de la misma forma.

7.1.4. Métodos equals() y hashCode() de un objeto

Como ya sabes, la clase *Object* es una superclase predeterminada de todas las clases. Es decir, todas las clases de Java directa o indirectamente heredan de la clase *Object*.

La clase *Object* define una serie de métodos que están disponibles en todas las clases en Java, por ejemplo, el famoso *toString()*. Pero en este punto **hay dos métodos más que debemos conocer:**

- ***hashCode():*** se utiliza para obtener un valor numérico único que representa a un objeto. La función *hash* está diseñada para mapear la entrada de manera eficiente a un valor *hash* único y consistente. Esto lo hace especialmente útil en la búsqueda de datos y la optimización de rendimiento.
- ***equals():*** se utiliza para comparar dos objetos y determinar si son iguales. Esta función es importante porque los objetos en Java se comparan por referencia, es decir, si dos objetos tienen la misma referencia (es decir, apuntan a la misma dirección de memoria), se consideran iguales.

Los métodos *equals()* y *hashCode()* son importantes porque permiten que los objetos se comparan y se indexen correctamente en las colecciones de Java. La implementación correcta de estos métodos es esencial para garantizar que las colecciones funcionen correctamente y produzcan resultados esperados.

Ejemplo de uso de *equals()* y *hashCode()* con *HashSet*

Si la clase *Alumno* no tuviera implementados los métodos *equals()* y *hashCode()*, se podrían insertar elementos duplicados (mismos valores de los atributos). Esto se debe a que, por defecto, los métodos *equals()* y *hashCode()* heredados de la clase *Object* están basados en las referencias de los objetos y no en su estado.

```
Set<Alumno> dam = new HashSet<>();
Alumno a1 = new Alumno("Pep", "222A", 25);
Alumno a2 = new Alumno("Sam", "666A", 18);
Alumno a3 = new Alumno("Sam", "666A", 18);
Alumno a4 = new Alumno("Kal", "666A", 20);

dam.add(a1); //Pep
dam.add(a2); //Pep, Sam
dam.add(a3); //Pep, Sam, Sam
dam.add(a4); //Pep, Sam, Sam, Kal

dam.add(a1); //No se inserta Pep, tienen la misma referencia
dam.add(new Alumno("Sam", "666A", 18)); //Se inserta Sam, no tienen la misma referencia
//Estado final de dam → Pep, Sam, Sam, Kal, Sam
```

Para verificar la existencia de duplicados en un *HashSet*, primero se compara el resultado del método *hashCode()*. Si los *hash* son diferentes, se considera que los objetos son diferentes y no se verifica más. Sin embargo, si los *hash* son iguales, se procede a utilizar el método *equals()* para determinar si los objetos son realmente iguales.

```
Set<Alumno> dam = new HashSet<>();
Alumno a1 = new Alumno("Pep", "222A", 25);
Alumno a2 = new Alumno("Sam", "666A", 18);
Alumno a3 = new Alumno("Sam", "666A", 18);
Alumno a4 = new Alumno("Kal", "666A", 20);

dam.add(a1); //Pep
dam.add(a2); //Pep, Sam
dam.add(a3); //Pep, Sam → no inserta Sam de nuevo, hay un Alumno igual
dam.add(a4); //Pep, Sam, Kal

dam.add(a1); //No se inserta Pep, tienen la misma referencia
dam.add(new Alumno("Sam", "666A", 18)); //No inserta Sam, hay un Alumno igual
//Estado final de dam → Pep, Sam, Kal
```

Cuándo es necesario implementar *equals()* y *hashCode()*

- Si no sobreescribimos *equals()* y *hashCode()*, el comportamiento por defecto (heredado de *Object*) puede no funcionar correctamente **si usamos objetos personalizados.**

```
1 import java.util.HashSet;
2
3 class Persona {
4     String nombre;
5
6     public Persona(String nombre) {
7         this.nombre = nombre;
8     }
9 }
10
11 public class Main {
12     public static void main(String[] args) {
13         HashSet<Persona> set = new HashSet<>();
14         set.add(new Persona("Ana"));
15         set.add(new Persona("Ana")); // se espera que no se duplique, p
16
17         System.out.println("Tamaño del conjunto: " + set.size()); // re
18     }
19 }
20 }
```

El **problema**: como *equals()* y *hashCode()* no están sobrescritos, los objetos *new Persona("Ana")* se consideran diferentes aunque el atributo nombre sea el mismo (por defecto, sólo se mira la referencia en memoria).

Sobrescribimos *equals()* y *hashCode()* en la clase *Persona* (siempre se hace igual):

```
1 import java.util.Objects;
2 import java.util.HashSet;
3
4 class Persona {
5     String nombre;
6
7     public Persona(String nombre) {
8         this.nombre = nombre;
9     }
10
11    @Override
12    public boolean equals(Object obj) {
13        if (this == obj) return true;
14        if (obj == null || getClass() != obj.getClass()) return false;
15        Persona persona = (Persona) obj;
16        return Objects.equals(nombre, persona.nombre);
17    }
18
19    @Override
20    public int hashCode() {
21        return Objects.hash(nombre);
22    }
23 }
24
25 public class Main {
26     public static void main(String[] args) {
27         HashSet<Persona> set = new HashSet<>();
28         set.add(new Persona("Ana"));
29         set.add(new Persona("Ana")); // ahora no se duplica
30
31         System.out.println("Tamaño del conjunto: " + set.size()); // resulta 1
32     }
33 }
```

Ahora *HashSet* funciona correctamente y evita duplicados, ya que, además de la referencia en memoria, ahora el método *Equals* compara también el valor de los nombres: *Objects.equals(nombre, persona.nombre)*.

- Si usamos tipos primitivos o clases estándar (String, Integer, etc.) NO es necesario. *String* y las clases de la API de Java ya implementan correctamente *equals()* y *hashCode()*.

```
1 import java.util.HashSet;
2
3 public class EjemploTiposPrimitivos {
4     public static void main(String[] args) {
5         // HashSet con Strings
6         HashSet<String> nombres = new HashSet<>();
7         nombres.add("Juan");
8         nombres.add("Ana");
9         nombres.add("Pedro");
10        nombres.add("Juan"); // no se añadirá porque ya existe
11
12        System.out.println("Nombres en el HashSet: " + nombres);
13        System.out.println("¿Contiene 'Ana'??: " + nombres.contains("Ana"));
14
15        // HashSet con Integer
16        HashSet<Integer> numeros = new HashSet<>();
17        numeros.add(10);
18        numeros.add(20);
19        numeros.add(30);
20        numeros.add(10); // no se añadirá porque ya existe
21
22        System.out.println("Números en el HashSet: " + numeros);
23        System.out.println("¿Contiene 20?: " + numeros.contains(20)); /
24    }
25 }
26 }
```

- Si sólo comparamos por referencia y no por contenido. Es decir, si usamos objetos sólo como identificadores únicos en memoria, no necesitaremos sobrescribir nada.

```
1 Persona p1 = new Persona("Carlos");
2 Persona p2 = p1; // ambas variables apuntan al mismo objeto en memoria
3 System.out.println(p1 == p2); // true
```

Un ejemplo de lo que ocurre, paso por paso

Vamos a suponer que tenemos la siguiente clase *Estudiante*, con la siguiente estructura:

```
1 public class Estudiante {  
2  
3     private String nia;  
4     private String nombre;  
5     private String apellidos;  
6  
7     public Estudiante (String nia, String nombre, String apellidos){  
8         this.nia = nia;  
9         this.nombre=nombre;  
10        this.apellidos=apellidos;  
11    }  
12  
13 }
```

Nos vamos al *main* y creamos una instancia de nuestra clase. Observa que Java nos permite usar los métodos *equals()* y *hashCode()*:

```
1 public class Main {  
2  
3     public static void main(String[] args) {  
4  
5         Estudiante estudiante1 = new Estudiante("12345667","Carlos","Perez");  
6  
7         estudiante1.equals();  
8         estudiante1.hashCode();  
9  
10    }  
11 }
```

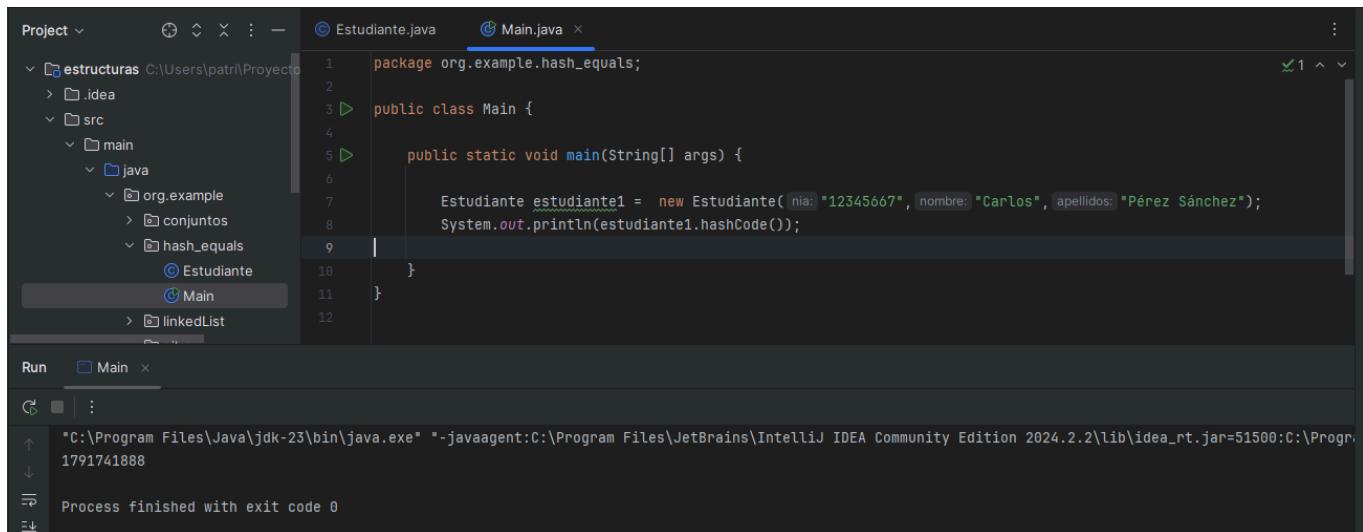
El método *hashCode* devolverá un número entero que identificará a nuestro estudiante según la implementación de este en la clase *Object*:



```
39  public class Object {
    Implementation As far as is reasonably practical, the hashCode method defined by class
    Requirements: Object returns distinct integers for distinct objects.

    See Also: equals(Object),
              System.identityHashCode()

106 @IntrinsicCandidate
107     public native int hashCode();
108 }
```



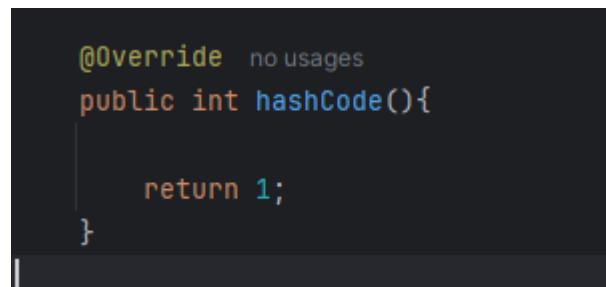
```
Project : estруктуры C:\Users\patr\Проекты
Main.java
```

```
1 package org.example.hash_equals;
2
3 public class Main {
4
5     public static void main(String[] args) {
6
7         Estudiante estudiante1 = new Estudiante("12345667", "Carlos", "Pérez Sánchez");
8         System.out.println(estudiante1.hashCode());
9     }
10 }
11
12 }
```

```
Run Main
Process finished with exit code 0
```

pero es recomendable que nosotros implementemos nuestra propia versión sobre escribiéndolo.

Podríamos hacerlo a mano, teniendo en cuenta que se debe devolver un valor numérico. Por ejemplo,

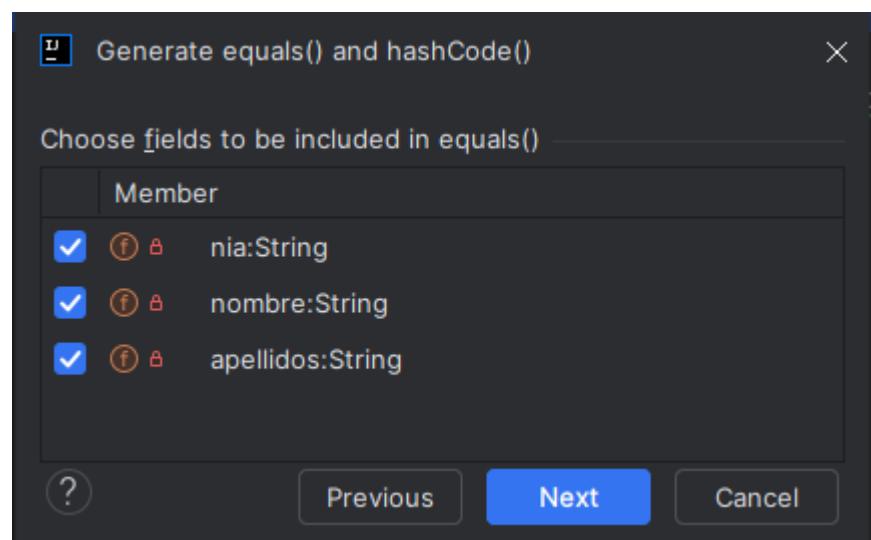
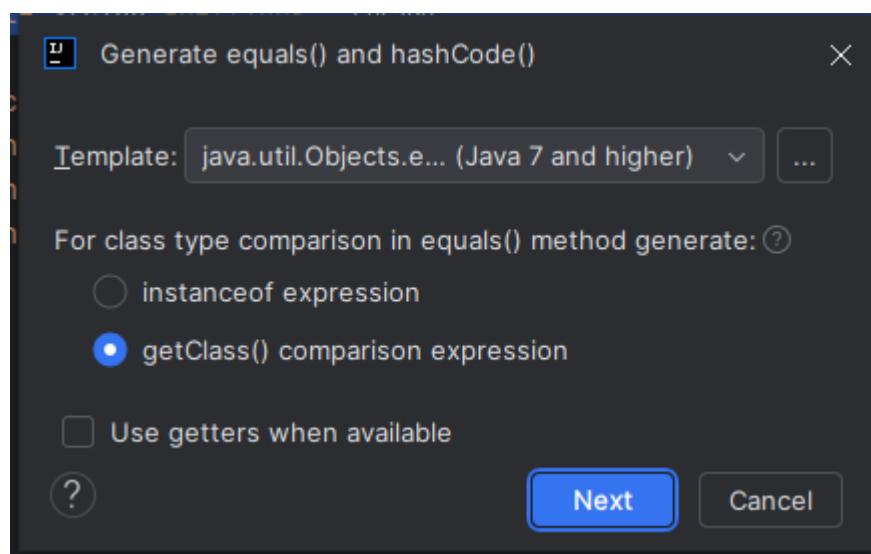


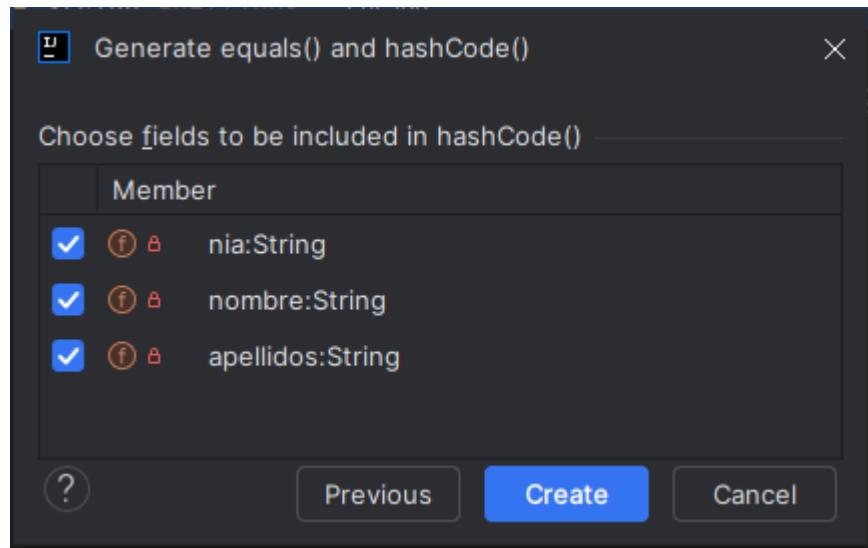
```
@Override no usages
public int hashCode(){
    return 1;
}
```

Pero dado que nos interesa que ese valor numérico sea lo más único posible para identificar bien a nuestros objetos, haremos uso del generador del *IDE IntelliJ*. **No se recomienda hacerlo manualmente.**

```
1 package org.example.hash_equals;
2
3 public class Estudiante { 2 usages
4
5     private String nia; 1 usage
6     private String nombre; 1 usage
7     private String
8
9     public Estudiante() {
10         this.nia = null;
11         this.nombre = null;
12         this.apellidos = null;
13     }
14 }
```

A code editor interface showing a Java class named 'Estudiante'. A context menu is open over the constructor definition, listing options: Constructor, Getter, Setter, Getter and Setter, equals() and hashCode(), toString(), Override Methods... (Ctrl+O), Delegate Methods..., Test..., and Copyright. The 'Constructor' option is highlighted.



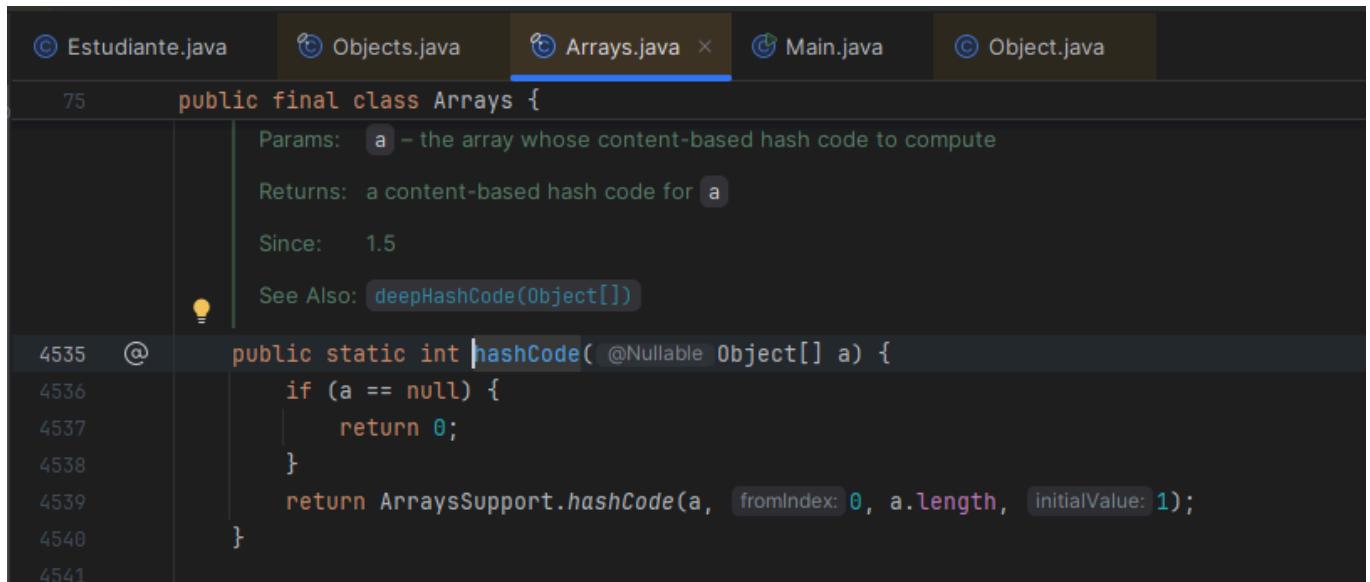


Lo que obtenemos es el siguiente código:

```
1 public class Estudiante {  
2  
3     private String nia;  
4     private String nombre;  
5     private String apellidos;  
6  
7     public Estudiante (String nia, String nombre, String apellidos){  
8         this.nia = nia;  
9         this.nombre=nombre;  
10        this.apellidos=apellidos;  
11    }  
12  
13 //    @Override  
14 //    public boolean equals(Object o) {  
15 //        if (this == o) return true;  
16 //        if (o == null || getClass() != o.getClass()) return false;  
17 //        Estudiante that = (Estudiante) o;  
18 //        return Objects.equals(nia, that.nia) && Objects.equals(nombre,  
19 //    }  
20  
21 //    @Override  
22 //    public int hashCode() {  
23 //        return Objects.hash(nia, nombre, apellidos);  
24 //    }  
25  
26 }
```

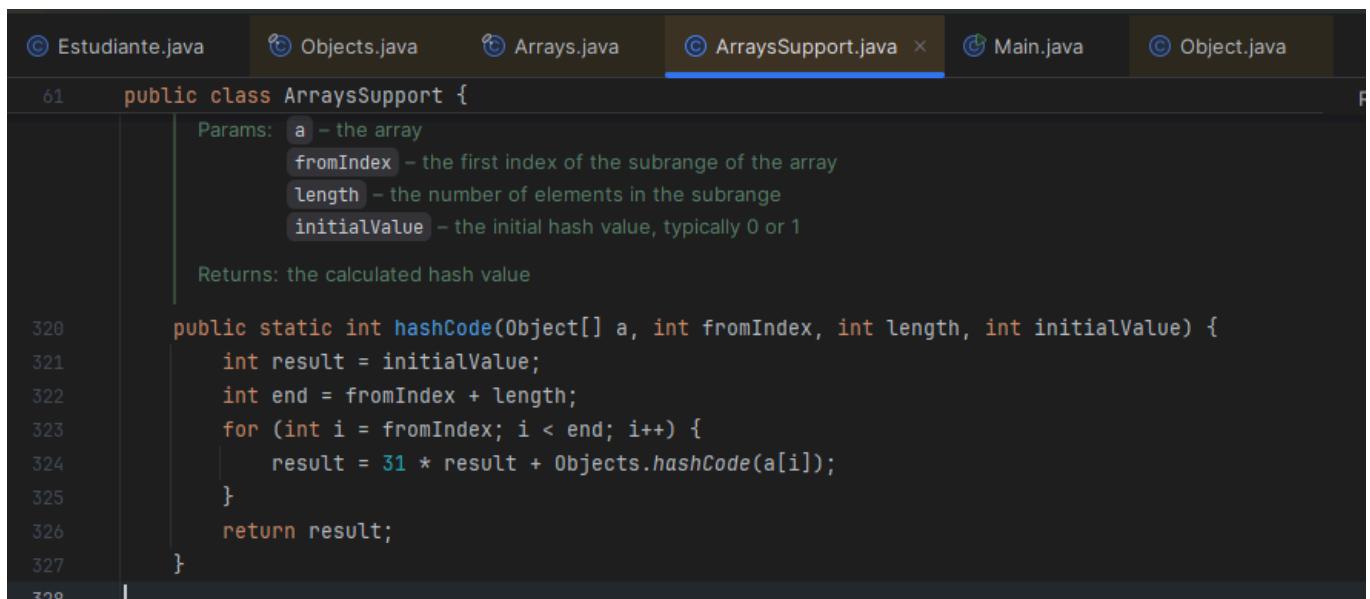
Comentamos de momento el `equals()` generado. La recomendación que nos hace *IntelliJ* para el método `hashCode()` es la siguiente:

- Se mira si cada uno de sus atributos es nulo (o no),



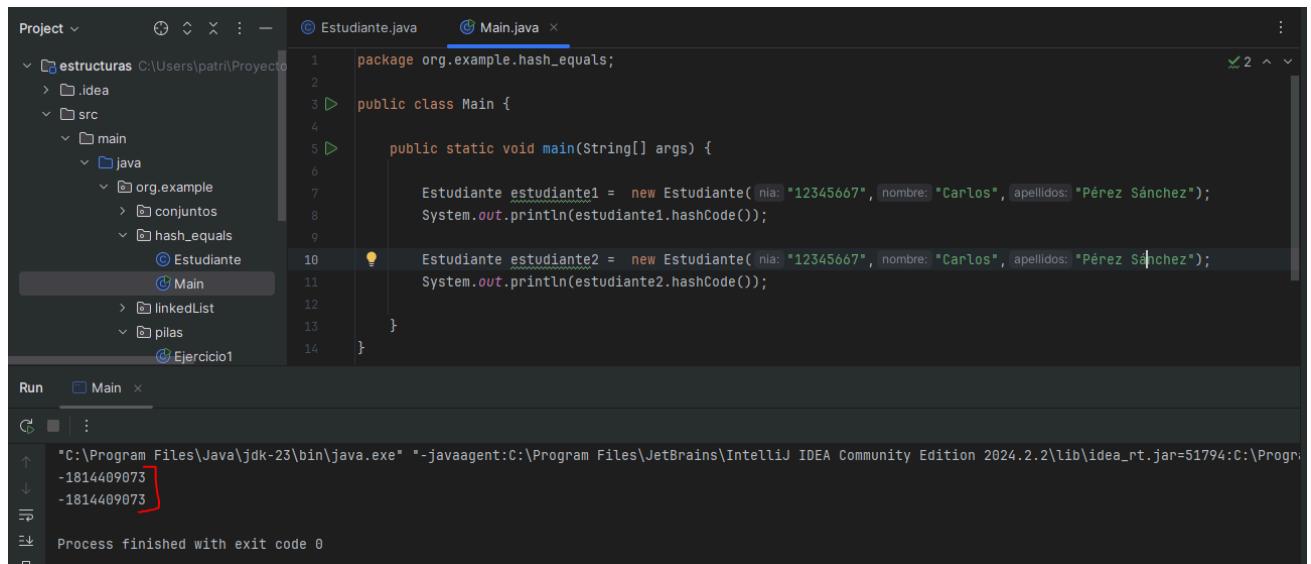
```
75 public final class Arrays {  
    ...  
    4535     @  
    4536         public static int hashCode( @Nullable Object[] a) {  
    4537             if (a == null) {  
    4538                 return 0;  
    4539             }  
    4540             return ArraysSupport.hashCode(a, fromIndex: 0, a.length, initialValue: 1);  
    4541     }  
    ...  
}
```

y seguidamente llama a un método que utiliza un multiplicador para dar valor al `hash` teniendo en cuenta los atributos de la clase. En este caso, el **31**.



```
61 public class ArraysSupport {  
    ...  
    320         public static int hashCode(Object[] a, int fromIndex, int length, int initialValue) {  
    321             int result = initialValue;  
    322             int end = fromIndex + length;  
    323             for (int i = fromIndex; i < end; i++) {  
    324                 result = 31 * result + Objects.hashCode(a[i]);  
    325             }  
    326             return result;  
    327         }  
    328     }  
    ...  
}
```

Para probarlo, vamos a crear otro estudiante con las mismas propiedades al que teníamos anteriormente. El resultado del `hash` debería ser igual entre ellos:



```

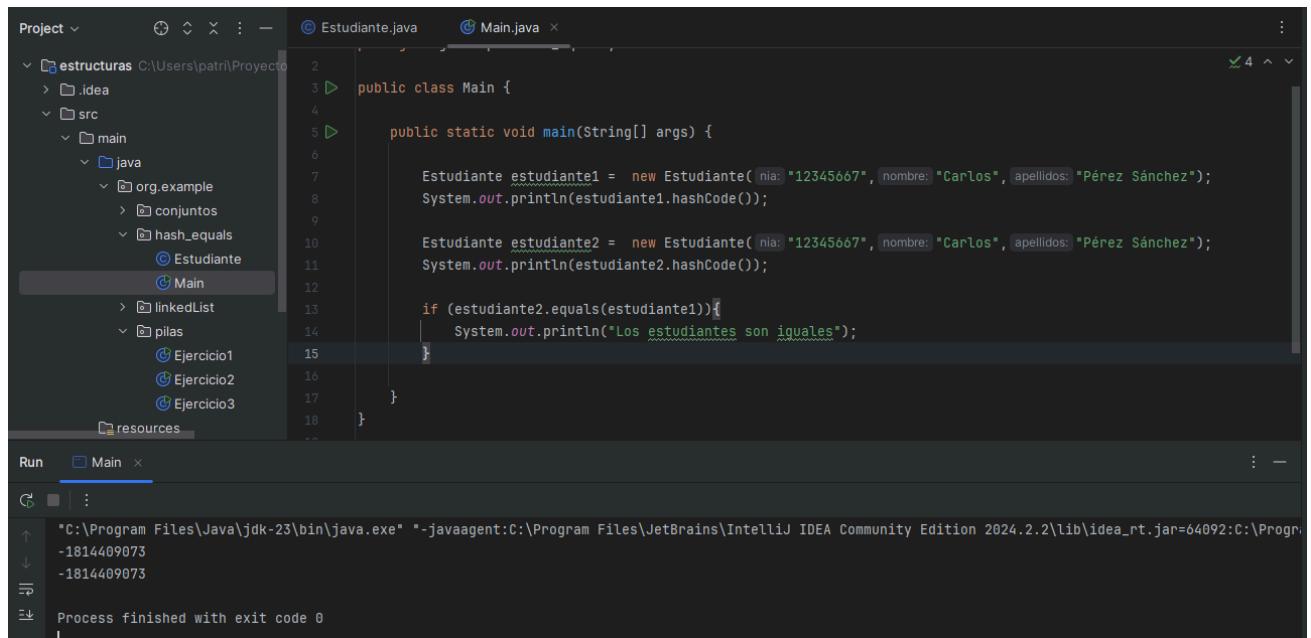
1 package org.example.hash_equals;
2
3 public class Main {
4
5     public static void main(String[] args) {
6
7         Estudiante estudiante1 = new Estudiante( nia: "12345667", nombre: "Carlos", apellidos: "Pérez Sánchez");
8         System.out.println(estudiante1.hashCode());
9
10    Estudiante estudiante2 = new Estudiante( nia: "12345667", nombre: "Carlos", apellidos: "Pérez Sánchez");
11    System.out.println(estudiante2.hashCode());
12
13 }
14 }
```

"C:\Program Files\Java\jdk-23\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2024.2.2\lib\idea_rt.jar=51794:C:\Program
-1814409073
-1814409073

Process finished with exit code 0

Ahora que ya controlamos que nuestros objetos obtengan el mismo *hash*, para terminar de identificar si son iguales deberemos sobreescibir el método *equals* para que no solamente se compruebe la referencia en memoria, sino también los valores de los atributos. Antes de recuperar el método que nos ha creado *IntelliJ*, vamos a hacer la siguiente prueba.

Vamos a añadir a nuestro *main* una condición para preguntar si nuestros objetos son iguales:



```

1 package org.example.hash_equals;
2
3 public class Main {
4
5     public static void main(String[] args) {
6
7         Estudiante estudiante1 = new Estudiante( nia: "12345667", nombre: "Carlos", apellidos: "Pérez Sánchez");
8         System.out.println(estudiante1.hashCode());
9
10    Estudiante estudiante2 = new Estudiante( nia: "12345667", nombre: "Carlos", apellidos: "Pérez Sánchez");
11    System.out.println(estudiante2.hashCode());
12
13    if (estudiante2.equals(estudiante1)){
14        System.out.println("Los estudiantes son iguales");
15    }
16
17 }
18 }
```

"C:\Program Files\Java\jdk-23\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2024.2.2\lib\idea_rt.jar=64092:C:\Program
-1814409073
-1814409073

Process finished with exit code 0

Fíjate que a pesar de ser aparentemente iguales, la condición no se cumple y no obtenemos la respuesta esperada. Esto se debe a que, aunque el *hash* ya es el mismo, el método *equals* por defecto sigue mirando solamente la referencia en memoria. En este caso, como nuestros objetos tendrán una dirección de memoria distinta (referenciados como *estudiante1* y *estudiante2*), la condición no se cumplirá.

Para que el método *equals* compruebe también el valor de los atributos, vamos a descomentar las líneas generadas por *IntelliJ* para sobreescibirlo:

```
1 package org.example.hash_equals;
2
3 import java.util.Objects;
4
5 public class Estudiante {
6
7     private String nia;
8     private String nombre;
9     private String apellidos;
10
11     public Estudiante(String nia, String nombre, String apellidos) {
12         this.nia = nia;
13         this.nombre = nombre;
14         this.apellidos = apellidos;
15     }
16
17     @Override
18     public boolean equals(Object o) {
19         if (this == o) return true;
20         if (o == null || getClass() != o.getClass()) return false;
21         Estudiante that = (Estudiante) o;
22         return Objects.equals(nia, that.nia) && Objects.equals(nombre,
23             that.nombre) && Objects.equals(apellidos, that.apellidos);
24     }
25
26     @Override
27     public int hashCode() {
28         return Objects.hash(nia, nombre, apellidos);
29     }
30 }
```

En este caso, la primera condición que va a mirar `equals` es igualmente si el objeto que hemos pasado apunta a la misma dirección de memoria que el actual. Después, mira si la clase es la misma (o no) con la instancia actual y el objeto que hemos pasado por parámetro. **Y además, ahora también comprueba que los atributos de cada uno de los objetos comparados también sean iguales.**

Si ahora volvemos a lanzar nuestro programa,

The screenshot shows the IntelliJ IDEA interface. The Project tool window on the left displays a package named 'estructuras' containing several Java files: 'idea', 'src', 'main', 'java', 'org.example', 'conjuntos', 'hash_equals', 'Estudiante', 'Main', 'linkedList', 'pilas', and 'Ejercicio1'. The Main.java file is open in the editor, showing the following code:

```
public class Main {
    public static void main(String[] args) {
        Estudiante estudiante1 = new Estudiante("12345667", "Carlos", "Pérez Sánchez");
        System.out.println(estudiante1.hashCode());

        Estudiante estudiante2 = new Estudiante("12345667", "Carlos", "Pérez Sánchez");
        System.out.println(estudiante2.hashCode());

        if (estudiante2.equals(estudiante1)) {
            System.out.println("Los estudiantes son iguales");
        }
    }
}
```

The Run tool window at the bottom shows the output of the program:

```
"C:\Program Files\Java\jdk-23\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2024.2.2\lib\idea_rt.jar=C:\Program
1814409073
1814409073
Los estudiantes son iguales
Process finished with exit code 0
```

Java ya detecta que se trata del mismo objeto, ya que además de la dirección de memoria, comprueba los valores de los atributos.

Modificar implementación de `equals()` para que solamente se tenga en cuenta ciertos atributos

Es posible que necesitemos considerar que dos objetos son iguales en función de alguno de los atributos, y no por todos. En el ejemplo anterior, podríamos establecer el `nia` para considerar si dos estudiantes son iguales o no, ya que podrían tener el mismo nombre y apellidos.

Para adaptar el método `equals` a nuestras necesidades, podremos modificarlo a mano siempre que sea necesario. En este caso, vamos a eliminar la condición que mira si el nombre y los apellidos también son iguales, y dejaremos solamente la condición del `nia`.

```
1  @Override
2  public boolean equals(Object o) {
3      if (this == o) return true;
4      if (o == null || getClass() != o.getClass()) return false;
5      Estudiante that = (Estudiante) o;
6      return Objects.equals(nia, that.nia);
7  }
```

De esta forma, solamente se considerará a dos estudiantes iguales si el `nia` que se ha guardado es el mismo.

The screenshot shows the IntelliJ IDEA interface with the Main.java file open. The code creates two `Estudiante` objects, `estudiante1` and `estudiante2`, with different NIA values but the same name and surname. The `hashCode()` method is called on both objects, and the output shows two different hash codes: -1814409073 and -1979407957 respectively.

```
public class Main {
    public static void main(String[] args) {
        Estudiante estudiante1 = new Estudiante(nia: "12345667", nombre: "Carlos", apellidos: "Pérez Sánchez");
        System.out.println(estudiante1.hashCode());
        Estudiante estudiante2 = new Estudiante(nia: "11111111", nombre: "Carlos", apellidos: "Pérez Sánchez");
        System.out.println(estudiante2.hashCode());
        if (estudiante2.equals(estudiante1)){
            System.out.println("Los estudiantes son iguales");
        }
    }
}
```

Run Main
Process finished with exit code 0

The screenshot shows the IntelliJ IDEA interface with the Main.java file open. The code creates two `Estudiante` objects, `estudiante1` and `estudiante2`, with the same NIA value but different names and surnames. The `hashCode()` method is called on both objects, and the output shows the same hash code: 4469266. This demonstrates that even though the objects are not equal based on their fields, they have the same hash code because the `hashCode()` method returns the same value for both.

```
public class Main {
    public static void main(String[] args) {
        Estudiante estudiante1 = new Estudiante(nia: "12345667", nombre: "Carlos", apellidos: "Pérez Sánchez");
        System.out.println(estudiante1.hashCode());
        Estudiante estudiante2 = new Estudiante(nia: "12345667", nombre: "Carla", apellidos: "Ruiz Martínez");
        System.out.println(estudiante2.hashCode());
        if (estudiante2.equals(estudiante1)){
            System.out.println("Los estudiantes son iguales");
        }
    }
}
```

Run Main
Process finished with exit code 0

OJO porque ahora el valor *hash* no será el mismo. Por eso, es importante que los dos métodos estén implementados juntos.

Ejercicios de colecciones haciendo uso de objetos



Ejercicio 1: impresora online

Implementa una clase Java para gestionar una impresora online, que puede recibir peticiones de impresión desde diferentes ordenadores.

Las peticiones serán impresas por orden de llegada. Cada **Petición** incluye la siguiente información:

- **id** de la máquina que solicita la impresión (por ejemplo, “PC3493”). Haz que sea constante para el usuario que ejecuta el programa.
- **nombre** del fichero a imprimir (por ejemplo, “file1.pdf”). Al crear una nueva petición, se debe comprobar que el formato del fichero es siempre **.pdf**. Si no tiene este formato, reportaremos un error y no se creará la petición. Usa el método **nombreArchivo.endsWith(".pdf")**.

El programa debe implementar los siguientes métodos:

- **añadirDocumento()**: recibe una petición de impresión como entrada y la añade al conjunto de peticiones pendientes de imprimirse. En caso de que alguna petición pendiente ya contenga el archivo que se intenta imprimir (por **id** y **nombre**), no se insertará de nuevo y se informará al usuario de que la **petición de impresión ya existe**.

Ejemplo:

Project ▾ + ⌂ × : - AppImpresora.java Peticion.java

Java

AppImpresora.java

```
public class AppImpresora {  
    public static void main(String[] args) {  
        System.out.println("**** BIENVENIDO A TU IMPRESORA ONLINE ****");  
        anyadirDocumento( nombre_archivo: "hola.txt");  
        anyadirDocumento( nombre_archivo: "doc.pdf");  
        anyadirDocumento( nombre_archivo: "doc2.pdf");  
        anyadirDocumento( nombre_archivo: "doc2.pdf");  
    }  
}
```

Run AppImpresora

```
C:\Program Files\Java\jdk-23\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2024.2.2\lib\idea_rt.jar=50979:C:\Program  
**** BIENVENIDO A TU IMPRESORA ONLINE ****  
El archivo hola.txt no es un pdf. No se puede añadir a la cola.  
Archivo doc.pdf añadido a la cola de impresión. [Peticion{id='PC1234', nombre='doc.pdf'}]  
Archivo doc2.pdf añadido a la cola de impresión. [Peticion{id='PC1234', nombre='doc2.pdf'}, Peticion{id='PC1234', nombre='doc2.pdf'}]  
La petición de impresión para el archivo doc2.pdf ya existe. [Peticion{id='PC1234', nombre='doc.pdf'}, Peticion{id='PC1234', nombre='doc2.pdf'}]  
Process finished with exit code 0
```

- **imprimirDocumento()**: coge la primera petición y muestra el nombre del fichero por consola (únicamente simula la impresión de la petición). La petición debe ser

eliminada del conjunto de peticiones al finalizar.

- **getNumPeticiones():** devuelve el número total de peticiones pendientes de imprimir.
- **verTodo():** muestra todas las peticiones que no han sido impresas.
- **imprimirTodo():** imprime todas las peticiones pendientes. Después de procesar cada petición, esta debe ser eliminada.

Escribe un **main** que incluya las llamadas necesarias para validar todos los métodos descritos anteriormente.

```
1 public static void main(String[] args) {  
2  
3     System.out.println("**** BIENVENIDO A TU IMPRESORA ONLINE **");  
4     anyadirDocumento("hola.txt");  
5     anyadirDocumento("doc.pdf");  
6     anyadirDocumento("doc2.pdf");  
7     anyadirDocumento("doc2.pdf");  
8     anyadirDocumento("doc3.pdf");  
9     anyadirDocumento("doc4.pdf");  
10    imprimirDocumento();  
11    System.out.println("Quedan " + getNumPeticiones() + " documentos en la cola.");  
12    verTodo();  
13    imprimirTodo();  
14  
15}
```

```
"C:\Program Files\Java\jdk-23\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2024.2.2\lib\idea_rt.jar=51141:C:\Program Files\Java\jdk-23\bin" -Dfile.encoding=UTF-8  
**** BIENVENIDO A TU IMPRESORA ONLINE ****  
El archivo hola.txt no es un pdf. No se puede añadir a la cola.  
Archivo doc.pdf añadido a la cola de impresión. [Peticion{id='PC1234', nombre='doc.pdf'}]  
Archivo doc2.pdf añadido a la cola de impresión. [Peticion{id='PC1234', nombre='doc.pdf'}, Peticion{id='PC1234', nombre='doc2.pdf'}]  
La petición de impresión para el archivo doc2.pdf ya existe.[Peticion{id='PC1234', nombre='doc.pdf'}, Peticion{id='PC1234', nombre='doc2.pdf'}]  
Archivo doc3.pdf añadido a la cola de impresión. [Peticion{id='PC1234', nombre='doc.pdf'}, Peticion{id='PC1234', nombre='doc2.pdf'}, Peticion{id='PC1234', nombre='doc3.pdf'}]  
Archivo doc4.pdf añadido a la cola de impresión. [Peticion{id='PC1234', nombre='doc.pdf'}, Peticion{id='PC1234', nombre='doc2.pdf'}, Peticion{id='PC1234', nombre='doc3.pdf'}, Peticion{id='PC1234', nombre='doc4.pdf'}]  
Imprimiendo... doc.pdf  
Quedan 3 documentos en la cola.  
La cola de impresión actual: [Peticion{id='PC1234', nombre='doc2.pdf'}, Peticion{id='PC1234', nombre='doc3.pdf'}, Peticion{id='PC1234', nombre='doc4.pdf'}]  
Imprimiendo... doc2.pdf  
Imprimiendo... doc3.pdf  
Imprimiendo... doc4.pdf  
La cola de impresión ha finalizado.  
  
Process finished with exit code 0
```



Ejercicio 2: recaudación del cine

Un cine de un pueblo pequeño nos propone hacer una aplicación para controlar las personas de una cola de un cine en los grandes estrenos de películas. Las personas

esperarán la cola para sacar una entrada, y tendremos que calcular el importe a pagar según la edad de la persona (mínimo 3 años):

Edad	Precio
Entre 3 y 10 años	1€
Entre 11 y 17 años	2,5€
Mayores de edad (18 años en adelante)	3,5€

Para probar, programaremos un método donde el número N de personas de la cola se elige al azar entre 0 y 250. Estas N personas se generan aleatoriamente con una edad entre 3 y 100 años. Se debe crear una clase *Persona* (*edad*) e implementar un método *generarCola()* que se invoque desde el programa principal para generar N *personas* en la cola con su respectiva edad.

Al final del programa, deberemos mostrar la cantidad total recaudada por el cine. Recuerda que la lista debe quedar vacía (una vez que una persona paga su entrada ya no está en la cola).

```
*C:\Program Files\Java\jdk-23\bin\java.exe* "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2024.2.2\lib\idea_rt.jar=51377:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2024.2.2\bin" -Dfile.encoding=UTF-8 C:\Users\Diego\OneDrive\Documentos\NetBeansProjects\CineAlPoble\src\main\java\com\diegocarrasco\cinealpoble\Cola.java
*** CINE AL POBLE ***
Hay 210 personas a la cola.[Persona{edad=31}, Persona{edad=80}, Persona{edad=49}, Persona{edad=9}, Persona{edad=41}, Persona{edad=38}, Persona{edad=49}, Persona{edad=100}, Persona{edad=50}, Persona{edad=60}, Persona{edad=70}, Persona{edad=80}, Persona{edad=90}, Persona{edad=100}, Persona{edad=110}, Persona{edad=120}, Persona{edad=130}, Persona{edad=140}, Persona{edad=150}, Persona{edad=160}, Persona{edad=170}, Persona{edad=180}, Persona{edad=190}, Persona{edad=200}, Persona{edad=210}, Persona{edad=220}, Persona{edad=230}, Persona{edad=240}, Persona{edad=250}]
La recaudación ha sido de 675.0€

Process finished with exit code 0
```



Ejercicio 3: las tareas del funcionario

Un amigo funcionario nos pide que le hagamos una *app* para ordenar sus tareas de generación de informes. Debemos gestionar cada *Informe*, que estará formado por un *código* numérico y una tarea con una *descripción* que puede ser de *tipo ADMINISTRATIVO, EMPRESARIAL y PERSONAL*. Debe comprobarse que la tarea es de alguno de estos tipos sí o sí. Además, las tareas se pueden duplicar, ya que el trabajo de este funcionario es muy monótono y se suelen repetir siempre las mismas durante el día.

Nuestro amigo quiere que seamos capaces de añadir y eliminar informes en forma de pila (el último en entrar, es el primero en salir).

Realiza un *main* de prueba para añadir 5 informes, donde:

- se muestre su contenido y se despachen todos en orden de salida,
- se añadan de nuevo 3 informes y se muestre el orden de salida.

Nuestro amigo nos pide, además, que aunque las tareas se puedan repetir, necesita una funcionalidad para saber en cualquier momento cuántos tipos de tareas (por código y tipo) debe realizar. Añádela a tu programa.



Obra publicada con Licencia Creative Commons Reconocimiento Compartir igual 4.0
<<http://creativecommons.org/licenses/by-sa/4.0/>>

7.2. Mapas

7.2. Mapas o diccionarios

Interfaz Map --> (clave-valor)

Como ya vimos durante la introducción a las estructuras del tema, un mapa es una estructura de datos que almacena información en forma de pares *clave-valor*, y es muy útil cuando necesitamos acceder a elementos de forma rápida. Es como un "diccionario" o una "agenda", donde cada entrada tiene dos partes: una clave (que es única) y un valor (que puede ser cualquier dato).

La estructura que usa Java para implementar los mapas es *HashMap*,

```
1 import java.util.HashMap;
2
3 public class EjemploHashMap {
4
5     public static void main(String[] args) {
6
7         HashMap<String, Integer> edades = new HashMap<>();
8
9         edades.put("Juan", 25);
10        edades.put("Ana", 30);
11        edades.put("Pedro", 28);
12
13        System.out.println("Edad de Juan: " + edades.get("Juan"));
14    }
15 }
16 }
```

En este ejemplo, las claves son los nombres ("Juan", "Ana", "Pedro") y los valores son las edades (25, 30, 28). La clave se convierte en un índice mediante una función *hash*, y cada valor se almacena en una celda de una *tabla hash* (una "lista" muy grande con muchas "casillas" donde se guardan las claves y los valores de forma súper

eficiente). La pregunta es, ¿cómo encontramos la casilla correcta para guardar o buscar un dato? Lo vemos en el siguiente apartado.

Los métodos principales de **HashMap** en Java son:

- **put(K, V)**: para insertar un par *clave-valor*. En caso de que una clave ya exista, pisaremos el valor antiguo por el nuevo.
- **get(K)**: para obtener el valor a partir de una clave.
- **remove(K key, V value)**: para eliminar sólo si el valor coincide.
- **containsKey(K)**: verifica si la clave existe.
- **containsValue(V value)**: verifica si existe un valor.
- **keySet()**: devuelve una lista con las claves. También sirve para iterar sólo con claves.
- **values()**: devuelve una lista con los valores. También sirve para iterar sólo con valores.

```
HashMap<String, Integer> edades = new HashMap<>();
edades.put("Juan", 25);
edades.put("Ana", 30);
edades.put("Pedro", 28);

System.out.println(edades); // {Juan=25, Ana=30, Pedro=28}

System.out.println(edades.get("Juan")); // 25
System.out.println(edades.get("Carlos")); // null

System.out.println(edades.containsKey("Ana")); // true
System.out.println(edades.containsKey("Carlos")); // false

System.out.println(edades.containsValue(30)); // true
System.out.println(edades.containsValue(40)); // false

edades.remove("Pedro");
System.out.println(edades); // {Juan=25, Ana=30}

edades.remove("Ana", 29); // no elimina nada porque el valor no coincide
System.out.println(edades); // {Juan=25, Ana=30}

for (String clave : edades.keySet()) {
    System.out.println("Clave: " + clave); //Clave: Juan
                                            // Clave: Ana
}
```

```

29 |     for (Integer valor : edades.values()) {
30 |         System.out.println("Valor: " + valor); //Valor: 25
31 |                                         //Valor: 30
32 |
33 |
34 |     edades.remove("Ana", 30); // elimina Ana porque el valor coincide
35 |     System.out.println(edades); // {Juan=25}

```

```

"C:\Program Files\Java\jdk-23\bin\java.exe" "-javaagent:s
{Ana=30, Pedro=28, Juan=25}
25
null
true
false
true
false
{Ana=30, Juan=25}
{Ana=30, Juan=25}
Clave: Ana
Clave: Juan
Valor: 30
Valor: 25
{Juan=25}

Process finished with exit code 0
.

```

- ***getOrDefault(key, defaultValue)***: es un método que permite obtener un valor asociado a una clave específica, pero si la clave no existe en el mapa, devuelve un valor por defecto en su lugar.

```

1 | HashMap<String, Integer> edades = new HashMap<>();
2 |
3 |     edades.put("Ana", 9);
4 |     edades.put("Luis", 7);
5 |
6 |     // obtener la edad de Ana (existe en el HashMap)
7 |     System.out.println(edades.getOrDefault("Ana", 18)); // salida: 9
8 |
9 |     // obtener la edad de Pedro (no existe, usa el valor predeterminado)
10 |    System.out.println(edades.getOrDefault("Pedro", 18)); // salida: 18

```

Es útil cuando trabajamos con conteos o acumulaciones, como en el problema **Va de modas...** de Acepta el Reto <<https://aceptaelreto.com/problem/statement.php?id=152>> .

Método de iteración en un *HashMap*

Hasta ahora, hemos visto los métodos *keySet()* y *values()* para obtener claves y valores por separado. Pero si necesitamos interaccionar con nuestro mapa para obtener claves y valores a la vez, usaremos un **for-each** con *.entrySet()* (*clave-valor*):

```
1 for (Map.Entry<String, Integer> entry : map.entrySet()) {  
2     System.out.println("Clave: " + entry.getKey() + ", Valor: " + er  
3 }  
4  
5 // Clave: Juan, Valor: 25  
6 // Clave: Ana, Valor: 30  
7 // Clave: Pedro, Valor: 28
```

Si has probado los iteradores anteriores, te habrás dado cuenta de algo importante: las estructuras *HashMap*, igual que pasaba con los *HashSet*, no mantienen el orden de inserción. Si necesitamos orden, usaremos **LinkedHashMap** (aunque su uso no es muy común).

Asociar clave con múltiples valores

```
1 HashMap<String, List<String>> amigos = new HashMap<>();  
2  
3 // añadimos los amigos de Juan  
4 amigos.put("Juan", new ArrayList<>());  
5 amigos.get("Juan").add("Pedro");  
6 amigos.get("Juan").add("Ana");  
7  
8 System.out.println(amigos); // {Juan=[Pedro, Ana]}
```

Uso de mapas manipulando archivos *JSON*

El uso principal de los *HashMap* actualmente es la manipulación de archivos tipo *JSON* característicos de las *APIs* y microservicios. También en acceso a bases de datos *NoSQL (MongoDB)*.

```
1  {
2      "nombre": "Juan",
3      "edad": 25
4 }
```

Otro ejemplo:

```
1  {
2      "usuario": {
3          "nombre": "Carlos",
4          "edad": 32,
5          "activo": true,
6          "hobbies": ["fútbol", "lectura", "música"],
7          "direccion": {
8              "ciudad": "Madrid",
9              "pais": "España"
10         }
11     }
12 }
```

Aquí tenemos:

- Un objeto "*usuario*".
- Claves con valores de distintos tipos (*String, Number, Boolean*).
- Un *array* "*hobbies*".
- Otro objeto anidado "*direccion*".

Una librería típica para manipular *JSON* es *Gson* (*com.google.gson.Gson*), de *Google*. Aquí tienes un ejemplo con *Gson* para convertir *JSON* a un objeto *Java*:

```
import com.google.gson.Gson;

class Usuario {
    String nombre;
    int edad;
    boolean activo;
}
```

```
8
9 public class Main {
10     public static void main(String[] args) {
11
12         String json = "{ \"nombre\": \"Carlos\", \"edad\": 32, \"act
13
14         Gson gson = new Gson();
15         Usuario usuario = gson.fromJson(json, Usuario.class);
16
17         System.out.println("Nombre: " + usuario.nombre);
18         System.out.println("Edad: " + usuario.edad);
19         System.out.println("Activo: " + usuario.activo);
20
21     }
22 }
```



Ejercicio 1

Dado un texto introducido por el usuario, cuenta cuántas veces aparece cada palabra y muestra el resultado.

Entrada de ejemplo:

hola mundo hola java hola código

Salida esperada:

```
hola: 3
mundo: 1
java: 1
código: 1
```



Ejercicio 2: contador de caracteres

Dada una palabra, cuenta cuántas veces aparece cada carácter.

Entrada de ejemplo:

banana

Salida esperada:

```
b: 1  
a: 3  
n: 2
```



Ejercicio 3. Registro de temperaturas por ciudad

Con todo el lío de la DANA, los negacionistas del cambio climático se están replanteando sus creencias y han decidido implementar un sistema sencillo que registre la temperatura en diferentes ciudades para ver si todo lo que se dice por parte de la comunidad científica es verdad. Como no se fían de los sensores automáticos, ellos mismos medirán la temperatura con un termómetro e irán insertando los datos a mano mediante una *app*.

- Cada ciudad tendrá una temperatura registrada.
- El usuario podrá *insertar, actualizar y consultar* temperaturas por ciudad.
- También podrá *mostrar* un listado de todas las temperaturas registradas en todas las ciudades.

Ejemplo de ejecución:

```
*** REGISTRO DE TEMPERATURAS ***
Elige una opción [insertar, actualizar, consultar, ver todas, salir]: insertar
Introduce los nuevos datos (ciudad-temperatura): Mutxamel-30

Elige una opción [insertar, actualizar, consultar, ver todas, salir]: insertar
Introduce los nuevos datos (ciudad-temperatura): San Juan-28

Elige una opción [insertar, actualizar, consultar, ver todas, salir]: consultar
Introduce la ciudad a consultar: Alicante
La ciudad de Alicante no tiene temperaturas registradas.

Elige una opción [insertar, actualizar, consultar, ver todas, salir]: consultar
Introduce la ciudad a consultar: Mutxamel
Temperatura en Mutxamel: 30º

Elige una opción [insertar, actualizar, consultar, ver todas, salir]: ver todas
Registro de temperaturas:
Mutxamel: 30º
San Juan: 28º

Elige una opción [insertar, actualizar, consultar, ver todas, salir]: salir
```



Ejercicio 4. Evitando el pucherazo

En los cursos de ciclos formativos del instituto se está llevando a cabo la elección de delegado, en la que cada estudiante escribe el nombre de su candidato en un papel. Como la mayoría de los estudiantes son de informática y no se fían del proceso de votación, deciden aprovechar sus conocimientos de programación para crear una pequeña app que contabilice los votos y determine cuántos votos recibe cada candidato de forma objetiva.

Entrada:

- Se escriben nombres de candidatos, uno por línea.
- Para finalizar la votación, se escribe la palabra "*fin*".

Entrada:

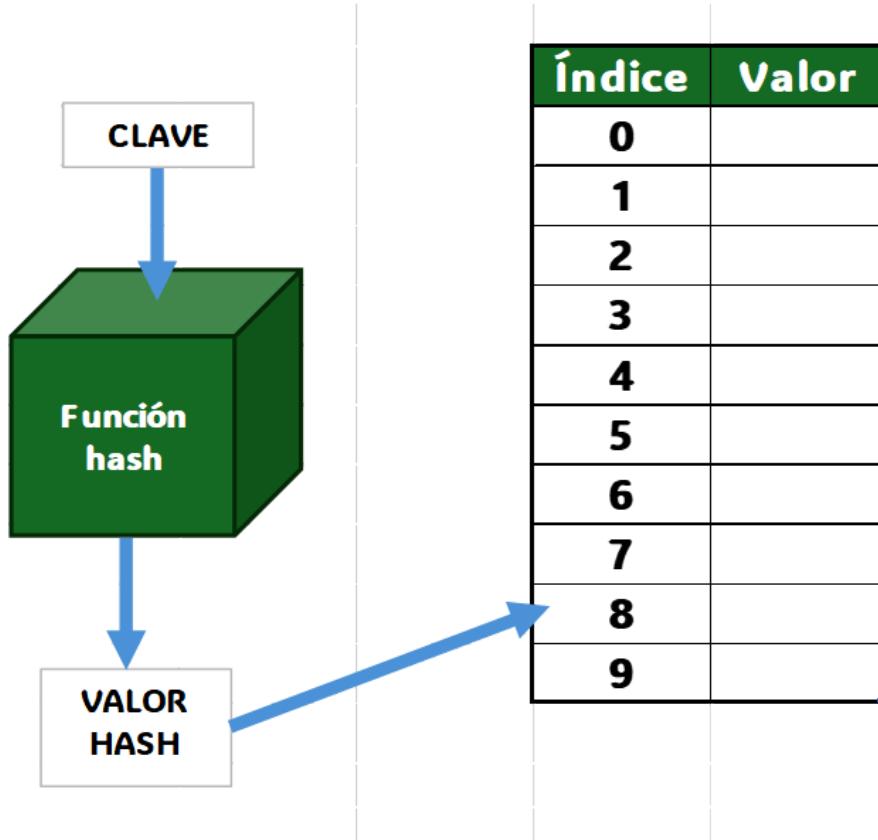
```
Ana  
Juan  
Pedro  
Ana  
Juan  
Ana  
fin
```

Salida:

```
Resultados:  
Ana: 3 votos  
Juan: 2 votos  
Pedro: 1 votos
```

Tablas Hash

Una tabla *hash* es simplemente una estructura de datos que organiza los elementos en "celdas" usando una función matemática llamada función *hash*. Esta función convierte las claves (como "Juan") en números, que se usan como índices para colocar los valores en un *array* o lista.



Funciona de la siguiente manera:

1. Se aplica una función *hash* a la clave

Ejemplo: si la clave es "Juan", la función *hash* genera un número (índice en el *array* a guardar el valor). Por ejemplo, **8**.

2. Se almacena el valor en la posición del array indicada por la función hash

Si $\text{hash}(\text{"Juan"}) = 8$, entonces "Juan" → 25 se guarda en la **posición 8** de la tabla.

Índice	Clave	Valor
0	-	-
1	-	-
2	-	-
3	-	-
4	-	-
5	-	-
6	-	-
7	-	-
8	Juan	25
9	-	-

3. Cuando queremos acceder a un valor, la función *hash* coge la clave ("Juan"), calcula el número de índice y encuentra rápidamente la casilla que corresponde

Cuando hacemos *get("Juan")*, la función *hash* obtiene el índice 8 y recupera el valor 25.

¿Y si con claves distintas se obtiene la misma posición?

Cuando dos claves diferentes producen el mismo índice, ocurre una **colisión**.

$$\text{hash}(\text{"Juan"}) = 8$$

$\text{hash}(\text{"Ana"}) = 8$ ambos datos quieren ocupar la misma posición en la tabla.

Una buena función *hash* debería generar posiciones aleatorias y uniformemente distribuidas en la tabla para minimizar las **colisiones**: cuando dos elementos diferentes generan la misma posición en la tabla. Si la función *hash* no es lo suficientemente buena, pueden producirse colisiones frecuentes y disminuir el rendimiento de la tabla *hash*.

Índice	Clave	Valor
0	-	-
1	-	-
2	-	-
3	-	-
4	-	-
5	-	-
6	-	-
7	-	-
8	Juan	25
9		

COLISIÓN

hash("Juan") = 8
hash("Ana") = 8

Soluciones para las colisiones

Existen dos formas principales para manejar colisiones:

- **Encadenamiento (*Chaining*)**. Usa una lista en cada índice para almacenar múltiples valores con la misma clave. En Java, *HashMap* usa internamente listas enlazadas.

```
1 // representación conceptual
2 tablaHash[8] -> ["Juan" -> 25] -> ["Ana" -> 30]
```

Índice	Clave	Valor
0	-	-
1	-	-
2	-	-
3	-	-
4	-	-
5	-	-
6	-	-
7	-	-
8	Juan --> Ana	25 --> 30
9	-	-

Así, la búsqueda en caso de colisión, revisa la lista enlazada.

- **Dirección abierta (*Open Addressing*)**. En lugar de usar una lista, si una posición está ocupada, se busca la siguiente disponible en la tabla.

Índice	Clave	Valor
0	-	-
1	-	-
2	-	-
3	-	-
4	-	-
5	-	-
6	-	-
7	-	-
8	Juan	25
9	Ana	30

Implementación interna de HashMap como Tabla Hash en Java

En Java, *HashMap* usa:

- Un array de "buckets": cada posición del *array* es un *bucket* que almacena un nodo *clave-valor*.
- *Nodos (<K, V>)*: son pares clave-valor con punteros (--) a posibles colisiones.
- *Rehashing*: cuando la tabla se llena demasiado (carga $\geq 75\%$), se duplica su tamaño y se vuelven a ubicar los elementos.

Índice	Bucket
0	-
1	-
2	-
3	-
4	-
5	-
6	-
7	-
8	Juan --> 25
9	

Cuando se inserta una clave:

1. Se calcula su *hash*.
2. Se almacena en el *bucket* correspondiente.
3. Si hay colisión, se usa una lista enlazada.

Índice	Bucket
0	-
1	-
2	-
3	-
4	-
5	-
6	-
7	-
8	Juan → 25, Ana → 30
9	

Ventajas y desventajas de *HashMap*

- ✓ Búsqueda rápida.
- ✗ El consumo de memoria puede ser mayor que con otras estructuras.
- ✗ Riesgo de colisiones que pueden degradar el rendimiento si hay muchas.

Diccionario Español - Inglés



Se pide hacer una aplicación Java que gestione un diccionario español-inglés.

La aplicación cargará en memoria un diccionario con parejas de palabras en español e inglés, y a continuación realizará un simple cuestionario al usuario mostrándole palabras en español y pidiéndole que escriba su correspondiente traducción al inglés.

Para ello, se tendrá que crear una clase *Diccionario* con las siguientes especificaciones:

- La clase definirá una colección de parejas de palabras en español-inglés.
- Esta clase tendrá una única propiedad *diccionario* que será un *HashMap* con pares *clave-valor*, ambos del tipo *String*.
- Por defecto, el *Diccionario* construye la propiedad *diccionario* como un *HashMap* vacío.

Métodos

- *nuevoPar()* - Este método recibirá una palabra en español y otra en inglés, y las introducirá en el *HashMap* como nuevo par *clave-valor*.
- *traduce()* - Este método recibirá una palabra en español, la buscará en la propiedad diccionario y devolverá su correspondiente valor en inglés.
- *palabraAleatoria()* - Este método no recibirá ningún parámetro. El método devolverá aleatoriamente una de las palabras en español del diccionario.
- *primeraLetraTraducion()* - Este método recibirá una palabra en español y devolverá la primera letra de su correspondiente palabra en inglés.

El programa principal debe instanciar un objeto de la clase *Diccionario*, y rellenarlo con parejas de palabras español-inglés. Una vez cargado el objeto *Diccionario*, el programa iniciará un pequeño cuestionario en el que le preguntará al usuario la traducción de una palabra en español. El usuario escribirá la respuesta en inglés y el programa le dirá si ha acertado o no.

El programa finaliza cuando el usuario escribe "*fin*" como respuesta.

La salida del programa será similar a la siguiente:

```
Perro: D...
Indique la respuesta: dog
¡CORRECTO!

Mesa: T...
Indique la respuesta: Tabla
¡NO! La respuesta correcta es Table

Coche: C...
Indique la respuesta: fin

FIN DEL PROGRAMA
Total preguntas: 2
Total aciertos: 1
Total errores: 1
Aciertos: 50%
```

Se proporciona la primera letra de la palabra en inglés como pista.



Obra publicada con Licencia Creative Commons Reconocimiento Compartir igual 4.0
<http://creativecommons.org/licenses/by-sa/4.0/>

7.3. Métodos útiles

7.3. Métodos útiles para la manipulación de colecciones

Cuando trabajamos con colecciones, a menudo necesitamos recorrer elementos de manera eficiente y organizarlos según ciertos criterios. Para lograrlo, Java nos ofrece tres herramientas clave:

- **Iterator**: permite recorrer colecciones sin exponer su estructura interna, facilitando la eliminación segura de elementos durante la iteración.
- **Comparable**: define un orden natural dentro de una clase, permitiendo comparar objetos entre sí de forma predeterminada.
- **Comparator**: proporciona una forma flexible de definir múltiples criterios de ordenación sin modificar la clase original.

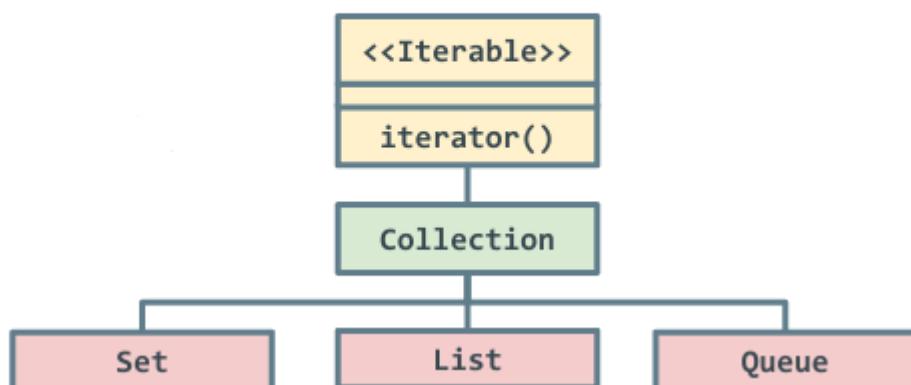
En los siguientes apartados veremos cada uno de estos conceptos en detalle.

7.3.1. Iteradores

Hasta ahora, para recorrer nuestras colecciones (da igual el tipo) hemos usado principalmente dos estructuras: *for-each* y *while* (*!colección.isEmpty()*).

En el caso del *while*, vimos que no nos quedaba más remedio que introducirlo para eliminar elementos de pilas (hacer *pop()*) y colas (hacer *poll()*), ya que, mediante el *for-each* Java se volvía loco al eliminar "cosas" mientras se recorre esa misma la lista (obteniendo una excepción *ConcurrentModificationException*). Por otro lado, *for-each* nos servía para recorrer nuestras listas con un cierto orden, cosa que con *while* (*!colección.isEmpty()*) no podemos hacer a no ser que vayamos borrando elementos y modificando la lista sobre la marcha. Como ves, con ambas estructuras estamos limitados para realizar según qué cosas.

Para dar solución a los problemas de cada estructura, tenemos disponible la *interfaz Iterable<E>* y su método *iterator()* que implementan todo tipo de clases que desciendan de *Collection*.



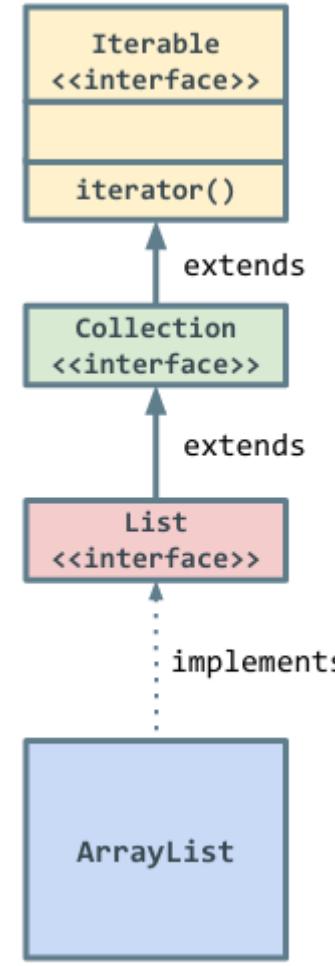
Pruebas.java x Collection.java x ArrayList.java List.java SequencedCollection.java

Author: Josh Bloch, Neal Gafter

Type parameters: <E> – the type of elements in this collection

```
257
258  public interface Collection<E> extends Iterable<E> {
259      // Query Operations
260 }
```

Por ejemplo, para *ArrayList*:



La forma de utilizar el método **iterator()** es creando un objeto de tipo **Iterator<E>**, que es el único método abstracto que contiene la interfaz **Iterable**:

```

jls          14.14.2 The enhanced for statement

42  public interface Iterable<T> {
    ...
    Returns an iterator over elements of type T.
    Returns: an Iterator.

    @NotNull
48  Iterator<T> iterator();
49

    Performs the given action for each element of the Iterable until all elements have been
    processed or the action throws an exception. Actions are performed in the order of iteration
  
```

```

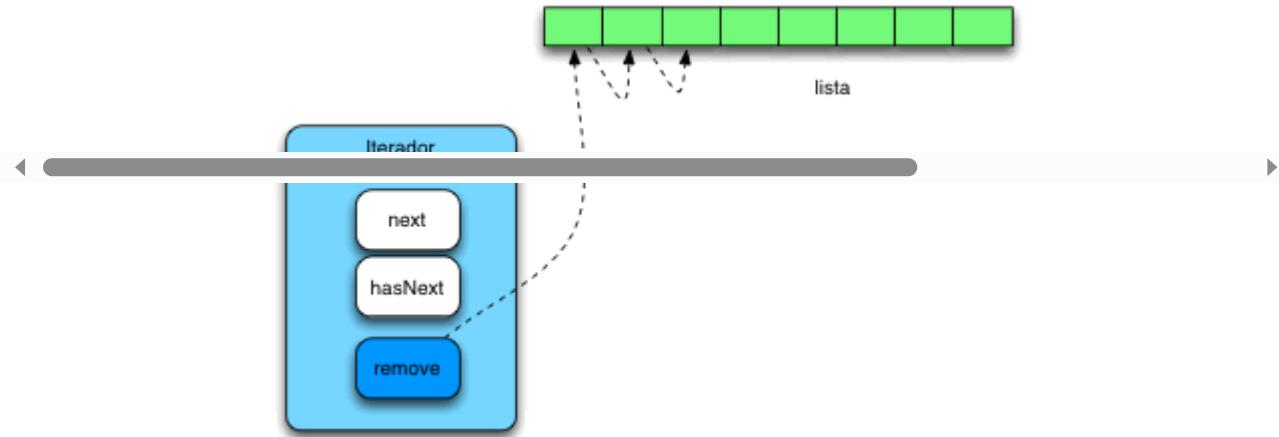
public class Main {

    public static void main(String[] args) {

        ArrayList<String> nombres = new ArrayList<>(Arrays.asList("Patr
        Iterator<String> it = nombres.iterator();
  
```

```
9 |     }  
  }
```

Este objeto permite recorrer elementos de una colección uno por uno, haciendo uso de los métodos `.hasNext()`, `.next()` y `.remove()`:



- `next()` - devuelve el siguiente elemento en la colección. Si no existe próximo elemento y se invoca, se produce una **NoSuchElementException**.
- `hasNext()` - devuelve `true` si existe un próximo objeto a retornar a través de la llamada a la función `next()`.
- `remove()` - Elimina el último objeto returned por la función `next()`. Si no se invoca `next()` antes de `remove()` o se invoca dos veces a `remove()` después de `next()`, se produce una **IllegalStateException**.

La forma de usar el iterador también es a través de un `while`, pero modificaremos la condición:

```
public class Main {  
  
    public static void main(String[] args) {  
  
        ArrayList<String> nombres = new ArrayList<>(Arrays.asList("Patr  
Iterator<String> it = nombres.iterator();  
  
        while (it.hasNext()) {  
            String elemento = it.next();  
            System.out.println(elemento);  
        }  
  
    }  
}
```

```
}
```

```
"C:\Program Files\Java\jdk-23\bin\java.exe
Patri
Pedro
Luis
Paula

Process finished with exit code 0
|
```

Y dirás... Pero esto es lo mismo que el *for-each*. Eso es porque todavía no ha entrado en acción el método *remove()*, el cual nos va a permitir eliminar elementos sin que salte una excepción de error:

```
1 public class Main {
2
3     public static void main(String[] args) {
4
5         ArrayList<String> nombres = new ArrayList<>(Arrays.asList("Patr
6             Iterator<String> it = nombres.iterator();
7
8             while (it.hasNext()) {
9                 String elemento = it.next();
10                System.out.println(elemento);
11
12                if (elemento.equals("Patri")) {
13                    it.remove();
14                }
15
16            }
17
18            System.out.println(nombres); // [Pedro, Luis, Paula]
19
20        }
21
22    }
```

Uso de *Iterator* con clases personalizadas

También se puede usar un **Iterator** para recorrer listas de objetos en Java. Ejemplo con una lista de objetos de una clase *Persona*...

```
1 class Persona {  
2     String nombre;  
3     int edad;  
4  
5     public Persona(String nombre, int edad) {  
6         this.nombre = nombre;  
7         this.edad = edad;  
8     }  
9  
10    @Override  
11    public String toString() {  
12        return nombre + " (" + edad + " años)";  
13    }  
14}
```

Uso de *Iterator* con objetos *Persona*:

```
import java.util.ArrayList;  
import java.util.Iterator;  
  
public class IteratorObjetos {  
    public static void main(String[] args) {  
  
        ArrayList<Persona> listaPersonas = new ArrayList<>();  
        listaPersonas.add(new Persona("Juan", 25));  
        listaPersonas.add(new Persona("Ana", 30));  
        listaPersonas.add(new Persona("Luis", 22));  
  
        Iterator<Persona> iterador = listaPersonas.iterator();  
  
        while (iterador.hasNext()) {  
            Persona persona = iterador.next();  
            System.out.println(persona);  
  
            // eliminar personas menores de 25 años  
            if (persona.edad < 25) {  
                iterador.remove();  
            }  
        }  
    }  
}
```

```
24     System.out.println("Lista final después de eliminar menores de ");
25     for (Persona p : listaPersonas) {
26         System.out.println(p);
27     }
28 }
29 }
```

```
Juan (25 años)
Ana (30 años)
Luis (22 años)
```

```
Lista final después de eliminar menores de 25 años:
Juan (25 años)
Ana (30 años)
```

Para estructuras Map...

En Java, `HashMap<K, V>` no implementa `Iterable` directamente, pero podemos usar `Iterator` para recorrer sus claves, valores o pares clave-valor.

```
1 public class IterarHashMapPares {
2     public static void main(String[] args) {
3
4         HashMap<Integer, String> mapa = new HashMap<>();
5         mapa.put(1, "Ana");
6         mapa.put(2, "Luis");
7         mapa.put(3, "Carlos");
8
9         Iterator<Map.Entry<Integer, String>> iterador = mapa.entrySet()
10
11        while (iterador.hasNext()) {
12            Map.Entry<Integer, String> entrada = iterador.next();
13            System.out.println("Clave: " + entrada.getKey() + ", Valor:
14
15        }
16    }
17 }
```

Cuidado con modificar la lista después de inicializar *Iterator*.
ConcurrentModificationException

Para la lista declarada previamente, realizar una operación como la siguiente no lanzará ninguna excepción:

```
1 public class Main {  
2  
3     public static void main(String[] args) {  
4  
5         ArrayList<String> nombres = new ArrayList<>(Arrays.asList("Patr"  
6             Iterator<String> it = nombres.iterator();  
7             nombres.set(1, "Juan");  
8  
9             while (it.hasNext()) {  
10                 String elemento = it.next();  
11                 System.out.println(elemento);  
12  
13                 if (elemento.equals("Patri")) {  
14                     it.remove();  
15                 }  
16  
17             }  
18  
19             System.out.println(nombres); // [Juan, Luis, Paula]  
20         }  
21     }  
22 }  
23 }
```

```
"C:\Program Files\Java\jdk-23\bin\java.exe"  
Patri  
Juan  
Luis  
Paula  
[Juan, Luis, Paula]  
  
Process finished with exit code 0  
|
```

Esto es porque modificar un elemento existente con *set()* no cambia la estructura de la lista, sólo reemplaza un valor. Sin embargo, añadir un nuevo elemento después de inicializar el *Iterator* generará una *ConcurrentModificationException*:

```

1 public class Main {
2
3     public static void main(String[] args) {
4
5         ArrayList<String> nombres = new ArrayList<>(Arrays.asList("Patr"
6             Iterator<String> it = nombres.iterator();
7             nombres.add("Juan");
8
9             while (it.hasNext()) {
10                 String elemento = it.next(); // aquí se lanza ConcurrentMod
11                 System.out.println(elemento);
12
13                 if (elemento.equals("Patri")) {
14                     it.remove();
15                 }
16
17             }
18
19             System.out.println(nombres); // [Pedro, Luis, Paula]
20
21         }
22
23     }

```

```

"C:\Program Files\Java\jdk-23\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community
Exception in thread "main" java.util.ConcurrentModificationException Create breakpoint
    at java.base/java.util.ArrayList$Itr.checkForComodification(ArrayList.java:1096)
    at java.base/java.util.ArrayList$Itr.next(ArrayList.java:1050)
    at org.example.Main.main(Main.java:17)

Process finished with exit code 1

```

Cualquier modificación estructural en la lista (como `add()` o `remove()`) después de crear el `Iterator` rompe la sincronización y dispara una `ConcurrentModificationException`. Por eso, **es importante usar siempre `iterator.remove()`**.

ListIterator

ListIterator es una versión mejorada de *Iterator* que permite recorrer una *List* en ambas direcciones (hacia adelante y hacia atrás), además de modificar la lista de forma segura durante la iteración con algunos métodos añadidos.

En este caso, los métodos *next()* y *hasNext()* se reemplazan por *previous()* y *hasPrevious()*.

```
1 public class ListIteratorEjemplo {  
2     public static void main(String[] args) {  
3  
4         List<String> frutas = new ArrayList<>(Arrays.asList("Manzana",  
5  
6             "Banana", "Cereza"));  
7  
8         ListIterator<String> listIterator = frutas.listIterator();  
9  
10        System.out.println("Recorriendo hacia adelante:");  
11        while (listIterator.hasNext()) {  
12            System.out.println(listIterator.next());  
13        }  
14  
15        System.out.println("Volviendo hacia atrás:");  
16        while (listIterator.hasPrevious()) {  
17            System.out.println(listIterator.previous());  
18        }  
19    }  
20}
```

```
"C:\Program Files\Java\jdk-23\bin\java  
Recorriendo hacia adelante:  
Manzana  
Banana  
Cereza  
Volviendo hacia atrás:  
Cereza  
Banana  
Manzana  
  
Process finished with exit code 0
```

Si no hubiéramos iniciado el recorrido hacia adelante y solamente queremos recorrer la lista hacia atrás, deberemos iniciar el iterador en la última posición:

```
1 | ListIterator<String> listIterator = frutas.listIterator(frutas.size());
```

Para **modificar elementos** tenemos disponible el método ***ListIterator.set()***:

```
1 | public class ListIteratorEjemplo {  
2 |     public static void main(String[] args) {  
3 |  
4 |         List<String> frutas = new ArrayList<>(Arrays.asList("Manzana",  
5 |  
6 |         ListIterator<String> listIterator = frutas.listIterator();  
7 |  
8 |         System.out.println("Recorriendo hacia adelante:");  
9 |         while (listIterator.hasNext()) {  
10 |             System.out.println(listIterator.next());  
11 |             listIterator.set("Mango"); // modifica la lista de forma se-  
12 |         }  
13 |  
14 |         System.out.println("Recorriendo hacia atrás:");  
15 |         while (listIterator.hasPrevious()) {  
16 |             System.out.println(listIterator.previous());  
17 |         }  
18 |  
19 |         System.out.println("Lista modificada: " + frutas);  
20 |     }  
21 | }  
22 | }
```

Lista modificada: [Manzana, Mango, Cereza]

Añadir elementos con ***.add()***:

```
public class AgregarEliminarConListIterator {  
    public static void main(String[] args) {  
  
        List<String> frutas = new ArrayList<>(Arrays.asList("Manzana",  
        ListIterator<String> listIterator = frutas.listIterator();  
  
        while (listIterator.hasNext()) {
```

```

10     String fruta = listIterator.next();
11
12     if (fruta.equals("Banana")) {
13         listIterator.add("Sandía"); // inserta después de "Banana"
14     }
15
16     if (fruta.equals("Cereza")) {
17         listIterator.remove(); // elimina "Cereza" de forma segura
18     }
19 }
20
21 System.out.println("Lista final: " + frutas); // [Manzana, Banana]
22
23 }

```

OJO: cuando utilizamos el método `add()`, el elemento se inserta en la posición actual del cursor, y el cursor se mueve después del elemento insertado.

Y además, `ListIterator` nos permite **obtener los índices** con `nextIndex()` (devuelve el índice del próximo elemento en la lista) y `previousIndex()` (devuelve el índice del elemento anterior en la lista).

- Ejemplo iterando hacia adelante (`nextIndex()`):

```

import java.util.*;

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

        List<String> frutas = new ArrayList<>(Arrays.asList("Manzana",
                                                               "Plátano",
                                                               "Banana",
                                                               "Cereza"));

        ListIterator<String> listIterator = frutas.listIterator();

        while (listIterator.hasNext()) {
            int actualIndex = listIterator.nextIndex();
            System.out.println("Antes de avanzar: nextIndex = " + actualIndex);

            String fruta = listIterator.next(); // avanza al siguiente elemento
            System.out.println("Elemento: " + fruta + ", previousIndex = " +
                               listIterator.previousIndex());
            System.out.println("-----");
        }
    }
}

```

```

    Antes de avanzar: nextIndex = 0
    Elemento: Manzana, previousIndex = 0
    -----
    Antes de avanzar: nextIndex = 1
    Elemento: Banana, previousIndex = 1
    -----
    Antes de avanzar: nextIndex = 2
    Elemento: Cereza, previousIndex = 2
    -----

```

- Ejemplo iterando hacia atrás (.previousIndex()):

```

1 public class ListIteratorReverso {
2     public static void main(String[] args) {
3
4         List<String> frutas = new ArrayList<>(Arrays.asList("Manzana",
5
6             ListIterator<String> listIterator = frutas.listIterator(frutas.
7
8             while (listIterator.hasPrevious()) {
9                 int actualIndex = listIterator.previousIndex();
10                System.out.println("Antes de retroceder: previousIndex = "
11
12                String fruta = listIterator.previous(); // retrocede un ele-
13
14                System.out.println("Elemento: " + fruta + ", nextIndex = "
15                System.out.println("-----");
16            }
17        }
18    }

```

```

    Antes de retroceder: previousIndex = 2
    Elemento: Cereza, nextIndex = 2
    -----
    Antes de retroceder: previousIndex = 1
    Elemento: Banana, nextIndex = 1
    -----
    Antes de retroceder: previousIndex = 0
    Elemento: Manzana, nextIndex = 0
    -----

```

- Otro ejemplo:

```
1 public class EjemploListIterator {  
2     public static void main(String[] args) {  
3  
4         List<String> frutas = new ArrayList<>(Arrays.asList("Manzana",  
5             ListIterator<String> iter = frutas.listIterator();  
6  
7             System.out.println("Al inicio:");  
8             System.out.println("nextIndex(): " + iter.nextIndex()); // 0 (p  
9             System.out.println("previousIndex(): " + iter.previousIndex());  
10  
11            System.out.println("Avanzamos con next():");  
12            System.out.println("Elemento: " + iter.next()); // Manzana  
13            System.out.println("nextIndex(): " + iter.nextIndex()); // 1  
14            System.out.println("previousIndex(): " + iter.previousIndex());  
15  
16            System.out.println("Avanzamos otra vez con next():");  
17            System.out.println("Elemento: " + iter.next()); // Banana  
18            System.out.println("nextIndex(): " + iter.nextIndex()); // 2  
19            System.out.println("previousIndex(): " + iter.previousIndex());  
20  
21            System.out.println("Retrocedemos con previous():");  
22            System.out.println("Elemento: " + iter.previous()); // Banana  
23            System.out.println("nextIndex(): " + iter.nextIndex()); // 1  
24            System.out.println("previousIndex(): " + iter.previousIndex());  
25  
26        }  
27    }
```



Ejercicio

María está aprendiendo mecanografía y usa un editor de texto muy básico. Sin embargo, este editor tiene algunas teclas especiales que afectan la forma en que se escriben los textos.

Dado un conjunto de pulsaciones de teclas, ¿cuál será el texto final que María escribirá en el editor?

Teclas especiales:

- < → Mueve el cursor una posición a la izquierda (si es posible).
- > → Mueve el cursor una posición a la derecha (si es posible).

→ Borra el carácter a la izquierda del cursor (como la tecla "retroceso").
Cualquier otra tecla se escribe en la posición actual del cursor.

Ejemplo de entradas y salidas:

hola - Salida: hola
ho<la - Salida: hla
holla# - Salida: holl
h#o<l>a - Salida: loa

Como salida se imprime el texto final resultante después de procesar todas las pulsaciones:



Acepta el reto: Teclado estropeado (144)

Teclado estropeado - ¡Acepta el reto!
[<https://aceptaelreto.com/problem/statement.php?id=144&cat=18>](https://aceptaelreto.com/problem/statement.php?id=144&cat=18)

7.3.2. Comparadores

Como ya hemos ido viendo durante el curso, en caso de necesitar ordenar los elementos de una lista o *array*, podemos usar la clase **Collections** y su método **sort()**. Por ejemplo, si nos pidieran ordenar la lista [1, 3, 5, 4, 2], tanto nosotros como el método **sort()** podríamos decir fácilmente que la respuesta es [1, 2, 3, 4, 5], ya que los números enteros tienen un orden natural. ¿Pero qué pasa con clases personalizadas?

Supongamos que estamos trabajando en una clase que representa a una **Persona** mediante su nombre y apellido. Hemos creado una clase básica para hacerlo e implementado correctamente los métodos *equals* y *hashCode*:

```
public class Persona {  
    private final String apellido;  
    private final String nombre;  
  
    public Persona(String nombre, String apellido) {  
        this.nombre = nombre;  
        this.apellido = apellido;  
    }  
  
    public String getNombre() {  
        return nombre;  
    }  
  
    public String getApellido() {  
        return apellido;  
    }  
  
    public String toString() {  
        return apellido + ", " + nombre;  
    }  
  
    @Override  
    public boolean equals(Object o) {  
        if (!(o instanceof Persona)) return false;  
        Persona p = (Persona) o;  
        return nombre.equals(p.nombre) && apellido.equals(p.apellido);  
    }  
  
    @Override  
    public int hashCode() {
```

```

31         return Objects.hash(nombre, apellido);
32     }
33 }
```

Ahora queremos ordenar una lista de objetos *Persona* por su *nombre* y *apellido*, como en el siguiente caso:

```

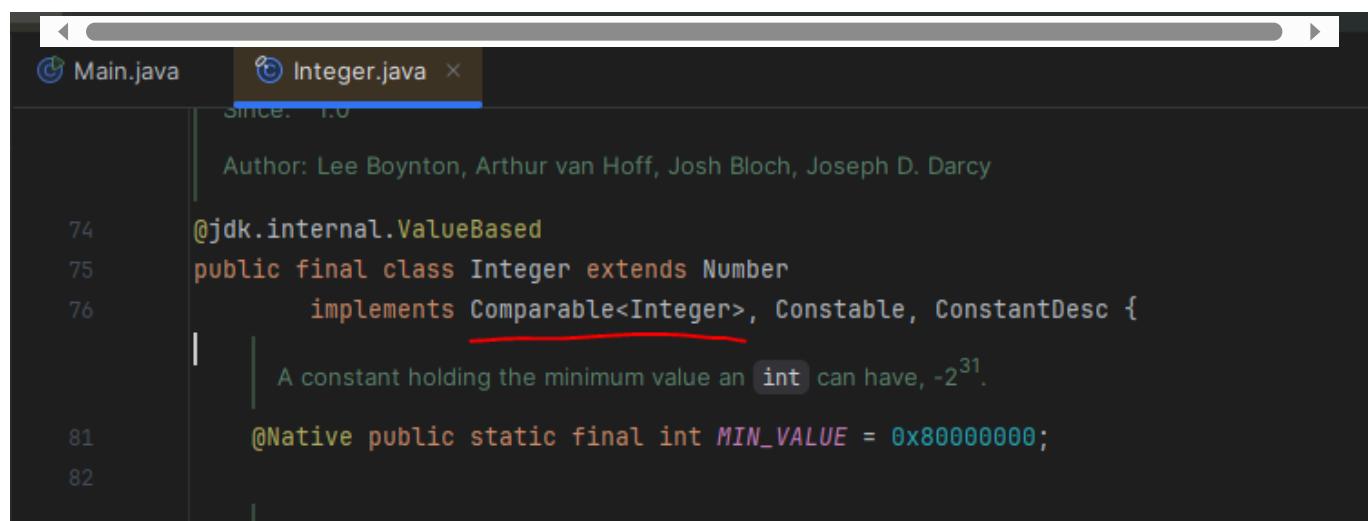
1 public static void main(String[] args) {
2     List<Persona> personas = Arrays.asList(
3         new Persona("Juan", "Pérez"),
4         new Persona("Ana", "Gómez"),
5         new Persona("Carlos", "López"),
6         new Persona("Beatriz", "López"),
7         new Persona("Pedro", "Pérez")
8     );
9
10    Collections.sort(personas); // esto no funcionará!!
11 }
```

PROBLEMA: desafortunadamente, el código anterior no compilará porque `Collections.sort(..)` sólo sabe ordenar listas si:

- los elementos son comparables (es decir, si implementan `Comparable`) ó,
- si se usa un método de comparación personalizado (`Comparator`).

Solución 1: implementando `Comparable`

En el caso de la lista con *Integer's* del primer ejemplo funcionaba, porque:



```

Main.java Integer.java ×
Author: Lee Boynton, Arthur van Hoff, Josh Bloch, Joseph D. Darcy

74 @jdk.internal.ValueBased
75 public final class Integer extends Number
76     implements Comparable<Integer>, Constable, ConstantDesc {
|     A constant holding the minimum value an int can have, -231.
81     @Native public static final int MIN_VALUE = 0x80000000;
82 }
```

pero nuestra clase *Persona* no implementa *Comparable*, por lo tanto, no funcionará. Para hacer que nuestra clase *Persona* tenga un orden definido (ordenar por *apellidos* y luego por el *nombre*), implementamos *Comparable<Persona>* y sobreescrivimos el método *compareTo(Persona p)*:

```
public class Persona implements Comparable<Persona> {
    private final String apellido;
    private final String nombre;

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

    public String getNombre() {
        return nombre;
    }

    public String getApellido() {
        return apellido;
    }

    public String toString() {
        return apellido + ", " + nombre;
    }

    @Override
    public boolean equals(Object o) {
        if (!(o instanceof Persona)) return false;
        Persona p = (Persona) o;
        return nombre.equals(p.nombre) && apellido.equals(p.apellido);
    }

    @Override
    public int hashCode() {
        return Objects.hash(nombre, apellido);
    }

    @Override
    public int compareTo(Persona otra) {

        // comparar primero por apellido
        int comparacionApellido = apellido.compareTo(otra.apellido);

        if (comparacionApellido != 0) { // si la comparación devuelve 0,
```

```

41         return comparacionApellido;
42     }
43
44     // si los apellidos son iguales, comparar por nombre
45     return nombre.compareTo(otra.nombre);
46
47 }
48 }
```

Como ves `elemento.compareTo(otro_elemento)` devolverá un número entero: si son iguales, devolverá un 0; si es mayor el de la izquierda (*this*) un número positivo; y si es mayor el de la derecha un número negativo.

Ahora, el *main* funciona correctamente:

```

1 public static void main(String[] args) {
2     List<Persona> personas = Arrays.asList(
3         new Persona("Juan", "Pérez"),
4         new Persona("Ana", "Gómez"),
5         new Persona("Carlos", "López"),
6         new Persona("Beatriz", "López"),
7         new Persona("Pedro", "Pérez")
8     );
9
10    Collections.sort(personas); // funciona correctamente
11    System.out.println(personas); // [Ana Gómez, Beatriz López, Carlos L
12
13 }
```

Solución 2: Usar un Comparator

Si no queremos o no podemos modificar la clase *Persona* (porque va a haber varios criterios de ordenación, por ejemplo), podemos proporcionar una clase aislada que implemente *Comparator<T>* y defina la forma de comparar los objetos reescribiendo el método *compare()*.

```
import java.util.Comparator;
```

```

4 public class PersonaComparator implements Comparator<Persona> {
5
6     @Override
7     public int compare(Persona p1, Persona p2) {
8
9         // comparar primero por apellido
10        int comparacionApellido = p1.getApellido().compareTo(p2.getApellido());
11
12        if (comparacionApellido != 0) {
13            return comparacionApellido;
14        }
15
16        // si los apellidos son iguales, comparar por nombre
17        return p1.getNombre().compareTo(p2.getNombre());
18    }
19}

```

Y en el *main*, pasaremos al método *sort()* un parámetro más de tipo *Comparator*.

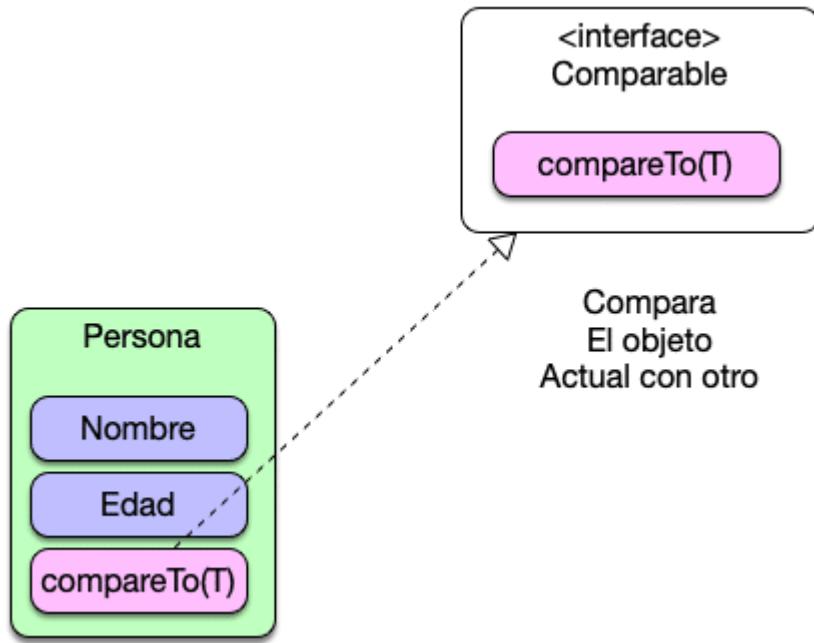
```

1 public static void main(String[] args) {
2     List<Persona> personas = Arrays.asList(
3         new Persona("Juan", "Pérez"),
4         new Persona("Ana", "Gómez"),
5         new Persona("Carlos", "López"),
6         new Persona("Beatriz", "López"),
7         new Persona("Pedro", "Pérez")
8     );
9
10    Collections.sort(personas, new PersonaComparator()); // ahora se pue
11    System.out.println(personas); // [Ana Gómez, Beatriz López, Carlos
12
13}

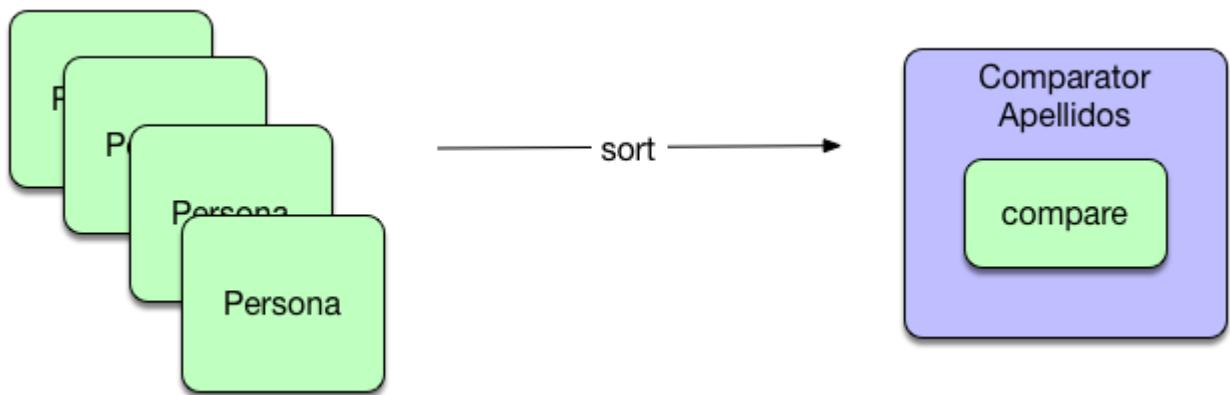
```

En resumen...

→ Usaremos *Comparable* sobreescribiendo *compareTo()* cuando queramos definir un orden natural en la clase y que *Collections.sort()* siempre se use ese criterio de comparación. Se implementa *Comparable* en la misma clase y se sobrescribe *compareTo()*.



→ Usaremos *Comparator* sobreescribiendo *compare()* cuando necesitemos varios criterios de ordenación sin modificar la clase original. Se usa una clase externa que implementa a *Comparator* y se pasa un objeto de ese tipo a *Collections.sort()* cada vez que necesitemos ordenar con ese criterio.



Ordenar descendente

En caso de que necesitemos ordenar al revés, una vez teniendo implementados *Comparable* o *Comparator* es tan fácil como realizar un *Collections.reverseOrder()* ó *PersonaComparator().reversed()* en *Collections.sort()*:

- Para Comparable:

```
1 | Collections.sort(personas, Collections.reverseOrder());
```

Esto funciona porque *reverseOrder()* invierte el criterio de comparación definido en *compareTo()* de la clase *Persona*.

- Para Comparator:

```
1 | Collections.sort(personas, new PersonaComparator().reversed());
```

La salida esperada en ambos casos será **[Pedro Pérez, Juan Pérez, Carlos López, Beatriz López, Ana Gómez]**

Ordenación de mapas

Los mapas (*HashMap*, *TreeMap*, etc.) no tienen un orden natural de por sí, pero se pueden ordenar según las claves o los valores utilizando *Comparable* o *Comparator*.

Un *TreeMap* ya ordena automáticamente sus claves:

```
1 public class EjemploTreeMap {  
2     public static void main(String[] args) {  
3  
4         Map<String, Integer> treeMap = new TreeMap<>(); // se ordena p  
5         treeMap.put("Juan", 30);  
6         treeMap.put("Ana", 25);  
7         treeMap.put("Carlos", 35);  
8  
9         System.out.println(treeMap); // {Ana=25, Carlos=35, Juan=30}  
10    }  
11 }  
12 }
```

Si las claves son *String*, *Integer*, etc., se ordenan por orden natural. Si no, deberemos implementar *Comparable*. Si necesitamos más de un criterio, crearemos una clase nueva que implemente *Comparator* con el que necesitemos y lo usaremos en el constructor del *TreeMap*:

```
1 Map<String, Integer> treeMap = new TreeMap<>(Comparator.reverseOrder())  
2 treeMap.put("Juan", 30);  
3 treeMap.put("Ana", 25);  
4 treeMap.put("Carlos", 35);  
5  
6 System.out.println(treeMap); // {Juan=30, Carlos=35, Ana=25} (ordenad
```

Ordenar un *HashMap* por valor usando *Comparator*

Un *HashMap* no mantiene el orden, así que convertimos sus entradas (*entrySet()*) en una lista, y luego la ordenamos con *Comparator*.

```
1 public class OrdenarHashMap {
2     public static void main(String[] args) {
3
4         Map<String, Integer> hashMap = new HashMap<>();
5         hashMap.put("Juan", 30);
6         hashMap.put("Ana", 25);
7         hashMap.put("Carlos", 35);
8
9         // convertir a lista
10        List<Map.Entry<String, Integer>> lista = new ArrayList<>(hashMa
11        lista.sort(Map.Entry.comparingByValue()); // ordena por valore
12
13        for (Map.Entry<String, Integer> entry : lista) {
14            System.out.println(entry.getKey() + " -> " + entry.getValue());
15        }
16
17        // Ana -> 25
18        // Juan -> 30
19        // Carlos -> 35
20    }
21 }
```

Y para ordenar al revés:

```
1 | lista.sort(Map.Entry.comparingByValue(Comparator.reverseOrder()));
```

7.3.3. Inmutabilidad en Java

Si nuestras clases tienen una lista (*List*, *Map*, *Set*, etc.), **debemos asegurarnos de devolver copias o versiones inmutables** para evitar modificaciones externas en caso de que no debamos permitirlas.

Por ejemplo:

```
1 | public class Grupo {  
2 |     private final List<String> miembros = new ArrayList<>();  
3 |  
4 |     public List<String> getMiembros() {  
5 |         return miembros; // peligroso! Se puede modificar desde fuera  
6 |     }  
7 | }
```

```
1 | Grupo g = new Grupo();  
2 | g.getMiembros().add("Juan"); // modifica la lista
```

devolviendo una lista inmutable con ***Collections.unmodifiableList()***.

```
1 | import java.util.*;  
2 |  
3 | final class Grupo {  
4 |     private final List<String> miembros;  
5 |  
6 |     public Grupo(List<String> miembros) {  
7 |         this.miembros = new ArrayList<>(miembros); // se almacena una  
8 |     }  
9 |  
10 |    public List<String> getMiembros() {  
11 |        return Collections.unmodifiableList(miembros); // se devuelve  
12 |    }  
13 | }
```

```
1 | Grupo g = new Grupo(Arrays.asList("Ana", "Luis"));
2 | g.getMiembros().add("Pedro"); // error: UnsupportedOperationException
```

A estas alturas estarás pensando... ¿si lo declaro como *final* cómo es posible que esté pudiendo manipular una lista?

Una lista declarada como *final* en Java puede ser modificada en cuanto a su contenido, pero no se puede reasignar a otra instancia. Es decir, *final* sólo evita que la variable apunte a otra instancia:

```
1 | final List<String> nombres = new ArrayList<>();
2 | nombres = new ArrayList<>(); // error: no se puede reasignar una lista
```

Pero sí podríamos hacer:

```
1 | public class Main {
2 |     public static void main(String[] args) {
3 |
4 |         final List<String> nombres = new ArrayList<>();
5 |
6 |         nombres.add("Juan"); // se puede modificar la lista
7 |         nombres.add("Ana");
8 |
9 |         System.out.println(nombres); // [Juan, Ana]
10 |
11 |         nombres.set(1, "Luis"); // se puede modificar un elemento
12 |         System.out.println(nombres); // [Juan, Luis]
13 |
14 |         nombres.remove("Juan"); // se puede eliminar elementos
15 |         System.out.println(nombres); // [Luis]
16 |
17 |     }
18 | }
```

¿Cómo hacer que la lista sea completamente inmutable?

Haciendo uso igualmente de `Collections.unmodifiableList()`, pero en el momento de la creación de la lista (`new`):

```
1 | List<String> nombres = Collections.unmodifiableList(new ArrayList<>(Arr
```

De esta forma no se podrán modificar los elementos, ni añadir o eliminar nuevos.



Obra publicada con Licencia Creative Commons Reconocimiento Compartir igual 4.0
<http://creativecommons.org/licenses/by-sa/4.0/>

PRÁCTICA: SERVICIO DE COMPRA ONLINE EN MERCADAW



MERCADAW

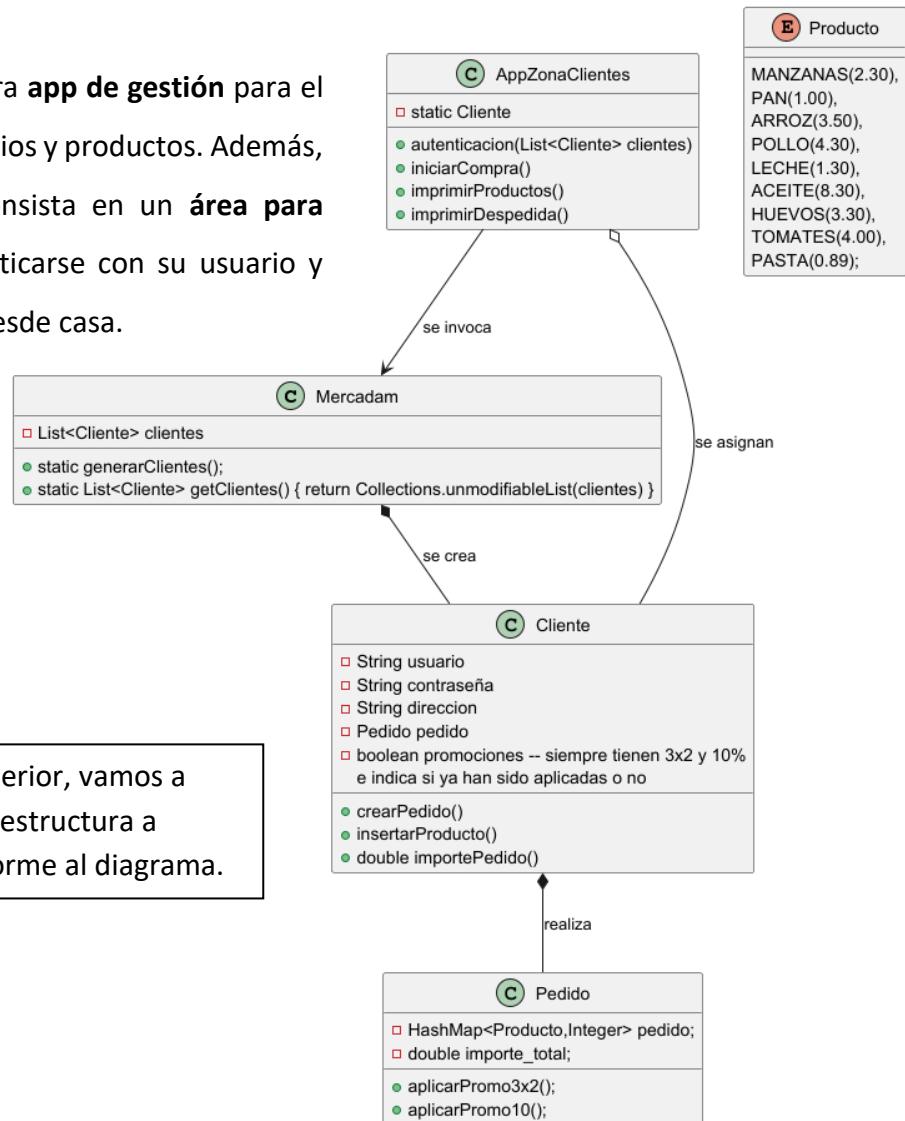
→ INTRODUCCIÓN

Tenemos nuevo supermercado en la ciudad. Puede que te suene el nombre, porque sus dueños, cabreados con las políticas de empresa del súper original, han querido montar su propia tienda.

Para no ser menos, tienen claro que quieren implementar la famosa compra online con su correspondiente reparto a domicilio. Como no tienen ni idea de informática, deciden acoger en prácticas a 4 estudiantes del instituto IES MUTXAMEL para que los asesoren un poco y construyan una app que, aunque de momento no sea bonita, implemente toda la lógica necesaria.

→ PROBLEMA A RESOLVER

El supermercado necesita una primera **app de gestión** para el personal, desde donde se crean usuarios y productos. Además, necesitan una segunda app que consista en un **área para clientes**, donde estos puedan autenticarse con su usuario y contraseña para realizar un pedido desde casa.



Teniendo en cuenta todo lo anterior, vamos a desarrollar un software cuya estructura a implementar se ha diseñado conforme al diagrama.

Para ir probando esta estructura, el programa principal **AppZonaClientes** debe iniciar creando una instancia de *Mercadaw* y generando a clientes aleatorios de prueba (`generarClientes()`). El constructor de

la clase *Mercadaw* no recibe nada y genera *usuario* y la *contraseña* de tamaño 8 con caracteres *random*.

Usa esto:

```
String caracteres = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789";
```

Pedido se iniciará a *null*, y *promociones* a *false*. La *dirección* siempre será “*Calle falsa, 123*”.

- *autenticacion(List<Cliente> clientes)*. Imprimirá y pedirá por pantalla lo siguiente:

```
*** COMPRA ONLINE DE MERCADAM ***

Usuario: patri
Constraseña: compra_patri
```

Cotejará los datos introducidos por el cliente contra la lista de clientes que recibe desde la clase *Mercadaw* para validar las credenciales. **Imprime la lista que recibes para no inventarte las pruebas.**

- En caso de que el *usuario* o *contraseña* no coincidan, se informará de que las credenciales no son correctas y el programa volverá a pedirlas. Tras **3 intentos**, el programa debe cerrarse mostrando un **error de autenticación**.

```
*** COMPRA ONLINE DE MERCADAM ***

Usuario: patri
Constraseña: compra_patri

Algo no coincide o no existe! Vuelve a intentarlo...

Usuario: patri
Constraseña: compra_patri2

Algo no coincide o no existe! Vuelve a intentarlo...

Usuario: patri
Constraseña: compra_patri3

ERROR DE AUTENTICACIÓN.
```

- En caso de que el usuario exista en la lista, se lo asignaremos a la variable de tipo *Cliente* (estático) que contiene la clase *AppZonaClientes* y llamaremos a *iniciarCompra()*.
- *iniciarCompra()* creará un nuevo pedido para el *Cliente*, inicializando su atributo *Pedido = new HashMap<Producto, Integer>*.

Además, llamará al método *imprimirProductos()* que mostrará el contenido del **enum**:

```

BIENVENID@ patri!

Añade productos a tu lista de la compra...

MANZANAS precio (2.30€),
PERAS precio (1.30€),
PAN precio (1.00€),
ARROZ precio (3.50€),
POLLO precio (4.30€),
LECHE precio (1.30€),
ACEITE precio (8.30€),
HUEVOS precio (3.30€),
TOMATES precio (4.00€),
PASTA precio (0.89€);

=====
Elige un producto:

```

- Recogeremos desde el programa principal la opción escogida por el usuario y llamaremos a ***insertarProducto(String producto)***. Este método añadirá el producto escogido al *Pedido*. En caso de que ya exista un producto del mismo tipo, incrementaremos la cantidad (+1ud).

CUIDADO: en caso de que el producto no exista, se debe mostrar un **ERROR e imprimir de nuevo la lista de productos disponibles**:

```

TOMATES precio (4.00€),
PASTA precio (0.89€);

=====
Elige un producto: MACARRONES

=====
El producto no existe! Elige otro.

Añade productos a tu carrito de la compra...

MANZANAS precio (2.30€),
PERAS precio (1.30€),
PAN precio (1.00€),
ARROZ precio (3.50€)

```

- Si todo va bien y el producto es correcto, se informa al cliente y se muestra un resumen del importe acumulado en el carrito (***importePedido()***):

```
TOMATES precio (4.00€),  
PASTA precio (0.89€);  
=====  
Elige un producto: PASTA  
=====  
Has añadido PASTA con un precio de 0.89€. Importe total del carrito: 0.89€. ¿Quieres añadir más productos a tu carrito de la compra? [S/N]:
```

Además, se pregunta al usuario si se quieren añadir más productos o finalizar. En caso de responder "S", se repite el proceso.

```
Has añadido PASTA con un precio de 0.89€. Importe total del carrito: 0.89€. ¿Quieres añadir más productos a tu carrito de la compra? [S/N]:  
S  
=====  
Añade productos a tu carrito de la compra...  
MANZANAS precio (2.30€),  
PERAS precio (1.30€),  
PAN precio (1.00€),  
ARROZ precio (3.50€),  
=====
```

```
LECHE precio (1.30€),  
ACEITE precio (8.30€),  
HUEVOS precio (3.30€),  
TOMATES precio (4.00€),  
PASTA precio (0.89€);  
=====
```

```
Elige un producto: TOMATES  
=====
```

```
Has añadido TOMATES con un precio de 4.00€. Importe total del carrito: 4.89€. ¿Quieres añadir más productos a tu carrito de la compra? [S/N]:
```

En caso de no querer añadir más productos, se debe actualizar el atributo **importe_total** del *Pedido* y **mostrar un resumen** de la compra realizada:

```
Has añadido TOMATES con un precio de 4.00€. Importe total del carrito: 4.89€. ¿Quieres añadir más productos a tu carrito de la compra? [S/N]:  
N  
=====  
RESUMEN DE TU CARRITO DE LA COMPRA:  
Productos:  
1 PASTA 0.89  
1 TOMATES 4.00  
3 PAN 1.00  
IMPORTE TOTAL: 7.89€
```

En este punto, mostraremos las siguientes opciones al cliente:

[1]. Aplicar promo.

[2]. Mostrar resumen ordenado por uds.

[X]. Terminar pedido.

- En caso de querer terminar, imprimiremos un mensaje de despedida dando las gracias e indicando la dirección del cliente.

```
=====
RESUMEN DE TU CARRITO DE LA COMPRA:

Productos:

1 PASTA  0.89
1 TOMATES 4.00
3 PAN 1.00

IMPORTE TOTAL: 7.89€

=====

¿QUÉ DESEA HACER?

[1]. Aplicar promo.
[2]. Mostrar resumen ordenado por uds.
[3]. Terminar pedido.

=====

Elige una opción: 3

=====

GRACIAS POR SU PEDIDO. Se lo mandaremos a la dirección Calle Falsa, 123.
```

- Si decidimos **aplicar promo**, deberemos comprobar que no se hayan aplicado ya al mismo cliente (promociones = *false*). Si ya las hemos aplicado, no faremos nada y mostraremos un mensaje para informar al cliente de que ya ha aplicado sus promos.

```
IMPORTE TOTAL: 7.89€
=====
¿QUÉ DESEA HACER?
[1]. Aplicar promo.
[2]. Mostrar resumen ordenado por uds.
[3]. Terminar pedido.
=====
Elige una opción: 1
=====
YA HAS APLICADO TUS PROMOS.
=====
RESUMEN DE TU CARRITO DE LA COMPRA:
| Productos ordenados por uds:
  3 PAN 1.00
  1 PASTA  0.89
```

En caso de que el cliente todavía no haya usado sus promociones, aplicaremos todas las que tenemos disponibles: **3x2 en productos y 10% de descuento**. Por lo tanto:

- Deberemos recorrernos nuestro pedido en busca de aquellos productos de los cuales existan 3 uds (o múltiplos de 3 uds) para recalcular el *importe_total* (sólo cobraremos 2).
- Al *importe_total* obtenido después de aplicar la promo anterior, deberemos aplicarle un 10% de descuento más.
- Modificaremos el atributo *promociones* del cliente a **true**.

Finalmente, imprimiremos de nuevo el resumen del pedido con las promos aplicadas y el importe total actualizado:

```
=====
¿QUÉ DESEA HACER?  
[1]. Aplicar promo.  
[2]. Mostrar resumen ordenado por uds.  
[3]. Terminar pedido.  

=====
Elige una opción: 1  

=====
PROMO 3X2 y 10% APLICADAS.  

=====
RESUMEN DE TU CARRITO DE LA COMPRA:  

Productos:  

1 PASTA 0.89  

1 TOMATES 4.00  

3 PAN 1.00  

IMPORTE TOTAL: 6.20€  

=====
```

- Si decidimos usar la opción 2, deberemos mostrar el resumen del pedido ordenado por uds descendentemente. Por ejemplo:

```
[1]. Aplicar promo.  
[2]. Mostrar resumen ordenado por uds.  
[3]. Terminar pedido.  

=====
Elige una opción: 2  

=====
RESUMEN DE TU CARRITO DE LA COMPRA:  

Productos ordenados por uds:  

3 PAN 1.00  

1 PASTA 0.89  

1 TOMATES 4.00  

IMPORTE TOTAL: 6.20€  

=====
```

BONUS. Implementa una funcionalidad añadida para que el programa permita eliminar productos del carrito de la compra. En caso de que algún tipo de producto quede con 0 uds, debe eliminarse completamente de la lista (mapa).

¿QUÉ DESEA HACER?

- [1]. Aplicar promo.
 - [2]. Mostrar resumen ordenado por uds.
 - [3]. Eliminar productos.
 - [X]. Terminar pedido.
- =====

Elige una opción:

=====

Realiza un programa en *Java* que implemente la lógica de la aplicación dada, usando *POO* y estructuras dinámicas de datos.

→ REALIZACIÓN DE LA PRÁCTICA

Sigue los siguientes pasos para realizar la práctica. **¡Ve guardando tu trabajo de vez en cuando para evitar que se borre el avance si se cierra el editor de textos u ocurre cualquier problema en tu equipo!**

1. Programa en Java la aplicación requerida
2. Sube un vídeo ejecutando tu aplicación, replicando pruebas y explicando todos los comportamientos que hayas implementado.



ENTREGA

REALIZA UN INFORME EN PDF CON LA INFO GENERADA Y LOS PASOS SEGUIDOS PARA REALIZAR ESTA PRÁCTICA. EXPLICA TU CÓDIGO. SÚBELO TODO A LA TAREA DE AULES DISPONIBLE.

ADEMÁS, PEGA LA URL DE TU PROYECTO EN GITHUB.

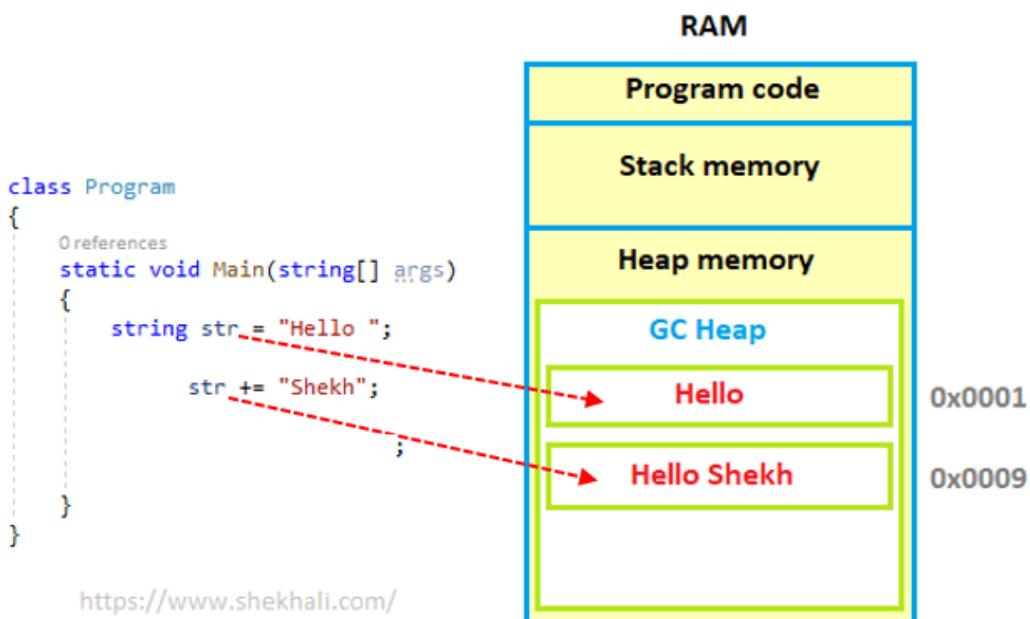
BONUS. MANIPULACIÓN DE CADENAS DE TEXTO CON *StringBuilder*



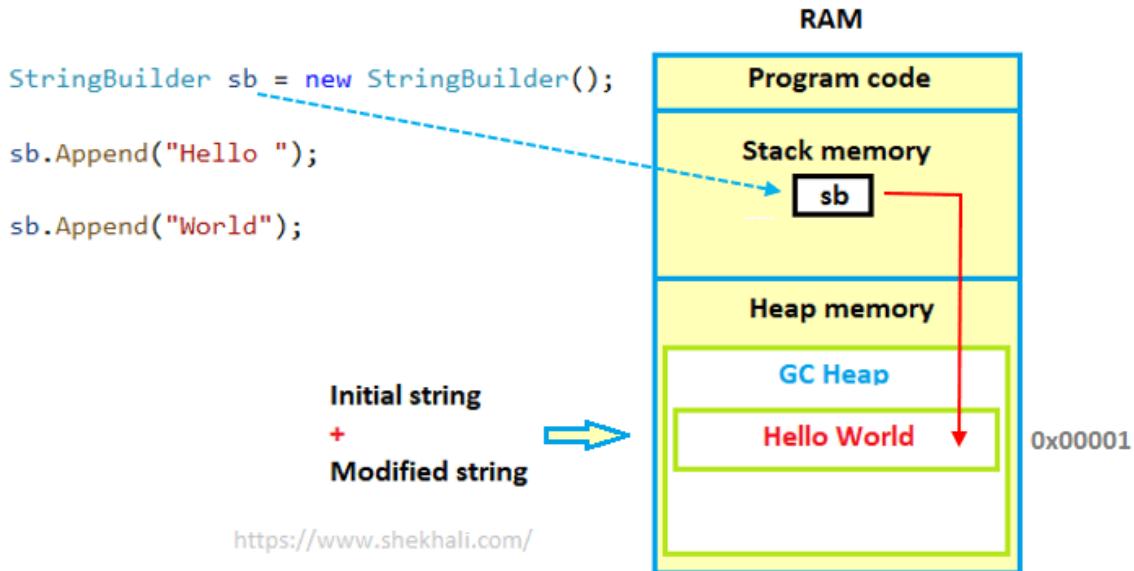
➔ *String* vs *StringBuilder*

Un *StringBuilder* es una clase en algunos lenguajes de programación, como *Java* y *C#*, que se usa para manipular cadenas de texto de manera eficiente. A diferencia de los objetos *String*, que son inmutables (no pueden modificarse después de su creación), un *StringBuilder* permite modificar su contenido sin necesidad de crear nuevos objetos en memoria.

Cada vez que concatenamos un *String* con otro mediante el símbolo suma (+), lo que pasa realmente es que se crea un nuevo objeto en memoria, lo que puede ser ineficiente en términos de rendimiento si se hacen muchas modificaciones.



StringBuilder, en cambio, usa un buffer dinámico que permite modificar el texto sin crear nuevos objetos constantemente.



→ Métodos de *StringBuilder* para manipular cadenas

- **append(String s)**: inserta texto al final.
- **insert(int index, String s)**: inserta texto en una posición específica.
- **replace(int start, int end, String s)**: reemplaza una parte de la cadena.
- **delete(int start, int end)**: elimina caracteres dentro de un rango.
- **reverse()**: invierte la cadena.
- **toString()**: convierte el *StringBuilder* en un *String* para poder imprimirla.

Ejemplos de uso:

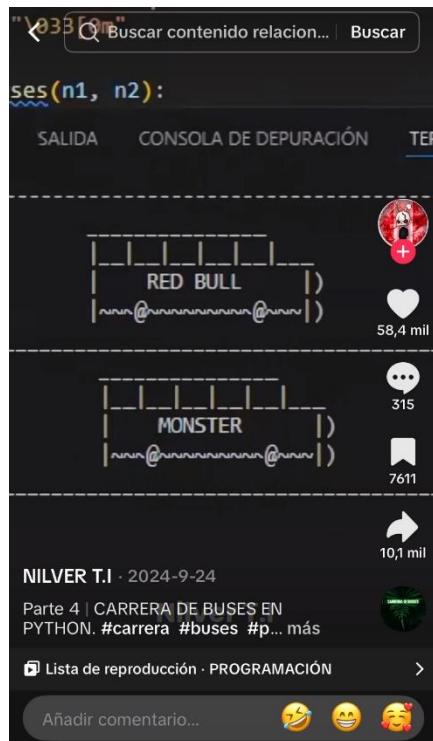
```
public class EjemploStringBuilder {
    public static void main(String[] args) {

        StringBuilder sb = new StringBuilder("Hola");
        sb.append(" Mundo"); // inserta " Mundo" al final
        sb.insert(5, " querido"); // inserta " querido" en la posición 5 (delante de el que ya estaba en esa posición que es Mundo)
        sb.replace(5, 13, "gran "); // reemplaza por "gran" todo lo que haya desde la posición 5 a la 13
        sb.delete(0, 5); // borra los primeros 5 caracteres: "Hola "

        System.out.println(sb.toString()); // imprime "granoMundo"

        sb.reverse();
        System.out.println(sb.toString()); // imprime "odnuM narg"
    }
}
```

→ La famosa carrera de autobuses de TikTok



A continuación, se proporciona el código *Java* que implementa un *StringBuilder* para generar un autobús que avanza por la pantalla:

```
public class Bus {  
  
    public static final int TAM = 97;  
  
    public static void main(String[] args) throws InterruptedException {  
  
        int a = 0; // POSICIÓN INICIAL DEL AUTOBÚS  
  
        System.out.println("\n<<<<<< AUTOBUSITO >>>>>>>");  
        Thread.sleep(3000);  
  
        while (a < TAM) {  
  
            a++; // avanzamos  
  
            limpiarPantalla();  
  
            if (a<TAM) {  
                System.out.println(dibujarCarrera(a));  
                Thread.sleep(70);  
            }  
        }  
  
        System.out.println("\033[32m" + "EL AUTOBUSITO HA LLEGADO A SU DESTINO!!" +  
"\033[0m");  
  
    }  
  
    public static String dibujarCarrera(int n1) {
```

```

        StringBuilder sb = new StringBuilder();

        sb.append("-".repeat(117)).append("\n");
        sb.append(" ".repeat(n1)).append("_____").append(" ".repeat(100 - n1)).append("|\\n");
        sb.append(" ".repeat(n1)).append("|__|__|__|__|__|__").append(" ".repeat(TAM - n1)).append("|\\n");
        sb.append(" ".repeat(n1)).append("|    IES MUTXAMEL |").append(" ".repeat(TAM - n1)).append("|\\n");
        sb.append(" ".repeat(n1)).append("|~~~@~~~~~@~~~|").append(" ".repeat(TAM - n1)).append("|\\n");
        sb.append("_".repeat(117));

        return sb.toString();
    }

    public static void limpiarPantalla() {
        try {
            new ProcessBuilder("cmd", "/c", "cls").inheritIO().start().waitFor();
        } catch (Exception e) {
            System.out.print("\033[H\033[2J");
            System.out.flush();
        }
    }
}

```

Explicación:

- *a* representa la posición del autobús.
- Se usa **Thread.sleep(3000)** para pausar 3 segundos, creando un efecto de animación.
- Mientras el autobús no llegue al final del tablero (**while (a < TAM)**), la ruta continúa avanzando.
- Se llama al método **limpiarPantalla()**, que borra la pantalla de la consola para mostrar la nueva posición del autobús.
- Sólo dibuja la ruta (**if (a < TAM)**) si el autobús no ha llegado aún a su destino (**TAM**).
- El método **dibujarCarrera(int n1)** dibuja la pista y la posición del autobús.
 - « **sb.append("-".repeat(117)).append("\n")** dibuja una línea superior de 117 guiones.
 - « A la forma del autobús se añaden líneas con espacios (**" ".repeat(n1)**) para posicionar el autobús según su avance.
 - « **sb.append("_".repeat(117))** dibuja la línea inferior con 117 barra bajas.
- Se imprime que ha llegado a su destino con color verde (**\033[32m**).
- **\033[0m** restablece el color de la consola.

ACTIVIDAD. Adapta el código dado para que se emule una carrera con dos autobuses como en el vídeo viral de *TikTok*.

```
C:\Users\patri\Proyectos\ proyecto_poo\src\main\java\org\example>java CarreraBuses.java
<<<<<< CARRERA DE AUTOBUSES >>>>>>
MONNEGRE FC vs FC MUTXAMEL
FIGHT!
```

```
C:\ Símbolo del sistema - java CarreraBuses.java
[...]
|---|---|---|---|---|---|
| MONNEGRE FC | )
|---|---|---|---|---|---|
|---|---|---|---|---|---|
| FC MUTXAMEL | )
|---|---|---|---|---|---|
[...]
```

```
C:\ Símbolo del sistema
¡¡GANADOR: FC MUTXAMEL!!
C:\Users\patri\Proyectos\ proyecto_poo\src\main\java\org\example>
```



ENTREGA

PEGA EN AULES LA URL DE TU CLASE JAVA DISPONIBLE EN GITHUB.

EXPLICA TU CÓDIGO EN UN FICHERO MARKDOWN Y SÚBELO AL MISMO DIRECTORIO DE GITHUB QUE LA CLASE JAVA.

Programación

EXAMEN TEMA 7 – ESTRUCTURAS DINÁMICAS DE DATOS

(03/04/2025)

1. (0,5p) ¿A partir de qué interfaces principales se implementan todo tipo de listas, pilas, colas, conjuntos y mapas/diccionarios?
2. (1,5p) ¿En qué consisten las estructuras *FIFO*? Indica el nombre de la clase que las implementa en *Java* y menciona sus métodos principales. Pon ejemplos de uso en aplicaciones reales y ayúdate de dibujos si lo necesitas.
3. (1p) El siguiente código *Java* contiene lógica de implementación de una pila:

```
public static int evaluar(String expresion) {  
    Stack<Integer> pila = new Stack<>();  
    for (String c : expresion.split(" ")) {  
        if (c.matches("\\d+")) {  
            pila.push(Integer.parseInt(c));  
        } else {  
            int b = pila.pop();  
            int a = pila.pop();  
            switch (c) {  
                case "-": pila.push(a - b); break;  
                case "*": pila.push(a * b); break;  
            }  
        }  
    }  
    return pila.pop();  
}  
  
public static void main(String[] args) {  
    String expresion = "4 2 - 5 *";  
    System.out.println(evaluar(expresion));  
}
```

Dibuja cómo va cambiando la pila mientras se ejecuta el programa e indica qué valor numérico retornará el método **evaluar()**.

4. (1p) ¿Qué diferencia hay entre una estructura *Queue* y *Deque*? Indica cómo cambian los métodos disponibles de una a otra.
5. (1p) ¿Por qué es importante implementar los métodos **.hashCode()** y **.equals()** en clases personalizadas? Relacionalo con las estructuras de tipo *Set* y pon ejemplos.

6. (1p) Indica la salida que imprime el siguiente programa:

```
public static void main(String[] args) {  
    String[] palabras = {"banana", "manzana", "pera", "banana"};  
    Set<String> hashSet = new HashSet<>();  
    Set<String> treeSet = new TreeSet<>();  
    Set<String> linkedHashSet = new LinkedHashSet<>();  
    for (String palabra : palabras) {  
        hashSet.add(palabra);  
        treeSet.add(palabra);  
        linkedHashSet.add(palabra);  
    }  
    System.out.println("HashSet: " + hashSet);  
    System.out.println("TreeSet: " + treeSet);  
    System.out.println("LinkedHashSet: " + linkedHashSet);  
}
```

7. (1p) Identifica los elementos característicos de los mapas y relaciona los con el funcionamiento de las tablas Hash. Explica qué ocurre cuando hay una colisión y cómo lo solucionan los lenguajes de programación.

8. (2p) ¿Qué estructura de datos será más eficiente usar en estos casos? Explica por qué.

- a) Contar la cantidad de veces que aparece cada palabra en un texto grande.
- b) Buscar rápidamente si un usuario está registrado.
- c) Almacenar un historial de navegación web manteniendo el orden de acceso.
- d) Implementar la función “deshacer” (*Ctrl + Z*) en un editor de texto

9. (0,5p) ¿Qué resultado devolverá el siguiente código *Java* para la lista dada?

```
public static void main(String[] args) {  
    ArrayList<String> nombres = new ArrayList<>(Arrays.asList("Patri", "Pedro"));  
    Iterator<String> it = nombres.iterator();  
    while (it.hasNext()) {  
        it.remove();  
    }  
}
```

10. (0,5p) ¿Cuándo es más eficiente usar *StringBuilder* respecto a *String*?

Programación

EXAMEN PRÁCTICO TEMA 7 – ESTRUCTURAS DINÁMICAS DE DATOS

(04/04/2025)



LEE ATENTAMENTE LAS SIGUIENTES INSTRUCCIONES ANTES DE EMPEZAR:



- **Recopila en un documento de texto las evidencias de todo el examen. Guárdalo de vez en cuando para no perder el avance de tu trabajo.**
- Cuando termines, **pásalo a PDF y sube el documento creado a la entrega de AULES.**

PARTE 1: Configuración del entorno (0,5p)

1. Crea un nuevo repositorio llamado “EXAMEN_UD7_[nombre]” desde *SourceTree*. El repositorio debe crearse en local y tener su espejo en remoto, por lo tanto, sincronízalo con *GitHub*.

Pega a continuación la URL a tu nuevo repositorio de GitHub:

2. Crea un nuevo proyecto Java (*Maven*) con *IntelliJ* -o el IDE que utilices- dentro del repositorio que acabas de crear. Llámalo “EXAMEN UD7”.
3. Crea en el proyecto dos paquetes nuevos llamados “chatbot” y “excursion” para ir añadiendo las clases correspondientes al problema 1 y el problema 2.

Sincroniza los cambios en tu repositorio remoto.

PARTE 2: Resolución de problemas

Programa en *Java* la solución a los siguientes problemas. Usa el proyecto que te acabas de crear en el apartado anterior.

Si no has conseguido crearlo correctamente, utiliza alguno de los proyectos que ya tenías para los ejercicios de clase y pega la URL de GitHub del repositorio al que vas a subir los cambios.

PROBLEMA 1 (3,5p). Simula un *Chatbot* básico que interactúe con el usuario y responda a preguntas comunes.

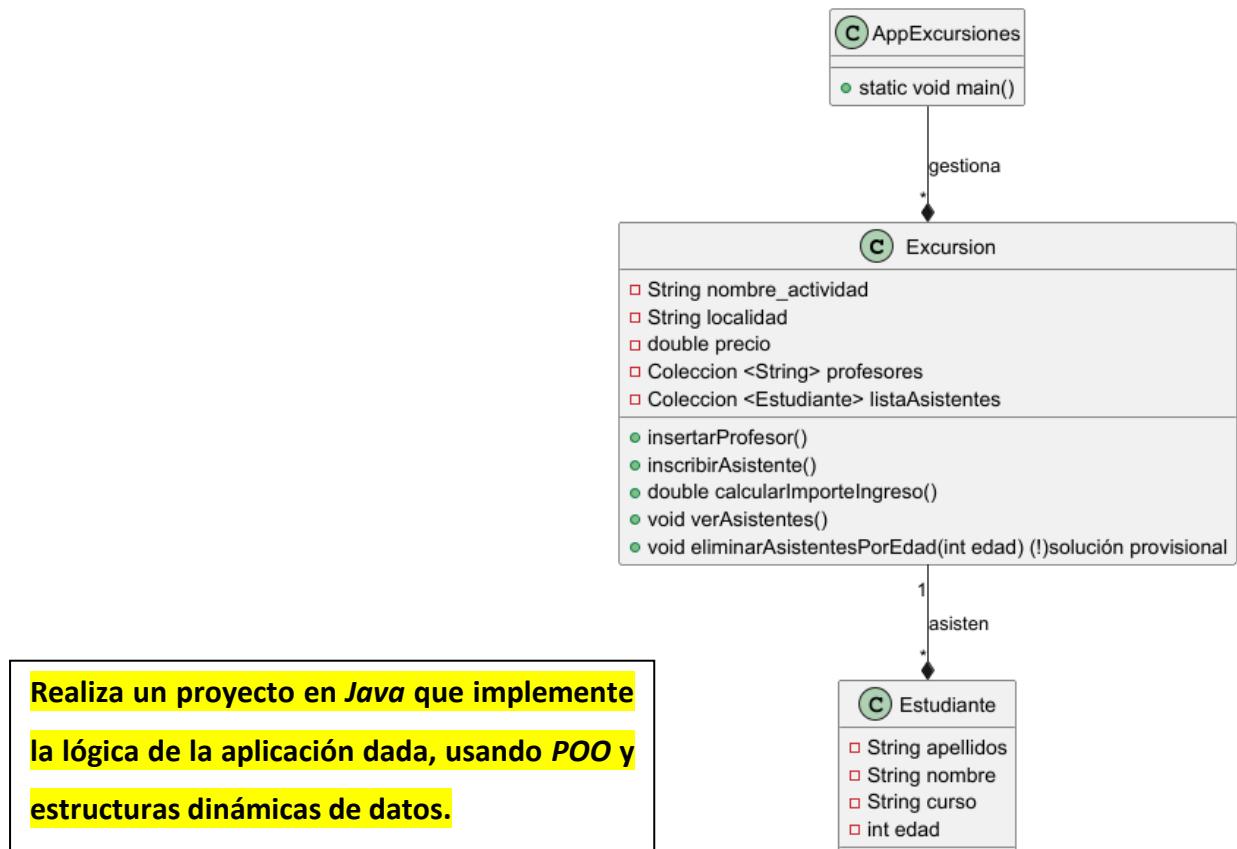
Ejemplo de comportamiento:

```
"C:\Program Files\Java\jdk-23\bin\java.exe" "-javaagent:C:\Program Files\JetBrai
Chatbot: Bienvenido! Escribe <salir> cuando quieras acabar la conversación...
TÚ: hola
Chatbot: Hola, ¿En qué te puedo ayudar?
TÚ: ¿cómo estás?
Chatbot: ¡Estoy listo para ayudarte!
TÚ: de lujo
Chatbot: No te entiendo :(
TÚ: Hola
Chatbot: Hola, ¿En qué te puedo ayudar?
TÚ: adiós
Chatbot: ¡Hasta luego!
TÚ: gracias
Chatbot: ¡De nada! Que tengas un buen día.
TÚ: me voy
Chatbot: No te entiendo :(
TÚ: salir
Chatbot: ¡Adiós!

Process finished with exit code 0
```

PROBLEMA 2 (6p). App de gestión de excursiones para el IES MUTXAMEL.

El centro ha decidido implementar un sistema para controlar al alumnado que va a una actividad extraescolar. La app la ha realizado un grupo de 1º DAM como parte del módulo *Proyecto Intermodular*, y la solución de diseño que han presentado es la siguiente:



Condiciones para la construcción de las clases y otras cosas a tener en cuenta

- (1p) Se debe controlar que no se inserten estudiantes duplicados en la lista de asistentes de la clase **Excursion**. Se considera que un estudiante es igual a otro si tiene los mismos **apellidos, nombre y curso**.
- (1p) El método **verAsistentes()** imprime la lista de asistentes, ordenada por *curso* y *apellidos*.
- (0,5p) Para ayudar al profesorado a calcular el importe que le deben ingresar a la empresa que organiza la excursión, nos han pedido implementar una funcionalidad **calcularImporteIngreso()** que devuelva la cantidad automáticamente en función del número de asistentes inscritos.
- (1p) Mientras se desarrolla la app, se recibe un aviso de un problema para una determinada actividad donde ya había inscritos bastantes estudiantes, ya que **han de eliminarse de la lista todos aquellos asistentes que todavía no hayan cumplido los 16 años** por motivos legales. A los desarrolladores se les ocurre, de forma provisional, añadir una funcionalidad **eliminarAsistentesPorEdad()** que recorra la lista de inscritos y vaya eliminando todos aquellos cuya edad sea menor a 16 años.
- (0,5p) De momento, nuestros compañeros de 1º DAM no tienen una app funcional con menús, pero se “apañan” para ir probando las funcionalidades implementadas con el siguiente código:

```
public class AppExcusiones {  
  
    public static void main(String[] args) {  
  
        System.out.println("***** APP EXCURSIONES *****");  
  
        Excusion terra_mitica = new Excusion("Terra Mítica", "Benidorm", 30.0);  
  
        terra_mitica.insertarProfesor();  
  
        terra_mitica.inscribirAsistente();  
        terra_mitica.inscribirAsistente();  
        terra_mitica.inscribirAsistente();  
        terra_mitica.inscribirAsistente();  
  
        terra_mitica.verAsistentes();  
  
        terra_mitica.eliminarAsistentesPorEdad(16);  
        System.out.println("\nDespués de eliminar a los menores de 16 años: ");  
        terra_mitica.verAsistentes();  
  
        System.out.println("\nImporte a ingresar para la actividad " +  
terra_mitica.getNombre_actividad() + " en " + terra_mitica.getLocalidad() + ":"  
" " + terra_mitica.calcularImporteIngreso() + " €.");  
    }  
}
```

1. (1p) ¿Qué solución propondrías sobre el diseño de clases de la app para evitar tener que lanzar el método *eliminarAsistentesPorEdad()* cada vez que tengamos una actividad limitada por edad?

2. (1p) Pruebas.

Lanza tu programa y explica las pruebas que vas haciendo conforme vas implementando código. Toma estas como referencia:

- a) Probar camino feliz (crear una excursión junto a sus respectivos profesores e insertar asistentes). Comprueba que se ha creado todo bien.
- b) Probar que no se insertan 2 estudiantes que se consideran iguales en la lista de asistentes.
- c) Probar que se eliminan correctamente de la lista de asistentes todos aquellos estudiantes que no alcancen la edad mínima para realizar una actividad.
- d) Probar que se ordena correctamente la lista de asistentes por *apellido y nombre*.
- e) Probar que se ordena correctamente la lista de asistentes por *curso*.

Ejecución de ejemplo

```
"C:\Program Files\Java\jdk-23\bin\java.exe" "-javaagent:C:\Program Files\JetBr
***** APP EXCURSIONES *****

Introduce el nombre del profesor/a para la excursión en Benidorm
Patricia
Profesor añadido correctamente a la excursión Terra Mítica

Creando estudiante...
Inserta los apellidos:
Boix Bernabeu
Inserta el nombre:
Luis
Inserta el curso:
1ºESO
Inserta su edad:
13
Añadido correctamente el estudiante Luis Boix Bernabeu del curso 1ºESO
```

```
Creando estudiante...
Inserta los apellidos:
Hernández Pastor
Inserta el nombre:
Andrea
Inserta el curso:
2ºESO
Inserta su edad:
14
```

Añadido correctamente el estudiante Andrea Hernández Pastor del curso 2ºESO

```
Creando estudiante...
```

```
Inserta los apellidos:
```

```
Hernández Pastor
```

```
Inserta el nombre:
```

```
Andrea
```

```
Inserta el curso:
```

```
2ºESO
```

```
Inserta su edad:
```

```
14
```

El estudiante ya existe en la lista de asistentes a la actividad Terra Mítica

```
Creando estudiante...
```

```
Inserta los apellidos:
```

```
Hernández Cuenca
```

```
Inserta el nombre:
```

```
Guillermo
```

```
Inserta el curso:
```

```
2ºESO
```

```
Inserta su edad:
```

```
16
```

Añadido correctamente el estudiante Guillermo Hernández Cuenca del curso 2ºESO

Lista de inscritos para la actividad Terra Mítica:

- [1]. 1ºESO - Boix Bernabeu, Luis (13 años).
- [2]. 2ºESO - Hernández Cuenca, Guillermo (16 años).
- [3]. 2ºESO - Hernández Pastor, Andrea (14 años).

Después de eliminar a los menores de 16 años:

Lista de inscritos para la actividad Terra Mítica:

- [1]. 2ºESO - Hernández Cuenca, Guillermo (16 años).

Importe a ingresar para la actividad Terra Mítica en Benidorm: 30.0 €.

7_4_lambdas_streams

7.4. Programación funcional

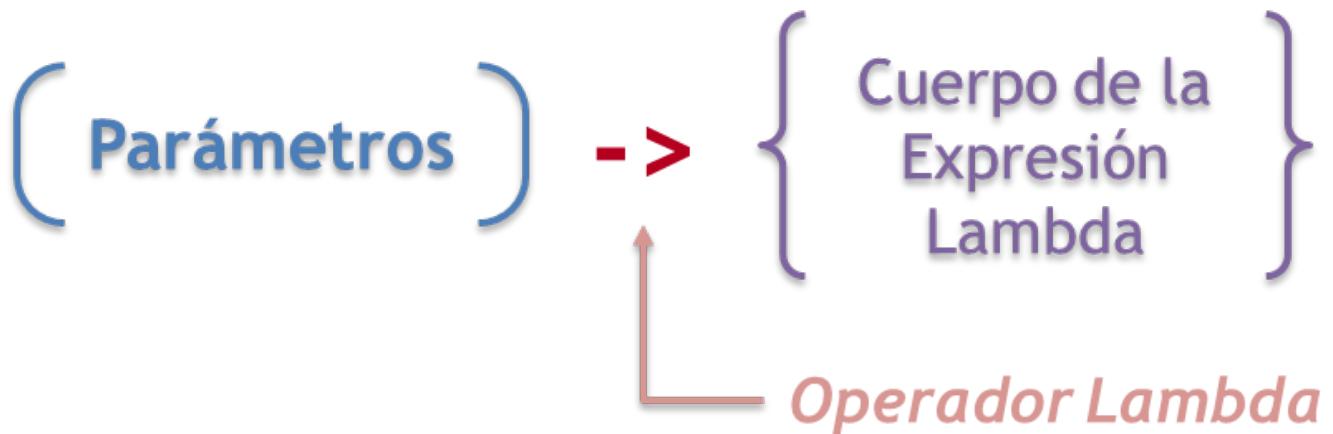
La **programación funcional** es un paradigma que trata la computación como la evaluación de funciones matemáticas y evita cambiar el estado o modificar datos (**inmutabilidad**), promoviendo código más declarativo, conciso y mantenable.

Desde Java 8, se introduce programación funcional principalmente a través de:

- **Expresiones Lambda:** funciones anónimas que se pueden pasar como argumentos. Se crearon como solución fácil a la implementación de clases anónimas para interfaces que solamente tengan un método abstracto.
- **Streams:** una forma funcional de procesar colecciones de datos (listas, arrays, etc.).

7.4.1. Funciones o expresiones lambda

Una expresión *lambda* es un bloque corto de código que toma parámetros y devuelve un valor.



La expresión *lambda* más simple contiene un solo parámetro y una expresión:

parámetro -> expresión

Para usar más de un parámetro, los envolvemos entre paréntesis:

(parámetro1, parámetro2) -> expresión

Para realizar operaciones más complejas, se puede usar un bloque de código con llaves:

(parámetro1, parámetro2) -> {bloque de código}

Y si la expresión *lambda* necesita devolver un valor, entonces el bloque de código debe tener una declaración *return*:

(parámetro1, parámetro2) -> {

bloque de código

return elemento_devuelto;

}

Para empezar a implementar ejemplos concretos con las expresiones anteriores, primero deberemos implementar una **interfaz funcional**.

Una interfaz funcional es una interfaz que solamente tiene un método abstracto, como podría ser la siguiente:

```
1  @FunctionalInterface
2  public interface Ejecutor {
3
4      void ejecutar();
5
6  }
```

Una vez que tenemos nuestra interfaz funcional, lo que podemos hacer simplemente con lo que sabemos es crear una clase anónima:

```
1  public class Main {
2      public static void main(String[] args) {
3
4          Ejecutor ej = new Ejecutor(){
5              @Override
6              public void ejecutar() {
7                  System.out.println("Hola desde una clase anónima.");
8              }
9          };
10     }
11 }
12 }
```

El problema de esto es que tenemos bastante código para solamente traer un método de la interfaz. Sin embargo, lo que podríamos hacer para acortarlo es crear directamente una expresión *lambda*:

```
1  public class Main {
2      public static void main(String[] args) {
3
4          Ejecutor ej = new Ejecutor(){
5              @Override
6              public void ejecutar() {
7                  System.out.println("Hola desde una clase anónima.");
8              }
9          };
10
11         Ejecutor lambda = () -> {
12             System.out.println("Hola desde una lambda");
13         };
14     }
15 }
```

```
15     ej.ejecutar();
16     lambda.ejecutar();
17 }
18 }
```

```
"C:\Program Files\Java\jdk-23\bin\jav
Hola desde una clase anónima.
Hola desde una lambda

Process finished with exit code 0
|
```

En este caso, como solamente realizamos una instrucción `println` no serían necesarias las **llaves**, por lo que nuestro código quedaría todavía más simplificado desde la *lambda*:

```
1 | Ejecutor lambda = () -> System.out.println("Hola desde una lambda");
```

Para probar las *lambdas* cuando tenemos **métodos abstractos con parámetros**, vamos a modificar el método `ejecutar()` de la interfaz para añadirle un parámetro:

```
1 | @FunctionalInterface
2 | public interface Ejecutor {
3 |
4 |     void ejecutar(String param1);
5 |
6 | }
```

La clase anónima y la *lambda* se empezarán a quejar tras el cambio, ya que también deben reflejar un parámetro en la llamada a `ejecutar()`. Las adaptamos:

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

        Ejecutor ej = new Ejecutor(){
            @Override
            public void ejecutar(String param1) {
                System.out.println("Hola desde " + param1);
            }
        };
    }
}
```

```
10    };
11
12    Ejecutor lambda = (param1) -> System.out.println("Hola desde ");
13
14    ej.ejecutar("una clase anónima");
15    lambda.ejecutar("una lambda");
16
17 }
```

```
"C:\Program Files\Java\jdk-23\bin\java.exe"
Hola desde una clase anónima
Hola desde una lambda

Process finished with exit code 0
```

También podríamos modificar el método `ejecutar()` de nuestra interfaz para retornar algún tipo de parámetro, por ejemplo un *String*:

```
1 @FunctionalInterface
2 public interface Ejecutor {
3
4     String ejecutar(String param1);
5
6 }
```

Una vez más, nuestra clase anónima y *lambda* se quejarán tal y como están implementadas, ya que deben adaptarse al nuevo cambio. Por lo tanto, las adaptamos para retornar un tipo *String*:

```
1 public class Main {
2     public static void main(String[] args) {
3
4         Ejecutor ej = new Ejecutor(){
5             @Override
6             public String ejecutar(String param1) {
7                 System.out.println("Hola desde " + param1);
8                 return param1.toUpperCase();
9             }
10        };
11    }
```

```
12     Ejecutor lambda = (param1) -> {
13         System.out.println("Hola desde " + param1);
14         return param1.toUpperCase();
15     };
16
17     String resultadoAnonima = ej.ejecutar("una clase anónima");
18     String resultadoLambda = lambda.ejecutar("una lambda");
19     System.out.println(resultadoAnonima);
20     System.out.println(resultadoLambda);
21
22 }
23 }
```

```
"C:\Program Files\Java\jdk-23\bin\java.exe
Hola desde una clase anónima
Hola desde una lambda
UNA CLASE ANÓNIMA
UNA LAMBDA

Process finished with exit code 0
```



Ejercicio 1

Modifica la interfaz *Ejecutor* para que el método *ejecutar()* reciba un parámetro de tipo *int* y retorne un *boolean*. Este método debe devolver *true* si el número es mayor a 10, o *false* en caso contrario.

Implementa una expresión *lambda* con un bloque de código que haga esta comprobación. En el método *main*, prueba con varios números (por ejemplo, 15 y 5) y muestra los resultados.

```
"C:\Program Files\Java\jdk-23\bin\java.exe
Comparando para 15:
true

Process finished with exit code 0
```



Ejercicio 2

Crea una nueva interfaz funcional llamada **VerificadorVoto** con un único método abstracto **boolean puedeVotar(String nombre, int edad)**.

Implementa una expresión *lambda* para que el método *puedeVotar()* retorne *true* si la persona tiene 18 años o más, y *false* si es menor de 18 años.

```
"C:\Program Files\Java\jdk-23\bin\java.exe"
Probando Maria con 14 años... false

Process finished with exit code 0
```

Vale... ¿y con objetos?

Vamos a coger el ejemplo del ejercicio anterior, donde la función *lambda* devuelve *true* o *false* en función de la edad de la persona que recibe por parámetro.

Crearemos una clase *Persona* con los atributos *nombre (String)* y *edad (int)*:

```
1  public class Persona {
2
3      private String nombre;
4      private int edad;
5
6      public Persona(String nombre, int edad) {
7          this.nombre = nombre;
8          this.edad = edad;
9      }
10
11     public String getNombre() {
12         return nombre;
13     }
14
15     public int getEdad() {
16         return edad;
17     }
18
19 }
```

Modificaremos la interfaz funcional llamada **VerificadorVoto** con un único método abstracto **boolean puedeVotar(Persona persona)**, que retorne *true* si la persona tiene 18 años o más.

```
1  @FunctionalInterface  
2  interface VerificadorVoto {  
3      boolean puedeVotar(Persona persona);  
4  }
```

Desde una clase *main*, crearemos una lista de personas con diferentes edades:

```
1  public class Main {  
2      public static void main(String[] args) {  
3  
4          List<Persona> personas = new ArrayList<>();  
5          personas.add(new Persona("Juan", 16));  
6          personas.add(new Persona("Ana", 18));  
7          personas.add(new Persona("Pedro", 25));  
8          personas.add(new Persona("María", 17));  
9          personas.add(new Persona("Luis", 30));  
10     }  
11  }
```

Usaremos una expresión *lambda* para verificar si cada persona puede votar y almacenaremos los nombres de las personas que sí pueden votar en una nueva lista:

```
public class Main {  
    public static void main(String[] args) {  
  
        List<Persona> personas = new ArrayList<>();  
        personas.add(new Persona("Juan", 16));  
        personas.add(new Persona("Ana", 18));  
        personas.add(new Persona("Pedro", 25));  
        personas.add(new Persona("María", 17));  
        personas.add(new Persona("Luis", 30));  
  
        // lambda que verifica si una persona puede votar (18 años o más)  
        VerificadorVoto verificador = persona -> persona.getEdad() >= 18;  
  
        // lista para almacenar los nombres de las personas que pueden votar  
        List<String> personasQuePuedenVotar = new ArrayList<>();  
  
        // verificando si cada persona puede votar usando la lambda  
        for (Persona persona : personas) {
```

```

19         if (verificador.puedeVotar(persona)) {
20             personasQuePuedenVotar.add(persona.getNombre());
21         }
22     }
23 }
24 }
25 }
```

(*)**FÍJATE.** En Java, cuando una *lambda* tiene una expresión simple (es decir, sólo una línea de código que se evalúa y retorna directamente), no es necesario escribir explícitamente la palabra *return*: Java lo hace solo. Sin embargo, si la *lambda* tuviera un bloque de código más complejo (con llaves {}), entonces sí es necesario usar *return* para devolver el valor.

Finalmente, imprimimos los nombres de las personas que pueden votar:

```

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

        List<Persona> personas = new ArrayList<>();
        personas.add(new Persona("Juan", 16));
        personas.add(new Persona("Ana", 18));
        personas.add(new Persona("Pedro", 25));
        personas.add(new Persona("María", 17));
        personas.add(new Persona("Luis", 30));

        VerificadorVoto verificador = persona -> persona.getEdad() >= 18;

        List<String> personasQuePuedenVotar = new ArrayList<>();

        for (Persona persona : personas) {
            if (verificador.puedeVotar(persona)) {
                personasQuePuedenVotar.add(persona.getNombre());
            }
        }

        // imprimir los nombres de las personas que pueden votar
        System.out.println("Personas que pueden votar:");
        for (String nombre : personasQuePuedenVotar) {
            System.out.println(nombre); // Ana, Pedro, Luis
        }
    }
}
```

```
    }  
}
```



Ejercicio 3. Filtrar y procesar una lista de productos

Crea una clase **Producto** que tenga los atributos *nombre (String)*, *precio (double)* y *categoria (String)*.

- a) Crea una interfaz funcional llamada **FiltroProducto** con un único método abstracto **boolean filtrar(Producto producto)**, que se encargará de filtrar los productos según algún criterio (por ejemplo, los productos que tienen un precio superior a un valor específico).
- b) Crea una lista de productos con diferentes valores de *precio* y *categoria*.
- c) Usa una expresión *lambda* para **filtrar** los productos de la lista que sean de una categoría específica (por ejemplo, "Electrónica") y tengan un precio superior a un valor dado (por ejemplo, 100€).
- d) Crea una nueva Lista para ir almacenando los productos que cumplen con los criterios
- e) Finalmente, imprime la lista filtrada con los nombres y precios de los productos que cumplen con estos criterios.

```
Productos filtrados (Electrónica y precio > 100):  
Nombre: Smartphone, Precio: 150.0  
Nombre: Laptop, Precio: 900.0  
Nombre: Tablet, Precio: 200.0
```

Lambdas con interfaces propias de Java

Además de las interfaces que podemos definir nosotros mismos (como en los ejemplos anteriores), Java proporciona varias interfaces funcionales predefinidas entre sus paquete `java.util` que son muy útiles al trabajar con *lambdas*, como nuestras conocidas **Comparable** y **Comparator**. Estas interfaces son muy utilizadas con *lambdas* para ordenar colecciones o elementos.

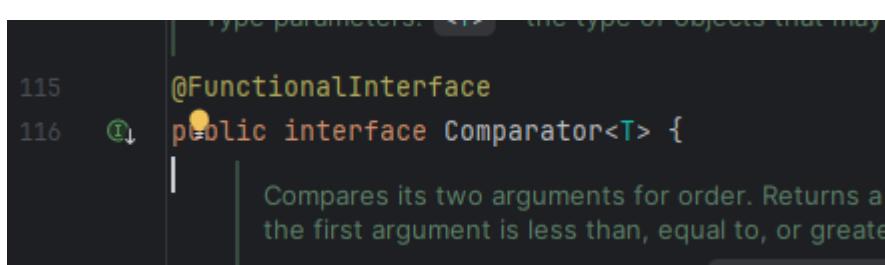
Ejemplo de *Comparator* usando *lambda*:

```
1 | List<String> nombres = Arrays.asList("Juan", "Ana", "Carlos");
2 | Collections.sort(nombres, (a, b) -> a.compareTo(b));
3 | System.out.println(nombres); // Ana, Carlos, Juan
```

¿Qué pasa internamente?

Este código está utilizando una expresión *lambda* como segundo parámetro del método `Collections.sort`. Ese segundo parámetro, como ya sabemos, es un objeto `Comparator<T>`, una interfaz funcional de Java que se usa para definir un criterio de orden.

Aunque la interfaz `Comparator<T>` tiene muchos métodos (como `thenComparing`, `reversed`, etc.), sólo uno es abstracto: `int compare(T o1, T o2)`. Ese es el único método que debe implementarse, y por eso *Comparator* es una interfaz funcional (es decir, con un único método abstracto), lo cual permite usar lambdas:



A partir de Java 8+, esa expresión se mejora más todavía y se convierte en:

```
1 | List<String> nombres = Arrays.asList("Juan", "Ana", "Carlos");
2 | nombres.sort(Comparator.comparing(s -> s));
3 | System.out.println(nombres); // [Ana, Carlos, Juan]
```

Ordenar por varios criterios

Si tuviéramos la clase Persona del ejemplo de objetos, y nos piden que ordenemos por nombre, aplicamos lo mismo que acabamos de ver:

```
1 ArrayList<Persona> personas = new ArrayList<>(Arrays.asList(new Persona
2
3 Collections.sort(personas, (p1, p2) -> p1.getNombre().compareTo(p2.getNombre()))
4
5 personas.sort(Comparator.comparing(p -> p.getNombre()));
```

pero si quisiéramos ordenar por más atributos, necesitamos otro método añadido: *thenComparing*

```
1 personas.sort(
2             Comparator.comparing(p -> p.getNombre())
3             .thenComparing(p -> p.getEdad())
4 );
5 // ó
6
7 personas.sort(
8             Comparator.comparing(Persona::getNombre)
9             .thenComparing(Persona::getEdad)
10            );
11 );
```



Ejercicio

Una empresa de tecnología necesita organizar su lista de empleados. Cada **Empleado** tiene un **nombre**, un **salario** y una **fecha** de contratación.

- Crea una lista de al menos cinco empleados con diferentes nombres, salarios y fechas de contratación.
- Implementa diferentes ordenaciones usando *lambdas* y muestra el resultado de cada ordenación por consola.
 - a) Ordenar por nombre alfabéticamente.
 - b) Ordenar por salario de menor a mayor.
 - c) Ordenar por fecha de contratación, con los más antiguos primero. Usa el método *reverse()*.

d) Ordenar por salario (de mayor a menor) y luego por nombre alfabéticamente.

7.4.2. Operaciones intermedias stream

Un **Stream** nos ayudan a manipular las colecciones, y son una secuencia de elementos que se pueden procesar de forma funcional (filtros, transformaciones, etc.).

¿Para qué sirven?

- Filtrar datos.
- Transformarlos.
- Contar, ordenar o agrupar.
- Procesarlos en cadena.
- Evitar bucles explícitos como *for* o *while*.

Por ejemplo:

```
1 public class Main {  
2  
3     public static void main(String[] args) {  
4  
5         List<String> nombres = new ArrayList<>(Arrays.asList("Ana", "Lu  
6         nombres.stream()  
7             .filter(nombre -> nombre.startsWith("A"))  
8             .forEach(System.out::println);  
9  
10    }  
11 }  
12 }
```

En este caso, el filtro que se ha aplicado es quedarse con los elementos de la lista que empiecen por "A", y los que se hayan seleccionado, se imprimen.

```
"C:\Program Files\Java\jdk-23\bin\java.e  
Ana  
Antonio  
  
Process finished with exit code 0
```

Otro ejemplo:

```
1 public class Main {  
2  
3     public static void main(String[] args) {  
4  
5         List<String> nombres = new ArrayList<>(Arrays.asList("Ana", "Lu  
6         List<String> filtrados = nombres.stream()  
7             .filter(n -> n.startsWith("L"))  
8             .map(String::toUpperCase)  
9             .collect(Collectors.toList());  
10  
11         filtrados.forEach(System.out::println);  
12     }  
13 }  
14  
15 }
```

```
"C:\Program Files\Java\jdk-23\bin\java.e  
LUIS  
  
Process finished with exit code 0
```

En este segundo caso, filtramos los que empiezan por L, los pasamos a mayúsculas y lo guardamos en otra lista.

Vale, ¿pero cómo funciona?

Un *Stream* tiene tres pasos fundamentales:

1. Origen de datos: el *stream* nace de una colección:

```
1 | List<String> nombres = Arrays.asList("Ana", "Luis", "Pedro");  
2 | nombres.stream()
```

2. Operaciones intermedias (se encadenan y no ejecutan):

- *.filter(...)* → Filtra datos
- *.map(...)* → Transforma datos
- *.sorted()* → Ordena
- *.distinct()* → Elimina duplicados
- etc.

3. Operación terminal (ejecuta)

- `.forEach(..)` → Ejecuta una acción
- `.collect(..)` → Recoge resultados
- `.count()` → Cuenta elementos
- `.sum()` → Suma elementos

Más ejemplos:

- Transformar, ordenar e imprimir

```
1 | List<String> frutas = Arrays.asList("manzana", "pera", "kiwi", "plátano")
2 |
3 | frutas.stream()                                // origen
4 |     .map(String::toUpperCase)      // intermedia: transforma
5 |     .sorted()                          // intermedia: ordena
6 |     .forEach(System.out::println);   // terminal
```

- Elevar al cuadrado sólo los pares y mostrar

```
1 | List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5, 6);
2 |
3 | numeros.stream()                                // origen
4 |     .filter(n -> n % 2 == 0)      // intermedia: sólo pares
5 |     .map(n -> n * n)            // intermedia: al cuadrado
6 |     .forEach(System.out::println); // terminal
```

- Obtener una nueva lista sin duplicados

```
1 | List<Integer> numeros = Arrays.asList(1, 2, 2, 3, 4, 4, 5);
2 |
3 | List<Integer> sinDuplicados = numeros.stream()
4 |                     .distinct()
5 |                     .collect(Collectors.toList()); // 
6 |
7 | System.out.println(sinDuplicados); // [1, 2, 3, 4, 5]
```

Comparación: *for* clásico vs. *stream*

For tradicional:

```
1 | for (String nombre : nombres) {  
2 |     if (nombre.length() > 4) {  
3 |         System.out.println(nombre.toUpperCase());  
4 |     }  
5 | }
```

Stream:

```
1 | nombres.stream()  
2 |     .filter(n -> n.length() > 4)  
3 |     .map(String::toUpperCase)  
4 |     .forEach(System.out::println);
```



Tarea

A partir de la misma lista del ejemplo, intenta escribir un stream que devuelva esto:

- a) Imprime los nombres que tienen más de 4 letras.

```
1 | nombres.stream()  
2 |     .filter(nombre -> nombre.length() > 4)  
3 |     .forEach(System.out::println);
```

- b) Transforma todos los nombres a minúsculas y lo guarda en otra lista.

```
1 | List<String> nombresMinusculas = nombres.stream()  
2 |                     .map(String::toLowerCase)  
3 |                     .collect(Collectors.toList())  
4 |  
5 | System.out.println(nombresMinusculas);
```

c) Cuenta cuántos nombres empiezan con "A".

```
1 long cantidad = nombres.stream()
2                         .filter(nombre -> nombre.startsWith("A"))
3                         .count();
4
5 System.out.println("Nombres que empiezan con 'A': " + cantidad);
```



Obra publicada con Licencia Creative Commons Reconocimiento Compartir igual 4.0
<http://creativecommons.org/licenses/by-sa/4.0/>

