

## ¿Qué son las colecciones?

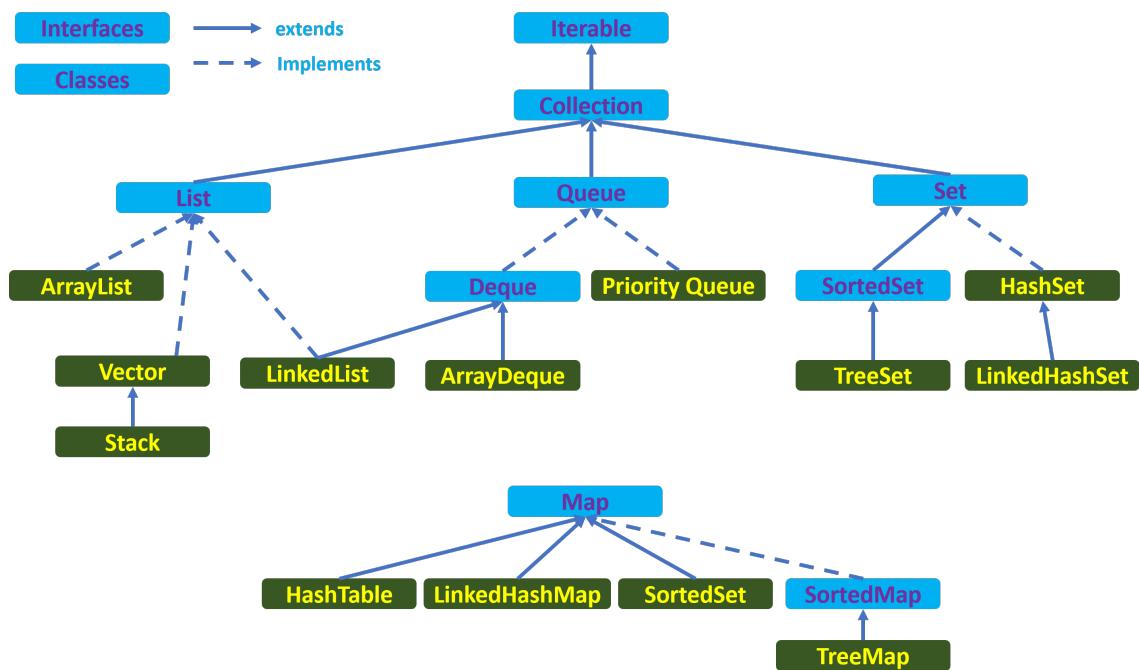
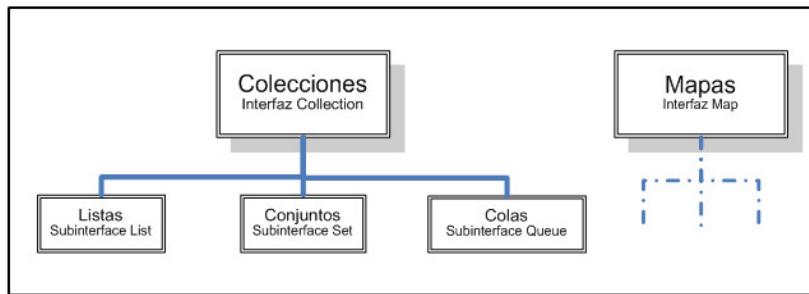
Almacenes de objetos dinámicos. La colección puede ampliarse o reducirse durante la ejecución del programa.

Inconvenientes: No podemos almacenar datos primitivos

Los arrays también son almacenes, pero no son dinámicos. Ventajas respecto a los arrays:

- Pueden cambiar de tamaño dinámicamente
- Podemos ordenar los objetos que están en una colección
- Se puede insertar y eliminar elementos de forma dinámica

Hay que tener en cuenta el framework de colecciones, todas las clases e interfaces que hay al respecto.



Una colección representa un **grupo de objetos**. Estos objetos son conocidos como elementos. Cuando queremos trabajar con un conjunto de elementos, necesitamos un almacén donde poder guardarlos. En Java, se emplea la interfaz genérica **Collection** para este propósito. Gracias a esta interfaz, podemos almacenar cualquier tipo de objeto y podemos usar una serie de métodos comunes, como pueden ser: añadir, eliminar, obtener el tamaño de la colección. Partiendo de la interfaz genérica **Collection** extienden otra serie de interfaces genéricas. Estas subinterfaces aportan distintas funcionalidades sobre la interfaz anterior.

a) Una **lista (list)** es una colección de objetos donde cada uno de ellos lleva un índice asociado. Podríamos tener una lista con los nombres de las personas que han hecho una compra de un servicio: usuarios --> (Juan, Sara, Rodolfo, Pedro).

La **funcionalidad más importante** exclusiva de las listas es el **acceso posicional** a sus elementos por medio de índices.

Cada elemento va asociado a un índice, usuario(0) sería Juan, usuario(1) sería Sara...

En una lista podemos insertar y eliminar objetos de posiciones intermedias.

Puede haber **duplicados** y el **orden** en el que se encuentran los elementos es relevante.

b) Un **conjunto (set)** es una colección de objetos que **no admite duplicados**.

Siguiendo el ejemplo anterior, un conjunto nos serviría para saber los usuarios distintos que han comprado el servicio.

No tendríamos la información ordenada y una misma persona no aparecería más de una vez aunque hubiera utilizado el servicio varias veces.

c) Una **cola (queue)** es una colección de objetos que se comportan como lo haría un grupo de personas en la cola de una **caja de un supermercado**. Los objetos se van poniendo en cola y el primero en salir es el primero que llegó.

d) Un **mapa (map)**, es una colección de objetos formados por **pares, clave-valor**. No hay duplicados. Un ejemplo de colección tipo mapa es un diccionario, las palabras son las claves, y el significado el valor.

**Tipos de colecciones.** Principales tipos de colecciones en Java.

## List

La interfaz List define una sucesión de elementos. A diferencia de la interfaz Set, la interfaz List sí admite elementos duplicados. A parte de los métodos heredados de Collection, añade métodos que permiten mejorar los siguientes puntos:

- Acceso posicional a elementos: manipula elementos en función de su posición en la lista.
- Búsqueda de elementos: busca un elemento concreto y devuelve su posición.
- Iteración sobre elementos: mejora el Iterator por defecto.
- Rango de operación: permite realizar ciertas operaciones sobre rangos de elementos dentro de la propia lista.

Dentro de la interfaz List existen varios tipos de implementaciones. Las más típicas:

- **ArrayList**: esta es la implementación típica. Se basa en un array redimensionable que aumenta su tamaño según crece la colección de elementos. Es la que mejor rendimiento tiene sobre la mayoría de situaciones.

**Muy eficiente para recorrer pero ineficiente para insertar o eliminar elementos que se no encuentren en el final.**

- usa internamente un array dinámico para almacenar los elementos.
  - Inserción y eliminación de elementos lenta
  - mejor opción para almacenar y acceder a datos o elementos consecutivos
  - muy rápida en las operaciones que supongan recorrer la lista para la lectura o modificación de elementos.
- **LinkedList**: esta implementación se basa en una lista doblemente enlazada de los elementos, teniendo cada uno de los elementos un puntero al anterior y al siguiente elemento.

**Lista doblemente enlazada, lenta para recorrer pero sumamente eficiente para insertar o eliminar elementos.**

- proporciona una manipulación más rápida porque utiliza una lista doblemente enlazada.
  - se puede utilizar como lista y cola porque implementa interfaz de List, Deque y Queue.
  - es mejor para manipulación de elementos, es decir, para insertar y eliminar elementos.
- Si vas a agregar elementos una sola vez, pero luego vas a acceder a ellos muchas veces entonces usa ArrayList
  - Si vas a estar continuamente agregando o eliminando, entonces LinkedList

## LinkedList

```
● add(String e) : boolean - LinkedList
● add(int index, String element) : void - LinkedList
● addAll(Collection<? extends String> c) : boolean - List
● addAll(int arg0, Collection<? extends String> arg1) : void - List
● addFirst(String e) : void - LinkedList
● addLast(String e) : void - LinkedList
● clear() : void - LinkedList
● clone() : Object - LinkedList
● contains(Object o) : boolean - LinkedList
● containsAll(Collection<?> arg0) : boolean - List
● descendingIterator() : Iterator<String> - LinkedList
● element() : String - LinkedList
● equals(Object arg0) : boolean - List
● forEach(Consumer<? super String> arg0) : void - Iterable
● get(int index) : String - LinkedList
● getClass() : Class<?> - Object
● getFirst() : String - LinkedList
● getLast() : String - LinkedList
● hashCode() : int - List
● indexOf(Object arg0) : int - LinkedList
● isEmpty() : boolean - List
● iterator() : Iterator<String> - List
● lastIndexOf(Object arg0) : int - LinkedList
● listIterator() : ListIterator<String> - List
● listIterator(int index) : ListIterator<String> - LinkedList
● notify() : void - Object
● notifyAll() : void - Object
● offer(String e) : boolean - LinkedList
● offerFirst(String e) : boolean - LinkedList
● offerLast(String e) : boolean - LinkedList
● parallelStream() : Stream<String> - Collection
● peek() : String - LinkedList
● peekFirst() : String - LinkedList
● peekLast() : String - LinkedList
● poll() : String - LinkedList
● pollFirst() : String - LinkedList
● pollLast() : String - LinkedList
● pop() : String - LinkedList
● push(String e) : void - LinkedList
● remove() : String - LinkedList
● remove(int index) : String - LinkedList
● remove(Object arg0) : boolean - LinkedList
● removeAll(Collection<?> arg0) : boolean - List
● removeFirst() : String - LinkedList
● removeFirstOccurrence(Object o) : boolean - LinkedList
● removeIf(Predicate<? super String> filter) : boolean - List
● removeLast() : String - LinkedList
● removeLastOccurrence(Object arg0) : boolean - LinkedList
● replaceAll(UnaryOperator<String> operator) : void - List
● retainAll(Collection<?> arg0) : boolean - List
● set(int index, String element) : String - LinkedList
● size() : int - LinkedList
● sort(Comparator<? super String> arg0) : void - List
● spliterator() : Spliterator<String> - LinkedList
● stream() : Stream<String> - Collection
● subList(int arg0, int arg1) : List<String> - List
● toArray() : Object[] - LinkedList
● toArray(IntFunction<T[]> generator) : T[] - Collection
● toArray(T[] arg0) : T[] - LinkedList
● toString() : String - AbstractCollection
● wait() : void - Object
● wait(long arg0) : void - Object
● wait(long timeoutMillis, int nanos) : void - Object
```

## ArrayList

```
● add(String e) : boolean - ArrayList
● add(int index, String element) : void - ArrayList
● addAll(Collection<? extends String> c) : boolean - ArrayList
● addAll(int index, Collection<? extends String> c) : boolean - List
● clear() : void - ArrayList
● clone() : Object - ArrayList
● contains(Object o) : boolean - ArrayList
● containsAll(Collection<?> arg0) : boolean - List
● ensureCapacity(int minCapacity) : void - ArrayList
● equals(Object o) : boolean - ArrayList
● forEach(Consumer<? super String> arg0) : void - Iterable
● get(int index) : String - ArrayList
● getClass() : Class<?> - Object
● hashCode() : int - ArrayList
● indexOf(Object o) : int - ArrayList
● isEmpty() : boolean - ArrayList
● iterator() : Iterator<String> - ArrayList
● lastIndexOf(Object o) : int - ArrayList
● listIterator() : ListIterator<String> - ArrayList
● listIterator(int index) : ListIterator<String> - ArrayList
● notify() : void - Object
● notifyAll() : void - Object
● parallelStream() : Stream<String> - Collection
● remove(int index) : String - ArrayList
● remove(Object o) : boolean - ArrayList
● removeAll(Collection<?> c) : boolean - ArrayList
● removeIf(Predicate<? super String> filter) : boolean - List
● replaceAll(UnaryOperator<String> operator) : void - List
● retainAll(Collection<?> c) : boolean - ArrayList
● set(int index, String element) : String - ArrayList
● size() : int - ArrayList
● sort(Comparator<? super String> c) : void - ArrayList
● spliterator() : Spliterator<String> - ArrayList
● stream() : Stream<String> - Collection
● subList(int fromIndex, int toIndex) : List<String> - ArrayList
● toArray() : Object[] - ArrayList
● toArray(IntFunction<T[]> generator) : T[] - Collection
● toArray(T[] a) : T[] - ArrayList
● toString() : String - AbstractCollection
● trimToSize() : void - ArrayList
● wait() : void - Object
● wait(long arg0) : void - Object
● wait(long timeoutMillis, int nanos) : void - Object
```

## Clase ArrayList

Un ArrayList es una estructura en forma de lista que permite almacenar elementos del mismo tipo. Su tamaño va cambiando a medida que se añaden o se eliminan esos elementos.

Aunque las colecciones permiten el uso de índices, no es necesario indicarlos siempre. Por ejemplo, en una colección del tipo ArrayList de String, cuando hay que añadir el elemento "Amapola", se puede hacer flores.add("Amapola"). Al no especificar índice, el elemento "Amapola" se añadiría justo al final de flores independientemente del tamaño y del número de elementos que se hayan introducido ya.

### Principales métodos de ArrayList

Las operaciones más comunes que se pueden realizar:

- add(elemento) : Añade un elemento al final de la lista.
- add(indice, elemento): Inserta elemento en una posición determinada, desplazando el resto de elementos hacia la derecha.
- clear(): Elimina todos los elementos pero no borra la lista.
- contains(elemento): Devuelve true si la lista contiene el elemento, false en caso contrario.
- get(indice): Devuelve el elemento de la posición que se indica entre paréntesis.
- indexOf(elemento): Devuelve la posición de la primera ocurrencia del elemento.
- isEmpty(): Devuelve true si la lista está vacía y false en caso contrario.
- remove(indice): Elimina el elemento que se encuentra en esa posición.
- remove(elemento): Elimina la primera ocurrencia de un elemento.
- removeIf(filtro): Elimina los elementos que cumplen una determinada condición.
- set(indice, elemento): Machaca el elemento que se encuentra en una determinada posición con el elemento que se pasa como parámetro.
- size(): Devuelve el tamaño (número de elementos) de la lista.
- toArray(): Devuelve un array con todos y cada uno de los elementos que contiene la lista.

A continuación se muestra un **ejemplo** en el que se puede ver cómo se declara un ArrayList y cómo se insertan y se extraen elementos.

```
import java.util.ArrayList;
public class Ejemplo1 {
    public static void main(String[] args) {
        ArrayList<String> miLista = new ArrayList<>();
        System.out.println("Número de elementos: " + a.size());
        a.add("rojo");
        a.add("verde");
        a.add("azul");
        System.out.println("Num de elementos: " + a.size());
        a.add("blanco");
        System.out.println("Num de elementos: " + a.size());
        System.out.println("El elemento que hay en la posición 0 es " + a.get(0));
        System.out.println("El elemento que hay en la posición 3 es " + a.get(3));
    }
}
```

El **método add** permite añadir elementos a un ArrayList. Por ejemplo, `a.add("amarillo")` añade el elemento "amarillo" **al final** de la lista o se puede utilizar también con un índice `a.add(1, "turquesa")`. En este caso, se inserta en la posición indicada. El ArrayList se reestructura de forma automática desplazando el resto de elementos.

Al crear un objeto de la clase ArrayList hay que indicar el tipo de dato que se almacenará en las celdas de esa lista. Para ello se utilizan los caracteres < y >. ¡OJO! No olvidar los paréntesis del final.

Es necesario importar la clase ArrayList para poder crear objetos de esta clase, import `java.util.ArrayList`.

En el siguiente **ejemplo** se muestra un ArrayList de números enteros.

```
import java.util.ArrayList;
public class Ejemplo2 {
    public static void main(String[] args) {
        ArrayList<Integer> a = new ArrayList<>(); //Se utiliza el wrapper Integer (con tipos primitivos NO se puede)
        a.add(18);
        a.add(22);
        a.add(-30);
        System.out.println("Un de elementos: " + a.size());
        System.out.println("El elemento que hay en la posición 1 es " + a.get(1));
    }
}
```

Para recorrer un ArrayList y pintar su contenido:

- Con for clásico

```
for(int i=0; i < miLista.size(); i++) {
    System.out.println(miLista.get(i));
}
```
- Con for each

```
for(String color: miLista) {
    System.out.println(color);
}
//Otra forma: con programación funcional
//miLista.forEach(System.out::println);
```
- Con iterador

```
Iterator<String> iterador = miLista.iterator();
while (iterador.hasNext())
    System.out.print(iterador.next() + " ");
```

### Eliminar elementos de un ArrayList.

Se utiliza el método `remove()` y se puede pasar como parámetro el índice del elemento que se quiere eliminar, o bien el valor del elemento.

`miLista.remove(2)` elimina el elemento que se encuentra en la posición 2 de la lista `miLista`  
`miLista.remove("blanco")` elimina el valor "blanco" de la lista (la primera ocurrencia)

Borrado selectivo si los elementos de una lista cumplen una determinada condición

Ejemplo: borrar todos los elementos que tengan "blanco"

```
miLista.removeIf(elemento-> elemento.contains("blanco"));
```

Ejemplo: Eliminar todos los números menores que 10.

```
miLista.removeIf(numero -> numero < 10);
```

Se puede también “machacar” una posición del ArrayList.

Al hacer `a.set(2, "turquesa")`, se borra lo que hubiera en la posición 2 y se coloca el valor “turquesa”. Sería equivalente a hacer `a[2] = "turquesa"` en caso de que `a` fuese un array tradicional.

### ArrayList de objetos

Una colección ArrayList puede contener objetos, instancias de clases definidas por el programador. Esto es muy útil para guardar datos de alumnos, productos, libros, etc.

En el siguiente ejemplo, definimos una lista de gatos. En cada celda de la lista se almacenará un objeto de la clase Gato.

```
import java.util.ArrayList;
public class Ejemplo10 {
    public static void main(String[] args) {
        ArrayList<Gato> g = new ArrayList<Gato>();
        g.add(new Gato("Garfield", "naranja", "mestizo"));
        g.add(new Gato("Pepe", "gris", "angora"));
        g.add(new Gato("Mauri", "blanco", "manx"));
        g.add(new Gato("Ulises", "marrón", "persa"));
        System.out.println("\nDatos de los gatos:\n");
        for (Gato gatoAux: g) {
            System.out.println(gatoAux+"\n");
        }
    }
}
```

### Ordenación de un ArrayList

Los elementos de una lista se pueden ordenar con el método `sort`.

El formato es el siguiente: `Collections.sort(lista)`;

Para poder utilizar este método es necesario incluir la línea `import java.util.Collections`

```
public class Ejemplo11 {
    public static void main(String[] args) {
        ArrayList<Integer> a = new ArrayList<Integer>();
        a.add(67);
        a.add(78);
        a.add(10);
        Collections.sort(a);
        System.out.println("\nNúmeros ordenados:");
        for (int numero: a) {
            System.out.println(numero);
        }
    }
}
```

También es posible ordenar una lista de objetos. En este caso es necesario indicar el criterio de ordenación en la definición de la clase.

Lo primero que hay que hacer es indicar que los objetos de la clase Gato se pueden comparar unos con otros. Para ello:

- implementamos la interfaz Comparable

```
public class Gato implements Comparable<Gato>
```

- Definimos el método `compareTo` lo el criterio de ordenación por defecto.

Tiene que devolver un 0 si los elementos que se comparan son iguales, un número negativo si el primer elemento que se compara es menor que el segundo y un número positivo en caso contrario.

```
public int compareTo(Gato g) {  
    return (this.nombre).compareTo(g.getNombre());  
}
```

### Método asList()

El método **asList()** de la clase [java.util.Arrays](#) se usa para devolver una lista de tamaño fijo que proviene de una matriz dada.

Este método actúa como puente entre las API basadas en matrices y colecciones , en combinación con Collection.toArray()

**NOTA:** El tipo de matriz debe ser una clase de envoltura (Integer, Float, etc.) en el caso de tipos de datos primitivos (int, float, etc.)

#### Ejemplos:

```
String a[] = new String[] { "A", "B", "C", "D" };  
List<String> list = Arrays.asList(a);
```

```
Integer a[] = new Integer[] { 10, 20, 30, 40 };  
List<Integer> list = Arrays.asList(a);
```

## Set

La interfaz **Set** define una colección que **no puede contener elementos duplicados**. Esta interfaz contiene, únicamente, los métodos heredados de **Collection** con la restricción de que los elementos duplicados están prohibidos. Cuando intentemos insertar un elemento repetido no se insertará pero tampoco saldrá ningún error.

Para comprobar si los elementos son elementos duplicados o no lo son, es necesario que esos elementos tengan implementados los métodos **equals** y **hashCode**.

Dentro de la interfaz **Set** existen varios tipos de implementaciones:

- **HashSet**: **almacena los elementos en una tabla hash**. Es la implementación con **mejor rendimiento de todas** pero **no garantiza ningún orden** a la hora de realizar iteraciones. Es la implementación **más empleada debido a su rendimiento** y a que, generalmente, no nos importa el orden que ocupen los elementos. Esta implementación proporciona tiempos constantes en las operaciones básicas.

Rápida, No duplicados, **No ordenación**, No acceso aleatorio

No se garantiza que los objetos que inserte en HashSet se inserten en el mismo orden. Los objetos se insertan en función de su código hash. Se permiten elementos NULL en HashSet.

- **TreeSet**: esta implementación **almacena los elementos ordenándolos en función de sus valores**. Es bastante más lento que **HashSet**. Los elementos almacenados deben implementar la interfaz **Comparable**. El criterio de ordenación por defecto será el proporcionado por el método `compareTo()`.

Esta implementación garantiza, siempre, un rendimiento de  $\log(N)$  en las operaciones básicas, debido a la estructura de árbol empleada para almacenar los elementos.

**Ordenado**, Poco eficiente

- **LinkedHashSet**: esta implementación **almacena los elementos en función del orden de inserción**. Es un poco más costosa que **HashSet**.

Ordenación por entrada, Eficiente al acceder, No eficiente al agregar

### Otras clases de tipo Set

- `EnumSet` (para tipos enumerados)
- `CopyOnWriteArraySet`, (para concurrencia, eficiente para la lectura)
- `ConcurrentSkipListSet` (para concurrencia, con muchos elementos no es eficiente)

## TreeSet

```
public class EjemploTreeSet {
    public static void main(String[] args) {
        TreeSet<String> miArbol = new TreeSet<>();

        miArbol.add("Juan");
        miArbol.add("Mari");
        miArbol.add("Bea");
        miArbol.add("Maria");

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

En el ejemplo anterior salen ordenados alfabéticamente porque son elementos de tipo String. String implementa la interfaz Comparable y tiene el método compareTo

Si hacemos un TreeSet con objetos creados por nosotros, hay que asegurarse de implementar la interfaz Comparable y desarrollar el criterio para el compareTo

```
class Articulo implements Comparable<Articulo>{
    private int numero_articulo;
    private String descripcion;

    public Articulo (int num, String desc) {
        numero_articulo=num;
        descripcion=desc;
    }
    public String getDescripcion () {
        return descripcion;
    }
    @Override
    public int compareTo(Articulo a) {
        return numero_articulo - a.numero_articulo;
    }
}
```

Si queremos ordenar los artículos con otro criterio no podemos desarrollar varios métodos compareTo. Creamos una clase que implemente la interfaz Comparator. Nos permitirá establecer el criterio que queramos. Después generamos un nuevo TreeSet utilizando esa clase comparadora en el constructor.

```
class MiComparador implements Comparator<Articulo>{
    @Override
    public int compare(Articulo a, Articulo b) {
        String descripcion1 = a.getDescripcion();
        String descripcion2 = b.getDescripcion();

        return descripcion1.compareTo(descripcion2);
    }
}
```

Y desde el programa principal indicamos:

```
MiComparador compara1 = new MiComparador();
TreeSet<Articulo> miArbol = new TreeSet<Articulo>(compara1);
//pasamos como parámetro el comparador al constructor del TreeSet
```

## Ejercicios propuesto para conjuntos (Set)

Crear una clase Clientes, con 3 atributos: nombre, email y tipo\_cliente.

- o Crear colección de tipo HashSet
- o Agregar objetos de tipo cliente a la colección
- o Recorrer la colección y mostrar los elementos.
- o Crear colección de tipo TreeSet y mostrar los elementos ordenados por email
- o No se permiten duplicados por nombre.

```
public class Clientes {  
    private String nombre;  
    private String email;  
    private String tipo_cliente;  
    public Clientes (String nombre, String email, String tipo) {  
        this.nombre = nombre;  
        this.email = email;  
        this.tipo_cliente = tipo;  
    }  
    . . . getters  
    . . . setters  
}  
public class Tienda {  
    public static void main(String[] args) {  
        Clientes c1 = new Clientes ("Paco","paco@gmail.com","Oro");  
        Clientes c2 = new Clientes ("María","maria@gmail.com","Platino");  
        Clientes c3 = new Clientes ("Pepe","pepe@hotmail.com","Bronce");  
        Clientes c4 = new Clientes ("Susana","susana@hotmail.com","Platino");  
    }  
}
```

No se puede hacer `Set<Clientes> colección_clientes1 = new Set<Clientes>();` porque Set es una interfaz, por eso no puedo instanciar, hay que usar alguna clase que implemente esa interfaz

```
Set<Clientes> ctoclientes = new HashSet<Clientes>();  
ctoclientes.add(c1);  
ctoclientes.add(c2);  
ctoclientes.add(c3);  
ctoclientes.add(c4);  
  
for (Clientes varclientes: ctoclientes) {  
    System.out.println(varclientes.getNombre()+"."+varclientes.getEmail());  
}  
  
TreeSet<Clientes> ctoclientes=new TreeSet<Clientes>(Arrays.asList(c1,c2,c3,c4));
```

En la clase Clientes implementamos el interfaz Comparable y añadimos el método CompareTo para ordenar según el email.

```
public int compareTo(Clientes otro) {  
    return email.compareTo(otro.email);  
}
```

Para el control de duplicados, implementamos los métodos equals y hashCode

Genero una nueva instancia pero con el mismo nombre de cliente "Paco".

```
Clientes c5= new Clientes ("Paco","paquillo@gmail.com","Platino");
```

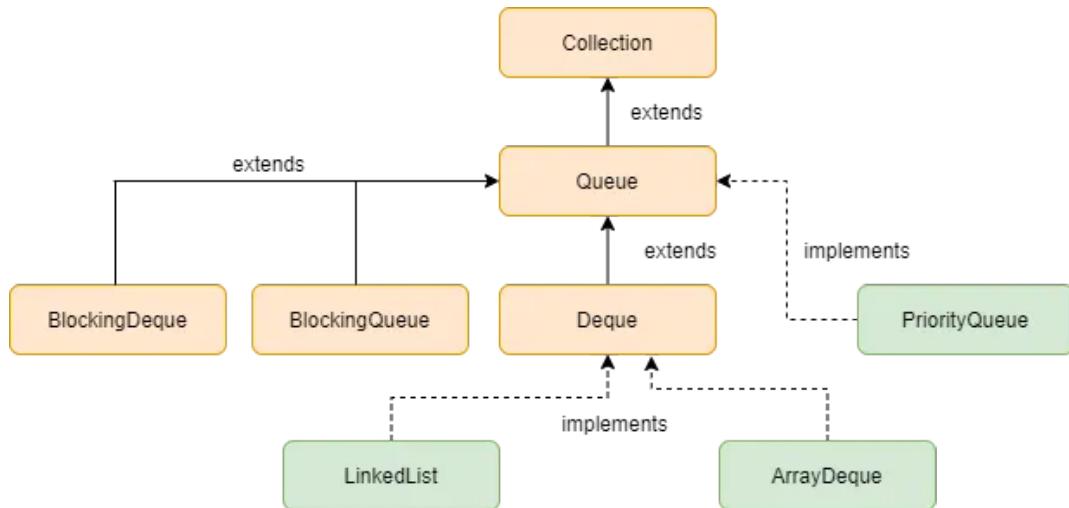
Podré añadir este nuevo cliente con add porque para Java esos objetos son diferentes, tienen posiciones de memoria distintas. Java sabe comparar String, Date, por ejemplo, porque están predefinidos, pero para una colección en particular **hay que sobreescribir los métodos equals y hashCode** para indicarle cual es el criterio que queremos poner para comparar.

Una vez sobreescritos, la colección set ya no dejará meter duplicados con los criterios que hayamos puesto.

Como en el enunciado me piden control de duplicados por nombre, desarrollamos esos métodos teniendo en cuenta sólo ese criterio.

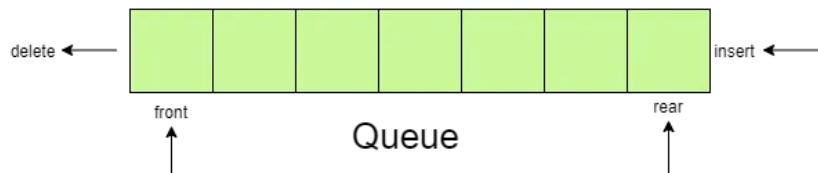
```
@Override  
public int hashCode() {  
    final int prime = 31;  
    int result = 1;  
    result = prime * result + ((nombre == null) ? 0 : nombre.hashCode());  
    return result;  
}  
  
@Override  
public boolean equals(Object obj) {  
    if (this == obj)  
        return true;  
    if (obj == null)  
        return false;  
    if (getClass() != obj.getClass())  
        return false;  
    Clientes other = (Clientes) obj;  
    if (nombre == null) {  
        if (other.nombre != null)  
            return false;  
    } else if (!nombre.equals(other.nombre))  
        return false;  
    return true;  
}
```

# Colas



## Java Queue (interfaz)

- Agrega elementos por la parte trasera y los borra por la parte delantera
- Implementa el concepto de primero en entrar, primero en salir (FIFO), algunas colas se pueden utilizar también para implementar pilas (LIFO).
- Admite todos los métodos de la interfaz de colección
- Mantiene una colección ordenada de elementos



## PriorityQueue (clase)

Es una clase que implementa la cola y procesa los elementos según una prioridad FIFO.

- **Prioridad por defecto (según el tipo de dato)** Por ejemplo con String será alfabéticamente.

```
PriorityQueue<String> nombres= new PriorityQueue<>();  
  
nombres.add("Paco");  
nombres.add("Patricia");  
nombres.add("Joe");  
nombres.add("Juan Antonio");  
nombres.add("Patri");  
  
System.out.println(nombres); //El orden en el que se muestra es alfabéticamente  
  
while (!nombres.isEmpty()) //Al sacar de la cola se verá la prioridad que hemos definido  
    System.out.println(nombres.poll());
```

**Salida por pantalla:** [Joe, Juan Antonio, Paco, Patricia, Patri]

Joe  
Juan Antonio  
Paco  
Patri  
Patricia

- Prioridad según clase comparadora

Acepta una clase comparadora en el constructor que determinará la prioridad de la cola.

```
//Creamos una cola donde la prioridad sea el tamaño del nombre
PriorityQueue<String> nombres= new PriorityQueue<>(new ComparaLongitud());

nombres.add("Paco");
nombres.add("Patricia");
nombres.add("Joe");
nombres.add("Juan Antonio");
nombres.add("Patri");

System.out.println(nombres); //El orden en el que se muestra no coincide con la entrada

while (!nombres.isEmpty()) //Al sacar de la cola se verá la prioridad que hemos definido
    System.out.println(nombres.poll());
```

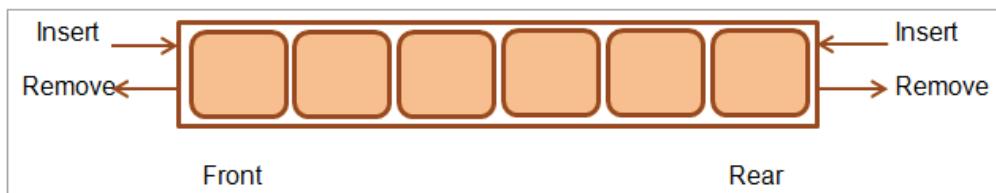
Salida por pantalla: [Joe, Patri, Paco, Juan Antonio, Patricia]

Joe  
Paco  
Patri  
Patricia  
Juan Antonio

## Deque (interfaz)

Es una interfaz que extiende la interfaz cola. La Deque o “cola de dos extremos” en Java es una estructura de datos en la que podemos insertar o eliminar elementos de ambos extremos.

Admite la implementación de la cola, que es el primero en entrar, el primero en salir (FIFO), y la implementación de la pila, que es el último en entrar, primero en salir (LIFO).



Inserción de elementos en un deque:

Arriba Abajo

|               |                                 |                                   |
|---------------|---------------------------------|-----------------------------------|
| Sin excepción | <code>offerFirst()</code>       | <code>offer(), offerLast()</code> |
| Excepción     | <code>addFirst(), push()</code> | <code>add(), addLast()</code>     |

Recuperar elementos de un deque:

Primer elemento (superior), sin eliminación Primer elemento (superior), eliminación

|               |                                    |   |
|---------------|------------------------------------|---|
| Sin excepción | <code>peek(), peekFirst()</code>   | <code>poll(), pollFirst()</code>            |
| Excepción     | <code>getFirst(), element()</code> | <code>remove(), removeFirst(), pop()</code> |

Último elemento (inferior), sin eliminación Último elemento (inferior), eliminación

|               |                         |                           |
|---------------|-------------------------|---------------------------|
| Sin excepción | <code>peekLast()</code> | <code>pollLast()</code>   |
| Excepción     | <code>getLast()</code>  | <code>removeLast()</code> |

`add()`, `addFirst()` y `addLast()` lanzan excepción si la cola está llena.

`offer()`, `offerFirst()` y `offerLast()` devuelven true si el elemento se inserta con éxito; si la cola deque está llena, estos métodos regresan false.

## ArrayDeque (clase)

La clase ArrayDeque implementa la interfaz Deque, lo que significa que podemos **insertar y eliminar los elementos de ambos lados**

*Nota:* Java tiene una clase para Pilas (Stack), ya en desuso. Según figura en el API de Java se recomienda para la implementación de pilas el uso de la interfaz Deque a partir de la implementación de ArrayDeque.

```
java.util
Class Stack<E>
java.lang.Object
    java.util.AbstractCollection<E>
        java.util.AbstractList<E>
            java.util.Vector<E>
                java.util.Stack<E>
All Implemented Interfaces:
    Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess
public class Stack<E>
extends Vector<E>
The Stack class represents a last-in-first-out (LIFO) stack of objects. It extends class Vector with five operations that allow a vector to be treated as a stack. The usual push and pop operations are provided, as well as a method to peek at the top item on the stack, a method to test for whether the stack is empty, and a method to search the stack for an item and discover how far it is from the top.
When a stack is first created, it contains no items.
A more complete and consistent set of LIFO stack operations is provided by the Deque interface and its implementations, which should be used in preference to this class. For example:
Deque<Integer> stack = new ArrayDeque<Integer>();
```

No tiene restricciones de capacidad y crece según sea necesario para admitir el uso. No es seguro para subprocessos, lo que significa que, en ausencia de sincronización externa, ArrayDeque no admite el acceso simultáneo de varios subprocessos.

```
import java.util.*;
public class ArrayDequeDemo {
    public static void main(String[] args)
    {
        // Initializing an deque
        Deque<Integer> de_que
            = new ArrayDeque<Integer>(10);

        // add() method to insert
        de_que.add(10);
        de_que.add(20);
        de_que.add(30);
        de_que.add(40);
        de_que.add(50);

        System.out.println(de_que);

        // clear() method
        de_que.clear();

        // addFirst() method to insert the
        // elements at the head
        de_que.addFirst(564);
        de_que.addFirst(291);

        // addLast() method to insert the
        // elements at the tail
        de_que.addLast(24);
        de_que.addLast(14);

        System.out.println(de_que);
    }
}
```

Ejecución:

```
[10, 20, 30, 40, 50]
[291, 564, 24, 14]
```

## Insertando elementos en ArrayDeque

```
4
5 public class PruebaColasDeque {
6
7     public static void main(String[] args) {
8
9         ArrayDeque<String> animales= new ArrayDeque<>();
10
11         animales.add("Perro");
12
13         animales.addFirst("Gato");
14
15         animales.addLast("Caballo");
16
17         System.out.println("ArrayDeque: " + animales);
18
19     }
20 }
```

```
Console ☒
<terminated> PruebaColasDeque [Java Application] /Users/macair/p2/pool/plugins/org.eclipse.justj.openjdk.hotspot.jre.full.macosx.x86_64_14.0.2.v20200815-0932/jre/bin/java (20 abr. 2023)
ArrayDeque: [Gato, Perro, Caballo]
```

## Recuperando elementos de una cola

```
7
8     public static void main(String[] args) {
9         ArrayDeque<String> animales= new ArrayDeque<>();
10
11         //Añade al final
12         animales.add("Perro");
13
14         //Añade al principio
15         animales.addFirst("Gato");
16
17         //Añade al final
18         animales.add("León");
19
20         //Añade al final
21         animales.addLast("Caballo");
22
23         System.out.println("ArrayDeque: " + animales);
24
25         // Obtiene el primer elemento de la cola
26         String firstElement = animales.getFirst();
27         System.out.println("Primer Elemento: " + firstElement);
28
29         // Obtiene el ultimo elemento
30         String lastElement = animales.getLast();
31         System.out.println("Último Elemento: " + lastElement);
```

```
Console ☒
<terminated> PruebaColasDeque [Java Application] /Users/macair/p2/pool/plugins/org.eclipse.justj.openjdk.hotspot.jre.full.macosx.x86_64_14.0.2.v20200815-0932/jre/bin/java (20 abr. 2023)
ArrayDeque: [Gato, Perro, León, Caballo]
Primer Elemento: Gato
Último Elemento: Caballo
```

## Ejercicios propuestos sobre Colas

**Implementa un ArrayDeque que sirva para organizar los pedidos a domicilio de un restaurante (un FIFO)**

Crea un menú con las siguientes opciones:

- a. Nuevo pedido, se introduce en la cola
- b. ¿Pedidos en cola?
- c. ¿Primer pedido en cola?
- d. Servir un pedido (mostrar número de pedido, desaparecerá de la cola)
  
- e. Pedido Vip
- f. Salir del programa

**Implementar una clase pila para añadir prendas de ropa para planchar, usando deque y con los siguientes métodos (un LIFO)**

- insertar: inserta elemento en la pila, no devuelve nada. Como parámetro recibe solo el número a insertar.
- quitar: quita elemento de la pila, devuelve -1 si está vacía. No recibe parámetros
- vacia: informa si la pila está vacía o no, devuelve booleano. No recibe parámetros
- numelementos: dice cuantos elementos hay en la pila. No recibe parámetros

Después implementa un programa que haga uso de esa clase.

## Map

La interfaz **Map** asocia claves a valores. Esta interfaz no puede contener claves duplicadas y cada una de dichas claves, sólo puede tener asociado un valor. Existen varios tipos de implementaciones:

- **HashMap**: almacena las **claves en una tabla hash**. Es la implementación con mejor rendimiento de todas pero no garantiza ningún orden a la hora de realizar iteraciones. Esta implementación proporciona tiempos constantes en las operaciones básicas.
  - No ordenada
  - Eficiente
- **TreeMap**: **almacena las claves ordenándolas en función de sus valores**. Es bastante más lento que *HashMap*. Las claves almacenadas deben implementar la interfaz Comparable. Garantiza un rendimiento de  $\log(N)$  en las operaciones básicas, debido a la estructura de árbol empleada para almacenar los elementos.
  - Ordenado por clave
  - Poco eficiente en todas sus operaciones
- **LinkedHashMap**: esta implementación **almacena las claves en función del orden de inserción**. Es un poco más costosa que *HashMap*.
  - Ordenado por inserción
  - Permite ordenación por uso (los elementos se reordenan según su frecuencia de uso/acceso)
  - Lectura eficiente y escritura poco eficiente.

### Otras clases de tipo Map

- **EnumMap** (para tipos enumerados)
- **ConcurrentHashMap** (para concurrencia, no permite nulos)

Ejemplo de uso con maps

<https://programandointentandolo.com/2013/02/ejemplo-de-uso-de-hashmap-en-java-2.html>

<https://parzibyte.me/blog/2020/01/07/hashmap-javascript-tutorial-ejemplos/>

<https://es.tutorialcup.com/java/hashmap-in-java.htm>

<https://jarroba.com/map-en-java-con-ejemplos/>

## Hashmap (clase)

Como es de tipo Map, almacena parejas clave, valor. No garantiza el orden.

- **No permite duplicado de claves pero pueden tener valores duplicados**
- **No mantiene ningún orden**, lo que significa que el orden en el que se insertan los datos no es el mismo que el orden en que se recuperan

Ejemplo Declaración de un Map (un HashMap) con clave "Integer" y Valor "String"

Principales métodos para trabajar con los mapas.

```
Map<Integer, String> nombreMap = new HashMap<Integer, String>();
nombreMap.size(); // Devuelve el numero de elementos del Map
nombreMap.isEmpty(); // Devuelve true si no hay elementos en el Map y false si si los hay
nombreMap.put(K clave, V valor); // Añade un elemento al Map
nombreMap.get(K clave); // Devuelve el valor de la clave que se le pasa como parámetro o 'null' si la clave no existe
nombreMap.clear(); // Borra todos los componentes del Map
nombreMap.remove(K clave); // Borra el par clave/valor de la clave que se le pasa como parámetro
nombreMap.containsKey(K clave); // Devuelve true si en el map hay una clave que coincide con K
nombreMap.containsValue(V valor); // Devuelve true si en el map hay un Valor que coincide con V
nombreMap.values(); // Devuelve una "Collection" con los valores del Map
```

```
//Creamos objeto HashMap con 2 argumentos, clave de tipo String y el
//valor será de tipo Empleado.
HashMap<String, Empleado> staff = new HashMap<String,Empleado>();

Empleado e1 = new Empleado("Paco",1000);
Empleado e2 = new Empleado("Pepe",1500);
Empleado e3 = new Empleado("María",2000);
Empleado e4 = new Empleado("Carmen",2500);

staff.put("id1", e1);
staff.put("id2", e2);
staff.put("id3", e3);

System.out.println(staff);

staff.remove("id2"); //Borro un elemento usando su clave
staff.get("id3").saluda(); //Invoca al método saluda de la clase empleado para el
//empleado que está con clave id3, que es el e3.
staff.put("id3", e4); //Sustituye al elemento que está en la clave 3
System.out.println(staff);

//System.out.println(staff.entrySet());

for (Map.Entry<String,Empleado> entrada : staff.entrySet()) {
    //staff.entrySet devuelve un set<Map.Entry>
    String clave = entrada.getKey();
    Empleado valor = entrada.getValue();

    System.out.println(clave+":"+valor);
}
```

staff.entrySet() devuelve un cto de objetos de tipo Map.Entry<String, Empleado>

- cada objeto representa una entrada del mapa de empleados, una clave-valor.
- cada objeto Map.Entry<String, Empleado> contiene 2 métodos principales:
  - getKey(): devuelve la clave de la entrada.
  - getValue(): devuelve el valor asociado a la clave

## Ejemplos con LinkedHashMap y TreeMap

```
LinkedHashMap<String, Integer> lista = new LinkedHashMap<String, Integer>();
    lista.put("gato1", 3);
    lista.put("gato3", 2);
    lista.put("gato2", 4);
Con LinkedHashMap salen ordenados según el orden de inserción.
for (Map.Entry<String, Integer> entrada : lista.entrySet()) {
    String clave = entrada.getKey();
    Integer valor = entrada.getValue();
    System.out.println(clave+"-"+valor);
}

TreeMap<String, Integer> lista2 = new TreeMap<String, Integer>();
    lista2.put("gato1", 3);
    lista2.put("gato3", 2);
    lista2.put("gato2", 4);
Con TreeMap salen ordenados por la clave (alfabéticamente en este caso
por ser de tipo String)
for (Map.Entry<String, Integer> entrada : lista2.entrySet()) {
    String clave = entrada.getKey();
    Integer valor = entrada.getValue();
    System.out.println(clave+"-"+valor);
}
```

## TreeMap de objetos como valor

```
TreeMap<Integer, Gato> gatos = new TreeMap<>();
Gato g1 = new Gato("g1","amarillo","r1");
Gato g2 = new Gato("g2","negro","r2");
Gato g3 = new Gato("g3","blanco","r3");
Gato g4 = new Gato("g4","verde","r4");
gatos.put(1, g1);
gatos.put(2, g2);
gatos.put(4, g4);
gatos.put(3, g3);
//El treemap ordena automáticamente por clave, pero no por valor.
//se puede ordenar por valor pero es más engorroso, por ejemplo creando una lista

//Se muestran ordenados por clave (Integer)
for (Map.Entry<Integer, Gato> entrada : gatos.entrySet()) {
    int clave = entrada.getKey();
    Gato valor = entrada.getValue();
    System.out.println(clave+"-"+valor);
}

List<Map.Entry<Integer, Gato>> list = new ArrayList<>(gatos.entrySet());
//https://stackoverflow.com/questions/109383/sort-a-mapkey-value-by-values

ComparaGato2 criterioColor = new ComparaGato2();
list.sort(criterioColor);

System.out.println("Imprimo ordenados por color");
for (Map.Entry<Integer, Gato> entrada : list) {
    int clave = entrada.getKey();
    Gato valor = entrada.getValue();
    System.out.println(clave+"-"+valor);
}
```

## Resumen formas de pintar el contenido de un mapa

Supongamos este mapa con clave String y valor String

```
HashMap<String, String> diccionario = new HashMap<String, String>();
diccionario.put("Lunes", "Primer día de la semana");
diccionario.put("Martes", "Segundo día de la semana");
diccionario.put("Miércoles", "Tercer día de la semana");
diccionario.put("Jueves", "Cuarto día de la semana");
```

### 1. Forma perezosa

```
System.out.println(diccionario);
```

### 2. Con values

```
for (String a : diccionario.values()) {
    System.out.println(a);
}
```

### 3. Con keyset

```
for (String a : diccionario.keySet()) {
    System.out.println(a + ":" + diccionario.get(a));
}
```

### 4. Con entrySet

```
for (Map.Entry<String, String> entrada : diccionario.entrySet()) {
    //diccionario.entrySet devuelve un set<Map.Entry>
    String clave = entrada.getKey();
    String valor = entrada.getValue();
    System.out.println(clave+":"+valor);
}
```

### 5. Con programación funcional

```
diccionario.forEach((k,v)->System.out.println(k+":"+v));
```

## Ejercicios propuestos con Mapas

### Gestión de alumnos

Un alumno se identifica por nombre y un NIF. Crea un programa usando Map que contenga el menú siguiente (puedes usar el tipo de dato que quieras)

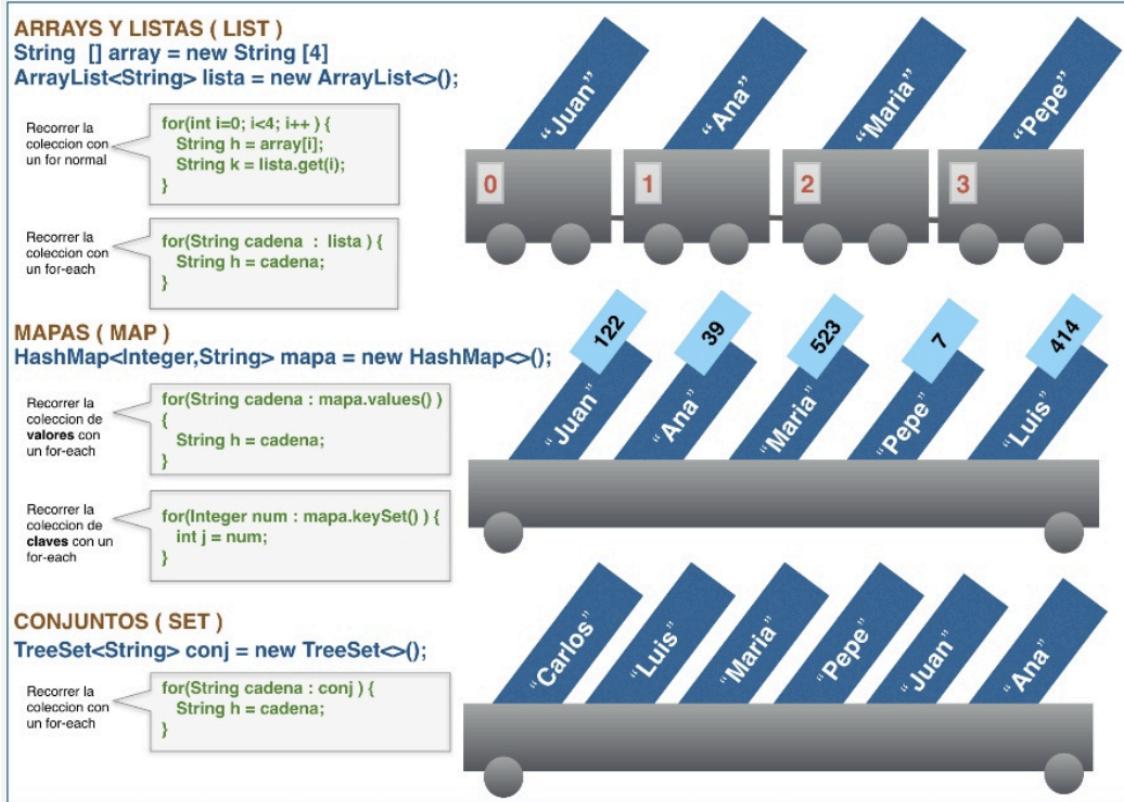
- 
- 1 – Nuevo alumno.
  - 2 – Buscar alumno por NIF
  - 3 – Buscar primer alumno por nombre
  - 4 – Buscar todos los alumnos por nombre
  - 5 – Modificar alumno
  - 6 – Borrar alumno
  - 7 – Listado de alumnos ordenado por NIF
  - 8 – Listado de alumnos ordenado por nombre
- Elija una opcion (0 para salir):

### Gestión de menú semanal

El mapa debe ser de tipo String la clave y objetos de tipo menú en el valor.  
Cada menú está formado por 1º, 2º y postre.

Se podrán añadir menús y consultar menú para un dia concreto

## Anexos



### Listas

- Una lista es una colección que almacena los elementos junto con un identificador numérico asociado a cada uno
- Las características de una Lista son:
  - ▶ El **indicador** (índice) es un **entero creciente**, muy parecido al de un array
  - ▶ Es de **tamaño variable**, no fijo como un array
  - ▶ Se puede **acceder** a los elementos (leer, añadir o borrar) por su **posición** a través del **índice**.
  - ▶ Puede **contener elementos duplicados**
  - ▶ **List** es la interfaz de la que cuelgan el resto de colecciones de lista; **ArrayList**, **LinkedList**, **Vector**, etc.

▶ Para construir una lista, se puede usar una clase "hija" de List

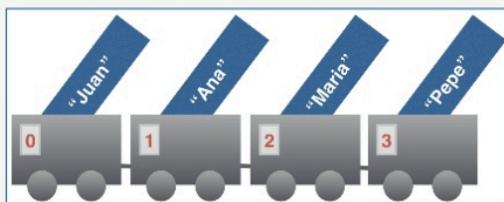
```
CLASELISTA<String> laLista2 = new CLASELISTA<String>();
```

Sustituir **CLASELISTA** por una de las clases de tipo lista

aunque normalmente se suele usar con polimorfismo (ambas son perfectamente válidas):

```
List<String> laLista2 = new CLASELISTA<String>();
```

En este segundo caso, se aplica upcasting



### Métodos más importantes de LIST (Y de todas sus clases heredadas)

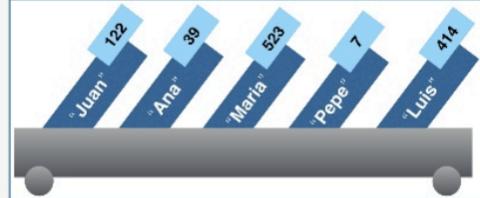
| CLASE | DEVUELVE | MÉTODO                            | DESCRIPCION  |
|-------|----------|-----------------------------------|--|
| List  | boolean  | <b>add (Object ob)</b>            | Mete un elemento al final de la lista<br><code>objLista.add("Ejemplo"); // Suponiendo que la lista contenga Strings</code>   |
| List  | void     | <b>add (int pos, Object elem)</b> | Mete un elemento en la posición indicada, moviendo el resto hacia adelante<br><code>objLista.add( 3, "Ejemplo"); // Suponiendo que la lista contenga Strings</code>  |
| List  | Object   | <b>get (int pos)</b>              | Devuelve el elemento de la posición pos. <b>Cuidado:</b> devuelve Object si ArrayList si no tiene tipo.<br><code>String res = objLista.get(0);</code>  |
| List  | object   | <b>remove (int posicion)</b>      | Elimina el elemento de la posición indicada. Devuelve dicho elemento.<br><code>objLista.remove(0);</code>  |
| List  | int      | <b>indexOf (Object ob)</b>        | Devuelve el indice de la primera ocurrencia del objeto ob en la lista, o -1 si no lo encuentra<br><code>int pos = objLista.indexOf("Ejemplo"); // Suponiendo que la lista contenga Strings</code>  |
| List  | Object   | <b>set (pos, elemento)</b>        | Sustituye el elemento de la posición pos. Devuelve error si la posición no existe<br><code>objLista.set( 1, "Otro Ejemplo"); // Suponiendo que la lista contenga Strings</code>  |
| List  | void     | <b>clear()</b>                    | Elimina los elementos de la lista<br><code>objLista.clear();</code>  |
| List  | boolean  | <b>contains (Object ob)</b>       | Informa de si un determinado contenido está en la colección. <b>Cuidado:</b> contains usa equals() para comparar, si se usan clases propias, es necesario sobreescribir equals()<br><code>boolean existe = objLista.contains("Ejemplo"); // Suponiendo una lista de Strings</code> |
| List  | int      | <b>size ()</b>                    | Devuelve el numero de elementos que tiene la<br><code>int tamano = objLista.size(); // Suponiendo una lista de Strings</code>  |

## Mapas

- En estas clases los objetos **no tienen un índice numérico**, sino que **tienen un índice popio**, que se añade junto al propio dato a almacenar en la colección.
- Este índice se llama **clave**, y hace las funciones de índice, pero no ordenado.
- La información que se almacena se denomina **valor**

- Las características de un Mapa son:

- La clave puede ser de cualquier tipo de **objeto, no tipo primitivo** (al igual que pasa con el valor)
- Es de **tamaño variable**, no fijo como un array
- Al **crear** el mapa, se debe **indicar el tipo** de la **clave**, y el tipo del **valor** almacenado en el mapa
- Se puede acceder a los elementos (leer, añadir o borrar) por su **posición** a través del **índice**.
- No** puede contener elementos con la **clave duplicada**
- Map** es la interfaz de la que cuelgan el resto de colecciones de mapas; **HashMap, TreeMap**, etc.
- Para construir una lista, se puede usar una clase "hija" de Map



```
CLASEMAPA<Integer, String> elMapa2 = new CLASEMAPA<>();
```

Sustituir CLASEMAPA por una de las clases de tipo mapa

aunque normalmente se suele usar con polimorfismo (ambas son perfectamente válidas):

```
Map<Integer, String> elMapa2 = new CLASEMAPA<>();
```

### Métodos más importantes de MAP (Y de todas sus clases heredadas)

| CLASE | DEVUELVE   | METODO                                  | DESCRIPCION   |
|-------|------------|---|---|
| Map   | int        | <b>size()</b>                           | Devuelve el tamaño de la colección.<br><code>objMapa.size();</code>   |
| Map   | object     | <b>remove(ind)</b>                      | Elimina el elemento con el índice "ind". Devuelve null si no existe el índice<br><code>objMapa.remove( 32 ); // Suponiendo que el Mapa tenga Integer en la clave</code>   |
| Map   | void       | <b>clear()</b>                          | Elimina todos los elementos de la colección<br><code>objMapa.clear();</code>  |
| Map   | boolean    | <b>containsKey (Object clave)</b>       | Indica si existe esa clave en la colección<br><code>boolean existeLaClave = objMapa.containsKey(14);</code>   |
| Map   | boolean    | <b>containsValue (Object valor)</b>     | Indica si existe ese objeto-valor entre los valores de la colección<br><code>boolean existeElValor = objMapa.containsValue("Pepe");</code>  |
| Map   | object     | <b>get (Object indice)</b>              | Devuelve el valor asociado con el índice indicado por parámetro, o null si no existe el índice<br><code>String elValor = objMapa.get(63); // Suponiendo que el mapa contenga Strings</code>   |
| Map   | object     | <b>put (Object clave, Object valor)</b> | Coloca una clave y valor en la colección. Si la clave ya existía se sobrescribe el antiguo valor. Devuelve el antiguo valor, si ya existía la clave, o null si no existía la clave<br><code>objMapa.put( 19, "Juan"); // Suponiendo que el Mapa tenga Integer y String</code> |
| Map   | Collection | <b>values()</b>                         | Devuelve una colección conteniendo únicamente los objeto-valor del mapa.<br><code>ArrayList&lt;String&gt; v = objMapa.values();</code>  |
| Map   | Set        | <b>keySet()</b>                         | Devuelve un conjunto (Set) conteniendo únicamente las objeto-clave del mapa.<br><code>TreeSet&lt;Integer&gt; v = objMapa.keys();</code>   |
| Map   | boolean    | <b>isEmpty()</b>                        | Indica si la colección está vacía<br><code>boolean estaVacioElMapa = objMapa.isEmpty();</code>  |

### TreeMap (de tipo Map)

- Un objeto **TreeMap** se crea igual que cualquier otro mapa
- Un **TreeMap** garantiza que los elementos del mismo están ordenados ascendente por la clave.

La clave se ordena ascendente (si permite un orden natural de ordenación) o si se trata de una clase, según la implementación de **Comparator** que tenga (ver mas adelante)

- Tiene muchos métodos además de los que sobrescrito de **Map**

### Métodos exclusivos de TreeMap

(aunque los más importantes son los heredados de Map)

| CLASE   | DEVUELVE | METODO                     | DESCRIPCION   |
|---------|----------|----------------------------|---|
| TreeMap | object   | <b>ceilingEntry(clave)</b> | Devuelve el valor asociado a la siguiente clave superior o igual en el orden a la pasada por parámetro, o null si no existe |
| TreeMap | object   | <b>ceilingKey (clave)</b>  | Devuelve la siguiente clave superior o igual en el orden a la pasada por parámetro, o null si no existe                     |
| TreeMap | object   | <b>floorEntry(clave)</b>   | Devuelve el valor asociado a la siguiente clave inferior o igual en el orden a la pasada por parámetro, o null si no existe |
| TreeMap | object   | <b>floorKey(clave)</b>     | Devuelve la siguiente clave inferior o igual en el orden a la pasada por parámetro, o null si no existe                     |
| TreeMap | Set      | <b>entrySet()</b>          | Devuelve como un conjunto (set) los valores de la colección.  |
| TreeMap | object   | <b>firstEntry()</b>        | Devuelve, tomando los elementos ordenados, el valor del elemento de clave mas baja.   |
| TreeMap | object   | <b>firstKey()</b>          | Devuelve, tomando los elementos ordenados, la clave mas baja (primera clave).   |
| TreeMap | object   | <b>lastEntry()</b>         | Devuelve, tomando los elementos ordenados, el valor del elemento de clave mas alta  |
| TreeMap | object   | <b>lastKey()</b>           | Devuelve, con los elementos ordenados, la clave mas alta (última clave).  |

### HashMap (de tipo Map)

- Un objeto **HashMap** se crea igual que cualquier otro mapa
- En el ejemplo, el Integer hace referencia al índice, mientras que el contenido de la tabla es un String
- En un **HashMap**, no se puede garantizar el orden de los elementos, de hecho, puede variar durante la propia ejecución.
- En un **HashMap** se permite que sea null tanto la clave como el valor.

### Métodos exclusivos de HashMap

(aunque los más importantes son los heredados de Map)

| CLASE   | DEVUELVE | METODO         | DESCRIPCION                 |
|---------|----------|----------------|-----------------------------|
| HashMap | Set      | <b>clone()</b> | Devuelve una copia del map. |

## ¿Como recorrer un Map?

- Al intentar usar `un for-each` con un mapa, se verá que **no funciona recorriendo el mapa**...¿por que?
- Un Mapa internamente **guarda dos colecciones (los indices y los valores)**. Al recorrer con un for-each, hay que decirle a este **cuál de las dos colecciones ha de recorrer**, si la de las claves o la de los valores
- Por lo tanto, no se puede recorrer directamente un mapa, **hay que crear una colección intermedia de claves o de valores**
- Esta colección intermedia **ha de ser del tipo Collection**, que es superclase de Listas y Conjuntos, con este uso:
  - \* con el método `.values()` se consigue una colección con los valores del mapa. La colección es de la clase Collection  
Ya vimos que este método devuelve un objeto de tipo `Collection` con la colección de valores del mapa, que sí tiene un índice para poder usarse en un for-each
  - \* con el método `.keySet()` se consigue una colección con los índices del mapa  
Ya vimos que este método devuelve un objeto de tipo `Set (hija de Collection)` con la colección de valores del mapa, que sí tiene un índice para poder usarse en un for-each
- Ejemplo: Partimos de este mapa ya creado:

```
Map<Integer, String> miMapa = new HashMap<>();  
miMapa.put(29, "Ana");  
miMapa.put(17, "Mercedes");  
miMapa.put(31, "Maria");
```

### ♦ Recorriendo los valores: usando `.values()`

```
// Creando un objeto Collection y usándolo en for-each  
Collection<String> losvalores = miMapa.values();  
for (String cadavalor : losvalores) {  
    System.out.println(cadavalor);  
}  
  
// Usando en el for-each directamente el método values()  
for (String cadavalor : miMapa.values()) {  
    System.out.println(cadavalor);  
}
```

consola  
Ana  
Mercedes  
Maria

### ♦ Recorriendo las claves: usando `.keySet()`

```
// Creando un objeto Collection y usándolo en for-each  
Collection<Integer> lasclaves = miMapa.keySet();  
for (Integer cadaclave : lasclaves) {  
    System.out.println(cadaclave);  
}  
  
// Usando en el for-each directamente el método keySet()  
for (Integer cadaclave : miMapa.keySet()) {  
    System.out.println(cadaclave);  
}
```

consola  
29  
17  
31

## MapEntry

- Ya se ha visto que de un mapa o se recorre la colección de valores o la de claves
- Pero, ¿y si quiero recorrer ambas colecciones a la vez? (conocer en un mismo bucle los valores y sus claves)
- Una solución es recorrer en un bucle las claves, y dentro del bucle, ir sacando los valores:

```
Map<Integer, String> miMapa = new HashMap<>();  
miMapa.put(29, "Ana");  
miMapa.put(17, "Mercedes");  
miMapa.put(31, "Maria");
```

```
for (Integer cadaclave : miMapa.keySet()) {  
    String cadavalar = miMapa.get(cadaclave);  
    System.out.print(cadavalar);  
    System.out.println(cadaclave);  
}
```

consola  
Ana29  
Mercedes17  
Maria31

- Pero hay una solución que aporta java, más efectiva, usar el método `entrySet()` en el mapa. Así se usa:
  - `entrySet()` en un mapa devuelve un conjunto de elementos llamados `Map.Entry`.
  - Cada `Map.Entry` es realmente un par de elementos (clave y valor), correspondiente a cualquier elemento del Mapa
  - De cada elemento del `Map.Entry` se puede extraer el valor con el método `getValue()` y la clave con el método `getKey()`

- Ejemplo: Analicemos como funciona `Map.Entry`, viendo los métodos necesarios al usarlo dentro de for-each;

- En la colección a recorrer, usar `entrySet()` en el mapa a recorrer
- En el elemento que entrega el for-each, poner como tipo `Map.Entry<tipoClave, tipoValor>`
- En el interior del for-each, usar con cada elemento `getKey()` (para valores) y `getValue()` (para claves):

```
// En este caso es siempre mejor usar el método entrySet directamente dentro del for-each  
for (Map.Entry<Integer, String> cadaEntry : miMapa.entrySet()) {  
    System.out.println("clave = " + cadaEntry.getKey() + ", valor = " + cadaEntry.getValue());  
}
```

consola  
Clave = 20, valor = Ana  
Clave = 10, valor = Mercedes  
Clave = 30, valor = Maria

## Ordenación natural

- Se han visto varios tipos de colecciones de datos, ademas de los arrays. Contienen información, pero ¿**pueden ordenarse los datos** de estas colecciones? ¿Puedo ordenar los datos a mi interés?
- Para poder ordenar elementos, la JVM ha de poder comparar dos elementos entre si para determinar cual es mayor de los dos, y cual menor.
- Esta comparación puede hacerse de modo automático, **depende del tipo de dato** que se quiera ordenar:
  - \* Tipos de datos **con ordenación natural**: Tipos que se pueden ordenar por un orden natural, como el que se da en los números o cadenas de texto (que se ordenan numéricamente o alfabéticamente, como los String o los int). También se incluyen aquí Date y otras clases donde Java aplica orden por defecto.
  - \* Tipos de datos **sin ordenación natural**: Si se trata de un tipo de dato creado por nosotros, como una Persona, una Tarjeta, un Coche, y se le pide a la JVM que la ordene, ¿que criterio ha de seguir?

Si le pido que ordene coches, ¿como lo ha de hacer? ¿por matricula? ¿por cilindrada? ¿por precio?

Estos tipos de datos necesitan que se defina programáticamente como se van a ordenar

### Cuidado con TreeSet y Comparator/Comparable

- Un TreeSet tiene sus elementos ordenados... ¡pero necesita poder ordenarlos !
- Si son de tipo de dato con orden natural, como int, funciona sin más, pero como añadamos objetos a un TreeSet, HEMOS de decirle con un Comparator o Comparable cómo debe ordenar los objetos !!
- El ejemplo de la derecha falla, por que no sabe como ordenar objetos de tipo Humano,
- Mas adelante en el capítulo de Comparable y Comparator se da solución a este problema

```
public class EjemploComparador {  
    public static void main (String[] args) {  
        Set<Humano> p = new TreeSet<> ();  
        p.add (new Humano ("Luis", "Ruiz")); //ERROR  
    }  
}  
  
class Humano {  
    private String nombre;  
    private String apellido;  
    public Humano (String nombre, String apellido) {  
        this.nombre = nombre;  
        this.apellido = apellido;  
    }  
}
```

<https://www.slideshare.net/JyocX/jyoc-javacap11-colecciones>

## Colecciones en Java 9

```
//Desde Java 9 ya se pueden crear las colecciones directamente con el método
//of pero genera colecciones inmutables, no deja añadir ni ordenarlas.

List<String> lista1 = List.of("Paco", "Ana", "Maria");
Set<String> set1 = Set.of("Paco", "Ana", "Maria"); //por defecto un hashset
Map<Integer, String> map1 = Map.of(1, "Paco", 2, "Ana", 3, "Maria");

lista1.forEach(System.out::println);
System.out.println("-".repeat(20));

set1.forEach(System.out::println);
System.out.println("-".repeat(20));

for (Integer a: map1.keySet())
    System.out.println(a+"-"+map1.get(a));

//Para permitir modificar y añadir elementos, hay que recurrir al constructor
//de la colección que queramos:
List<String> lista_modificable = new ArrayList<String>(List.of("Paco", "Ana", "Maria"));
lista_modificable.add("Otro");
Collections.sort(lista_modificable);
System.out.println("Lista modificable ordenada");
lista_modificable.forEach(System.out::println);

//También se pueden crear con Streams, desde Java 8
List<String> lista2 = Stream.of("Paco", "Ana", "Maria").sorted().collect(Collectors.toList());
System.out.println(lista2.getClass().getName()); //ha generado un arrayList

lista2.forEach(System.out::println);
```

## Iterator

Un iterator es como un índice que utilizamos para recorrer las colecciones. Se utiliza mucho como una forma estandarizada de recorrer las colecciones.

- Puede utilizar estos iteradores para cualquier clase de colección
- Con Iterator se puede **leer** y **eliminar** operaciones.
- Si estás usando un bucle no puedes eliminar la colección mientras que con un iterador sí se puede.

¿Cómo se definen y se utilizan los iterator sobre las colecciones?

```
List<String> list = new ArrayList<>();
list.add("Jane");
list.add("Heidi");
list.add("Hannah");

Iterator<String> iterator = list.iterator();
while(iterator.hasNext()) {
    String next = iterator.next();
    System.out.println( next );
}
```

```
Set<String> set = new HashSet();
set.add("Jane");
set.add("Heidi");
set.add("Hannah");

Iterator<String> iterator2 = set.iterator();
while(iterator2.hasNext()){
    System.out.println( iterator2.next() );
}
```

```

Map<String, String> map = new HashMap<>(); I
map.put("key1", "value1");
map.put("key2", "value2");

Iterator<String> keyIterator    = map.keySet().iterator();
Iterator<String> valueIterator = map.values().iterator();

Iterator<Map.Entry<String, String>> entryIterator = map.entrySet().iterator();

```

```

public static void main(String[] args) {
    // TODO Auto-generated method stub
    Map map = new HashMap<>();
    map.put(1, "a");
    map.put(2, "b");
    map.put(3, "c");
    // Las claves en todos los pares clave-valor forman un conjunto
    Set set = map.keySet();
    Iterator iter = set.iterator();
    while(iter.hasNext()){
        System.out.println (iter.next()); // Imprime las claves en el mapa (1,2,3)
    }

    // Imprime el valor
    // un conjunto de todos los valores
    Collection col = map.values();
    // Método anulado toString
    System.out.println (col); // Imprime los valores de a, b, c
}

```

```

// Obtenga todas las claves y valores
// entrySet puede obtener un conjunto que consta de todos los pares clave-valor
// Almacena todos los datos (clave-valor)
Set<Map.Entry<Integer, String>> entrySet=map.entrySet();
Iterator<Map.Entry<Integer, String>> iter=entrySet.iterator();
while(iter.hasNext()) {
    Map.Entry<Integer, String> entry=iter.next();
    System.out.println ("Clave:" + entry.getKey ());
    System.out.println ("Valor:" + entry.getValue ());
}

```

Si creamos la lista genérica, el iterator también lo creamos genérico, y luego dentro del bucle hacemos casting al tipo correspondiente que hayamos utilizado, en este ejemplo String.

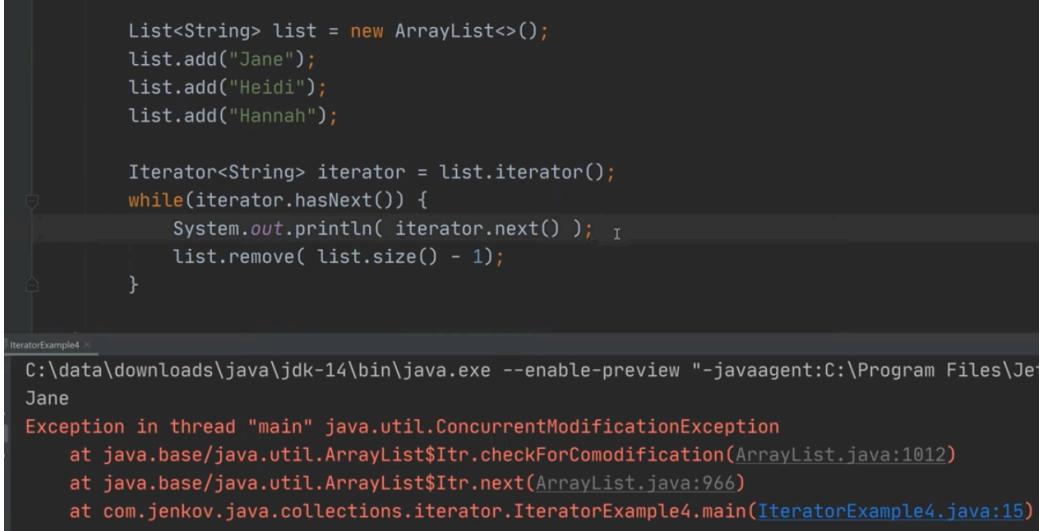
```

ListI list2 = new ArrayList();
list2.add("Jane");

Iterator iterator2 = list2.iterator();
while(iterator2.hasNext()) {
    String next = (String) iterator2.next();
}

```

No se puede modificar una colección mientras iteramos pero **sí podemos borrar utilizando el iterador**



The screenshot shows an IDE interface with a code editor and a terminal window. The code in the editor is:

```
List<String> list = new ArrayList<>();
list.add("Jane");
list.add("Heidi");
list.add("Hannah");

Iterator<String> iterator = list.iterator();
while(iterator.hasNext()) {
    System.out.println( iterator.next() );
    list.remove( list.size() - 1 );
}
```

The terminal window shows the output and an error stack trace:

```
C:\data\downloads\java\jdk-14\bin\java.exe --enable-preview "-javaagent:C:\Program Files\Java\jenkov\java\collections\iterator\IteratorExample4.jar"
Jane
Exception in thread "main" java.util.ConcurrentModificationException
        at java.base/java.util.ArrayList$Itr.checkForComodification(ArrayList.java:1012)
        at java.base/java.util.ArrayList$Itr.next(ArrayList.java:966)
        at com.jenkov.java.collections.iterator.IteratorExample4.main(IteratorExample4.java:15)
```

## Eliminar elementos utilizando un iterador

El método `Iterator.remove()` es un método opcional que elimina el elemento devuelto por la llamada anterior a `Iterator.next()`. Por ejemplo, el siguiente código rellena una lista de cadenas y luego elimina todas las cadenas vacías.

```
List<String> names = new ArrayList<>();
names.add("name 1");
names.add("name 2");
names.add("");
names.add("name 3");
names.add("");
System.out.println("Old Size : " + names.size());
Iterator<String> it = names.iterator();
while (it.hasNext()) {
    String el = it.next();
    if (el.equals("")) {
        it.remove();
    }
}
System.out.println("New Size : " + names.size());
```

Salida:

```
Old Size : 5
New Size : 3
```

El método `remove()` solo puede ser llamado (una vez) después de una llamada `next()`. Si se llama antes de llamar a `next()` o si se llama dos veces después de la llamada `next()`, entonces la llamada `remove()` lanzará una `IllegalStateException`.

Se puede listar al revés.

`ListIterator` es el iterador más poderoso pero **sólo es aplicable para clases implementadas List**. Por tanto, no es un iterador universal.

```
public static void main(String[] args) {
    List<String> list = new ArrayList<>();
    list.add("Jane");
    list.add("Heidi");
    list.add("Hannah");

    System.out.println("====");
    ListIterator<String> listIterator = list.listIterator();
    while(listIterator.hasNext()) {
        System.out.println(listIterator.next());
    }
    while(listIterator.hasPrevious()) {
        System.out.println(listIterator.previous());
    }
}
```