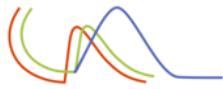


---

## COLECCIONES Y GENÉRICOS

### Tabla de contenido

INTRODUCCION .....	2
API COLLECTIONS .....	2
Jerarquía principal del API Collections.....	2
Ejemplo de uso de Set.....	3
Ejemplo de uso de List .....	4
Ejemplo de uso de Map.....	4
Ordenar Colecciones.....	5
Comparable (orden natural) .....	6
Comparator (orden personalizado) .....	6
Genéricos .....	7
Iteradores.....	8
Recomendaciones .....	8
STREAMS Y COLECCIONES .....	8
De una colección a un Stream .....	8
De un Stream a una colección .....	9
Recolectores comunes (Collectors) .....	10
Streams desde otras fuentes .....	11
Conversión inversa: Stream a Array .....	11
Resumen del uso combinado Colecciones ↔ Stream .....	11
QUÉ ES UN COLLECTOR EN JAVA.....	12
el método collect() .....	12



## INTRODUCCION

Los arrays son estructuras muy útiles en Java, pero presentan ciertas limitaciones que los hacen poco flexibles en escenarios dinámicos:

- Todos los elementos deben ser del **mismo tipo**.
- Su tamaño es **fijo** y debe definirse en el momento de su creación.
- No pueden **redimensionarse** sin crear un nuevo array.

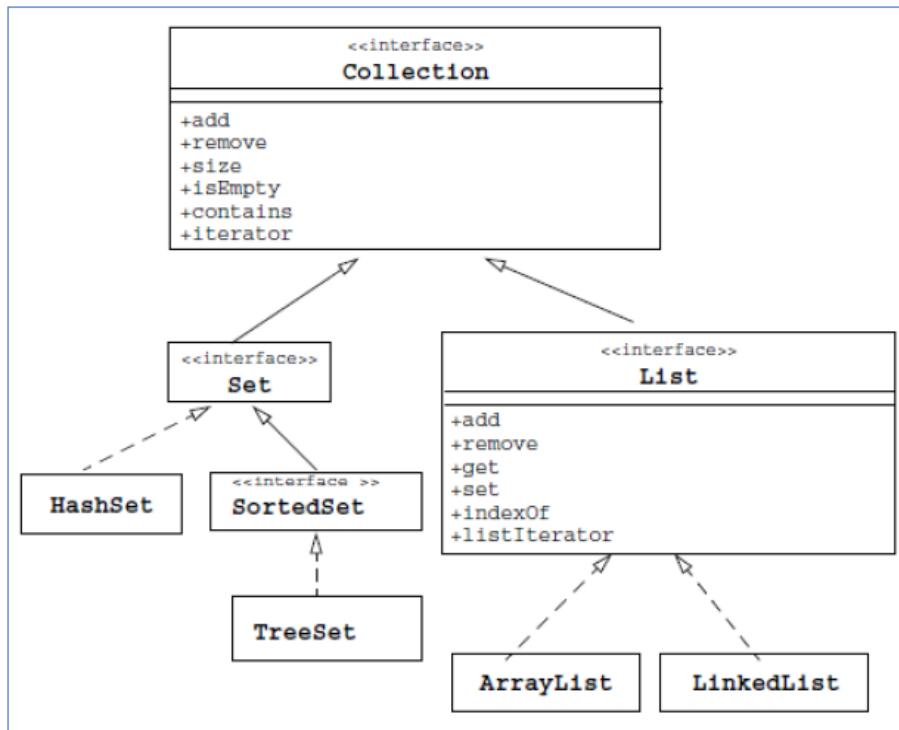
Por ello, la API de **Colecciones (Collections Framework)** proporciona un conjunto de clases e interfaces que permiten manejar grupos de objetos de forma **dinámica, flexible y tipada**.

Las **colecciones** permiten almacenar múltiples elementos dentro de una misma estructura de datos. Entre sus ventajas respecto a los arrays:

- No requieren definir un tamaño inicial.
- Son **estructuras dinámicas**: podemos añadir o eliminar elementos sin preocuparnos por la capacidad.
- Admiten distintos tipos de comportamiento (ordenado, sin duplicados, por clave-valor, etc.).
- Desde Java 5, gracias a los **genéricos**, las colecciones permiten especificar el tipo de datos que contendrán, mejorando la seguridad en tiempo de compilación.

## API COLLECTIONS

Jerarquía principal del API Collections





El Java Collections Framework (JCF) se organiza en torno a varias interfaces fundamentales:

Interfaz	Descripción
Collection<E>	Representa un grupo de elementos (el nivel más general).
List<E>	Colección ordenada que permite elementos duplicados.
Set<E>	Colección que no permite duplicados y no garantiza el orden.
SortedSet<E> / NavigableSet<E>	Subinterfaces de Set que mantienen los elementos ordenados.
Queue<E> / Deque<E>	Representan colas y colas dobles, útiles en estructuras FIFO/LIFO.
Map<K,V>	Colección de pares clave-valor, sin claves duplicadas. No hereda de Collection.

### Implementaciones más utilizadas

Tipo	Implementación común	Características
List	ArrayList, LinkedList	Ordenadas, permiten duplicados.
Set	HashSet, LinkedHashSet, TreeSet	Sin duplicados; algunas mantienen orden.
Queue / Deque	ArrayDeque, LinkedList	Para colas FIFO o pilas LIFO.
Map	HashMap, LinkedHashMap, TreeMap	Almacenan pares clave-valor.

### Ejemplo de uso de Set

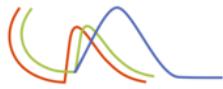
Un Set no permite elementos duplicados:

```
import java.util.*;

public class EjemploSet {
    public static void main(String[] args) {
        Set<String> nombres = new HashSet<>();
        nombres.add("Ana");
        nombres.add("Luis");
        nombres.add("Ana"); // Ignorado

        System.out.println("Contenido del Set: " + nombres);
    }
}
```

Salida posible:  
Contenido del Set: [Luis, Ana] (el orden puede variar)



## Ejemplo de uso de List

Una List mantiene el orden de inserción y permite duplicados:

```
java

import java.util.*;

public class EjemploList {
    public static void main(String[] args) {
        List<String> frutas = new ArrayList<>();
        frutas.add("Manzana");
        frutas.add("Pera");
        frutas.add("Manzana");

        for (String f : frutas) {
            System.out.println(f);
        }
    }
}
```

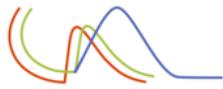
Salida:

```
nginx

Manzana
Pera
Manzana
```

## Ejemplo de uso de Map

Un Map almacena pares *clave → valor*. No puede haber claves duplicadas.



```
java

import java.util.*;

public class EjemploMap {
    public static void main(String[] args) {
        Map<Integer, String> mapa = new HashMap<>();
        mapa.put(1, "Java");
        mapa.put(2, "Python");
        mapa.put(1, "C++"); // Sobrescribe el valor anterior

        for (var entrada : mapa.entrySet()) {
            System.out.println(entrada.getKey() + " → " + entrada.getValue());
        }
    }
}
```

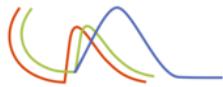
Salida:

```
mathematica
```

```
1 → C++
2 → Python
```

## Ordenar Colecciones

En Java podemos ordenar colecciones utilizando las interfaces **Comparable** y **Comparator**.



---

### Comparable (orden natural)

```
public class Cliente implements Comparable<Cliente> {  
    private String nombre;  
    private double ventas;  
  
    public Cliente(String nombre, double ventas) {  
        this.nombre = nombre;  
        this.ventas = ventas;  
    }  
  
    @Override  
    public int compareTo(Cliente otro) {  
        return Double.compare(this.ventas, otro.ventas);  
    }  
  
    @Override  
    public String toString() {  
        return nombre + " (" + ventas + ")";  
    }  
}  
  
import java.util.*;  
  
public class EjemploComparable {  
    public static void main(String[] args) {  
        Set<Cliente> clientes = new TreeSet<>();  
        clientes.add(new Cliente("Ana", 2000));  
        clientes.add(new Cliente("Luis", 5000));  
        clientes.add(new Cliente("Marta", 1500));  
  
        System.out.println(clientes);  
    }  
}
```

### Comparator (orden personalizado)

Permite definir múltiples formas de ordenar sin modificar la clase original:



```
import java.util.*;  
  
public class EjemploComparador {  
    record Cliente(String nombre, String nif, double ventas) {}  
  
    public static void main(String[] args) {  
        List<Cliente> clientes = List.of(  
            new Cliente("Ana", "123A", 2000),  
            new Cliente("Luis", "456B", 5000),  
            new Cliente("Marta", "789C", 1500)  
        );  
  
        // Ordenar por nombre  
        clientes.stream()  
            .sorted(Comparator.comparing(Cliente::nombre))  
            .forEach(System.out::println);  
  
        // Ordenar por ventas descendentes  
        clientes.stream()  
            .sorted(Comparator.comparing(Cliente::ventas).reversed())  
            .forEach(System.out::println);  
    }  
}
```

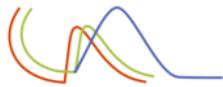
## Genéricos

Los **genéricos** permiten definir clases, interfaces y métodos que operan sobre **tipos parametrizados**, garantizando seguridad de tipos y evitando conversiones de tipo (*casting*).

Ejemplo básico:

```
List<String> nombres = new ArrayList<>();  
nombres.add("Carlos");  
// nombres.add(123); // Error de compilación  
  
for (String nombre : nombres) {  
    System.out.println(nombre.toUpperCase());  
}
```

Ejemplo con Map genérico:



```
java

Map<Integer, String> cursos = new HashMap<>();
cursos.put(1, "Java");
cursos.put(2, "Spring Boot");

String curso = cursos.get(1); // No requiere casting
```

## Iteradores

Un **Iterator** permite recorrer los elementos de una colección de forma segura:

```
List<String> nombres = List.of("Ana", "Luis", "Carlos");

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

Desde Java 5, el bucle **for-each** simplifica este proceso:

```
java

for (String nombre : nombres) {
    System.out.println(nombre);
}
```

## Recomendaciones

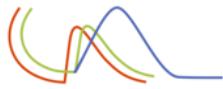
- Usa `List.of()`, `Set.of()` y `Map.of()` (Java 9+) para crear colecciones inmutables.
- Prefiere `var` para inferencia de tipos local (Java 10+).
- Utiliza **Streams** (`stream()`, `filter()`, `map()`, `sorted()`) para un manejo funcional y expresivo de colecciones.
- No confundas **Collections** (la clase utilitaria) con el **framework de colecciones** (interfaces y clases concretas).

## STREAMS Y COLECCIONES

A partir de **Java 8**, el paquete `java.util.stream` introduce una API que permite procesar datos de colecciones (y otras fuentes) de forma **declarativa y eficiente**.

Un **Stream** no es una colección, sino una *secuencia de elementos* que se pueden **filtrar, transformar, ordenar o reducir** sin modificar la colección original.

De una colección a un Stream



---

Cualquier colección del framework (List, Set, Map, etc.) puede transformarse fácilmente en un Stream mediante el método `.stream()` o `.parallelStream()`.

```
import java.util.*;
import java.util.stream.*;

public class EjemploStreamDesdeColeccion {
    public static void main(String[] args) {
        List<String> nombres = List.of("Ana", "Luis", "Carlos", "Lucia");

        // Convertir la colección a Stream y procesarla
        nombres.stream()
            .filter(n -> n.length() > 3)
            .map(String::toUpperCase)
            .sorted()
            .forEach(System.out::println);
    }
}
```

Descripción:

- `.filter(...)` selecciona elementos según una condición.
- `.map(...)` transforma cada elemento.
- `.sorted()` ordena los resultados.
- `.forEach(...)` aplica una acción terminal (en este caso, imprimir).

*Recuerda:* El método `.stream()` devuelve un **Stream secuencial**, mientras que `.parallelStream()` devuelve un **Stream paralelo** que puede aprovechar múltiples núcleos del procesador.

#### De un Stream a una colección

Una vez procesado el *stream*, podemos **recolectar** los resultados en una nueva colección utilizando los **Collectors** de la clase `java.util.stream.Collectors`.

Ejemplo básico:



java

```
import java.util.*;
import java.util.stream.*;

public class EjemploStreamAColeccion {
    public static void main(String[] args) {
        List<String> nombres = List.of("Ana", "Luis", "Carlos", "Lucía");

        // Filtrar y convertir el resultado a una nueva lista
        List<String> resultado = nombres.stream()
            .filter(n -> n.startsWith("L"))
            .collect(Collectors.toList());

        System.out.println("Resultado: " + resultado);
    }
}
```

Salida:

```
Resultado: [Luis, Lucía]
```

### Recoletores comunes (Collectors)

Método	Descripción	Tipo devuelto
Collectors.toList()	Convierte el Stream en una List.	List<E>
Collectors.toSet()	Convierte el Stream en un Set (sin duplicados).	Set<E>
Collectors.toMap(k -> clave, v -> valor)	Convierte el Stream en un Map.	Map<K,V>
Collectors.joining(", ")	Une los elementos en un String.	String
Collectors.groupingBy(...)	Agrupa elementos por una clave.	Map<K, List<E>>
Collectors.partitioningBy(...)	Separa en dos grupos según una condición booleana.	Map<Boolean, List<E>>
Collectors.counting()	Cuenta los elementos.	Long



Salida:

css

```
A → [Ana, Alberto]  
B → [Bea]  
C → [Carlos, Carmen]
```

### Streams desde otras fuentes

Además de las colecciones, los Streams pueden crearse desde otras fuentes:

```
// Desde un array  
Stream<String> streamArray = Arrays.stream(new String[]{"Java", "Python", "C++"});  
  
// Desde valores directos  
Stream<Integer> streamNumeros = Stream.of(10, 20, 30);  
  
// Desde archivos (usando NIO)  
try (Stream<String> lineas = java.nio.file.Files.lines(java.nio.file.Path.of("datos.txt"))) {  
    lineas.forEach(System.out::println);  
} catch (Exception e) {  
    e.printStackTrace();  
}
```

### Conversión inversa: Stream a Array

Si en lugar de colecciones necesitamos un array, también podemos hacerlo fácilmente:

```
java  
  
String[] array = nombres.stream()  
    .filter(n -> n.length() > 3)  
    .toArray(String[]::new);
```

### Resumen del uso combinado Colecciones ↔ Stream

Conversión	Método / Ejemplo
Colección → Stream	list.stream()
Stream → List	.collect(Collectors.toList())
Stream → Set	.collect(Collectors.toSet())



Conversión	Método / Ejemplo
Stream → Map	.collect(Collectors.toMap(k -> ..., v -> ...))
Stream → Array	.toArray(String[]::new)
Stream paralelo	.parallelStream()

## QUÉ ES UN COLLECTOR EN JAVA

Un **Collector** es un **objeto que define cómo un Stream debe ser reducido o recolectado en una estructura final** (por ejemplo, una List, un Set, un Map, o incluso un único valor).

El **Collector** forma parte de la clase utilitaria `java.util.stream.Collectors`, que incluye muchos métodos estáticos para construir diferentes tipos de *collectors*.

el método `collect()`

Todo **Stream** en Java tiene un método terminal llamado `.collect(Collector<T, A, R>)`.

Su propósito es transformar el Stream en una **colección, mapa, cadena o valor agregado**.