
UT 8 – COLECCIONES DE DATOS

Contenido

Jerarquías de colecciones	1
Tipos genéricos o parametrizados	2
Clases con parámetros genéricos	3
Interfaces con genéricos.....	4
Interfaz Comparator	4
Comodines.....	7
Métodos básicos de la interfaz Collection	7
Iterador en Java	8
Tipos de colecciones.	10
Listas (interfaz List)	10
Conjuntos (interfaz Set)	10
Colas (interfaz Queue).....	11
Mapas (interfaz Map).....	11
Referencias.....	11

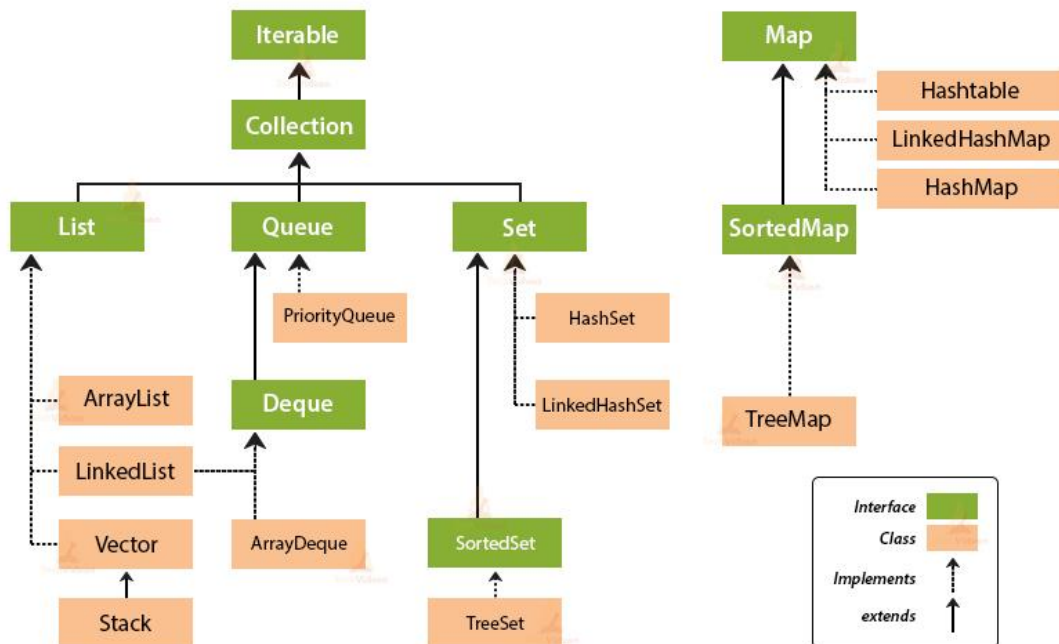
Jerarquías de colecciones

A menudo necesitamos guardar información, pero no sabemos de antemano el espacio que va a ocupar en memoria.

Ya sabemos que los arrays no valen para esto, porque se crean para guardar un número determinado de elementos, y su tamaño no puede modificarse.

Lo que necesitamos en este caso son **estructuras dinámicas de datos**, es decir, objetos que guardan datos que se pueden ir insertando y eliminando, cambiando de tamaño en tiempo de ejecución. Estas estructuras dinámicas comparten un conjunto de métodos declarados en la interfaz [Collection](#). Todas ellas implementan dicha interfaz, aunque de distinta forma.

Collection Framework Hierarchy in Java



No existe una clase colección, sino todo un marco de trabajo (*framework*) con una estructura jerárquica de interfaces que se implementan en distintas clases según lo que se necesite hacer en cada una.

Lo que tienen en común es que se utilizan para agrupar y almacenar objetos para después manipularlos mediante operaciones como inserción, eliminación, búsqueda u ordenación.

El conjunto de interfaces y clases del framework Collection está en el paquete `java.util`.

Tipos genéricos o parametrizados

En el framework Collection se trabaja con tipos de datos genéricos o parametrizados, que son clases, interfaces o métodos en los que el tipo de datos sobre los que operan se especifica como parámetro, y se indica mediante el operador de *diamante* `<>`.

El tipo de dato a pasar por parámetro puede ser de cualquier clase o interfaz, pero nunca un tipo primitivo.

Se suele utilizar la letra T para el tipo genérico, pero puede ser cualquier otra, aunque es costumbre reservar la letra E para elementos de colecciones, K para claves, V para valores o N para números.

Clases con parámetros genéricos

Por ejemplo, queremos definir la clase Contenedor, que permite guardar un solo objeto de cualquier tipo. Tendrá un atributo objeto del tipo que metamos por parámetro, y dos métodos: guardar() y extraer():

```
public class Contenedor<T> {  
    private T objeto;  
  
    public Contenedor() {  
    }  
    public void guardar(T objeto) {  
        this.objeto = objeto;  
    }  
    public T extraer() {  
        return objeto;  
    }  
}
```

Ahora creamos varios objetos:

Por ejemplo, uno de tipo Integer:

```
public static void main(String[] args) {  
    Contenedor<Integer> numero = new Contenedor<Integer>();  
    System.out.println(numero.extraer());  
    numero.guardar(123);  
    System.out.println(numero.extraer());  
}
```

Podemos pasar como parámetro cualquiera de las clases con las que hemos trabajado en en los temas anteriores. Por ejemplo, si tuviéramos una clase Paciente con un atributo nombre, podríamos crear un objeto contenedor de este tipo:

```
Contenedor<Paciente> paciente = new Contenedor<Paciente>();  
paciente.guardar(new Paciente());  
System.out.println(paciente.extraer().nombre);
```

En la implementación de una clase puede haber más de un tipo genérico; en este caso se especifican los parámetros separados por comas.

```
class NombreClase<T, U...> {  
}
```

Interfaces con genéricos

También se pueden definir interfaces con tipos genéricos y la sintaxis es idéntica:

```
public interface InterfazGenérica<T> {  
    void métodoAbstracto();  
    // etc.  
}
```

Y aquí, en vez de ponernos a crear interfaces genéricas de ejemplo, vamos a aprender a utilizar una interfaz genérica de la API de Java: **Comparator**.

Interfaz Comparator

Es frecuente que tengamos que ordenar objetos de la misma clase con distintos criterios en el mismo programa. Para esto sirve la interfaz Comparator.

Antes de utilizarla hay que importarla

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

Contiene un único método abstracto:

```
int compare(Object o1, Object o2);
```

que recibe como parámetros dos objetos para determinar cuál va antes y cuál después en un proceso de ordenación. Devuelve un entero, que será:

- Negativo, si objeto1 va antes que objeto2.
- Positivo, si objeto 1 va después que objeto2.
- 0, si ambos objetos son iguales.

Llamamos *comparador* a cualquier objeto de una clase que implemente la interfaz Comparator.

Igualmente, tendremos que hacer casting (conversión de tipos) para los objetos Object pasados por parámetro.

Cuando queramos una ordenación en sentido decreciente, la interfaz Comparator ya lleva implementado un método para ello: `reversed()`.

Veamos un ejemplo de cómo utilizar esta interfaz.

Supongamos que tenemos una clase Socio, con los siguientes atributos: número identificativo del socio, nombre y fecha de nacimiento, y un constructor parametrizado. También queremos un método `toString()` personalizado.

Suponemos también que **queremos ordenar a los socios por por fecha de nacimiento.**

Implementamos una clase que implementa Comparator. Me obligará a desarrollar el método abstracto `compare`, y ahí es donde indicamos que el atributo `fechaNacimiento` (aprovechamos también los métodos `isBefore` e `isAfter` de la clase `LocalDate`).

```

public class ComparaFNacimiento implements Comparator {
    @Override
    public int compare(Object o1, Object o2) {
        // Lo primero que hacemos es un casting a Socio
        Socio socio1 = (Socio) o1;
        Socio socio2 = (Socio) o2;
        // ahora seguimos la regla:
        // si socio1 va antes que socio2, que devuelva -1
        // si es al revés, que devuelva 1
        // si son iguales, que devuelva 0
        if (socio1.fechaNacimiento.isBefore(socio2.fechaNacimiento)) {
            return -1;
        } else if (socio1.fechaNacimiento.isAfter(socio2.fechaNacimiento)) {
            return 1;
        } else {
            return 0;
        }
    }
}

```

Con esto ya tenemos una herramienta para ordenar socios por fecha de nacimiento (como ejercicio para practicar, crea otras dos clases comparadoras: una para ordenar por número de socio y otra para ordenar por nombre).

Vamos a probarla con los objetos de un ArrayList de Socios (con Comparator se puede utilizar el método sort() de ArrayList): creamos un objeto de esta clase y se la pasamos como parámetro al método sort():

```

Socio socio1 = new Socio(3, "Caleb", LocalDate.of(2000, 12, 1));
Socio socio2 = new Socio(2, "Benito", LocalDate.of(1999, 1, 1));
Socio socio3 = new Socio(1, "Aladina", LocalDate.of(1943, 6, 14));
Socio socio4 = new Socio(4, "Doris", LocalDate.of(2012, 12, 13));

```

```

ArrayList<Socio> socios = new ArrayList<>();

```

```

socios.add(socio1);
socios.add(socio2);
socios.add(socio3);
socios.add(socio4);

```

Vamos primero a mostrar los socios tal cual han entrado en el ArrayList.

```

for (Socio elemento: socios) {
    System.out.println(elemento);
}

```

Resultado:

```

Socio n° 3: Caleb (1-12-2000)
Socio n° 2: Benito (1-1-1999)
Socio n° 1: Aladina (14-6-1943)
Socio n° 4: Doris (13-12-2012)

```

Ahora creamos un objeto del tipo de la clase que nos va a servir para ordenar:

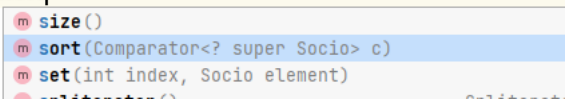
```

ComparaFNacimiento comparaFNacimiento = new ComparaFNacimiento();

```

Y para utilizarlo (y que ordene los elementos) le pasamos este objeto al método sort()

```
// queremos ordenar por fecha de nacimiento
ComparaFNacimiento comparaFNacimiento = new ComparaFNacim
socios.s
```



```
socios.sort(comparaFNacimiento);
```

Resultado (se ordenan por fecha de nacimiento):

```
Socio nº 1: Aladina (14-6-1943)
Socio nº 2: Benito (1-1-1999)
Socio nº 3: Caleb (1-12-2000)
Socio nº 4: Doris (13-12-2012)
```

Ahora vamos a ordenar en orden descendente de fecha de nacimiento:

```
Comparator comparaFNacimientoDesc = comparaFNacimiento.reversed();
socios.sort(comparaFNacimientoDesc);
```

Resultado:

```
Socio nº 4: Doris (13-12-2012)
Socio nº 3: Caleb (1-12-2000)
Socio nº 2: Benito (1-1-1999)
Socio nº 1: Aladina (14-6-1943)
```

Completa este ejemplo/ejercicio con lo siguiente:

- Haz lo necesario para ordenar los socios por el atributo nombre, de forma ascendente y descendente. Muestra un ejemplo de utilización.
- Haz lo necesario para ordenar los socios por el número de socio, ascendente y descendente. Muestra un ejemplo de utilización.
- Añade un menú para gestionar esto, de forma que el usuario pueda elegir cómo quiere ordenar los socios.

Otro ejemplo de utilización de la interfaz Comparator lo tenemos en el siguiente enlace:

[CU00918C interface Comparator api java metodo compare ejemplo ejercicio.pdf \(aprenderaprogramar.com\)](#)

Comodines

Los comodines o wildcards se utilizan en la declaración de atributos, variables locales o parámetros pasados a una función. Se representan con el símbolo "?", que significa "cualquier tipo". Por ejemplo, la expresión:

```
Contenedor<?> c;
```

Declara una variable de la clase parametrizada Contenedor, cuyo parámetro puede ser de cualquier tipo. Esto implica que todos los objetos Contenedor son alguna subclase de Contenedor<?> (es como si fuera la superclase en la jerarquía de Contenedor).

En este sentido, podemos encontrar expresiones que significarían:

<? extends T>	significa	"cualquier clase que herede de T, incluyendo a esta"
<? super T>	significa	"T o cualquier superclase de T"

Por ejemplo, la expresión:

```
Contenedor<? extends Number> c;
```

declara un objeto de tipo Contenedor de cualquier clase numérica. Podría ser Contenedor<Integer>, Contenedor<Double>, etc.

Métodos básicos de la interfaz Collection

<https://docs.oracle.com/en/java/javase/20/docs/api/java.base/java/util/Collection.html>

Varios de ellos los hemos visto y utilizado con ArrayList. Otros los veremos más adelante, cuando tratemos la interfaz Stream.

boolean **add**(**E** e) : Añade el elemento especificado a la colección.

Boolean **addAll**(**Collection**<? extends **E**> c) : Agrega todos los elementos de la colección especificada a esta colección.

void **clear**() : Elimina todos los elementos de la colección.

boolean **contains**(**Object** o) : Devuelve true si la colección contiene el elemento especificado.

boolean **containsAll**(**Collection**<?> c) : Devuelve true si la colección contiene todos los elementos de la colección especificada.

boolean **equals**(**Object** o) : Compara el objeto especificado con la colección para ver si son iguales. (RECUERDA QUE HEMOS VISTO CÓMO SOBREScribir ESTE MÉTODO PARA QUE COMPARE POR EL ATRIBUTO QUE QUERAMOS)

int **hashCode**() : Devuelve el valor del código hash para la colección.

boolean **isEmpty**() : Devuelve true si la colección no contiene elementos.

Iterator<**E**> **iterator**() : Devuelve un iterador sobre los elementos de la colección.

boolean remove(Object o) : Elimina una sola instancia del elemento especificado de la colección, si está presente.

boolean removeAll(Collection<?> c) : Elimina todos los elementos de la colección que también están contenidos en la colección especificada.

default boolean removeIf(Predicate<? super E> filter) : Elimina todos los elementos de la colección que satisfacen el predicado dado.

boolean retainAll(Collection<?> c) : Retiene solo los elementos de la colección que están contenidos en la colección especificada.

int size() : Devuelve el número de elementos de esta colección.

default Stream<E> stream() : Devuelve un secuencial Stream con esta colección como fuente.

Object[] toArray() : Devuelve un array que contiene todos los elementos de esta colección.

default <T> T[] toArray(IntFunction<T[]> generator) : Devuelve una matriz que contiene todos los elementos de esta colección, utilizando la generatorfunción proporcionada para asignar la matriz devuelta.

<T> T[] toArray(T[] a) : Devuelve una matriz que contiene todos los elementos de esta colección; el tipo de tiempo de ejecución de la matriz devuelta es el de la matriz especificada.

Iterador en Java

Un iterador es un objeto que sirve para recorrer una colección. Al ser un objeto, proporciona una serie de métodos que nos permitir hacer más cosas que las que proporciona un bucle for (sin tener que pensar demasiado).

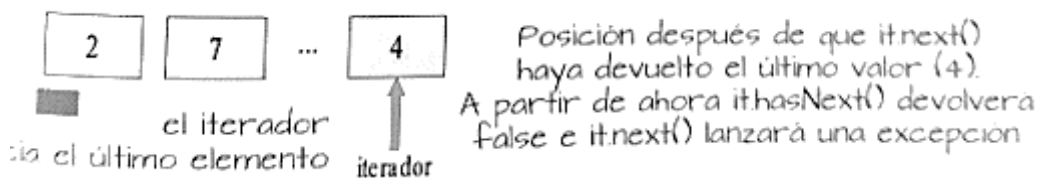
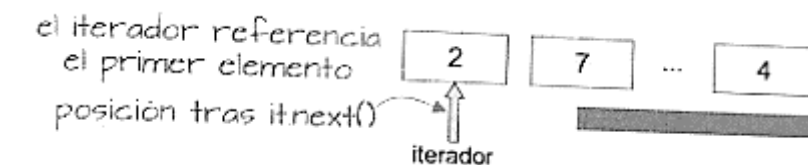
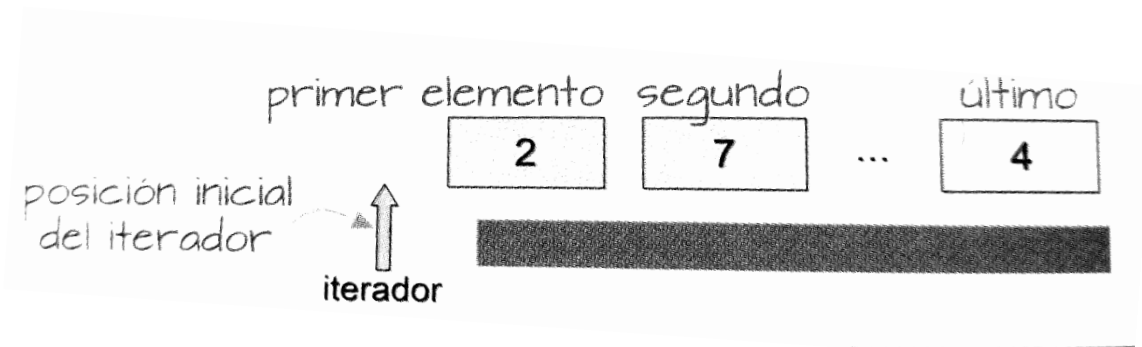
Para crear un iterador debemos invocarlo a través de un objeto de algún tipo de colección.

Por ejemplo, si queremos recorrer nuestro ArrayList de socios, creamos el iterador con el método iterator() (necesitaremos importar la interfaz Iterator del paquete java.util) y, a partir de aquí, con el objeto creado podemos recorrer la colección apoyándonos en sus métodos.

```
Iterator<Socio> iterator = socios.iterator();
```

iterator.

(m) <u>hasNext</u> ()	boolean
(m) <u>next</u> ()	Socio
(m) <u>forEachRemaining</u> (Consumer<? super Socio> action)	void
(m) <u>equals</u> (Object obj)	boolean
(m) <u>remove</u> ()	void



boolean hasNext() : Devuelve true si la iteración tiene más elementos.

E next() : Devuelve el siguiente elemento de la iteración.

default void remove() : Elimina de la colección el último elemento devuelto por el iterador.

default void forEachRemaining(Consumer<? super E> action) : Realiza la acción dada para cada elemento restante hasta que se hayan procesado todos los elementos o la acción genere una excepción.

En la práctica, los métodos next() y hasNext() se utilizan conjuntamente, de forma que sólo se llama al método next() si antes se ha comprobado con hasNext() que hay un siguiente elemento. Si no lo hacemos así y llamamos a next() estando al final de la colección se lanzará la excepción NoSuchElementException.

Ejemplo: vamos a recorrer el arraylist de socios, mostrando cada uno de ellos.

```
while (iterator.hasNext()) {
    Socio s = iterator.next(); // avanzamos al siguiente elemento
    System.out.println(s);
}
```

Tipos de colecciones.

Los tipos fundamentales de estructuras dentro del framework Collections son tres:

Listas (interfaz List)

Las listas almacenan información en un orden determinado. A diferencia de los conjuntos (Set), sí admite elementos duplicados. Aparte de los métodos heredados de Collection, tiene métodos propios (muchos de ellos ya los hemos utilizado en ArrayList, que es un tipo de Lista): add(E elem), add(int index, E elem), addAll(int index, Collection<? extends E> c), get(int index), set(int index, E elem), remove(int index), indexOf(Object o), lastIndexOf(Object o), equals(Object otraLista), sort(Comparator<? Super E> c)....

Existen dos tipos de listas:

- **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. Ejemplo:

```
ArrayList<Socio> lista = new ArrayList<Socio>();
```

- **LinkedList:** es una lista doblemente enlazada de los elementos, teniendo cada uno de los elementos un puntero al anterior y al siguiente elemento. Ejemplo:

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

El cuándo usar una implementación u otra de List variará en función de la situación en la que nos encontremos. Generalmente, ArrayList será la implementación que usemos en la mayoría de situaciones.

Conjuntos (interfaz Set)

Un conjunto es una colección que **no puede contener elementos duplicados**. Esta interfaz contiene, únicamente, los métodos heredados de Collection añadiendo la restricción de que los elementos duplicados están prohibidos. Esto implica que si intentamos insertar en un conjunto un elemento que ya existe, no nos dejará.

Como la interfaz Set no se puede instanciar, utilizaremos la clase HashSet cuando queramos definir un conjunto:

- **HashSet:** almacena los elementos en una tabla hash. Es la implementación más empleada debido a su rendimiento y a que, generalmente, no nos importa el orden que ocupen los elementos.

```
HashSet<Socio> set = new HashSet<Socio>();
```

Otros tipos de conjuntos son TreeSet o LinkedHashSet, pero no vamos a trabajar con ellos en este curso.

Colas (interfaz Queue)

Son estructuras de tipo FIFO (First In, First Out), de forma que solamente se pueden añadir elementos, que se ubicarán al principio de la cola, y extraer o consultar el elemento que está al final.

Mapas (interfaz Map)

Los mapas o diccionarios son estructuras dinámicas cuyos elementos, llamados *entradas* (objetos del tipo Map.Entry) son pares Clave-Valor en vez de valores individuales.

De los diversos tipos de mapas que hay, vamos a utilizar:

- **HashMap:** admite claves que son objetos, y como valor otros objetos. El orden viene dado por la función hashCode() de los objetos almacenados.

```
HashMap<Integer, Socio> map = new HashMap<Integer, Socio>();
```

Los mapas son una estructura especial muy utilizada. Para aprender a trabajar con ellos vamos a meternos en estos enlaces:

- [Métodos principales de HashMap en Java.](#)
- [Formas de recorrer un HashMap.](#)
- [Utilizando la clase Map.Entry \(EN\)](#)

Referencias

Jiménez Martín, Alfonso y Pérez Montes, Francisco Manuel (2021). *Programación*. Editorial Paraninfo.