

CURSO DE PROGRAMACIÓN FULL STACK

# COLECCIONES

PARADIGMA ORIENTADO A OBJETOS



# GUÍA DE COLECCIONES

## COLECCIONES

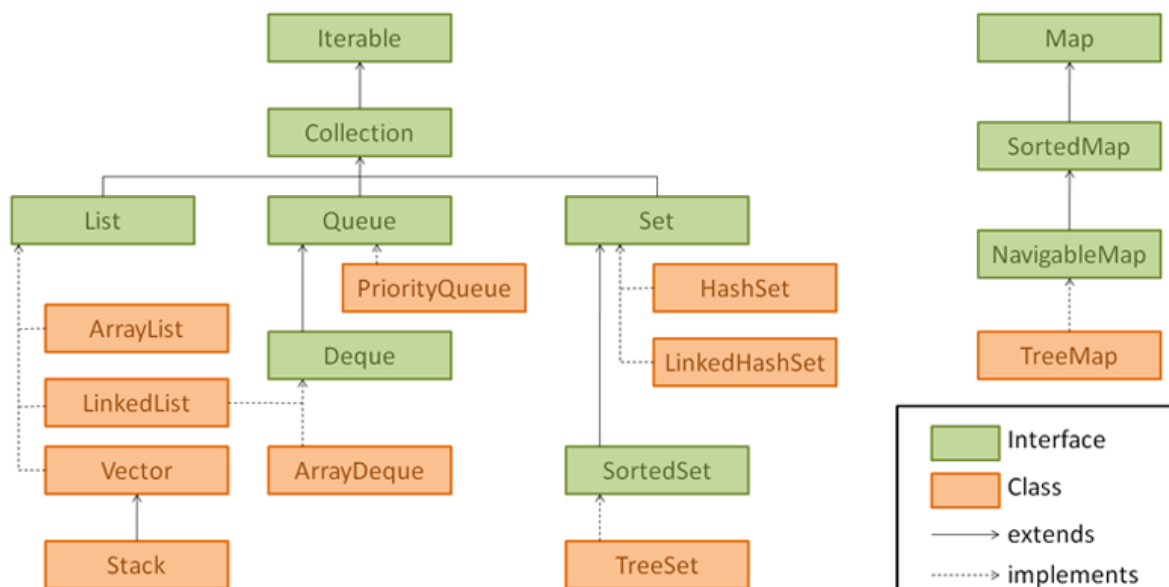
Previo a esta guía nosotros manejábamos nuestros objetos de uno en uno, no teníamos manera de manejar varios objetos a la vez, para esto existen las colecciones.

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. Las colecciones nos dan la opción de almacenar cualquier tipo de objeto, esto incluye los objetos de tipo de datos. Por lo que, para crear una colección de un tipo de dato, no podremos usar los datos primitivos de los tipos de datos, sino sus objetos. Por ejemplo: en vez de **int**, hay que utilizar **Integer**.

Las colecciones nos proveen con una serie de métodos comunes, para trabajar con los elementos de la colección, como, por ejemplo: agregar y eliminar elementos o obtener el tamaño de la colección, etc. Las colecciones son una especie de arrays de tamaño dinámico. Estas son para del Java Collections Framework dentro del paquete `java.util`. El Collections Framework es una arquitectura compuesta de interfaces y clases. Dentro de este framework están las colecciones que vamos a trabajar, las listas, conjuntos y mapas. **Nota:** el concepto de interfaces lo vamos a explicar más adelante.

## ¿QUÉ ES UN FRAMEWORK?

Un framework es un marco de trabajo el cual contiene un conjunto estandarizado de conceptos, prácticas y criterios para hacer frente a un tipo de problemática particular y resolver nuevos problemas de índole similar.



## LISTAS

Las listas modelan una colección de objetos ordenados por posición. La principal diferencia entre las listas y los arreglos tradicionales, es que la lista crece de manera dinámica a medida que se van agregando objetos. No necesitamos saber de antemano la cantidad de elementos con la que vamos a trabajar. El framework trae varias implementaciones de distintos tipos de listas tales como ArrayList, LinkedList.

- ArrayList: se implementa como un arreglo de tamaño variable. A medida que se agregan más elementos, su tamaño aumenta dinámicamente. Es el tipo más común.

**Ejemplo de un ArrayList de numeros:**

```
ArrayList<Integer> numeros = new ArrayList();
```

- LinkedList: se implementa como una [lista de doble enlace](#). Su rendimiento al agregar y quitar es mejor que ArrayList, pero peor en los métodos set y get.

**Ejemplo de una LinkedList de numeros:**

```
LinkedList<Integer> numeros = new LinkedList();
```

## CONJUNTOS

Los conjuntos modelan una colección de objetos de una misma clase donde cada elemento aparece sólo una vez, no puede tener duplicados, a diferencia de una lista donde los elementos podían repetirse. El framework trae varias implementaciones de distintos tipos de conjuntos:

- HashSet, se implementa utilizando una [tabla hash](#) para darle un valor único a cada elemento y de esa manera evitar los duplicados. Los métodos de agregar y eliminar tienen una complejidad de tiempo constante por lo que tienen mejor performance que el TreeSet.

**Ejemplo de un HashSet de cadenas:**

```
HashSet<String> nombres = new HashSet();
```

- TreeSet se implementa utilizando una [estructura de árbol](#) (árbol rojo-negro en el libro de algoritmos). La gran diferencia entre el HashSet y el TreeSet, es que el TreeSet mantiene todos sus elementos de manera ordenada (forma ascendente), pero los métodos de agregar, eliminar son más lentos que el HashSet ya que cada vez que le entra un elemento debe posicionarlo para que quede ordenado.

**Ejemplo de un TreeSet de numeros:**

```
TreeSet<Integer> numeros = new TreeSet();
```

- LinkedHashMap está entre HashSet y TreeSet. Se implementa como una tabla hash con una lista vinculada que se ejecuta a través de ella, por lo que proporciona el orden de inserción.

Ejemplo de un `LinkedHashSet` de cadenas:

```
LinkedHashSet<String> frases = new LinkedHashSet();
```

## MAPAS

Los mapas modelan un objeto a través de una llave y un valor. Esto significa que cada valor de nuestro mapa, va a tener una llave única para representar dicho valor. Las llaves de nuestro mapa no pueden repetirse, pero los valores sí. Un ejemplo sería una persona que tiene su dni/rut (llave única) y como valor puede ser su nombre completo, puede haber dos personas con el mismo nombre, pero distinto dni/rut.

A la hora de crear un mapa tenemos que pensar que el primer tipo de dato será el valor de la llave y el segundo el valor del valor (valga la redundancia).

Son una de las estructuras de datos importantes del Framework de Collections. Las implementaciones de mapas son `HashMap`, `TreeMap`, `LinkedHashMap` y `HashTable`.

- `HashMap` es un mapa implementado a través de una tabla hash, las llaves se almacenan utilizando un algoritmo de hash para las llaves y evitar que se repitan.

Ejemplo de un `HashMap` de personas:

```
HashMap<Integer,String> personas = new HashMap();
```

- `TreeMap` es un mapa que ordena los elementos de manera ascendente a través de las llaves.

Ejemplo de un `TreeMap` de personas:

```
TreeMap<Integer,String> personas = new TreeMap();
```

- `LinkedHashMap` es un `HashMap` que conserva el orden de inserción.

Ejemplo de un `TreeMap` de personas:

```
LinkedHashMap<Integer,String> personas = new LinkedHashMap();
```

## AÑADIR UN ELEMENTO A UNA COLECCIÓN

Las colecciones constan con funciones para realizar distintas operaciones, en este caso si queremos añadir un elemento a las listas o sets vamos a tener que utilizar la función `add(T)`, pero para los mapas vamos a utilizar la función `put(llave,valor)`.

**Listas:**

```
ArrayList<Integer> numeros = new ArrayList();
```

```
int num = 20;
```

```
numeros.add(num);
```

### Conjuntos:

```
HashSet<Integer> numeros = new HashSet();  
  
int num = 20;  
  
numeros.add(20);
```

### Mapas:

```
HashMap<Integer, String> alumnos = new HashMap();  
  
int dni = 34576189;  
  
String nombreAlumno = "Pepe"  
  
alumnos.put(dni, nombreAlumno);
```

## RECORRER UNA COLECCIÓN

Si quisiéramos mostrar todos los elementos que le hemos agregado y que componen nuestra colección vamos a tener que recorrerla.

Para recorrer una colección, vamos a tener que utilizar el bucle `forEach`. El bucle comienza con la palabra clave `for` al igual que un bucle *for* normal. En lugar de declarar e inicializar una variable contador del bucle, declara una variable vacía que es del mismo tipo que la colección, seguido de dos puntos y seguido del nombre de la colección. La variable recibe en cada iteración un elemento de la colección, de esa manera si nosotros mostramos esa variable, podemos mostrar todos los elementos de nuestra colección.

Para recorrer mapas vamos a tener que usar el objeto `Map.Entry` en el `for each`. A través de el `entry` vamos a traer los valores y las llaves, si no podemos tener un `for each` para cada parte de nuestro mapa sin utilizar el objeto `Map.Entry`.

Para saber más sobre la clase `Map` y el objeto `Entry`: [Map.Entry](#)

### For Each:

```
for ( Tipo de dato variable : Colección){  
  
}
```

### Listas:

```
ArrayList<String> lista = new ArrayList();  
  
for (String cadena : lista) {  
  
    System.out.println(cadena); // mostramos los elementos a traves de la variable  
  
}
```

### Conjuntos:

```
HashSet<Integer> numerosSet = new HashSet();

for (Integer numero : numerosSet) {

    System.out.println(numero); // mostramos los elementos a traves de la variable

}
```

### Mapas:

```
HashMap<Integer, String> estudiantes = new HashMap();

// Recorrer las dos partes del mapa

for (Map.Entry<Integer, String> entry : estudiantes.entrySet()) {

    System.out.println("documento=" + entry.getKey() + ", nombre=" +
        entry.getValue());

    // entry.getKey trae la llave y entry.getValue trae los valores del mapa

    // Mostrar solo las llaves

    for (Integer dni : estudiantes.keySet()) {

        System.out.println("Documento = " + dni);

    }

    // Mostrar solo los valores

    for (String nombres : estudiantes.values()) {

        System.out.println("Nombre: " + nombres);

    }

}
```

## ITERATOR

El **Iterator** es una interfaz que pertenece al **framework de colecciones**. Este, nos permite recorrer, acceder a la información y eliminar algún elemento de una colección. Ya que, cuando queremos eliminar algún elemento, mediante el `for each`, nos va a tirar un error. Gracias a que el **Iterator** es parte de el framework de colecciones, todas las colecciones vienen con un metodo **iterator()**, este devuelve un iterador para recorrer esa colección, este iterador que devuelve la colección, lo recibe el objeto **iterator** y nos deja iterar sobre nuestra colección.

Para poder usar el **Iterator** es importante crear el objeto de tipo **Iterator**, con el mismo tipo de dato que nuestra colección.

El **iterator** contiene tres metodos muy utiles para lograr esto:

1. **boolean hasNext()**: Retorna verdadero si al **iterator** le quedan elementos por iterar

2. **Object next()**: Devuelve el siguiente elemento en la coleccion, mientras le metodo hasNext() retorne true. Este metodo es el que nos sirve para mostrar el elemento,
3. **void remove()**: Elimina el elemento actual de la coleccion.

#### Ejemplo:

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

lista.add("A");
lista.add("B");

// Iterator para recorrer la lista
Iterator iterator = lista.iterator(); Devolvemos el iterador

System.out.println("Elementos de la lista : ");

// Usamos un while para recorrer la lista, siempre que el hasNext()
// devuelva true.

while (iterator.hasNext())

// Mostramos los elementos con el iterator.next()

    System.out.print(iterator.next() + " ");

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

## ELIMINAR UN ELEMENTO DE UNA COLECCIÓN

Como pudimos ver más arriba para eliminar un elemento de una colección vamos a tener que utilizar la funcion **remove()** del Iterator, esto se aplica para el resto de nuestras colecciones. Para los mapas, vamos a utilizar el Map. Entry para usar con el iterator. Además, podemos elegir sobre si queremos preguntar por la llave o por el valor, para validar que elemento eliminar.

#### Conjuntos:

```
HashSet<Integer> numerosSet = new HashSet();

Iterator<Integer> it = hashEnteros.iterator();

while (it.hasNext()) {

if (it.next() == 3) { // Borramos si está el numero 3

it.remove();

}

}
```

### Mapas:

Con los mapas vamos a tener que usar el `remove` propio de los Mapas pero no podemos recorrerlos, este `remove`, solo sirve para eliminar elementos del mapa según su llave.

```
HashMap<Integer, String> estudiantes = new HashMap();

Int llaveABorrar = 123;

estudiantes.remove(llaveABorrar)

}
```

## ORDENAR UNA COLECCIÓN

Los elementos, que vamos agregando a nuestra colección se van mostrar según se fueron agregando y nosotros capaz necesitemos mostrar o tener todos los elementos ordenados. Para ordenar una colección, vamos a tener que utilizar la función

**`Collections.sort(coleccion)`**. La función recibe la colección y la ordena para después poder mostrarla ordenada de manera ascendente.

Algunas colecciones, como los conjuntos o los mapas no pueden utilizar el `sort()`. Ya que por ejemplo los `HashSet`, manejan valores Hash y el `sort()` no sabe ordenar por hash, si no por elementos. Por otro lado, los mapas al tener dos datos, el `sort()` no sabe por cual de los datos ordenar.

Entonces, para ordenar los conjuntos, deberemos convertirlos a listas para poder ordenar esa lista por sus elementos. Y a la hora de ordenar un mapa como tenemos dos datos para ordenar, vamos a convertir el `HashMap` a un `TreeMap`.

**Nota:** recordemos que los `TreeSet` y `TreeMap` se ordenan por si mismos.

### Listas:

```
ArrayList<Integer> numeros = new ArrayList();

Collections.sort(numeros);
```

### Conjuntos:

```
HashSet<Integer> numerosSet = new HashSet();

ArrayList<Integer> numerosLista = new ArrayList(numerosSet); Se convierte el
HashSet a Lista.

Collections.sort(numerosLista);
```

### Mapas:

```
HashMap<Integer, String> alumnos = new HashMap();

// Se convierte el HashMap a TreeMap

TreeMap<Integer, String> alumnosTree = new TreeMap(alumnos);
```



# COLECCIONES CON OBJETOS

De la misma manera que podemos crear colecciones con los tipos de datos de Java, podemos crear colecciones, de algún objeto que hayamos creado previamente. Esto, nos serviría para manejar varios objetos al mismo tiempo y acceder a ellos de una manera más sencilla.

## AÑADIR UN OBJETO A UNA COLECCIÓN

Para añadir un objeto a una colección tenemos que primero crear el objeto que queremos trabajar y después crear una colección donde su tipo de dato sea dicho objeto.

La manera de agregar los objetos a la colección es muy parecida a lo que habíamos visto previamente.

A excepción de las colecciones **Tree**, ya sean TreeSet o TreeMap. Ya que al tratar de agregar elementos un Tree que su tipo de dato sea algún objeto creado por nosotros. Y los Tree, siendo colecciones que se ordenan a si mismas, debemos informarle al Tree como va a ordenarse. Pensemos que el objeto posee más de un dato(atributos), entonces el Tree, no sabe por qué atributo va a ordenarse. Para solucionar esto vamos a necesitar un Comparator que le de la pauta de como ordenarse y sobre que atributo. El Comparator está explicado más abajo en la guía y muestra como agregárselo a los Tree.

### Listas:

```
ArrayList<Libro> libros = new ArrayList();  
  
Libro libro = new Libro();  
  
libros.add(libro);
```

### Conjuntos:

```
HashSet<Perro> perros = new HashSet();  
  
Perro perro = new Perro();  
  
perros.add(perro);
```

### Mapas:

```
HashMap<Integer, Alumno> alumnos = new HashMap();  
  
int dni = 34576189;  
  
Alumno alumno = new Alumno("Pepe", "Honguito");  
  
alumnos.put(dni, alumno);
```

## RECORRER UNA COLECCIÓN CON OBJETOS

Para recorrer una colección donde su tipo de dato sea un objeto creado por nosotros, vamos a seguir utilizando los métodos que conocemos, el for each o el iterator. Pero a la hora de mostrar el objeto con un System.out.println, no nos va a mostrar sus atributos. Sino que, nos va a mostrar el nombre de la clase, el nombre del objeto, una arroba y un código hash para representar los valores del objeto.

### Ejemplo:

```
ArrayList<Libro> libros = new ArrayList();  
Libro libro = new Libro();  
libros.add(libro);  
for (Libro libro : libros) {  
    System.out.println(libro);  
}
```

Cuando queremos mostrar el libro, que está siendo recorrido por el for each, nos mostraria algo asi: Libreria.Libro@14ae5a5

Para solucionar este problema, vamos a tener que sobrescribir(Override), un metodo de la clase String, dentro de la clase de nuestro objeto. Este metodo va a transformar, el nombre de la clase, el nombre del objeto y el hash, en una cadena legible para imprimir. Para poder usar este metodo vamos a ir a nuestra clase, ahí hacemos click derecho, insert code y le damos a **toString()**. Eso nos va a generar un metodo **toString()** con los atributos de nuestro objeto y que retorna una cadena para mostrar el objeto.

### Ejemplo:

```
@Override  
public String toString() {  
    return "Libro{" + "titulo=" + titulo + '}';  
}
```

Este metodo se va a llamar solo, sin necesidad que lo llamemos nosotros, siempre que querramos mostrar nuestro objeto en un System.out.println. Y mostrará la linea que se ve en el return.

### Ejemplo:

```
ArrayList<Libro> libros = new ArrayList();  
Libro libro = new Libro();  
libro.setTitulo("La Odisea");  
libros.add(libro);  
for (Libro libro : libros) {  
    System.out.println(libro);  
}
```

Output: Libro{titulo= La Odisea}

# COMPARATOR

A la hora de querer ordenar una colección de objetos en Java, no podemos utilizar la función `sort`, ya que el `sort` se utiliza para ordenar colecciones con elementos uniformes. Pero los objetos pueden tener dentro distintos tipos de datos (atributos). Entonces, nuestra función `sort` no sabe por qué tipo de dato o atributo ordenar. Para esto, utilizamos la interfaz **Comparator** con su función **compare** en nuestra clase entidad.

Supongamos que tenemos una clase `Perro`, que tiene como atributos el nombre del perro y la edad. Nosotros queremos ordenar los perros por edad, deberemos crear el método `compare` de la clase `Comparator` en la clase `Perro`.

## Ejemplo:

Public Class `Perro`

```
public static Comparator<Perro> compararEdad = new Comparator<Perro>() {  
  
    @Override  
  
    public int compare(Perro p1, Perro p2) {  
  
        return p2.getEdad().compareTo(p1.getEdad());  
  
    }  
  
};  
}
```

## Explicación del método:

- El método crea un objeto estático de la interfaz `Comparator`. Este nos va permitir utilizar a través de un sobre escribir (`Override`) el método `compare`, el mismo nos deja comparar dos objetos para poder ordenarlos. Este objeto se crea `static` para poder llamar al método solo llamando a la clase, sin tener que crear otro objeto `Comparator`, en este caso la clase `Perro`.
- Dentro de la creación de objeto se crea un método de la clase `Comparator` llamado `compare`, arriba del método se puede ver la palabra `Override`. `Override`, se usa cuando desde una subclase (`Perro`), queremos utilizar un método de otra clase (`Comparator`) en nuestra subclase.
- El método recibe dos objetos de la clase `Perro` y retorna una comparación entre los dos usando los `get` para traer el atributo que queríamos comparar y usa la función `compareTo`, que devuelve 0 si la edad es la misma, 1 si la primera edad es mayor a la segunda y -1 si la primera edad es menor a la segunda.
- Si quisiéramos cambiar el atributo que usa para ordenar, pondríamos otro atributo en el `get` del `return`.

## USO DEL METODO COMPARATOR

Como el comparator se va a usar para ordenar nuestras colecciones, se va a poner en el llamado de la función Collections.sort() y se va a poner en la inicialización de cualquier tipo de Tree.

### Listas:

```
ArrayList<Perro> perros = new ArrayList();

perros.sort(Perro.compararEdad); Se llama al metodo estatico a traves de la
clase y se pone la lista a ordenar.
```

### Conjuntos:

```
HashSet<Perro> perrosSet = new HashSet();

ArrayList<Perro> perrosLista = new ArrayList(perrosSet);

perrosLista.sort(Perro.compararEdad);
```

### Crear un TreeSet

En los TreeSet necesitamos crearlos con el comparator porque como el TreeSet se ordena solo, necesitamos decirle al TreeSet, bajo que atributo se va a ordenar, por eso le pasamos el comparator en el constructor.

```
TreeSet<Perro> perros = new TreeSet(Perro.compararEdad);

Perro perro = new Perro();

perros.add(perro);
```

### Mapas:

```
HashMap<Integer, Alumno> alumnos = new HashMap();

ArrayList<Alumno> nombres = new ArrayList(map.values()); Se usa una función de
los mapas para traer todos valores.

nombres.sort(Alumno.compararDni);
```

## COLECCIONES EN FUNCIONES

A la hora de querer pasar una colección a una función, deberemos recordar que Java es fuertemente tipado, por lo que deberemos poner el tipo de dato de la colección y que tipo de colección es cuando la pongamos como argumento.

### Listas:

```
Public void llenarLista(ArrayList<Integer> numeros){

numeros.add(20)

}
```

Main

```
ArrayList<Integer> numeros = new ArrayList();  
llenarLista(numeros); // Le pasamos la lista a la función
```

### Conjuntos:

```
Public void llenarHashSet(HashSet<String> palabras){  
    palabras.add("Hola")  
}
```

Main

```
HashSet<String> palabras = new HashSet();  
llenarHashSet(palabras); // Le pasamos el conjunto a la función
```

### Mapas:

```
Public void llenarMapa(HashMap<Integer, String> alumnos){  
    alumnos.add(1, "Pepe");  
}
```

Main

```
HashMap<Integer, String> alumnos = new HashMap();  
llenarMapa(alumnos); // Le pasamos el conjunto a la función
```

## DEVOLVER UNA COLECCIÓN EN FUNCIONES

Para devolver una colección en una función, tenemos que hacer que el tipo de dato de nuestra función sea la colección que queremos devolver, teniendo también el tipo de dato que va a manejar dicha colección.

### Listas:

```
Public ArrayList<Integer> llenarLista(){  
    ArrayList<Integer> numeros = new ArrayList();  
    numeros.add(20);  
    return numeros; // Devolvemos la lista llena.  
}
```

### Conjuntos:

```
Public HashSet<String> llenarHashSet(){  
    HashSet<String> palabras = new HashSet();  
    palabras.add("Hola")  
    return palabras }
```

### Mapas:

```
Public HashMap<Integer,String> llenarMapa(){  
    HashMap<Integer, String> alumnos = new HashMap();  
    alumnos.add(1, "Pepe");  
    return alumnos;  
}
```

### Proyecto con ejemplos de colecciones

El proyecto contiene un paquete con un main para cada tipo de colección y un paquete más que muestra como usar una lista con un objeto. Además contiene algunos métodos que no están explicados en la teoría.

Link: [Colecciones](#)

## CLASE COLLECTIONS

La clase collections es parte del framework de colecciones y también es parte del paquete java.util. Esta clase nos provee de métodos que reciben una colección y realizan alguna operación o devuelven una colección, según el método. Vamos a mostrar algunos de los métodos pero hay muchos más.

Método	Descripción.
fill(List<T> lista, Objeto objeto)	Este método reemplaza todos los elementos de la lista con un elemento específico.
frequency(Collection<T> coleccion, Objeto objeto)	Este método retorna la cantidad de veces que se encuentra un elemento específico en una colección.
replaceAll(List<T> lista, T valorViejo, T valorNuevo)	Este método reemplaza todas las apariciones de un elemento específico en una lista, con otro valor.
reverse(List<T> lista)	Este método invierte el orden de los elementos de una lista.
reverseOrder()	Este método retorna un comparator que invierte el orden de los elementos de una colección.
shuffle(List<T> lista)	Este método modifica la posición de los elementos de una lista de manera aleatoria.
sort(List<T> lista)	Este método ordena los elementos de una lista de manera ascendente.

## METODOS EXTRAS COLECCIONES

En la guía mostramos las acciones más realizadas con colecciones, con la ayuda de sus métodos, pero también existen otros métodos en las colecciones para realizar otras acciones. **Nota:** los métodos de los List y los Set, son los mismos, quitando el get y el set.

### Listas y Conjuntos:

Método	Descripción.
size()	Este método retorna el tamaño de una lista / conjunto.
clear()	Este método se usa para remover todos los elementos de una lista / conjunto.
remove(int índice)	Este método remueve un elemento de un índice específico. Esto mueve los elementos de manera que no queden índices sin elementos,

<code>remove(elemento)</code>	Este método remueve un elemento de una lista
<code>get(int índice)</code>	Este método retorna un elemento de la lista según un índice de la lista.
<code>set(int índice, elemento)</code>	Este método guarda un elemento en la lista en un índice específico.
<code>isEmpty()</code>	Este método retorna verdadero si la lista / conjunto está vacía y falso si no lo está.
<code>contains(elemento)</code>	Este método recibe un elemento dado por el usuario y revisa si el elemento se encuentra en la lista o no. Si el elemento se encuentra retorna verdadero y si no falso.

### Mapas:

<b>Método</b>	<b>Descripción.</b>
<code>clear()</code>	Este método se usa para remover todos los elementos de un mapa.
<code>containsKey(Llave)</code>	Este método recibe una llave dada por el usuario y revisa si la llave se encuentra en la lista o no. Si la llave se encuentra retorna verdadero y si no falso.
<code>containsValue(Valor)</code>	Este método recibe un valor dado por el usuario y revisa si el valor se encuentra en el mapa o no. Si el elemento se encuentra retorna verdadero y si no falso.
<code>get(Llave)</code>	Este método retorna un elemento del mapa según una llave dentro del mapa.
<code>isEmpty()</code>	Este método retorna verdadero si el mapa está vacío y falso si no lo está.
<code>size()</code>	Este método retorna el tamaño de un mapa.
<code>values()</code>	Este método crea una colección según los valores del mapa. Ósea, que retorna una lista, por ejemplo, con todos los valores del mapa.
<code>clear()</code>	Este método se usa para remover todos los elementos de un mapa.



# PREGUNTAS DE APRENDIZAJE

- 1) Cual de estos paquetes es el contenedor de las colecciones:
  - a) java.lang
  - b) java.util
  - c) java.net
  - d) java.awt
  
- 2) Las colecciones en Java son:
  - a) Un grupo de objetos.
  - b) Un grupo de clases.
  - c) Un grupo de interfaces.
  - d) Ninguna de las anteriores.
  
- 3) En el framework de colecciones de Java un Set es
  - a) Una colección que no puede contener elementos duplicados
  - b) Una colección ordenada que puede contener elementos duplicados
  - c) Un objeto que mapea conjuntos de clave valor y no puede contener valores duplicados
  - d) Ninguna de las anteriores
  
- 4) Cual, de estos métodos, borra un elemento de una colección:
  - a) .clear();
  - b) .delete();
  - c) .remove();
  - d) .reset();
  
- 5) Para agregar elementos en una lista se usa la función:
  - a) lista.lenght();
  - b) lista.size();
  - c) lista.add();
  - d) lista.iterator();
  
- 6) En Java un Iterator es:
  - a) Una interface que proporciona los métodos para borrar elementos de una colección
  - b) Una interface que proporciona los métodos para recorrer los elementos de una colección y posibilita el borrado de elementos
  - c) Una interface que proporciona los métodos para ordenar los elementos de la colección.
  - d) Ninguna de las anteriores

# EJERCICIOS DE APRENDIZAJE

En este módulo vamos a continuar modelando los objetos con el lenguaje de programación Java, pero ahora vamos a utilizar las colecciones para poder manejarlas de manera más sencilla y ordenada.

VER VIDEOS:

- A. [Introducción](#)
- B. [Listas 1](#)
- C. [Listas 2](#)
- D. [Listas 3](#)

1. Diseñar un programa que lea y guarde razas de perros en un ArrayList de tipo String. El programa pedirá una raza de perro en un bucle, el mismo se guardará en la lista y después se le preguntará al usuario si quiere guardar otro perro o si quiere salir. Si decide salir, se mostrará todos los perros guardados en el ArrayList.

VER VIDEO: [Iterar o Recorrer Colecciones](#)

2. Continuando el ejercicio anterior, después de mostrar los perros, al usuario se le pedirá un perro y se recorrerá la lista con un Iterator, se buscará el perro en la lista. Si el perro está en la lista, se eliminará el perro que ingresó el usuario y se mostrará la lista ordenada. Si el perro no se encuentra en la lista, se le informará al usuario y se mostrará la lista ordenada.
3. Crear una clase llamada Alumno que mantenga información sobre las notas de distintos alumnos. La clase Alumno tendrá como atributos, su nombre y una lista de tipo Integer con sus 3 notas.  
En el main deberemos tener un bucle que crea un objeto Alumno. Se pide toda la información al usuario y ese Alumno se guarda en una lista de tipo Alumno y se le pregunta al usuario si quiere crear otro Alumno o no.

Después de ese bucle tendremos el siguiente método en la clase Alumno:

Método notaFinal(): El usuario ingresa el nombre del alumno que quiere calcular su nota final y se lo busca en la lista de Alumnos. Si está en la lista, se llama al método. Dentro del método se usará la lista notas para calcular el promedio final de alumno. Siendo este promedio final, devuelto por el método y mostrado en el main.

Ejemplo de una colección como atributo: [ColeccionAtributo](#)

VER VIDEO: [Comparator](#)

4. Un cine necesita implementar un sistema en el que se puedan cargar películas. Para esto, tendremos una clase Película con el título, director y duración de la película (en horas). Implemente las clases y métodos necesarios para esta situación, teniendo en cuenta lo que se pide a continuación:

En el main deberemos tener un bucle que crea un objeto Pelicula pidiéndole al usuario todos sus datos y guardándolos en el objeto Pelicula.

Después, esa Pelicula se guarda una lista de Peliculas y se le pregunta al usuario si quiere crear otra Pelicula o no.

Después de ese bucle realizaremos las siguientes acciones:

- Mostrar en pantalla todas las películas.
- Mostrar en pantalla todas las películas con una duración mayor a 1 hora.
- Ordenar las películas de acuerdo a su duración (de mayor a menor) y mostrarlo en pantalla.
- Ordenar las películas de acuerdo a su duración (de menor a mayor) y mostrarlo en pantalla.
- Ordenar las películas por titulo, alfabéticamente y mostrarlo en pantalla.
- Ordenar las películas por director, alfabéticamente y mostrarlo en pantalla.

Nota: recordar el uso del Comparator para ordenar colecciones con objetos.

VER VIDEO: [Conjuntos](#)

5. Se requiere un programa que lea y guarde países, y para evitar que se ingresen repetidos usaremos un conjunto. El programa pedirá un país en un bucle, se guardará el país en el conjunto y después se le preguntará al usuario si quiere guardar otro país o si quiere salir, si decide salir, se mostrará todos los países guardados en el conjunto.

Después deberemos mostrar el conjunto ordenado alfabéticamente para esto recordar como se ordena un conjunto.

Y por ultimo, al usuario se le pedirá un país y se recorrerá el conjunto con un Iterator, se buscará el país en el conjunto y si está en el conjunto se eliminará el país que ingresó el usuario y se mostrará el conjunto. Si el país no se encuentra en el conjunto se le informará al usuario.

VER VIDEO: [Mapas](#)

6. Se necesita una aplicación para una tienda en la cual queremos almacenar los distintos productos que venderemos y el precio que tendrán. Además, se necesita que la aplicación cuente con las funciones básicas.

Estas las realizaremos en el main. Como, introducir un elemento, modificar su precio, eliminar un producto y mostrar los productos que tenemos con su precio (Utilizar Hashmap). El HashMap tendrá de llave el nombre del producto y de valor el precio. Realizar un menú para lograr todas las acciones previamente mencionadas.

## EJERCICIOS DE APRENDIZAJE EXTRA

Estos van a ser ejercicios para reforzar los conocimientos previamente vistos. Estos pueden realizarse cuando hayas terminado la guía y tengas una buena base sobre lo que venimos trabajando. Además, si ya terminaste la guía y te queda tiempo libre en las mesas, puedes continuar con estos ejercicios extra, recordando siempre que no es necesario que los termines para continuar con el tema siguiente. Por último, recordá que la prioridad es ayudar a los compañeros de la mesa y que cuando tengas que ayudar, lo más valioso es que puedas explicar el ejercicio con la intención de que tu compañero lo comprenda, y no sólo mostrarlo. ¡Muchas gracias!

1. Diseñar un programa que lea una serie de valores numéricos enteros desde el teclado y los guarde en un ArrayList de tipo Integer. La lectura de números termina cuando se introduzca el valor -99. Este valor no se guarda en el ArrayList. A continuación, el programa mostrará por pantalla el número de valores que se han leído, su suma y su media (promedio).
2. Crear una clase llamada CantanteFamoso. Esta clase guardará cantantes famosos y tendrá como atributos el nombre y discoConMasVentas. Se debe, además, en el main, crear una lista de tipo CantanteFamoso y agregar 5 objetos de tipo CantanteFamoso a la lista. Luego, se debe mostrar los nombres de cada cantante y su disco con más ventas por pantalla. Una vez agregado los 5, en otro bucle, crear un menú que le de la opción al usuario de agregar un cantante más, mostrar todos los cantantes, eliminar un cantante que el usuario elija o de salir del programa. Al final se deberá mostrar la lista con todos los cambios.
3. Implemente un programa para una Librería haciendo uso de un HashSet para evitar datos repetidos. Para ello, se debe crear una clase llamada Libro que guarde la información de cada uno de los libros de una Biblioteca. La clase Libro debe guardar el título del libro, autor, número de ejemplares y número de ejemplares prestados. También se creará en el main un HashSet de tipo Libro que guardará todos los libros creados. En el main tendremos un bucle que crea un objeto Libro pidiéndole al usuario todos sus datos y los seteandolos en el Libro. Después, ese Libro se guarda en el conjunto y se le pregunta al usuario si quiere crear otro Libro o no.

La clase Librería contendrá además los siguientes métodos:

- Constructor por defecto.
- Constructor con parámetros.
- Métodos Setters/getters
- Método prestamo(): El usuario ingresa el título del libro que quiere prestar y se lo busca en el conjunto. Si está en el conjunto, se llama con ese objeto Libro al método. El método se incrementa de a uno, como un carrito de compras, el atributo ejemplares prestados, del libro que ingresó el usuario. Esto sucederá cada vez que se realice un préstamo del libro. No se podrán prestar libros de los que no queden ejemplares disponibles para prestar. Devuelve true si se ha podido realizar la operación y false en caso contrario.

- Método `devolucion()`: El usuario ingresa el título del libro que quiere devolver y se lo busca en el conjunto. Si está en el conjunto, se llama con ese objeto al método. El método decrementa de a uno, como un carrito de compras, el atributo `ejemplares prestados`, del libro seleccionado por el usuario. Esto sucederá cada vez que se produzca la devolución de un libro. No se podrán devolver libros donde que no tengan ejemplares prestados. Devuelve `true` si se ha podido realizar la operación y `false` en caso contrario.
- Método `toString` para mostrar los datos de los libros.

4. Almacena en un `HashMap` los códigos postales de 10 ciudades a elección de esta página: <https://mapanet.eu/index.htm>. Nota: Poner el código postal sin la letra, solo el número.

- Pedirle al usuario que ingrese 10 códigos postales y sus ciudades.
- Muestra por pantalla los datos introducidos
- Pide un código postal y muestra la ciudad asociada si existe sino avisa al usuario.
- Muestra por pantalla los datos
- Agregar una ciudad con su código postal correspondiente más al `HashMap`.
- Elimina 3 ciudades existentes dentro del `HashMap`, que pida el usuario.
- Muestra por pantalla los datos