ALL COLLECTIONS

ÍNDICE

1. ¿Qué es una colección?

Las **colecciones** son estructuras similares a los arrays pero con la característica de que son dinámicos (su tamaño y cantidad elementos puede variar en el tiempo).

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 sub-interfaces aportan distintas funcionalidades sobre la interfaz anterior.

En **JAVA**, las colecciones se emplean mediante la interfaz "**Collection**", permite implementar una serie de métodos comunes como: añadir, borrar, tamaño de la colección…

En **JAVA** disponemos de un **framework** en el cuál nos permite almacenar o recuperar objetos de cualquier clase. Dichas colecciones están dentro del paquete **(java.util).**

Estas colecciones nos permiten:

- Fomenta la reutilización de software.
- Aumentar la calidad y velocidad del programa
- Permite la interoperabilidad (compartición de datos e intercambio de información) con librerías de terceros.
- Reducir el esfuerzo de programación.
- Reduce el esfuerzo de aprender y usar otras librerías

Forma de utilizarla (sintaxis):

1. Pondremos de que será el tipo de colección que deseamos usar.
2. Luego entre mayor y menor que, pondremos la clase de objetos que vamos a almacenar.
3. Pondremos el mejor nombre que nosotros consideremos.
4. Y por último pondremos el típico constructor con el mismo nombre entre el mayor y el menor que.

```
Coleccion<Clase> nombre = new Coleccion<Clase>();
```

Ejemplo:

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

- Cuando ponemos el tipo de colección que deseemos en eclipse nos dará un error, tenemos que importar nuestro paquete de java.util.

2. Tipos de colecciones

2.1 Interfaz List

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.

Implementaciones a la interfaz List:

- **ArrayList** → Es una implementación de List basada en un array dinámico. Proporciona acceso rápido y constante en tiempo de ejecución a elementos por índice.

- **LinkedList** → Esta implementación permite que mejore el rendimiento en ciertas ocasiones. 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.

2.2 Interfaz Set

Set → Define 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.
Es importante destacar que, para comprobar si los elementos son elementos duplicados o no lo son, es necesario que dichos elementos tengan implementada, de forma correcta, los métodos equals y hashCode. Para comprobar si dos Set son iguales, se comprobará si todos los elementos que los componen son iguales sin importar el orden que ocupen dichos elementos.

Implementaciones a la interfaz Set:

- **HashSet** → Esta implementación 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

- **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

- **LinkedHashSet** → Esta implementación almacena los elementos en función del orden de inserción. Es, simplemente, un poco más costosa que HashSet.

2.3 Interfaz Map

Map → Asocia claves a valores. Esta interfaz no puede contener claves duplicadas y cada una de dichas claves, sólo puede tener asociado un valor como máximo.

Implementaciones a la interfaz Map:

- **HashMap** → Esta implementación 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 siempre y cuando la función hash disperse de forma correcta los elementos dentro de la tabla hash. Es importante definir el tamaño inicial de la tabla ya que este tamaño marcará el rendimiento de esta implementación.

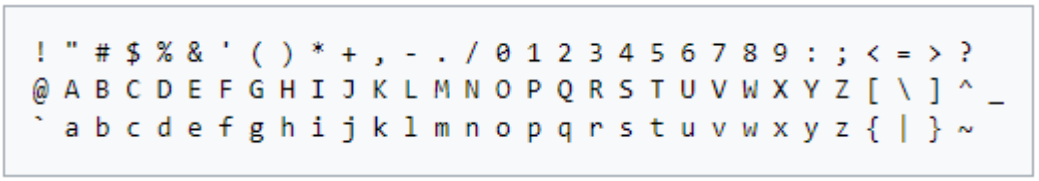
- **TreeMap** → Esta implementación almacena las claves ordenándose en función de sus valores. Es bastante más lento que HashMap. Las claves almacenadas deben implementar la interfaz **Comparable.**

- **LinkedHashMap** → Esta implementación almacena las claves en función del orden de inserción. Es, simplemente, un poco más costosa que HashMap.

3. Métodos.

- Vamos a realizar ejemplos de algunos métodos de Collections.

- **Max** → Esté método sirve para buscar el mayor de la colección, sirve con números y letras. Mediante esta tabla se determina cual es el mayor y el menor.




```
public Alumno buscarApellidoMax() {
    return Collections.max(listaAlumnos);
}
```

  - Esté nos devuelve el máximo de los apellidos.

```
Apellido más lejano: Alumnos [nombre=Marta, apellidos=Vázquez,
idAlumno=6]
```

- **Min** → Este hace lo mismo que el max pero a la inversa, por lo que nos devuelve el menor de la colección. Sigue también la misma tabla para determinar cuál es el pequeño.

```
public Alumno buscarApellidoMin() {
    return Collections.min(listaAlumnos);
}
```

  - Y este nos devuelve lo contrario al máximo, es decir, el mínimo de ellos.

```
Apellido más cercano: Alumnos [nombre=Candi, apellidos=Alcantarilla,
idAlumno=4]
```

- **ReplaceAll** → Este método sustituye todos los elementos de un valor especificado en una lista por otro, en el método tendremos que especificar primero la lista que en nuestro caso es de alumnos, luego el antiguo valor que queremos reemplazar y por último, el nuevo valor que queremos introducir.

```
public void reemplazarAlumno(Alumno alumNuevo, int pos) {
     Collections.replaceAll(listaAlumnos, listaAlumnos.get(pos - 1), alumNuevo);
}
```

```
-----------------------
DATOS DEL NUEVO ALUMNO
-----------------------
Indique el nombre
Carlos
Indique el apellido
Gómez
En que posición esta el alumno que desea reemplazar
2
1-Alumnos [nombre=Lúcas, apellidos=Pérez, idAlumno=1]
2-Alumnos [nombre=Carlos, apellidos=Gómez, idAlumno=7]
3-Alumnos [nombre=Sebastián, apellidos=Millán, idAlumno=3]
4-Alumnos [nombre=Candi, apellidos=Alcantarilla, idAlumno=4]
5-Alumnos [nombre=Lucía, apellidos=López, idAlumno=5]
6-Alumnos [nombre=Marta, apellidos=Vázquez, idAlumno=6]
```

  - Aquí vemos cómo el alumno nuevo se introduce en su posición donde nosotros lo habíamos indicado (posición 2), por lo que borraría al antiguo alumno.

- **Reverse** → Invierte la lista, ejemplo; Sí nuestra lista tiene el orden de 1,2,3,4 y 5, la lista saldría 5, 4, 3, 2 y 1.

```
public void invertirLista() {
    Collections.reverse(listaAlumnos);
}
```

```
1-Alumnos [nombre=Lúcas, apellidos=Pérez, idAlumno=1]
2-Alumnos [nombre=Pablo, apellidos=Gónzalez, idAlumno=2]
3-Alumnos [nombre=Sebastián, apellidos=Millán, idAlumno=3]
4-Alumnos [nombre=Candi, apellidos=Alcantarilla, idAlumno=4]
5-Alumnos [nombre=Lucía, apellidos=López, idAlumno=5]
6-Alumnos [nombre=Marta, apellidos=Vázquez, idAlumno=6]
```

```
1-Alumnos [nombre=Marta, apellidos=Vázquez, idAlumno=6]
2-Alumnos [nombre=Lucía, apellidos=López, idAlumno=5]
3-Alumnos [nombre=Candi, apellidos=Alcantarilla, idAlumno=4]
4-Alumnos [nombre=Sebastián, apellidos=Millán, idAlumno=3]
5-Alumnos [nombre=Pablo, apellidos=Gónzalez, idAlumno=2]
6-Alumnos [nombre=Lúcas, apellidos=Pérez, idAlumno=1]
```

  - Podemos comprobar cómo se ha invertido el orden de la lista.
  - La primera imagen es la lista sin invertir y la segunda lista ya invertida.

- **Shuffle** → Esté método desordena la lista aleatoriamente.

```
public void desordenarLista() {
    Collections.shuffle(listaAlumnos);
}
```

  - Aquí vemos como se ha desordenado la lista y los alumno salen en distinto orden.

```
1-Alumnos [nombre=Sebastián, apellidos=Millán, idAlumno=3]
2-Alumnos [nombre=Marta, apellidos=Vázquez, idAlumno=6]
3-Alumnos [nombre=Carlos, apellidos=Gómez, idAlumno=7]
4-Alumnos [nombre=Candi, apellidos=Alcantarilla, idAlumno=4]
5-Alumnos [nombre=Lucía, apellidos=López, idAlumno=5]
6-Alumnos [nombre=Lúcas, apellidos=Pérez, idAlumno=1]
```

- **Swap** → Este método sirve para intercambiar las posiciones de los elementos en una colección, le debes pasar primero la lista, luego la posición del primer elemento y por último la posición del elemento por el que quieras intercambiar la posición.

```
public void cambiarPosicion(int posicion, int posNuevo) {
    Collections.swap(listaAlumnos, posicion -1 , posNuevo - 1);
}
```

```
1-Alumnos [nombre=Sebastián, apellidos=Millán, idAlumno=3]
2-Alumnos [nombre=Marta, apellidos=Vázquez, idAlumno=6]
3-Alumnos [nombre=Carlos, apellidos=Gómez, idAlumno=7]
4-Alumnos [nombre=Candi, apellidos=Alcantarilla, idAlumno=4]
5-Alumnos [nombre=Lucía, apellidos=López, idAlumno=5]
6-Alumnos [nombre=Lúcas, apellidos=Pérez, idAlumno=1]
```

```
Posición del alumno que desea mover
2
Posición nueva por el que lo desea mover
5
1-Alumnos [nombre=Sebastián, apellidos=Millán, idAlumno=3]
2-Alumnos [nombre=Lucía, apellidos=López, idAlumno=5]
3-Alumnos [nombre=Carlos, apellidos=Gómez, idAlumno=7]
4-Alumnos [nombre=Candi, apellidos=Alcantarilla, idAlumno=4]
5-Alumnos [nombre=Marta, apellidos=Vázquez, idAlumno=6]
6-Alumnos [nombre=Lúcas, apellidos=Pérez, idAlumno=1]
```

  - Aquí observamos como la posición 2 (id 6) pasaría a la posición 5 que es donde nosotros lo hemos indicado.

- **AddAll** → Sirve para agregar los elementos a la colección de forma en la que puedes ahorrarte código.

```
List<Alumno> alumnos = new ArrayList<Alumno>();

Alumno a1 = new Alumno("Lúcas", "Pérez", idAlumno);
idAlumno++;
Alumno a2 = new Alumno("Pablo", "Gónzalez", idAlumno);
idAlumno++;
Alumno a3 = new Alumno("Sebastián", "Millán", idAlumno);
idAlumno++;
Alumno a4 = new Alumno("Candi", "Alcantarilla", idAlumno);
idAlumno++;
Alumno a5 = new Alumno("Lucía", "López", idAlumno);
idAlumno++;
Alumno a6 = new Alumno("Marta", "Vázquez", idAlumno);
idAlumno++;



// Los agregamos con addAll para mostrar su funcionamiento, pero hacerlo con
// add y new Alumno funciona igual.

Collections.addAll(alumnos, a1, a2, a3, a4, a5, a6);
GestionAlumnos gA = new GestionAlumnos (alumnos);
```

```
1-Alumnos [nombre=Lúcas, apellidos=Pérez, idAlumno=1]
2-Alumnos [nombre=Pablo, apellidos=Gónzalez, idAlumno=2]
3-Alumnos [nombre=Sebastián, apellidos=Millán, idAlumno=3]
4-Alumnos [nombre=Candi, apellidos=Alcantarilla, idAlumno=4]
5-Alumnos [nombre=Lucía, apellidos=López, idAlumno=5]
6-Alumnos [nombre=Marta, apellidos=Vázquez, idAlumno=6]
```

- Os dejo el enlace hacía el ejemplo que hay subido en mi gitHub: https://github.com/Candiia/ProyectoCollections