ULP Virtual

Tecnicatura Universitaria en Desarrollo de Software

Programación II

Guía 5.2

Colecciones

Universidad de LA PUNTA

GOBIERNO DE SAN LUIS

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, pero, para esto existen las *colecciones*.

Una *colección* representa un grupo de objetos. Esto 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, pero no podemos guardar elementos de tipo primitivo; para solucionar esto aparecieron las clases de envoltura (*wrappers*) en el API de java a partir de la versión 5.

Para todos los tipos de **datos primitivos**, existen unas clases llamadas **Wrapper**, también conocidas como envoltorio, ya que **proveen una serie de mecanismos que nos permiten envolver a un tipo de dato primitivo** permitiéndonos con ello el tratarlos como si fueran **objetos**.

Como sabemos, existen 8 tipos primitivos predefinidos en Java**,** cada uno de los tipos de tipos de datos primitivos **tienen asociados su correspondiente clase Wrapper para poder realizar las conversiones de tipos primitivos a objetos (tipos no primitivos).**

Tipos de datos	
Primitivos	Objetos
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

Estas clases Wrappers tienen algunos métodos muy útiles. Por ejemplo:

Para convertir (parsear) una cadena en un valor numérico primitivo.

```
String entero=”124”;
int enteroPrimitivo= Integer.parseInt(entero);

String doubleCadena=”75.50”;
double doublePrimitivo=Double.parseDouble(doubleCadena);
```

Conclusión: Todas las clases Wrappers que representan números tienen un método estático llamado parseXXX, que permite convertir una String en un valor primitivo.

El **boxing y unboxing** es automático, es decir, el proceso de guardar un primitivo dentro de un Wrapper o sacarlo es automático. Por ejemplo:

```
//Boxing
Integer nro= 12;
```

Es lo mismo que:

```
Integer nro= new Integer(12);
```

```
//Unboxing
Integer nro=12;
int otroNro= nro;
```

Es lo mismo que:

```
int otroNro= nro.intValue();
```

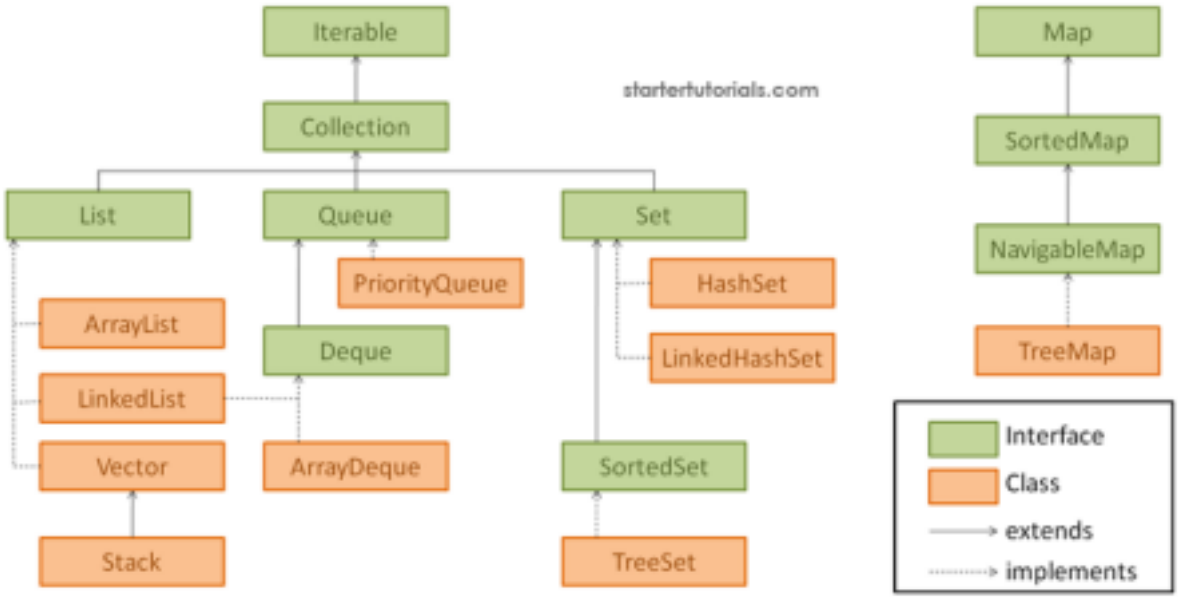
Ahora, retomando el tema principal, dijimos que una colección es un grupo de objetos, pero para obtener una colección vamos a utilizar unas clases propias de Java. Estas clases, que van a ser el almacén de los objetos, nos proveen con una serie de *métodos* comunes, para trabajar con los elementos de la colección, como, por ejemplo: *agregar* y *eliminar* elementos u obtener el tamaño de la colección, entre otros.

Las colecciones son una especie de arreglos de tamaño dinámico. Estas son parte 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.

Las clases del **Java Collections Framework** son las siguientes:



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 y Vector*.

• **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; con la lógica para trabajar con listas genéricas, es decir podemos insertar y extraer elementos de cualquier parte de la lista ya que implementa Queue. Itera más lento que ArrayList pero las inserciones y borrados son más rápidos.

Ejemplo de una LinkedList de números:

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

• **Vector**: Es básicamente lo mismo que un ArrayList, pero sus métodos están sincronizados. En general, es preferible utilizar ArrayList en vez de Vector, dado que los métodos sincronizados reducen la performance de nuestro programa.

Ejemplo de una Vector de números:

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

CONJUNTOS

Los *conjuntos* o en ingles *Set* modelan una colección de objetos de una misma clase donde cada elemento aparece *solo una vez,* no puede tener *duplicados*, a diferencia de una lista donde los elementos podían repetirse. Utiliza el método *equals()* de sus elementos para determinar si dos objetos son idénticos. 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 rendimiento que el Treeset. Los elementos de esta colección no se recuperan en el orden en que fueron insertados, sino de acuerdo a su ubicación en la tabla hash.

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 en un orden natural, es decir, si se agregan números podrían estar ordenados de menor a mayor o a la inversa, si se ingresan cadenas, en un orden lexicográfico; si se ingresan Personas, podrían estar ordenadas por dni;etc. Para lograr esto, los elementos agregados deben implementar Comparable o establecer en el constructor de TreeSet la regla de ordenamiento; por esa razón 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();
```

• **LinkedHashSet** 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.

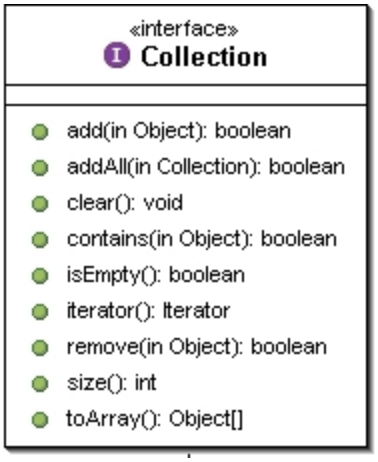
Ejemplo de un LinkedHashSet de cadenas:

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

En conclusión si necesitas coleccionar tus elementos sin que se repitan y no te interesa en qué forma se ordenan, entonces la implementación a usar podría ser HashSet ahora si te interesa que no se repitan pero quieres recuperar los elementos en el orden que los insertaste, entones la

implementación adecuada es LinkedHashSet; pero si lo que quieres es que no estén repetidos sus elementos pero necesitas que tengan algún orden especial, lo que debes utilizar es un TreeSet.

Tanto las Listas como los Conjuntos descienden de la interface Collection que propone una serie de métodos que cada una de estas familias implementarán de forma diferente.

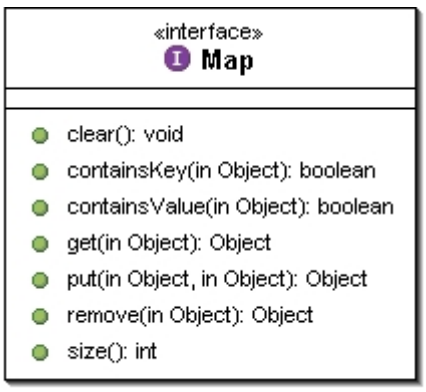


MAPAS

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 nunca con el mismo DNI/RUT.

Los mapas al tener dos datos, también vamos a tener que especificar el tipo de dato tanto de la llave y del valor, pueden ser de tipos de datos distintos. A la hora de crear un mapa tenemos que pensar que el primer tipo dato será el de la llave y el segundo el valor.

Son una de las estructuras de datos importantes del Framework de Collections. Las implementaciones de Map son HashMap, TreeMap, LinkedHashMap y HashTable; Map no desciende de la interface Collection como List y Set.



• **HashMap** es un mapa implementado a través de una tabla hash, las llaves se almacenan utilizando un algoritmo de hash solo para las llaves y evitar que se repitan; por lo tanto, los objetos utilizados como llaves tienen que tener sobreescrito los métodos equals y hashCode. Los elementos se recuperan por el orden implementado en la tabla hash.

Ejemplo de un HashMap de personas:

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

• **TreeMap** es un mapa que ordena los elementos en un orden natural a través de las llaves; por lo tanto los objetos utilizados como llaves deben implementar la interface Comparable.

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 LinkedHashMap de personas:

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

AÑADIR UN ELEMENTO A UNA COLECCIÓN

Las colecciones constan de métodos para realizar distintas operaciones, en este caso si queremos añadir un elemento a las listas o conjuntos vamos a tener que utilizar el método add(T), pero para los mapas vamos a utilizar el método put(llave,valor).

Listas:

```
ArrayList<Integer> numeros = new ArrayList(); //Lista de tipo Integer

int num = 20;

numeros.add(num); //Agregamos el numero 20 a la lista, en la posición 0
```

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); //Agregamos la llave y el valor
```

ELIMINAR UN ELEMENTO DE UNA COLECCIÓN

Cada colección consta con métodos para poder remover elementos del tipo que sea la colección.

Listas:

Las listas constan de dos métodos:

remove(int índice): Este método remueve un elemento de un índice especifico. Esto mueve los elementos, de manera que no queden índices sin elementos.

remove(elemento): Este método remueve la primer aparición de un elemento a borrar en una lista

Índice:

```
ArrayList<Integer> numeros = new ArrayList();
int num = 20;
numeros.add(num); // Este numero se encuentra en el índice 0
numeros.remove(0); // Eliminamos el numero que esté en el índice 0
```

Elemento:

```
ArrayList<Integer> numeros = new ArrayList();
int num = 30;
numeros.add(num);
numeros.remove(30); // Eliminamos el numero 30 o el primer 30 que encuentre
```

Conjuntos:

Ya que los conjuntos no constan de índices, solo se puede borrar por elemento.

remove(elemento): Este método remueve la primer aparición de un elemento a borrar en un conjunto

```
HashSet<Integer> numeros = new HashSet();
int num = 50;
numeros.add(50);
numeros.remove(50); //Eliminamos el numero 50
```

Mapas:

La parte más importante de los elementos de un mapa es la llave del elemento, que es la que hace el elemento único, por eso en los mapas solo podemos remover un elemento por su llave.

remove(llave): Este método remueve la primer aparición de la llave de un elemento a borrar en un mapa.

```
HashMap<Integer, String> estudiantes = new HashMap();
estudiantes.put(123,"Perez");
estudiantes.remove(123); //Borramos la llave 123
```

RECORRER UNA COLECCIÓN

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

Para recorrer una colección, podemos utilizar el bucle *forEach*. El bucle comienza con la palabra clave *for* al igual que un bucle for normal. Pero, 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 del *entry* vamos a traer los valores y las llaves, si no, podemos tener un foreach para cada parte de nuestro mapa sin utilizar el objeto **Map.Entry**.

For Each:

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

Listas:

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

Conjuntos:

```
HashSet<Integer> numerosSet = new HashSet();
for (Integer numero : numerosSet) {
          System.out.println(numero);
          // mostramos los elementos a través de la variable
}
```

Mapas:

```
HashMap<Integer, String> alumnos = new HashMap();
// Recorrer las dos partes del mapa
for (Map.Entry<Integer, String> entry : alumnos.entrySet()) {

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

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

}
```

Sin Map.Entry:

```
// mostrar solo las llaves
for (Integer dni : alumnos.keySet()) {
        System.out.println("Documento: " + dni);
}
// mostrar solo los valores
for (String nombres : alumnos.values()) {
        System.out.println("Nombre: " + nombres);
}
```

ITERATOR

El *Iterator* es una interfaz que pertenece al *framework* de *colecciones*. Este, nos permite recorre ,acceder a la información y eliminar algún elemento de una colección. Gracias al Iterato podemos
eliminar un elemento, mientras recorremos la colección. Ya que, cuando queremos eliminar algún elemento mientras recorremos una colección con el bucle ForEach, nos va a tirar un error.

Como el Iterator es parte del framework de colecciones, todas las colecciones vienen con el método *iterator(),* este, devuelve las instrucciones para iterar sobre esa colección. Este método iterator(), devuelve la colección, lo recibe el objeto Iterator y usando el objeto Iterator, podemos recorrer 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 métodos muy útiles 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 colección, mientras el método hasNext() retorne true. Este método es el que nos sirve para mostrar el elemento,

3***. void remove():*** Elimina el elemento actual de la colección.

Ejemplo Listas:

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

Como pudimos ver más arriba para eliminar un elemento de una colección vamos a tener que utilizar la función *remove()* del Iterator. Esto se aplica para el resto de nuestras colecciones. Los mapas son los únicos que no podemos eliminar mientras las iteramos.

Listas:

```
ArrayList<String> palabras = new ArrayList();
Iterator<String> it = palabras.iterator();
while (it.hasNext()) {
    if (it.next().equals("Hola")) {
    // Borramos si está la palabra Hola it.remove();
    }
}
```

Conjuntos:

```
HashSet<Integer> numerosSet = new HashSet();
Iterator<Integer> it = numerosSet.iterator();
while (it.hasNext()) {
    if (it.next() == 3) { // Borramos si está el numero 3
        it.remove();
    }
}
```

ORDENAR UNA COLECCIÓN

Recordemos que los TreeSet y TreeMap se ordenan por sí mismos ya que los elementos que añadimos deben implementar la interface Comparable o a través del criterio de ordenamiento impuesto a través del constructor de estas colecciones. En cambio los elementos que vamos agregando a una colección de la familia Liste el único orden que tiene es el de inserción, pero podemos solucionar eso, pues el framework de colecciones disponen de una clase de nombre Collections que dispone de una serie de métodos estáticos para ordenar listas y otras operaciones muy útiles; entonces, para ordenar una colección del tipo List, vamos a tener que utilizar la función Collections.sort(List), recibe la colección cuyos elementos deben implementar Comparable; ahora que pasa si los elementos que están guardados en un List no implementan Comparable? … pues en ese caso, el método sort esta sobrecargado[1] y permite que pasemos como segundo parámetro el criterio de ordenamiento a través de un clase que implemente Comparator.

Ahora que ya sabemos reordenar una Lista, que pasa si queremos ordenar los conjuntos (Set), para ello, deberemos convertirlos a listas, para poder ordenar esa lista por sus elementos. Y a la hora

[1] La sobrecarga de métodos es la creación de varios métodos con el mismo nombre pero con diferente lista de tipos de parámetros. Java utiliza el número y tipo de parámetros para seleccionar cuál definición de método ejecutar.

de ordenar un mapa como tenemos dos datos para ordenar (llaver, valor), vamos a convertir el HashMap a un TreeMap.

Listas:

```
ArrayList<Integer> numeros = new ArrayList();
Collections.sort(numeros);
```

Conjuntos:

```
HashSet<Integer> numerosSet = new HashSet();
// Se convierte el HashSet a Lista.
ArrayList<Integer> numerosLista = new ArrayList(numerosSet);
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 CLASES CREADAS POR NOSOTROS

De la misma manera que podemos crear colecciones con los tipos de objetos del API Java como String, Integer, etc., podemos crear colecciones de algún objeto, de una clase creada por nosotros, previamente. Esto, nos servirá para manejar varios objetos al mismo tiempo y acceder a ellos de una manera más sencilla. Por ejemplo, tener una lista de alumnos, siendo cada Alumno un objeto con sus atributos.

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.

Las colecciones Tree, ya sean TreeSet o TreeMap, son las únicas que no vamos a poder agregar como siempre. Ya que, los Tree, siendo colecciones que se ordenan a sí mismas, debemos informarle al Tree como va a ordenarse. Ahora, pensemos que un objeto posee más de un dato(atributos), entonces, el Tree, no sabe por qué atributo debe ordenarse.

Para solucionar esto, vamos a necesitar un *Comparator*, este, le dará 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("Juan", "Lopez");
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 mostraría algo así: Libreria.Libro@14ae5a5

Para solucionar este problema, vamos a tener que sobrescribir(Override) el método toString() dentro de la clase de nuestro objeto. Este método 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 método vamos a ir a nuestra clase, ahí hacemos click derecho, **insert code** y le damos a **toString()**. Eso nos va a generar un método 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 método se va a llamar solo, sin necesidad que lo llamemos nosotros, siempre que queramos mostrar nuestro objeto en un System.out.println. Y mostrará la línea 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, y la clase de estos objetos no implementa Comparable, no podemos utilizar el método sort de Collections. Para esto, utilizamos la interfaz *Comparator* con su método *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 implementar el método compare de la interface Comparator en la clase Perro.

Ejemplo:

Una forma de hacerlo podría ser, crear una clase adicional que implemente Comparator y al sobreescribir el método compareTo estableceremos allí el criterio de ordenamiento.

En un archivo Java aparte:

```
public class EdadAscendente implements Comparator<Perro> {
    public int compare(Perro p1, Perro p2) {
     if(p1.getEdad() < p2.getEdad()) return -1;
```

```
        if(p1.getEdad() == p2.getEdad()) return 0;
        return 1;
 }
}
```

Otra forma es crear una *clase interna anónima*:

```
Comparator<Perro> compararEdad = new Comparator<Perro>() {  @Override
        public int compare(Perro p1, Perro p2) {
                        if(p1.getEdad() < p2.getEdad()) return -1;
                        if(p1.getEdad() == p2.getEdad()) return 0;
                        return 1;
         }
        };
```

Explicación del método:

• 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, 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.

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();
//Se llama al metodo estatico a traves de la clase y se pone la lista a  //ordenar.
Collections.sort(perros, compararEdad);
O
//Instanciando un objeto de la clase adicional como segundo parámetro del método sort de Collections..
Collections.sort(perros, new EdadAscendente());
```

Conjuntos:

```
HashSet<Perro> perrosSet = new HashSet();
ArrayList<Perro> perrosLista = new ArrayList(perrosSet);
Collections.sort(perrosLista, compararEdad);
```

```
//Instanciando un objeto de la clase adicional como segundo parámetro del método sort de Collections..
Collections.sort(perrosLista, new EdadAscendente());
```

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 como habíamos explicado anterioremente.

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

Mapas:

```
HashMap<Integer, Alumno> alumnos = new HashMap();
//Se usa una función de los mapas para traer todos valores.
ArrayList<Alumno> nombres = new ArrayList(map.values());
Collections.sort(compararDni);
```

COLECCIÓNES EN MÉTODOS

A la hora de querer pasar una colección como parámetro de un método, 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 al método
```

Conjuntos:

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

llenarHashSet(**palabras**); // Le pasamos el conjunto al método

Mapas:

```
Public void llenarMapa(HashMap<Integer, String> alumnos){
alumnos.put(1, “Pepe”);
}
Main
HashMap<Integer, String> alumnos = new HashMap();
llenarMapa(alumnos); // Le pasamos el mapa al método
```

DEVOLVER UNA COLECCIÓN EN LOS MÉTODOS

Para devolver una colección en un método, 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.put(1, “Pepe”);
return alumnos;
}
```

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 se muestran 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, ya que ambas heredan de la interface **Collection** (observe que el nombre de la interface es en singular, mientras que la clase con funcionalidades extras es Collections). Excepto los get y 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.
get(int índice)	Este método retorna un elemento de la lista según un índice de la lista.
set(int índice, elemento)	Este método guarda un elemento en la lista en un índice específico.
isEmpty()	Este método retorna verdadero si la lista / conjunto está vacío y falso si no lo está.
contains(elemento)	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:

Método	Descripción.
clear()	Este método se usa para remover todos los elementos de un mapa.
containsKey(Llave)	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.

containsValue(Valor)	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.
get(Llave)	Este método retorna un elemento del mapa según una llave dentro del mapa.
isEmpty()	Este método retorna verdadero si el mapa está vacío y falso si no lo está.
size()	Este método retorna el tamaño de un mapa.
values()	Este método crea una colección según los valores del mapa. Ósea, que retorna una lista, por ejemplo, con todos los valores del mapa.
clear()	Este método se usa para remover todos los elementos de un mapa.

BIBLIOGRAFIA:

Christopher Alexander, “A Pattern Language”, 1978

Erich Gamma, “Design Patterns: Elements of Reusable OO Software”

Martin Fowler, “Analysis Patterns: Reusable Object Models”, Addison Wesley,

1997

Kathy Sierra, “OCA/OCP JAVA SE 7 PROGRAMMER I & II STUDY GUIDE (EXAMS 1Z0-803 & 1Z0-8”, McGraw Hill, 2014