Tema 9 : Colecciones.

Listas , Conjuntos y Mapas.

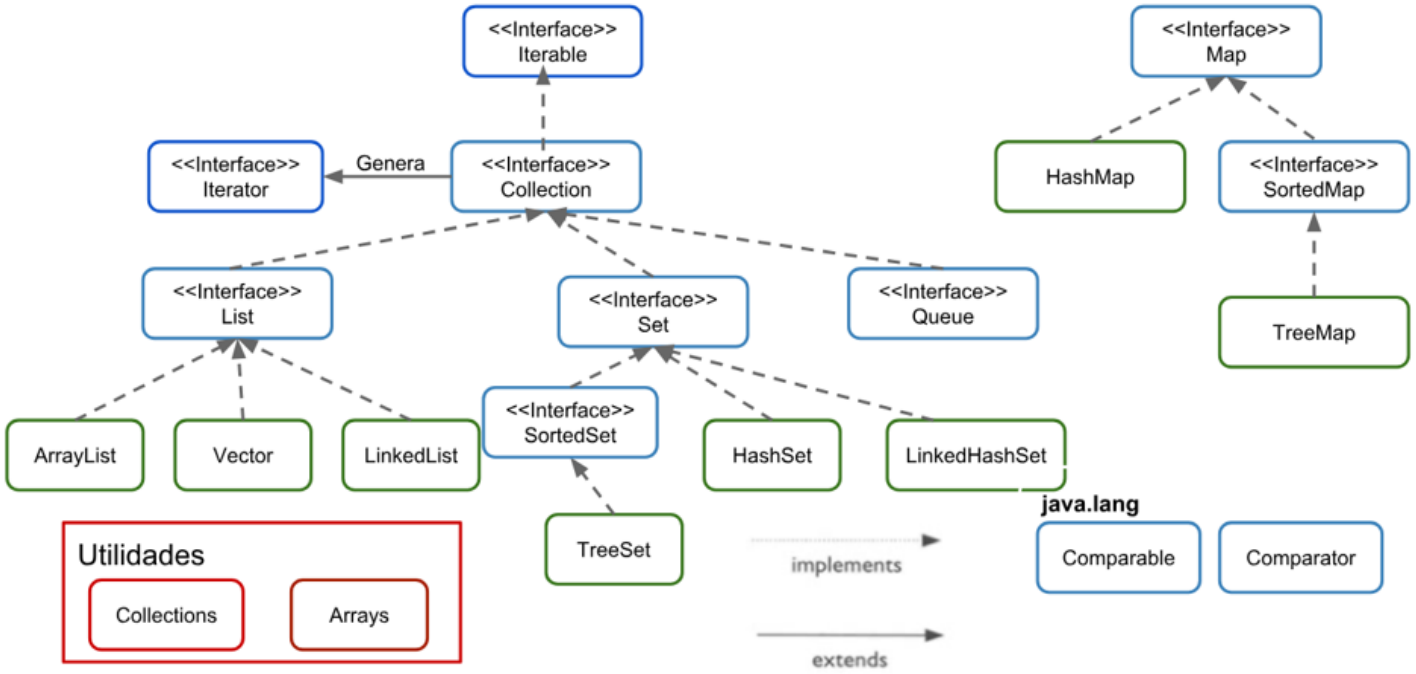
Hemos visto en el tema 3 del curso que las matrices o arrays nos permitían guardar un conjunto de valores pero no se pueden redimensionar, esto es , tienen un tamaño fijo que no puede ser cambiado. Tampoco se podía cambiar el tipo de elemento que guardaban. Además las labores de inserción o eliminación y búsqueda de elementos dependían del programador.

La API de Java en su paquete ***java.util*** nos ofrece una serie de *clases contenedoras* que nos permitirán **simular** el comportamiento de matrices dinámicas y proporcionan además formas sofisticadas para realizar búsquedas e inserciones eficientes en la matriz subyacente con la que trabajan. Estas clases son conocidas como **colecciones**.

Las colecciones pueden clasificarse en 3 grandes grupos atendiendo a la forma en la que almacenan , buscan y modifican los objetos internamente:

- **listas**: Son colecciones ordenadas de elementos , también conocidas como secuencias , en las que podemos controlar en qué posiciones queremos guardar los elementos. Se puede acceder y buscar los elementos a través de un **índice**. Otra característica de las listas es que permiten elementos duplicados así como el valor **null.** Todas las listas heredan de la interfaz **List<E> donde <E> representa el tipo de instancia que se va a almacenar en la lista.**
- **conjuntos**: Son colecciones que no permiten duplicados. La interfaz que define todas las propiedades comunes de los conjuntos es **Set<E> .**
- **mapas o diccionarios**: Permite almacenar parejas clave - valor , de forma que un valor puede ser obtenido a partir de la clave que se le asoció al guardarlo en el mapa . Las claves no pueden repetirse , los valores sí. No son colecciones estrictamente hablando ya que no hereda de la interfaz **Collection**. La interfaz que define el comportamiento básico de todas las clases de mapas es **Map<E>.**

La jerarquía de clases simplificada del framework de colecciones es el siguiente[1]



En esta gráfica podemos ver como las listas y los conjuntos , es decir , las interfaces **List<E>** y **Set<E>** implementan la interfaz **Collection** mientras que los mapas no lo hacen. Implementar la interfaz Collection supone que vamos a poder ***iterar*** de forma segura por la colección gracias a un objeto conocido como **Iterator**. O lo que es lo mismo podremos utilizar la estructura for each para recorrer la colección. Como vemos los mapas no disponen de esta funcionalidad por lo que tendremos que recorrerlos de alguna otra manera.

Listas.

Vamos a centrarnos en dos de las implementaciones de la interfaz List más utilizadas **ArrayList** y **LinkedList**.

[1] Para profundizar más se recomienda la lectura de los temas dedicados a colecciones del libro *Piensa en Java*

ArrayList.

Son listas basadas en un array interno y que proporcionan métodos para agregar , eliminar o buscar elementos en el array . Así mismo también disponen de dos tipos de iteradores para recorrerlo de forma segura.

Cada elemento guardado en el ArrayList es indexado y puede recuperarse a través del índice asignado. Cuando se agrega un elemento el ArrayList lo guarda en la primera posición vacía del array. También se pueden insertar elementos en posiciones concretas , aunque esta operación es menos eficiente. Cuando se elimina un elemento de una posición los siguientes elementos se desplazan para ocupar su lugar de forma que no queden posiciones vacías por el medio.

La **capacidad** del ArrayList se va adaptando conforme se van añadiendo elementos. La forma en la que crece el tamaño del array no está especificado aunque normalmente aumenta su capacidad en un **50**% cuando se queda sin espacio para agregar más elementos.

Para crear una instancia de un ArrayList disponemos de 3 constructores:

Constructores ArrayList
public **ArrayList**():
public **ArrayList**(**int** capacidad):
public **ArrayList**(**Collection<? extends E> col**) [2]:

Los principales métodos de que disponemos en esta clase son :

Métodos ArrayList
boolean **add**(**E** e): Inserta una instancia de tipo E en la primera posición vacía.
void **add**(**int** pos , **E** e): Inserta una instancia de tipo E en la posición especificada.
boolean **addAll**(**Collection<? extends E>** col) **:** Agrega todos los elementos de la colección especificada al final del ArrayList
void **addAll**(**int** pos **, Collection<? extends E>** col) **:** Agrega todos los elementos de la colección especificada al final del ArrayList

[2] *Esta notación se explicará en el tema de Genéricos.*

void **clear() :** Elimina todos los elementos del ArrayList.
boolean **contains(Object o):** Busca un objeto en la lista.
E **get**(**int** indice) : Devuelve una referencia al elemento cuyo índice es pasado como parámetro.
int **indexOf**(**Object o**) **:** Devuelve la posición de la primera aparición del objeto que se le pasa como argumento o -1 en caso de no encontrarlo.
Iterator **iterator**(): Devuelve un iterador para recorrer la colección subyacente.
ListIterator **listIterator**() : Devuelve un ListIterator para recorrer la colección subyacente.
boolean **remove**(**Object** o) : Elimina el objeto indicado.
E **remove**(**int** indice) : Elimina el elemento en la posición indicada.
E **set(int** indice **, E** elemento **) :** Reemplaza el elemento situado en la posición indicada por el nuevo elemento pasado como parámetro.
int **size**() : Devuelve el número de elementos en la lista
List<E> **subList**(**int** indiceInicio, **int** indiceFinal) : Devuelve una vista de los elementos que se encuentran entre las posiciones indicadas . El indice de inicio es inclusivo y el indice final exclusivo.
void **trimToSize() :** Ajusta la capacidad de la lista al número de elementos almacenado.

Ejemplo 1: ArrayList de Strings

```
ArrayList<String> libros = new ArrayList<String>();
ArrayList<String> otrosLibros = new ArrayList<String>(2);
 //Agregar libros
  libros.add("Los pilares de la Tierra");
  libros.add("El médico");
  libros.add("Sinuhé el egipcio");
  libros.add(2,"La conjura de los necios");
```

El método **add** va añadiendo elementos al final del array , si indicamos el índice en el que lo queremos añadir desplaza los elementos hacia la derecha es decir les suma uno a su

indice , esta sobrecarga del método lanza una **IndexOutOfBoundsException** cuando el índice indicado es mayor que el número de elementos.

```
//Número de elementos
   System.out.println("En la colección hay: " + libros.size() + "
elementos.");
      libros.add(6,"Los miserables");


En la colección hay: 5 elementos.
Exception in thread "main" java.lang.IndexOutOfBoundsException: Index: 6, Size: 5
        at java.util.ArrayList.rangeCheckForAdd(Unknown Source)
        at java.util.ArrayList.add(Unknown Source)
        at listas._01ArrayListStrings.main(_01ArrayListStrings.java:22)
```

Utilizando **addAll** podemos pasar una lista entera a otra lista:

```
List<String> otrosLibros = new ArrayList<String>(2);
otrosLibros.add("Piensa en Java");
otrosLibros.add("Java 8");
otrosLibros.add("Como programar en Java"); //Se redimensiona para que
tenga espacio
libros.addAll(otrosLibros);
```

Para reemplazar un elemento por otro **set(**int pos , Object o**):**

```
libros.set(0, "Parque Jurásico");
```

Para recuperar un elemento o saber en qué posición está : **get**(int pos) y **indexOf**(Object o) :

```
//Busqueda de elementos
  String libro = libros.get(4);

  System.out.println("Posicion 4 : " + libro);

  int posicion = libros.indexOf("Java 8");
  System.out.println("El libro Java 8 está en la posición : " +
```

```
posicion);
```

Para eliminar tenemos el método **remove :**

```
//Eliminar
      libros.remove("Piensa en Java");

      libros.remove(0);

      libros.removeAll(otrosLibros);
```

Para recorrer la colección podemos utilizar un for each:

```
      System.out.println("Coleccion de libros:");

      for(String s:libros){
            System.out.println(s);
      }
```

Ejemplo 2: ArrayList con números.

Los arraylist no pueden almacenar variables primitivas como **int** o **float** por lo que hay que envolverlas en su wrapper correspondiente **Integer , Float etc…** , esta operación se realiza implícitamente.

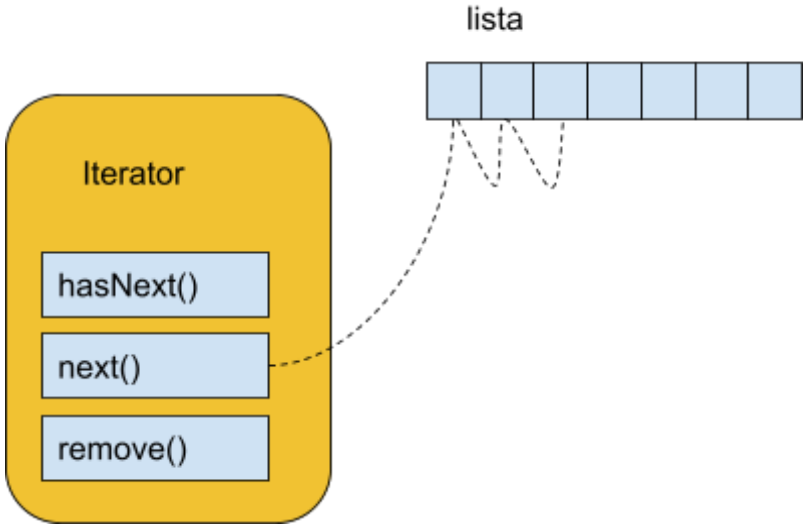
```
ArrayList<Integer> numero = new ArrayList<Integer>();
   numero.add(3);
   numero.add(new Integer(4));
   numero.add(9);

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

Iterator.

Hemos visto cómo desde Java 5 para recorrer colecciones tenemos la estructura for each que es muy cómoda e intuitiva de utilizar. Esta estructura utiliza internamente un objeto de tipo ***Iterator***[3] que es un objeto que permite recorrer una lista de forma segura en su totalidad a través de dos métodos:

- **hasNext():** Devuelve true si hay objeto.
- **next():** Devuelve el siguiente objeto**.**



Vamos a ver un código de ejemplo :

```
//Recorrer la lista con un iterator
      Iterator<String> it = libros.iterator();

      while(it.hasNext()){
            System.out.println(it.next());

      }
```

Como podemos observar el código es menos claro que con el for each , sin embargo no siempre podremos utilizar un for each, por ejemplo cuando queremos modificar la

[3] *El diseño de esta clase se explica en el Anexo I*

colección que estamos recorriendo. Siguiendo con el ejemplo anterior , imaginémonos que queremos eliminar todos los libros que empiecen por "E" [4]:

```
for(String s : libros){
            if(s.startsWith("E")){
                  libros.remove(s);
            }
      }

Exception in thread "main" java.util.ConcurrentModificationException
        at java.util.ArrayList$Itr.checkForComodification(Unknown Source)
        at java.util.ArrayList$Itr.next(Unknown Source)
        at listas._01ArrayListStrings.main(_01ArrayListStrings.java:70)
```

El código suelta una excepción en tiempo de ejecución ya que se está modificando la colección que se desea recorrer.

La interfaz **Iterator** dispone del método **remove()** que permite eliminar elementos mientras recorremos la colección:

```
//Eliminar con un iterator
      it = libros.iterator();
      while(it.hasNext()){
            if(it.next().startsWith("E")){
                  it.remove();
            }
      }
```

De esta forma podemos eliminar objetos mientras iteramos sobre la colección. Cabe destacar como para volver a recorrer la colección **debemos de volver a llamar al método iterator()** ya que de esta forma volvemos a posicionarlo al principio de la colección.

[4] Esto no es válido para versiones superiores a la versión 8

ListIterator.

Los ArrayList disponen de otro tipo de iterador que permite recorrer la colección en ambas direcciones y realizar modificaciones en la matriz. Este iterador conocido como **ListIterator** añade los métodos **hasPrevious**() , **previous**() , **add(**E e**)** , **set(**int pos , **E** e**)**.

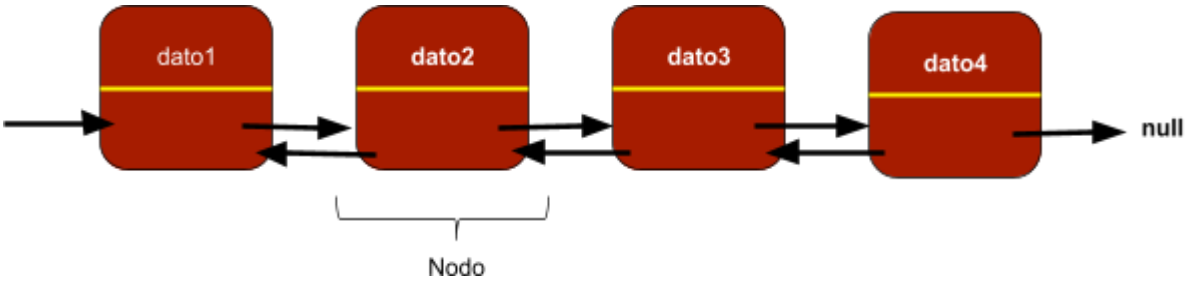
```
//Ejemplo de listIterator
ListIterator<String> it2 = libros.listIterator();

	while(it2.hasNext()){
		System.out.println(it2.next());
	}


	while(it2.hasPrevious()){
		System.out.println(it2.previous());
	}
```

LinkedList[5].

Esta implementación de la interfaz List está basada en el concepto de lista **doblemente** enlazada en la que cada objeto almacena una referencia al objeto anterior y al siguiente.



Esta forma de estructurar los datos ofrece una **gran eficiencia agregando o eliminando elementos a la colección**. Es **menos eficiente** en procesos de lectura de la colección , es decir **en búsquedas** ya que se deben de recorrer todos los nodos hasta encontrar el nodo deseado.

Para crear una LinkedList tenemos dos constructores:

[5]Funcionamiento interno listas enlazadas

Constructores LinkedList
public **LinkedList**():
public **LinkedList**(Collection <? extends e>):

Además de los métodos que ya vimos en el ArrayList como add , clear , remove ofrece los siguientes métodos entre otros:

Métodos LinkedList
void **addFirst**(E e): Añade un elemento al principio
void **addLast**(E e): Añade un elemento al final
E **getFirst**(): Devuelve el primer elemento de la lista
E **getLast**(): Devuelve el último elemento de la lista
boolean **offerFirst():** Añade un elemento al principio
boolean **offerLast**(): Añade un elemento al final
E **pollFirst**(): Devuelve y elimina el primer elemento de la lista.
E **pollLast**() : Devuelve y elimina el último elemento de la lista

```
LinkedList<String> pelis = new LinkedList<String>();
      pelis.add("Matar a un ruiseñor");
      pelis.add("El resplandor");
      pelis.add("Interstellar");


      System.out.println(pelis.getLast());
      System.out.println(pelis.getFirst());


      pelis.offerFirst("Speed");
      pelis.offerLast("Regreso al Futuro");
```

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

Comparación entre ArrayList y LinkedList:

- **Búsquedas**: Los ArrayList son más rápidos a la hora de realizar búsquedas ya que el método get ofrece un rendimiento de O(1)[6] mientras que en un **LinkedList** es de O(n) [7], es decir el tiempo aumenta con el número de elementos.
- **Eliminación e inserciones**: Los linkedlist ofrecen una eficiencia de O(1) mientras que en los ArrayList es de O(n) , debido a que tienen que recolocar el resto de elementos para mantener la contigüidad.

Por lo tanto si nuestra estructura de datos va a ser estable con pocas operaciones de inserción o eliminación debemos decantarnos por un **ArrayList** , mientras que si va a ser una estructura muy dinámica con inserciones y eliminaciones frecuentes la **LinkedList** es la más idónea.

Conjuntos.

Los conjuntos son colecciones que no permiten valores duplicados. Vamos a ver los dos más importantes que son TreeSet y HashSet.

HashSet

Proporciona una implementación de conjunto respaldada por una tabla **hash** , lo que permite determinar de forma rápida si un elemento ya ha sido almacenado.
Este tipo de conjunto no mantiene el orden de inserción sino que ordena los elementos en función a un algoritmo basado en el código hash de los elementos que almacena , a pesar de esto es mucho más rápida que un **TreeSet**.

Los constructores disponibles son :

Constructores HashSet
public **HashSet**():
public **HashSet**(**Collection<? extends E>** col):

[6] *O(1)Es tiempo constante*
[7] *O(n) El tiempo aumenta de forma lineal si aumenta el número de elementos.*

public **HashSet**(int capacidad):

Esta clase no añade métodos nuevos.

Ejemplo 1: *HashSet con Strings y numeros.*

```
HashSet<String> nombres = new HashSet<String>();
   nombres.add("Pedro");
   nombres.add("Luis");
   nombres.add("Juan");
   System.out.println(nombres);

HashSet<Integer> numeros = new HashSet<Integer>();
  Random r = new Random();
   for(int i = 0;i<10;i++) {
    numeros.add(r.nextInt(100));

     }
     System.out.println(numeros);

[Luis, Pedro, Juan]
[1, 17, 84, 69, 39, 23, 73, 13, 61]
```

Vemos como el orden en el que se muestran no se corresponde ni con el orden de inserción ni con el criterio natural de las variables , y cómo también no se almacenan valores repetidos.

Ejemplo 2: *HashSet con Películas*

Vamos ahora a almacenar en un HashSet objetos de tipo Película :

Clase Película:

```
public class Pelicula {

     private String codigo;
     private String titulo;
```

```
      private double valoracion;
      public Pelicula(String codigo, String titulo, double
valoracion) {

            this.codigo = codigo;
            this.titulo = titulo;
            this.valoracion = valoracion;
      }

…
```

```
Pelicula p1 = new Pelicula("HE345","Interestellar",9.75);
Pelicula p2 = new Pelicula("PIUYT","Regreso al Futuro",10);
Pelicula p3 = new Pelicula("QIWOW","Cadena Perpetua",9.8);

  HashSet<Pelicula> videoclub = new HashSet<Pelicula>();
  videoclub.add(p1);
  videoclub.add(p2);
  videoclub.add(p3);
  videoclub.add(p1);

for(Pelicula p: videoclub) {
            System.out.println(p);
      }

Pelicula [codigo=HE345, titulo=Interestellar valoracion: 9.75]
Pelicula [codigo=QIWOW, titulo=Cadena Perpetua valoracion: 9.8]
Pelicula [codigo=PIUYT, titulo=Regreso al Futuro valoracion: 10.0]
```

Vemos como intentamos añadir dos veces **p1** pero simplemente se añadió una vez. Vamos a intentar que si dos películas tienen el mismo código tampoco las guarde. Para ello tenemos que conseguir que dos películas sean consideradas iguales si tienen el mismo código.

Métodos equals y hashcode.

Para asignar un criterio de igualdad a una clase debemos modificar el método equals y también el hashcode para que sean coherentes , es decir , que cuando dos objetos sean considerados iguales tengan el mismo hashcode.

Vamos a crear dos objetos películas que contengan exactamente los mismos valores:

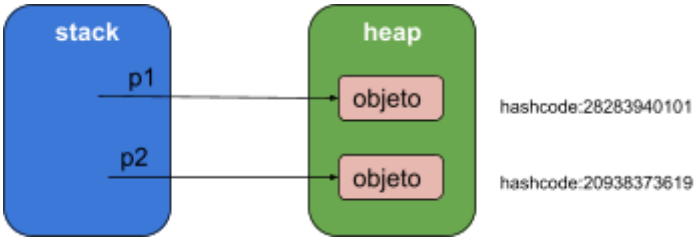
```
Pelicula p1 = new Pelicula("HE345","Interestellar",9.75);
Pelicula p2 = new Pelicula("HE345","Interestellar",9.75);
```

Vamos a mostrar por pantalla lo que devuelven los métodos equals y hashcode sobre estos objetos:

```
System.out.println("p1.equals(p2) = " + p1.equals(p2));
System.out.println("p1==p2) = " + (p1==p2));
System.out.println("p1.getHAshCode() " + p1.hashCode());
System.out.println("p2.hashCode() " + p2.hashCode());
```

```
p1.equals(p2) = false
p1==p2) = false
p1.hashCode()   2018699554
p2.hashCode()1311053135
```

Vemos como el equals devuelve false y los hashcode son distintos. Esto es totalmente lógico ya que tenemos dos referencias apuntando a dos objetos distintos.



Las referencias se almacenan en la memoria de acceso rápido **stack** y los objetos en la zona de memoria conocida como **heap d**e acceso un poco más lento**.**

Si igualamos las dos referencias:

```
p1 = p2;
```

Lo que veríamos sería que p1 yp2 son el mismo objeto:

```
p1.equals(p2) = true
p1==p2) = true
p1.hashCode()  2018699554
p2.hashCode()2018699554
```

Vemos como además se genera el mismo hashcode.

Ahora vamos a sobreescribir ambos métodos para que dos objetos de tipo película sean considerados iguales si almacenan el mismo valor para el código.

Método equals:

```
@Override
	public boolean equals(Object obj) {

		Pelicula other = (Pelicula) obj;

		if (!codigo.equals(other.codigo))
			return false;
		return true;
	}
```

Se sobreescribe de forma que devuelva true si los códigos de las películas son iguales[8].

Ahora vamos a sobreescribir el hashcode para que dos películas con el mismo código tengan el mismo hashcode:

```
@Override
	public int hashCode() {
		return this.codigo.hashCode();
```

[8] *Si utilizais el eclipse para sobreescribir el método equals os genera un código más completo y seguro que este , ya que tiene en cuenta valores null etc...*

```
    }
```

De esta forma la salida del ejemplo anterior sería:

```
Pelicula p1 = new Pelicula("HE345","Interestellar",9.75);
Pelicula p2 = new Pelicula("HE345","Interestellar",9.75);
  System.out.println("p1.equals(p2) = " + p1.equals(p2));
  System.out.println("p1==p2) = " + (p1==p2));
  System.out.println("p1.getHAshCode() " + p1.hashCode());
  System.out.println("p2.hashCode() " + p2.hashCode());
```

```
p1.equals(p2) = true
p1==p2) = false
p1.getHAshCode() 68599767
p2.hashCode() 68599767
```

Vemos como el método equals devuelve true y como tienen el mismo hashcode , el operador **==** sigue comparando las referencias y por lo tanto devuelve **false**[9]

Vamos ahora a intentar guardar en un HashSet dos películas con el mismo código:

```
HashSet<Pelicula> videoclub = new HashSet<Pelicula>();
 videoclub.add(new Pelicula("QWE","Ciudadano Kane",9));
 videoclub.add(new Pelicula("QWE","123 Berlin",8));

for(Pelicula p : videoclub) {
           System.out.println(p);
     }

Pelicula [codigo=QWE, titulo=Ciudadano Kane valoracion: 9.0]
```

Sólo almacena una de las dos películas que se intenta guardar.

[9] *En java no se puede modificar el comportamiento de un operador , en otros lenguajes como* ***c#*** *esto si es posible*

TreeSet.

Este tipo de conjunto ordena los elementos por su orden natural , es decir, el especificado en el método **compareTo** de la interfaz **Comparable<E>** o por el especificado en el método **compare** de la interfaz **Comparator.**

Para construir un **TreeSet** disponemos de los siguientes constructores:

Constructores de TreeSet
public **TreeSet**(): Crea un conjunto ordenado. Los objetos deben implementar la interfaz Comparable.
public **TreeSet** (**Collection**<? extends E> c) : Crea un conjunto ordenado a partir de la colección recibida.
public **TreeSet**(**Comparator**<? super E> c) : Crea un conjunto ordenado de acuerdo al criterio especificado en el implementación de la interfaz Comparator.

Ejemplo 1: TreeSet con Strings

```
TreeSet<String> alumnos = new TreeSet<String>();
      alumnos.add("Braulio Otero");
      alumnos.add("Alejandro Sánchez");
      alumnos.add("Domingo Duró");

      for(String s : alumnos) {
            System.out.println(s);
      }
```

```
Alejandro Sánchez
Braulio Otero
Domingo Duró
```

Vemos como el orden en el que lo muestra no coincide con el de inserción sino que está ordenado alfabéticamente.

Vamos a crear un TreeSet con números:

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

    for(int i = 1;i<10;i++) {
         numeros.add(r.nextInt(10));
     }

    for(Integer f : numeros) {
        System.out.print(f + " ");
     }
```

```
0 2 5 6 7 8 9
```

Ejemplo 2: TreeSet con clase Pelicula.

Vamos a intentar almacenar objetos Película en un TreeSet:

```
TreeSet<Pelicula> pelis = new TreeSet<Pelicula>();
Pelicula p1 = new Pelicula("HE345","Interestellar",9.75);
Pelicula p2 = new Pelicula("PIUYT","Regreso al Futuro",10);
Pelicula p3 = new Pelicula("QIWOW","Cadena Perpetua",9.8);

          pelis.add(p1);
          pelis.add(p2);
          pelis.add(p3);
```

```
Exception in thread "main" java.lang.ClassCastException: clases.Pelicula cannot be cast to java.lang.Comparable
        at java.util.TreeMap.compare(Unknown Source)
        at java.util.TreeMap.put(Unknown Source)
        at java.util.TreeSet.add(Unknown Source)
        at conjuntos._05TreeSetPeliculas.main(_05TreeSetPeliculas.java:16)
```

Salta la excepción **ClassCastException** ya que no tenemos definido un criterio de comparación que pueda utilizar para ordenar los objetos de tipo Película.

Por tanto debemos implementar la interfaz **Comparable<Pelicula>** en la clase Pelicula y programar el método **compareTo. En este caso vamos a utilizar el campo valoración para ordenar las películas.**

Ejemplo 3: TreeSet con criterio ordenación alternativo.

```
@Override
	public int compareTo(Pelicula p) {
		// TODO Auto-generated method stub
		Double d = new Double(this.valoracion);
		return d.compareTo(p.valoracion);
	}
```

Ahora ya podemos guardar películas en nuestro TreeSet y los almacena ordenándolos por la valoración de menos a más :

```
Pelicula [codigo=HE345, titulo=Interestellar valoracion: 9.75]
Pelicula [codigo=QIWOW, titulo=Cadena Perpetua valoracion: 9.8]
Pelicula [codigo=PIUYT, titulo=Regreso al Futuro valoracion: 10.0]
```

Comparator.

Si quisiéramos establecer otro criterio alternativo de ordenación , debemos de crear una clase que implemente la interfaz **Comparator** y pasarle un objeto de esta clase al constructor del TreeSet :

```
import java.util.Comparator;

public class ComparadorPeliculas implements Comparator<Pelicula> {

	@Override
	public int compare(Pelicula o1, Pelicula o2) {
		// TODO Auto-generated method stub
		return o1.getTitulo().compareTo(o2.getTitulo());
	}

}
```

Una ez creado el comparador , creamos un TreeSet pasándole en el constructor un objeto de tipo ComparaPeliculas :

```
TreeSet<Pelicula> videoclub = new TreeSet<Pelicula>();
```

```
 Pelicula p1 = new Pelicula("HE345","Interestellar",9.75);
 Pelicula p2 = new Pelicula("PIUYT","Regreso al Futuro",10);
 Pelicula p3 = new Pelicula("QIWOW","Cadena Perpetua",9.8);
           videoclub.add(p1);
           videoclub.add(p2);
           videoclub.add(p3);

TreeSet<Pelicula> videoclub2 = new TreeSet<Pelicula>(new
ComparadorPeliculas());

videoclub2.addAll(videoclub);

           for(Pelicula p : videoclub) {
                System.out.println(p);
           }
           System.out.println();
           System.out.println("Videoclub alternativo: ");

           for(Pelicula p : videoclub2) {
                System.out.println(p);
           }
Pelicula [codigo=HE345, titulo=Interestellar valoracion: 9.75]
Pelicula [codigo=QIWOW, titulo=Cadena Perpetua valoracion: 9.8]
Pelicula [codigo=PIUYT, titulo=Regreso al Futuro valoracion: 10.0]

Videoclub alternativo:
Pelicula [codigo=QIWOW, titulo=Cadena Perpetua valoracion: 9.8]
Pelicula [codigo=HE345, titulo=Interestellar valoracion: 9.75]
Pelicula [codigo=PIUYT, titulo=Regreso al Futuro valoracion: 10.0]
```

Mapas.

Los mapas son un grupo de parejas **clave/valor** en la que las claves no pueden estar duplicadas y cada clave puede tener asociado un único valor. La interfaz base es **Map<k,v>** que ho desciende de Collection , es decir no son propiamente colecciones y por lo tanto no podremos utilizar la sintaxis del for each para recorrerlas , pero sí podremos utilizarla para recorrer las claves y los valores como veremos a continuación.

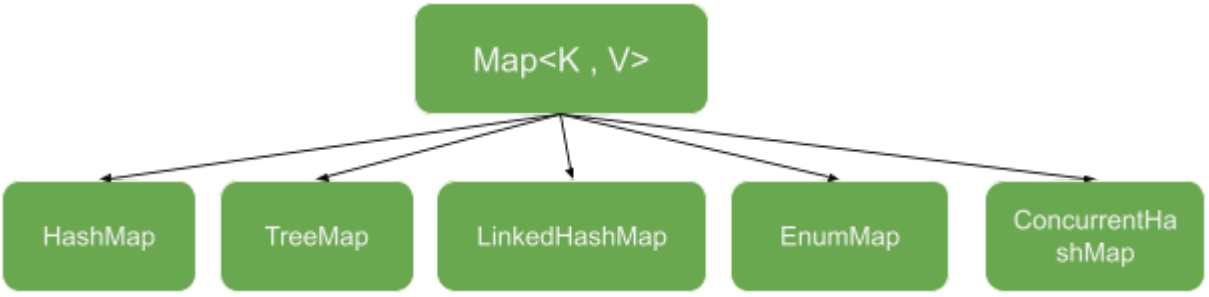
La interfaz **Map<K,V>** expone los siguientes métodos:

Métodos interfaz Map<K,V>
void **clear():** Elimina todas las entradas del diccionario**.**
boolean **containsKey**(**Object** o): Devuelve true si encuentra la clave que recibe como argumento.
boolean **containsValue**(**Object** o): Devuelve true si encuentra el valor que recibe como argumento.
V **get**(**Object** key) : Devuelve el valor asociado a la clave.
V **put(K** key **, V** value**) :** Introduce un valor asociado a una clave , si la clave ya existía es asociada al nuevo valor. Este método devuelve el valor que tenía asociado la clave o **null** si no estaba la clave previamente guardada.
void **putAll(Map<? extends k , ? extends v>** mapa**) :** Permite guardar todas las entradas de un mapa en otro mapa.
V **remove**(Object key) : Elimina la entrada de **key** de este mapa.
int **size**(): Devuelve el número de entradas **clave/valor** en este mapa.
Collection<V> **values**(): Devuelve una colección con los valores almacenados.
Set<K> **keySet**() : Devuelve un conjunto con las claves de las entradas del mapa.

¿Por qué el método values() devuelve un **Collection**<V> y el método keySet() un **Set**<K>?

La respuesta es sencilla , porque los valores pueden estar repetidos y por lo tanto si los guardásemos en un conjunto íbamos a perder información.

Las clases de mapas que implementan la interfaz map son:



Aquí nos centraremos en los **HashMap** , los demás podéis consultar en la documentación su funcionamiento.

HashMap

Proporciona una implementación de Map basada en una estructura de datos de tabla hash. Admite todas las operaciones definidas en Map y valores **null.**

Una tabla hash mapea las claves con valores enteros mediante el método hashcode().

El orden en el que son organizadas las entradas no está grantizado.

Vamos a crear un mapa en el que se asocia un DNI con un nombre de persona.

```
HashMap<String,String> dnis = new HashMap<String , String>();

    dnis.put("12345678A", "Adrián Jocarias");
    dnis.put("99999999A","María Sánchez");
    dnis.put("55555555A" , "Esteban Portabales");
    //Intento guardar un dni que ya está
    dnis.put("12345678A","Cuidado que sustituyo");

   System.out.println(dnis);

{99999999A=María Sánchez, 55555555A=Esteban Portabales,
12345678=Cuidado que sustituyo, 12345678A=Adrián Jocarias}
```

Vemos como se elimina la primera entrada y se sustituye por el nuevo valor asociado a la clave repetida , por esto es aconsejable comprobar si la clave ya existe antes de guardar una nueva.

```
//Intento guardar un dni que ya está
            if(!dnis.containsKey("12345678A")) {
                  dnis.put("12345678","Cuidado que sustituyo");
            }
            else {
                  System.out.println("Clave repetida");
            }
```

Para realizar búsquedas por clave utilizamos el método **get :**

```
String nombre = dnis.get("12345678A");
```

Si no lo encuentra devuelve el valor **null.**

Para recorrer un Map tenemos dos opciones : obtener claves y valores y recorrerlas por separado:

```
//Recuperar claves
    Set<String> claves = dnis.keySet();
     for(String clave : claves) {

            }
 //Recuperar los valores
  Collection<String> valores = dnis.values();
```

O utilizar el método **entrySet()** que devuelve todas las entradas en un conjunto **Set<Map.Entry<K,V>** siendo **Map.Entry** el objeto que se utiliza internamente para almacenar las entradas.

```
Set<Entry<String, String>> entradas = dnis.entrySet();

  for(Entry<String,String> e : entradas) {
     System.out.println(e.getKey() + " " + e.getValue());
     }
```

Ejercicios Propuestos.

1. **Programad los siguientes métodos:**
    a. Método que reciba un ***Collection<Integer>*** un entero ***n*** y un entero ***límite*** y rellene la colección con **n** números aleatorios entre 0 y el límite.
    b. Método que reciba un ***Collection<Integer>*** y muestre por pantallas sus valores.

c. Método que recibe un ***Set<Double>*** y lo llena con 100 números double aleatorios.
d. Método que recibe un ***Set<Double>*** y muestra sus valores con una cifra decimal.

A continuación probado los métodos anteriores desde el main:

e. Crear un ArrayList con el método del apartado a. A partir de este ArrayList cread un HashSet y un TreeSet utilizando el método addAll

f. Mostrar por pantalla las colecciones anteriores.

g. Crear ahora un TreeSet con el método del apartado **c ,** a partir de este crear un Hashset y mostrar ambas colecciones utilizando el método del apartado **e.**

h. Crear ahora un ArrayList de strings y meterle 10 nombres , alguno de ellos repetidos.

i. Crear un TreeSet y un HashSet a partir del ArrayList anterior

j. Mostrar ambas coleeciones utilizando un Iterator.

2. **Crear un ArrayList de números decimales.**

a. A continuación introducir 10 números aleatorios entre 0 y 10.
b. Calcular la media.
c. Calcular el valor más alto generado.
d. Mostrar por pantalla todos los valores almacenados.
e. Eliminar todos aquellos que sean superiores a 7.
f. Volver a mostrar la matriz.

3. **Crear dos ArrayList de números enteros.**

a. Llenarlos con 10 números aleatorios entre 0 y 20. (Podeis utilizar función Ejercicio 1)
b. Crear una tercera lista con los valores de las 2 listas anteriores.
c. Crear una cuarta lista con los valores de la primera lista que no están en la segunda.

d. Mostrar los valores de las 4 listas.

4. **Diseñar una Clase llamada *Contacto* con los siguientes atributos:**
   a. String nombre;
   b. String telefono;
   c. String correo;

   Añadirle los constructores y métodos de acceso junto con un toString.

   Crear a continuación la clase **Agenda** que tiene como atributo:

   a. ArrayList<Contacto> contactos .

   Añdir un constructor por defecto que crea una instancia del ArrayList y un método **getContactos** que devuelve un ArrayList con todos los contacts de la agenda.

   Esta clase tiene los siguientes métodos:

   - *void* **addContacto**(Contacto c): Almacena un contacto nuevo en contactos.
   - *contacto* **getContacto**(String telefono); Devuelve el contacto que tiene el teléfono que se le pasa como argumento.
   - int **getPosicionContacto**(String telefono): Devuelve la posición del contacto cuyo telefono coincide con el que se le pasa como parámetro.
   - void **eliminarContacto**(String telefono): Elimina el contacto con el telefono especificado.
   - ArrayList<Contacto> **buscarContactos**(String nombre): Devuelve una lista con todos los contactos que contienen todo o parte del String que recibe como parámetro.
   - void **modificarContacto**(String telefono , String nombre , String correo): Busca un contacto por el teléfono y modifica el nombre y el correo.
   - void **imprimeContactos**(): Método que muestra por consola todos los contactos almacenados

En la clase **Principal** crear una Agenda con 5 contactos. Probad los métodos anteriores.

5. Modificar la clase Contacto para que dos objetos de tipo Contacto sean considerados iguales si tienen el mismo teléfono.
   a. Crear un Set de contactos (ej anterior) .

b. Asociarle una instancia de tipo HashSet.

Set<Contacto> agenda = new HashSet<Contacto>();

c. Almacenar 5 contactos. Comprobad que no se almacenan contactos con números repetidos.
d. Mostrar por pantalla los contactos almacenados.
e. Haced que los Contactos sean comparables por el nombre.
f. Copiar los contactos anteriores en un TreeSet.
g. Mostrar por pantalla el TreeSet creado.

6. Crear un HashMap que permita almacenar el dni y la nota (float) de un alumno.
   a. Añadir 5 entradas a la colección.
   b. Realizar una búsqueda por el dni.
   c. Eliminar una entrada de la colección.
   d. Mostrar todos los datos almacenados.

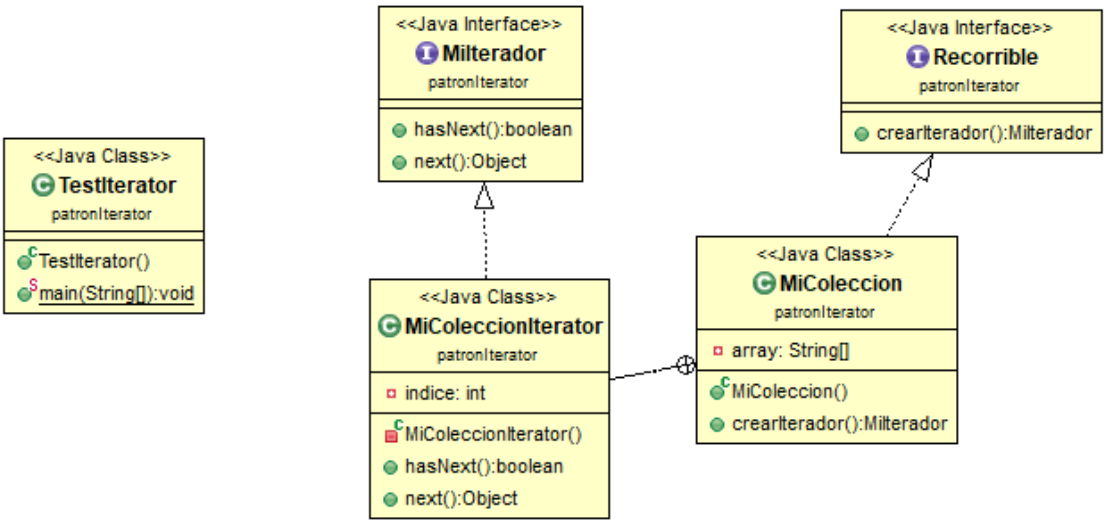
7. Crear un HashMap en el que la clave sea el DNI y esté asociado a un ArrayList de números decimales:

   HashMap<String,ArrayList<Integer> mapa;

   a. Crear un ArrayList llamado notas1 y llenarlo con 10 notas.
   b. Crear otro ArrayList llamado notas2 y llenarlo con 10 notas.
   c. Meted en el mapa dos entradas asociando las notas anteriores a un dni.
   d. A continuación mostrad las notas de un determinado dni.
   e. Calcular la nota media de las dos entradas del mapa.

Anexo I : Patrón de diseño iterator.

A continuación mostramos el diagrama UML del patrón Iterator y el código de cada una de las clases:



MiIterador.java

```
public interface MiIterador {
	boolean hasNext();
	Object next();
}
```

Recorrible.java

```
public interface Recorrible {
	MiIterador crearIterador();
}
```

MiColeccion.java

```
public class MiColeccion implements Recorrible {

	private String array[] = {"Uno","Dos","Tres","Cuatro"};
```

```
@Override
public MiIterador crearIterador() {
	// TODO Auto-generated method stub
	MiColeccionIterator mc = new MiColeccionIterator();
	return mc;
}

private class MiColeccionIterator implements MiIterador{

	private int indice=0;
	@Override
	public boolean hasNext() {
		// TODO Auto-generated method stub
		if(indice < array.length) {
			return true;
		}
		else {
		return false;
		}
	}
	@Override
	public Object next() {
		// TODO Auto-generated method stub
		if(this.hasNext()) {
			return array[indice++];
		}
		else
		return null;
	}
}
```

TestIterator.java

```
MiColeccion mc = new MiColeccion();
MiIterador mit = mc.crearIterador();

	while(mit.hasNext()) {
		System.out.println(mit.next());
```

```
        }
```

Bibliografía:

- **Piensa en Java** , *Bruce Eckel*
- **Estructuras de datos**, *Mark Allen Weiss*,

Lecturas recomendadas para profundizar en el tema:

- 5 cosas que no sabías de las colecciones.
- 5 cosas que no sabías de las colecciones II
- Comparativa entre conjuntos: TreeSet Vs HashSet Vs LinkedHashSet