1. [Tipos de colecciones en java](https://www.luaces-novo.es/guia-de-colecciones-en-java/" \l "tipos_colecciones)
   1. [Listas](https://www.luaces-novo.es/guia-de-colecciones-en-java/#tipo_listas)

Permite duplicados. Estan ordenadas. Collections.sort(). ListIterator()=>modificación

Bajo rendimiento.

* 1. [Sets](https://www.luaces-novo.es/guia-de-colecciones-en-java/#tipo_sets)

No duplicados. equals(). Mas eficiente que listas =>.add().

No acceso aleatorio. Algunos son ordenados (poco eficiente).

* 1. [Maps](https://www.luaces-novo.es/guia-de-colecciones-en-java/#tipo_maps)

Asocian valor-clave. No claves duplicadas=> usan un Set.

Rendimiento bajo.

* 1. [Colas](https://www.luaces-novo.es/guia-de-colecciones-en-java/#tipo_colas)

Gran rendimiento. LIFO o FIFO.

Queues: fin. Dequeues: inicio o fin.

Iteracion y acceso aleatorio lentos.

1. [Listas](https://www.luaces-novo.es/guia-de-colecciones-en-java/#listas)
   1. [ArrayList](https://www.luaces-novo.es/guia-de-colecciones-en-java/#arraylist): Buena performance.
   2. [LinkedList](https://www.luaces-novo.es/guia-de-colecciones-en-java/#linkedlist): Añade al principio, mas rápida que ArrayList. Insercion pos 0.
   3. [Vector](https://www.luaces-novo.es/guia-de-colecciones-en-java/#vector): deprecated. Solo arraylist concurrente
   4. [CopyOnWriteArrayList](https://www.luaces-novo.es/guia-de-colecciones-en-java/#copyonwritearraylist): Mas eficiente en lectura. Como vector.
2. [Sets](https://www.luaces-novo.es/guia-de-colecciones-en-java/#sets)
   1. [HashSet](https://www.luaces-novo.es/guia-de-colecciones-en-java/#hashset): Rapida, no duplicados. No ordenado.
   2. [LinkedHashSet](https://www.luaces-novo.es/guia-de-colecciones-en-java/#linkedhashset): Ordenacion por orden de entrada. Velocidad **iteración** mayor. Añadir elementos no eficiente.
   3. [TreeSet](https://www.luaces-novo.es/guia-de-colecciones-en-java/#treeset): Arbol rojo-negro. **Ordenado**, menos iteración al hashSet.
   4. [EnumSet](https://www.luaces-novo.es/guia-de-colecciones-en-java/#enumset): Enum.
   5. [CopyOnWriteArraySet](https://www.luaces-novo.es/guia-de-colecciones-en-java/#copyonwritearrayset): Bueno en Lectura. Pesimo escritura, eliminado y contains.
   6. [ConcurrentSkipListSet](https://www.luaces-novo.es/guia-de-colecciones-en-java/#concurrentskiplistset): Concurrente y **ordenable**.
3. [Maps](https://www.luaces-novo.es/guia-de-colecciones-en-java/#maps)
   1. [HashMap](https://www.luaces-novo.es/guia-de-colecciones-en-java/#hashmap): Array clave-valor. No ordenación de clave. No concurrente.
   2. [LinkedHashMap](https://www.luaces-novo.es/guia-de-colecciones-en-java/#linkedhashmap): Claves ordenadas por orden Insercion de clave. Itera mas rápido que HashMap, inserta y elimina peor.
   3. [TreeMap](https://www.luaces-novo.es/guia-de-colecciones-en-java/#treemap): Claves ordenadas. Bueno iterando. Peor en lo demás en relación a Hashmap.
   4. [EnumMap](https://www.luaces-novo.es/guia-de-colecciones-en-java/#enummap): Alto rendimiento. Claves Enum.
   5. [WeakHashMap](https://www.luaces-novo.es/guia-de-colecciones-en-java/#weakhashmap): Referencias blandas de claves-valores. Elegibles por el Garbage. Para crear registros que borran propiedades.
   6. [HashTable](https://www.luaces-novo.es/guia-de-colecciones-en-java/#hashtable): Deprecated. No usar.
   7. [ConcurrentHashMap](https://www.luaces-novo.es/guia-de-colecciones-en-java/#concurrenthashmap): Map concurrente. Acepta null.
4. [Colas](https://www.luaces-novo.es/guia-de-colecciones-en-java/#colas) o queue
   1. [ArrayDeque](https://www.luaces-novo.es/guia-de-colecciones-en-java/#arraydeque) Rendimiento excepcional. LIFO (deque) y FIFO (queue)
   2. [LinkedBlockingDeque](https://www.luaces-novo.es/guia-de-colecciones-en-java/#linkedblockingdeque): LIFO concurrente.
   3. [LinkedList](https://www.luaces-novo.es/guia-de-colecciones-en-java/#linkedlist2): LIFO rendimiento inferior al ArrayDeque.
   4. [PriorityQueue](https://www.luaces-novo.es/guia-de-colecciones-en-java/#priorityqueue): Comparator. Primer elemento depende de cualquier factor (tamaño, prioridad, tiempo de inserción…).
   5. [PriorityBlockingQueue](https://www.luaces-novo.es/guia-de-colecciones-en-java/#priorityblockingqueue): PriorityQueue concurrente.

Referencia: <https://www.luaces-novo.es/guia-de-colecciones-en-java/#tipo_listas>

**EJEMPLOS**

1. Ejemplos de MAP => A. **HashMap , B. TreeMap, C. LinkedHashMap, D. Comparable(Orden Natural).**

CLASE\_07\_EJEMPLO\_1

**Codigo Netbeans:**



**Codigo Eclipse:**

****

1. Ejemplo de Comparable => ORDEN NATURAL

CLASE\_07\_EJEMPLO\_2

**Codigo Netbeans:**



**Codigo Eclipse:**

****

1. Ejemplo de Comparator (Aplicado a Listas y Set)=> ORDEN DEFINIDO por nosotros

Codigo: **Clase09\_comparator => compare(o1, o2) => no es orden natural**

**Codigo Netbeans**



**Codigo Eclipse**

****

Cabe aclarar que el método **sort()** esta sobrecargado en Collections. Esto significa que puede aplicarse para **comparable** como para **comparator**.

1. Ejemplo de INTERFAZ QUEUE:

**INTERFACE QUEUE**

Esta diseñada para contener elementos antes de que se procesen. Es una de las interfaces mas nueva de todas.

<https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html>

Las implementaciones mas conocidas con **LinkedList** y **PriorityQueue**.



Este cuadro lo que significa es que para cada operación hay 2 formas de trabajar. Arrojando una excepción o retornando un valor especial.

 Declaración de clase:

|  |  |
| --- | --- |
| 1 | public interface Queue<E> extends Collection<E>, Iterable<E> |

• Interfaz Queue ordena cosas por el orden en que van a ser procesados. Por defecto ordena como FIFO pero puede ordenar de otras formas.

Queue a = new PriorityQueue();

LIFO

Metodo x => a ( job1, job2, job3, jobx …

• Queue generalmento NO PERMITE NULOS pero alguna de sus implementaciones como LinkedList si que los permiten.

• Algunos métodos de Queue:

add(E e)

element()=> recupera sin borrar la cabeza del Queue

offer(E e) => Inserta el elemento especificado en el Queue si hay tamaño suficiente.

Peek()=> Recupera sin borrar la cabeza del queue o devuelve null si el queue esta vacio

Poll()=> Idem previo pero elimina la cabeza.

Remove()=> recupera y elimina la cabeza del queue.

Ejemplo de PriorityQueue => **codigo PriorityQueue**

**Codigo Netbeans:**



**Codigo Eclipse:**

****

**Ejemplo 2** priorityQueue

**Codigo Netbeans:**



**Codigo Eclipse:**

****

1. Recorrer Collections:

**Codigo Netbeans:**

****

**Codigo Eclipse:**



Vemos 3 formas de recorrer colecciones.

1. La forma tradicional => con un FOR

Sirve para listas solamente, no sirve para map, set o queue.

No es del todo recomendable.

1. Mediante un método **iterator().** Es un método de la interfaz List().

Se define un objeto de tipo Iterator<E> que pertenece a la interfaz Iterator<E>.

Lo que ocurre es que el método iterator() hace una implementación de la interfaz Iterator<E> en este caso para una List.

El iterator significa que quien hizo la colección (en este caso una lista) sabe la mejor forma de recorrer sus elementos. No necesariamente de manera secuencial, porque cada implementación de la collection es distinta.

1. Mediante un **for.. each**. Es la forma mas novedosa. Me permite hacer el recorrido de manera mas sencilla.
2. **EJERCICIO MAP:**

**Codigo Netbeans:**

****

**Codigo Eclipse:**

****

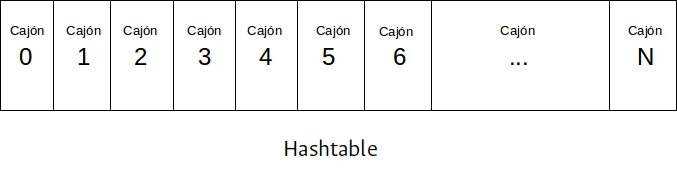
Es importante que en la clase del objeto de referencia se especifiquen los métodos **equals**() y **hashCode**() para que funcione correctamente la implementacion del Map.

El **hashCode()** es un identificador de 32 bits que se almacena en un hash. Toda clase debe proveer un método hashCode() para retornar el nro correspondiente. Este nro puede ser por ejemplo el dato de uno de los atributos de la clase, ej: id.

Lo que hace el hashCode() es la doble validación con el equals. Entonces si el metod equals dio OK, luego el hashCode retorna el integer de cada objeto. Si el integer es el mismo devuelve.

Tambien el orden puede ser el inverso, primero se valida con el hashCode y luego se valida con el equals.

Todo objeto diferente a otro debería tener un hashCode distinto.

* **Respecto al primer método**, el objetivo de hashCode es devolver un número entero que 'identifica' al objeto cuando se guarda en una estructura de datos conocida como [HashMap](https://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html) (u otros como [Hashtable](https://docs.oracle.com/javase/7/docs/api/java/util/Hashtable.html), [HashSet](https://docs.oracle.com/javase/7/docs/api/java/util/HashSet.html)), cuyo objetivo es almacenar un conjunto de valores (de forma similar a un ArrayList o un arreglo). Estas estructuras, salvo algunas diferencias, persisten los datos de la siguiente forma:
* [](https://i.stack.imgur.com/Fj8kn.jpg)

**Codigo Netbeans:**

****

**Codigo Eclipse:**

****

En el siguiente ejercicio vemos como agregamos una persona p3 adicional con la key “David” existente previamente. Entonces 1ro va a chequear con el HashCode si existe el hashcode previamente. Si no existe directamente inserta la key nueva. Si existe ya el control pasa al método equals().

**Clas09\_Hash\_code (Simpson)**

**Codigo Netbeans:**

****

**Codigo Eclipse:**

****

**EJEMPLO HASHCODE**

La implementación del metodo hashcode() e equals() tiene una gran importancia en collections. Porque mejoran la performance en la escritura de sus elementos.

**hashCode():** devuelve un id único para cada objeto. Gracias a este metodo podemos identificar unívocamente a cada objeto.

Si vemos el metodo en la clase object veremos que es un metodo **nativo**. Esto quiere decir que esta implementado en lenguajes C y C++. Tiene su implementación en dichos lenguajes. La cual es utilizada en tiempo de ejecución.

1. Creamos una clase Persona:

public class Persona {

private String nombre;

private int edad;

private String dni;

public Persona(String nombre, int edad, String dni) {

this.nombre = nombre;

this.edad = edad;

this.dni = dni;

}

public String getDni() {

return dni;

}

public void setDni(String dni) {

this.dni = dni;

}

public String getNombre() {

return nombre;

}

public void setNombre(String nombre) {

this.nombre = nombre;

}

public int getEdad() {

return edad;

}

public void setEdad(int edad) {

this.edad = edad;

}

2. Luego desde el main obtenemos el codigo hash para dos objetos.

Persona a = new Persona("Juan", 22);

Persona b = new Persona("Juan", 22);

System.out.println("Codigo Hash de a: "+a.hashCode());

System.out.println("Codigo hash de b: "+b.hashCode());

Vemos que el codigo hash es distinto para cada objeto. Lo mismo sucederá si creamos mas objetos.

El **hashcode()**  **no nos devuelve** el id del objeto en memoria.

Cuando creamos un **objeto** este se crea en tiempo de ejecución en la **heap.** Alli se le asigna una dirección (ej 2234). El **hashcode** number surge de multiplicar ese nro **address nr** por un nro y se le suma otro nro (ej: 2234 \* 31 + 1). El **hashcode number** surge del **heap address nr** pero después de adicionarle, multiplicarle otros nros.

1. **Overriding** the hashcode() method:

Si lo sobreescribimos de la siguiente manera en **Persona**:

@Override

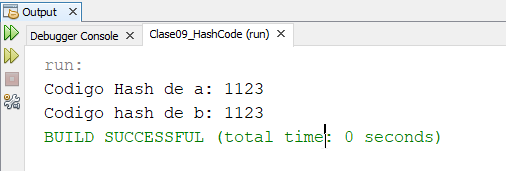
public int hashCode() {

//producimos nuestro propio hashcode nr.

return 1123;

}

Vemos que siempre nos devolverá ese nro como hashcode nr para todos los objetos.



Pero esta es una buena implementación o mala implementación del **hashcode metod?.**

No, porque queremos que cada objeto tenga un id único para cada uno.

1. Entonces podríamos mejorar la implementación del metodo **hashcode** en la clase **Persona**

@Override

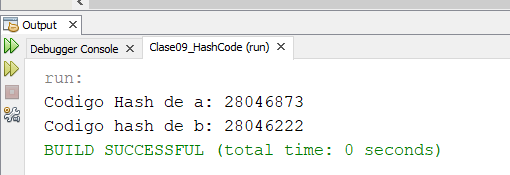
public int hashCode() {

//producimos nuestro propio hashcode nr.

return Integer.valueOf(this.dni);

}

De esta forma nos entregara un nro hashcode distinto para cada objeto creado.



Esto puede considerarse una **buena implementación** de hashcode method.

1. Clases que también **sobreescriben** el **hashcode() method**:

String class:

Comentamos lo que tenemos en el **main** y escribimos:

**main**

/\*

Persona a = new Persona("Juan", 22, "28046873");

Persona b = new Persona("Juan", 22, "28046222");

System.out.println("Codigo Hash de a: "+a.hashCode());

System.out.println("Codigo hash de b: "+b.hashCode());

\*/

String empleado = new String("Analista");

System.out.println("Hashcode de un string: " + empleado.hashCode());

}

}

Si creamos diferentes String objects.

String empleado = new String("Analista");

System.out.println("Hashcode de un string: " + empleado.hashCode());

String empleado1 = new String("Diseñador");

System.out.println("Hashcode de otro string: " + empleado1.hashCode());

Esto también nos dará un id único.

El metodo **hashCode()** tiene su propia implementación en la clase String.

La pregunta que nos tenemos que hacer es el hashcode() method para los Strings siempre devuelve un único id?.

Para el siguiente caso:

**main**

String str1 = "FB";

String str2 = "Ea";

System.out.println(str1.hashCode() + " " + str2.hashCode());

Vemos que el hashcode nr es el mismo. Esto nos indica que no siempre el metodo **hashcode()** devuelve el mismo id, por lo tanto no podemos confiar 100% en el mismo. Por lo tanto dos objetos distintos pueden tener el mismo hashcode. No esta garantizada la unicidad en todos los casos.

Como norma general si dos objetos son iguales el codigo hash debe ser el mismo. Pero si dos objetos tienen el mismo hashcode no esta garantizada su igualdad.