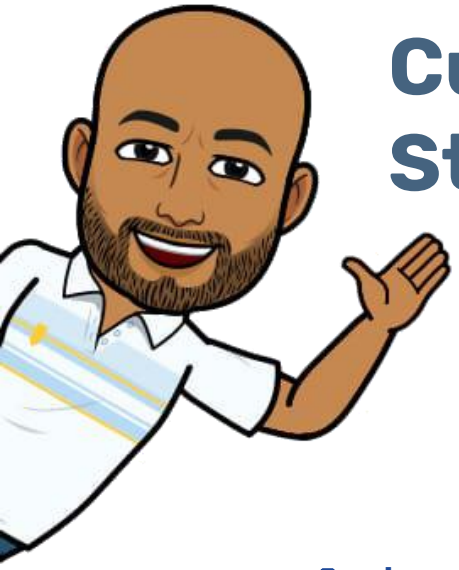


Curso de Java Standard

Ing. Octavio Robleto

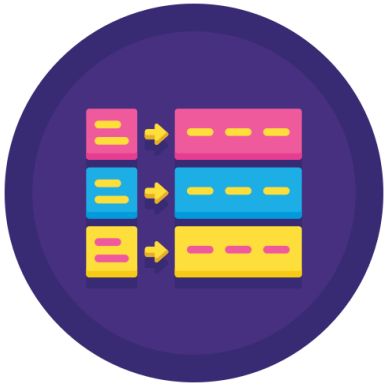
octavio.robleto@gmail.com

https://octaviorobleto.com

Introducción

En el desarrollo de aplicaciones muchas veces vamos a querer representar los objetos en una estructura que permita la búsqueda de forma un poco fácil y mas optima a través de su clave sin necesidad de recorrerlo.

Los mapas o también llamados diccionarios en Java nos permite almacenar elementos asociando a cada clave un valor.



La interfaz **Map** no admite claves duplicadas y es especialmente útil para ir almacenando datos sin la preocupación de que alguna de las claves posea mas de un valor asociado.

Cuando tratamos de agregar un objeto cuando la clave ya existe lo que hace el mapa es actualizar el valor asociado por el nuevo.

La clase **AbstractMap** Proporciona una implementación esquelética de la interfaz **Map** y simplemente agrega implementaciones para los métodos **equals** y **hashCode**.

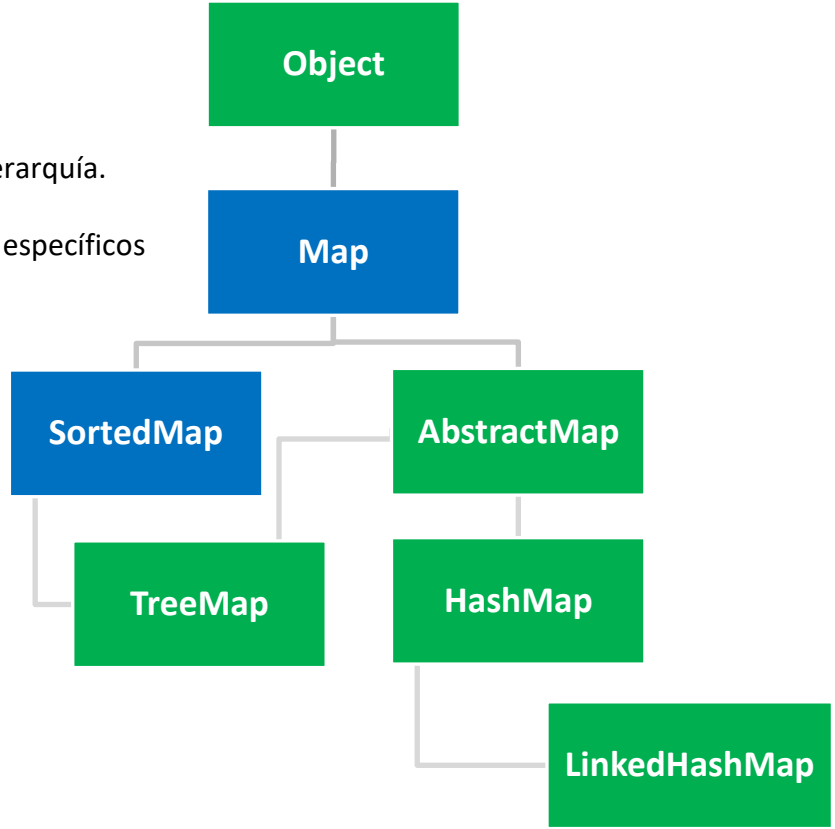
La interfaz **SortedMap** permite que las clases que la implementen tengas los elementos ordenados.

Los mapas permiten claves y valores nulos, recordando siempre que solo puede existir una clave null

Jerarquía

Estas clases e interfaces están estructuradas en una jerarquía.

A medida que se va descendiendo a niveles más específicos aumentan los requerimientos y comportamientos.



Es importante aclarar que existen otros mapas por lo cual tocaremos solo las mas usados

Métodos Map

Tipo	Método	Descripción
boolean	isEmpty()	Devuelve verdadero si este Mapa no contiene elementos.
void	clear()	Elimina todos los elementos del mapa.
Int	size()	Devuelve el número de elementos del mapa.
V	put(K key, V value)	Asocia el valor especificado con la clave especificada en este mapa, si existía ya la clave reemplaza el valor y retorna el objeto reemplazado
void	putAll(Map<? extends K,? extends V> m)	Agrega todos los elementos del mapa especificado a este mapa
V	get(Object key)	Devuelve el valor que contenga la clave especificada o null si no existe
boolean	containsKey(Object key)	Devuelve verdadero si este mapa contiene la clave especificada.
boolean	containsValue(Object value)	Devuelve verdadero si este mapa contiene el objeto especificado.
boolean	equals(Object o)	Compara el objeto especificado con esta colección para la igualdad.
int	hashCode()	Devuelve el valor del código hash para esta colección.
V	remove(Object key)	Elimina la asignación de una clave de este mapa si esta presenta y retorna el valor eliminado.
Set<Map.Entry<K,V>>	entrySet()	Devuelve una vista de colección del mapa en una colección
Collection<V>	values()	Devuelve una colección con los objetos del mapa.
Set<K>	keySet()	Devuelve en una colección Set las claves del mapa.

Métodos Map

```
K clave1, clave2, clave3, clave4;
E elemento1, elemento2, elemento3, elemento4;

Mapa<E> mapa1 = new Implementacion<>();
Mapa<E> mapa2 = new Implementacion<>();

// agrega objetos al mapa
mapa1.put(clave1, elemento1);
mapa1.put(clave2, elemento2);
mapa1.put(clave3, elemento3);
mapa1.put(clave4, elemento4);

// Devuelve el valor asiganado a la clave
System.out.println(mapa1.get(clave1));

// Mostrar los objetos en una coleccion
System.out.println(mapa1.values());

// Agregar un mapa en otro mapa
mapa2.putAll(mapa1);

// Mostrar si el mapa esta vacio  o sin elementos
System.out.println(mapa1.isEmpty());
```

```
// Longitud o tamaño del mapa
System.out.println(mapa1.size());

// Contiene Clave
System.out.println(mapa1.containsKey(clave1));

// Contiene Objeto
System.out.println(mapa1.containsValue(elemento1));

// Devuelve un set
System.out.println(mapa1.entrySet());

// Elimina un Objeto por su clave
System.out.println(mapa1.remove(clave4));

// Devuelve las claves del mapa en una coleccion Set
System.out.println(mapa1.keySet());

// Eliminar todos los objetos y sus claves del mapa
mapa1.clear();
```

Iteradores

Los mapas no son colecciones por lo tanto no extienden de **iterable** que es la interfaz que implementa el métodos **iterator**, lo que si podemos hacer es que atreves del método **keySet** que nos devuelve una colección Set con las claves obtener un iterador.

De igual forma lo único que podemos hacer es recorrer el mapa pero no podemos eliminar o modificar objetos ya que tendríamos un problema de concurrencia.

```
Iterator<E> iterador = mapa.keySet().iterator();

while (iterador.hasNext()) {
    E claveAuxiliar = iterador.next();
    System.out.println("[Clave: " + claveAuxiliar + ", Valor: " + mapa.get(claveAuxiliar) + "]");
}
```

Map.Entry

La interfaz interna Map.Entry<K,V> nos proporciona ciertos métodos para acceder a las claves y valores del mapa, además de permitirnos utilizar el for mejorado o for-each para el fácil recorrido del mapa, también nos proporciona un método para reemplazar un valor en dicho mapa.

Tipo	Método	Descripción
K	getKey()	Devuelve la clave de la iteración correspondiente.
E	getValue()	Devuelve el valor de la iteración correspondiente.
V	setValue(V value)	Reemplaza el valor de la iteración correspondiente y retorna el valor reemplazado.

```
for (Entry<K, E> entradaMapa : mapa.entrySet()) {
    K claveAuxiliar = entradaMapa.getKey();
    E valorAuxiliar = entradaMapa.getValue();

    System.out.println("[" + claveAuxiliar + ", " + valorAuxiliar + "]");

    if (valorAuxiliar.equals(Eelemento3)) {
        System.out.println("Reemplazado: "+entradaMapa.setValue(elemento2));

    }
}
```

HashMap

Este implementación almacena los elementos en una tabla hash (es un contenedor asociativo “tipo Diccionario” que permite un almacenamiento y posterior recuperación eficiente de elementos); este acceso hace que esta clase sea ideal para búsqueda, inserción y borrado de elementos.

A diferencia de la colección **HashSet** esta tabla hash no esta sincronizada lo que permite que existan claves **null**.

Representa un par Clave, Valor donde las claves son únicas (no puede tener claves duplicadas) sin ordenar (por ejemplo si hacemos un recorrido de los objetos dentro del mapa no siempre los obtendremos en igual orden) y tiene una iteración más rápida que otros mapas.

```
Map<Integer, String> nombres = new HashMap<Integer, String>();
AbstractMap<Integer, String> nombresA = new HashMap<Integer, String>();
HashMap<Integer, String> nombresB = new HashMap<Integer, String>();
```

Recordando Equals y hashCosde

- **public boolean equals(Object obj)**
- **public int hashCode()**

Ya hemos visto que podemos comparar con el método **equals** una cadena de caracteres, este método devuelve true o false según sea el caso. Ej: **"Octavio".equals("octavio");** en este caso devuelve false ya que el método es sensible a las mayúsculas y minúsculas.

Pero, ¿Cómo sabe Java que un objeto propio es igual a otro?

Primero que nada debemos saber que este es un método que heredamos de la clase **Object** y que las clases de Java sobreescriben para que los podamos utilizar, así que nosotros debemos hacer lo mismo, indicándole al método cuales son los atributos que debe comparar para que un objeto sea igual a otro.

Pero también debemos sobreescribir otro método el **hashCode**, ya que si dos objetos son iguales se debe generar el mismo numero hash para poder localizar el elementos en la tabla hash.

Por ejemplo nosotros podemos decir que un Auto es Igual a otro si poseen la misma Patente, en otros casos los objetos son iguales si poseen todos los atributos iguales; siempre dependerá del caso de uso.

Recordando Equals y hashCode

Los IDE nos proporcionan alguna manera de generar de forma automática estos métodos.

Por ejemplo en Eclipse podemos ir a:
Source –> Generate hashCode() and equals()
En el panel seleccionamos los atributos que queremos que se tengan en cuenta en estos métodos.

Es importante aclarar que si los atributos a comparar también son objetos propios debemos sobreescribir en esos objetos dichos métodos, de lo contrario la interfaz no detectara que son iguales.

Además podemos utilizar el método **equals** a partir de ahora cuando lo creamos conveniente.

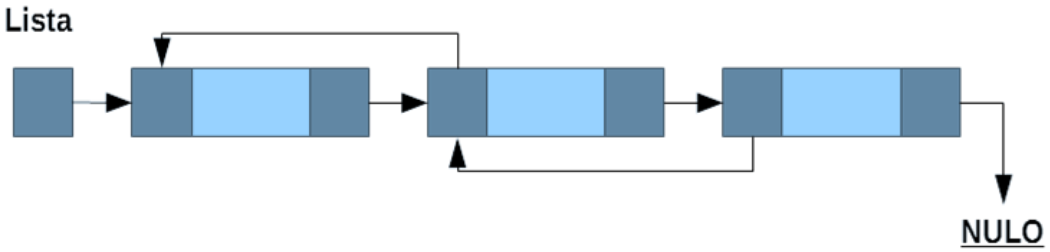
```java
@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + ((patente == null) ? 0 : patente.hashCode());
    return result;
}

@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    Auto other = (Auto) obj;
    if (patente == null) {
        if (other.patente != null)
            return false;
    } else if (!patente.equals(other.patente))
        return false;
    return true;
}
```

LinkedHashMap

Esta implementación almacena los elementos en función del orden de inserción lo que la hace un poco más costosa que **HashMap**.

```
Map<Integer, String> nombres = new LinkedHashMap<Integer, String>();
AbstractMap<Integer, String> nombresA = new LinkedHashMap<Integer, String>();
LinkedHashMap<Integer, String> nombresB = new LinkedHashMap<Integer, String>();
```

Define el concepto de elementos añadiendo una lista doblemente enlazada en la ecuación que nos asegura que los elementos siempre se recorren de la misma forma.



TreeMap

Esta implementación almacena los elementos ordenándolos en función de sus claves (Implementando el algoritmo del árbol rojo – negro: árbol de búsqueda binaria, también llamado árbol binario ordenado), por lo que es bastante más lento que **HashMap**

Este orden podrá ser Natural o Alternativo.

```
Map<Integer, String> nombres = new TreeMap<Integer, String>();
AbstractMap<Integer, String> nombresA = new TreeMap<Integer, String>();
TreeMap<Integer, String> nombresB = new TreeMap<Integer, String>();
```

```
TreeMap<Integer, String> nombresA = new TreeMap<Integer, String>(); // orden natural
TreeMap<Integer, String> nombresB = new TreeMap<Integer, String>(new ordenAlternativo());
```

Recordando Orden

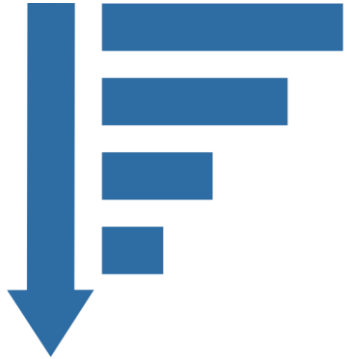
Java proporciona dos interfaces (**Comparable<T>** y **Comparator<T>**) para dar un orden a los objetos de un tipo creado por el usuario, Java ya proporciona un orden natural a los tipos envoltorio como Integer y Double o al tipo inmutable String.

Comparable (Orden Natural) esta interfaz debe ser implementada por la clase y compara el objeto **this** con otro que toma como parámetro el método **compareTo.** PD: El orden natural definido debe ser coherente con la definición de igualdad. Si **equals** devuelve true compareTo debe devolver cero.

Comparator (Orden Alternativo) esta interfaz debe ser implementada en una nueva clase comparando los dos objetos que toma como parámetros el método **compare.**

Los métodos compareTo y compare comparan dos objetos o1 y o2 (en compareTo o1 es **this**) y devuelve un entero que es:

- Negativo si o1 es menor que o2
- Cero si o1 es igual a o2
- Positivo si o1 es mayor que o2



Si el atributo a comparar es un String podemos apoyarnos en el método **compareTo** que posee de lo contrario con una resta entre atributos numéricos basta.

Cuando comparamos o1 contra o2 el orden es ascendente si invertimos la comparación ordenaría de forma descendente.

Recordando Orden

Comparable

```
public abstract class Auto implements MantenimientoMecanico, Archivo, Comparable<Auto>{
```

```
@Override
public int compareTo(Auto auto) {
    return this.patente.getNumero().compareTo(auto.getPatente().getNumero());
    // return this.puestos - auto.puestos;
}
```

Comparator

```
public class OrdenAutoMarca implements Comparator<Auto> {

    @Override
    public int compare(Auto auto1, Auto auto2) {
        return auto1.getMarca().compareTo(auto2.getMarca());
        // return auto1.puestos - auto2.puestos;
    }

}
```