educación IT
CODE YOUR FUTURE

Bootcamp Java Developer

Fase 1 - Java Analyst

Módulo 9

Mapas

Introducción

En el desarrollo de aplicaciones, muchas veces, es necesario representar los objetos en una estructura que permita una búsqueda fácil y optimizada a través de una clave sin necesidad de recorrerlos.

Los mapas, o también llamados *diccionarios*, permiten almacenar elementos asociando a cada clave un valor.

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

Cuando se intenta agregar un objeto, **si la clave ya existe, lo que hace el mapa es actualizar el valor asociado por el nuevo**.

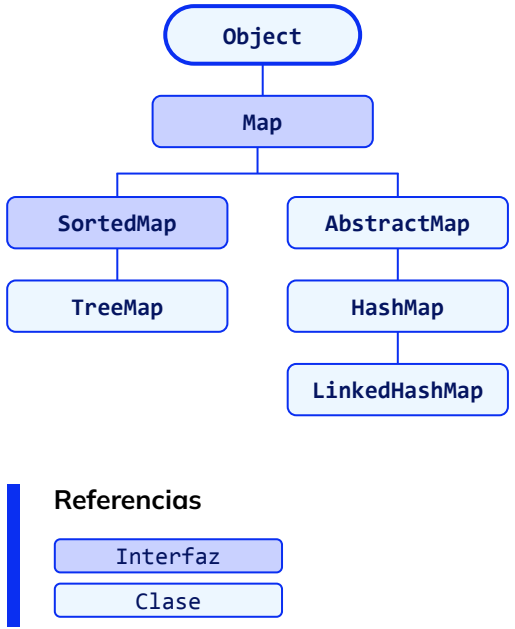
La clase **`AbstractMap`** proporciona una implementación esquelética de la interfaz `Map` y simplemente agrega implementaciones para los métodos `equals` y `hashCode`.

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

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**.



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

Más métodos en la siguiente pantalla.

Métodos Map

Tipo	Método	Descripción
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.

Ejemplos

```
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 sí se puede hacer es que, a través 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>`** 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, 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.

Ejemplo

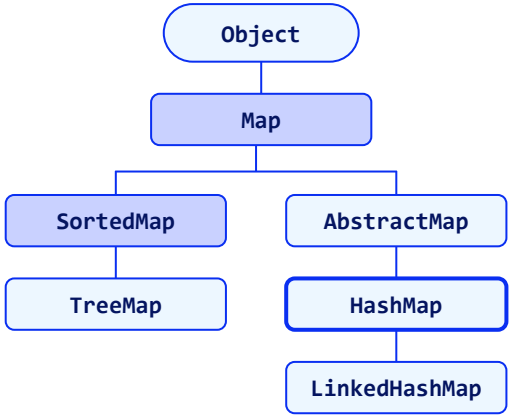
```
for (Entry<K, E> entradaMapa : mapa.entrySet()) {
    K claveAuxiliar = entradaMapa.getKey();
    E valorAuxiliar = entradaMapa.getValue();

    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 la clase sea ideal para **búsqueda, inserción y borrado de elementos**.

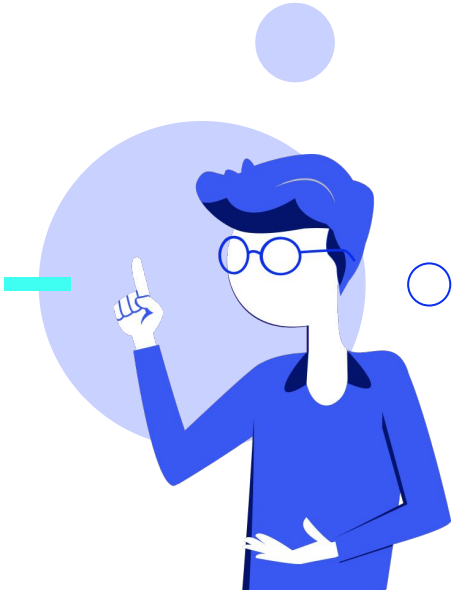
A diferencia de la colección `HashSet`, **la tabla hash no está sincronizada,** por eso permite que existan claves **`null`**.

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.



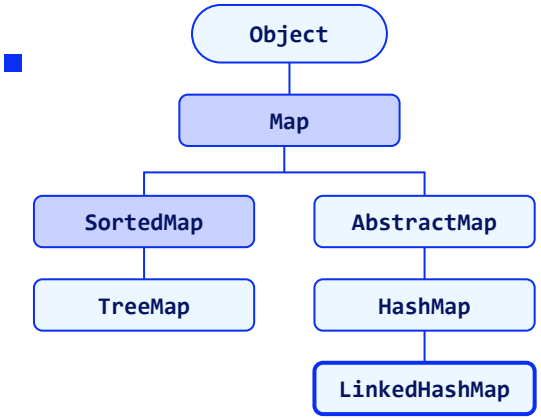
Ejemplo

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

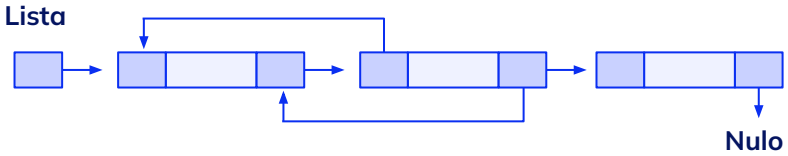


LinkedHashMap

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



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.



Ejemplo

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

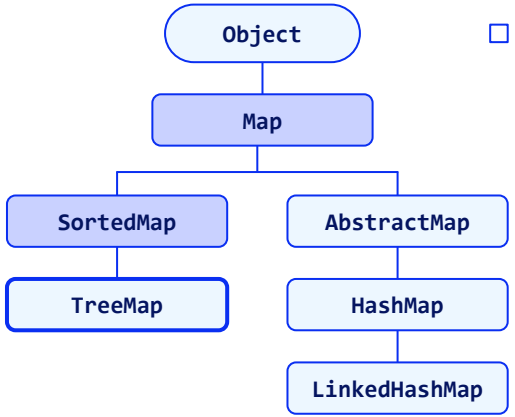
TreeMap

Esta implementación almacena los elementos ordenándolos en función de sus **claves.**

Implementa 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());
```



¡Sigamos trabajando!