Tema 7. Colecciones I

1. El tipo Map

1.1 Definición

El tipo de dato *Map*, incluido en el paquete java.util, permite modelar el concepto de aplicación: una relación entre los elementos de dos conjuntos de modo que a cada elemento del conjunto inicial le corresponde uno y solo un elemento del conjunto final. Los elementos del conjunto inicial se denominan claves (*keys*) y los del conjunto final valores (*values*).

En la figura 1 puede verse un ejemplo de *Map*. En este caso las claves son cadenas de caracteres, cada una de las cuales representa un valor numérico en diversas representaciones textuales, y los valores son enteros. Cada cadena tiene asociada el valor numérico correspondiente.

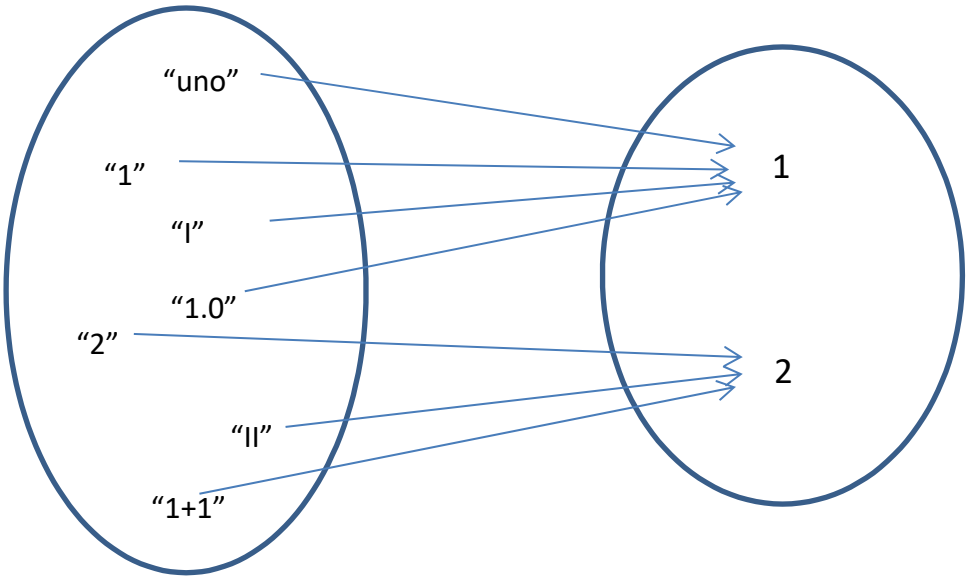


Figura 1. Modelo de un Map

Este mismo ejemplo se podría representar como una tabla con dos columnas:

Clave	Valor
“uno”	1
“1”	1
“I”	1
“1.0”	1
“2”	2
“II”	2
“1 + 1”	2

Vemos que la tabla refleja la misma información que el diagrama anterior.

Como entre las claves no puede haber elementos duplicados (una clave no puede estar asociada con más de un valor), las claves forman un conjunto (*Set*). Sin embargo sí que puede haber valores duplicados, por los que estos están en una C*ollection*.

Por tanto, un *Map* está definido por un conjunto de claves, una colección de valores y un conjunto de pares clave-valor (también llamadas entradas), que son realmente los elementos de los que está compuesto.

Dentro de la jerarquía de tipos de Java, *Map* no extiende a Collection ni a Iterable. Por tanto, no se puede aplicar un for extendido sobre sus elementos, es decir, sobre sus pares, salvo que accedamos explícitamente a ellos. Los Map se utilizan en muchas situaciones. Por ejemplo, los listines telefónicos (clave: nombre del contacto, valor: número de teléfono), listas de clase (clave: nombre del alumno, valor: calificaciones), ventas de un producto (clave: código de producto, valor: total de euros de recaudación), índices de palabras por páginas en un libro (clave: palabra, valor: lista de números de página donde aparece), la red de un Metro (clave: nombre de la estación, valor: conjunto de líneas que pasan por esa estación, o al revés, clave: número de línea, valor: lista de estaciones de esa línea), etc. Tiene también muchas otras aplicaciones en

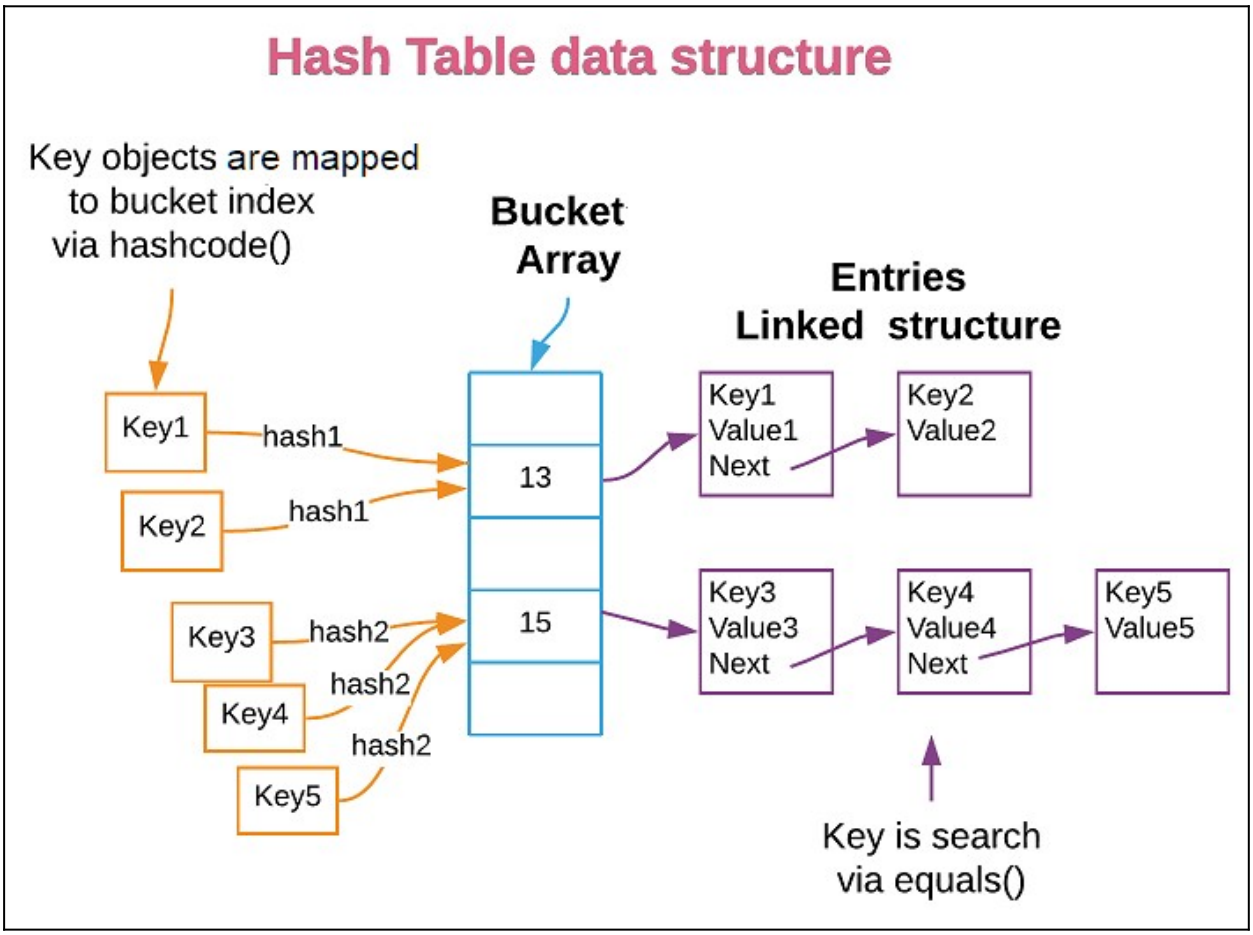
informática: representar diccionarios o propiedades, almacenar en memoria tablas de bases de datos, cachés, etc.

Centrándonos en Java, la interfaz Map tiene dos tipos genéricos, que suelen denominarse K (de Key) y V (de Value): Map<K, V>. El Map del ejemplo anterior sería Map<String, Integer>.

La clase que implementa la interfaz Map es HashMap (aunque también se puede utilizar TreeMap, que se verá más adelante). La clase HashMap obliga a definir de forma correcta el método hashCode del tipo de las claves para evitar comportamientos incorrectos, como claves repetidas. Asimismo, al igual que pasa en el tipo Set, los objetos mutables que se introducen en un Map son "vistas" o referencias a objetos y, por tanto, si estos cambian, el Map puede dejar de ser coherente, esto es, podría tener claves repetidas.

Un Map cuyas claves sean String y cuyos valores sean Integer se definiría e inicializaría de la siguiente forma:

```
Map<String, Integer> m = new HashMap<String, Integer>();
```



1.2 Métodos del tipo Map

Los métodos del tipo Map se relacionan a continuación. La información completa sobre este tipo se puede encontrar en la API de Java[1].

`void`	**`clear`**`()` Elimina todos los elementos (pares o entradas) del Map.

boolean	**containsKey**(**Object** key) Devuelve `true` si el Map contiene la clave especificada.
boolean	**containsValue**(**Object** value) Devuelve `true` si una o más claves del Map tienen asociadas el valor especificado.
V	**get**(**Object** key) Devuelve el valor asociado con la clave especificada o `null` si esa clave no está asociada con ningún valor (la clave no está en el conjunto de claves).
boolean	**isEmpty**() Devuelve `true` si el Map no contiene ningún par.
Set<**K**>	**keySet**() Devuelve un Set que es una vista de las claves que contiene el Map.
V	**put**(**K** key, **V** value) Asocia el valor con la clave especificada. Devuelve el valor previamente asociado con la clave si esta ya estaba en el Map o `null`, en caso contrario.

[1] http://docs.oracle.com/javase/7/docs/api/java/util/Map.html

`V`	**`remove`**`(Object key)` Elimina el par que tiene como clave el parámetro especificado. Devuelve el valor previamente asociado con la clave o `null`, si la clave no existía.
`int`	**`size`**`()` Devuelve el número de pares del Map.
`Collection<V>`	**`values`**`()` Devuelve una Collection que es una vista de los valores del Map.
`Set<Map.Entry<K,V>>`	**`entrySet`**`()` Devuelve una vista del conjunto de todos los pares del Map (el tipo Map.Entry se verá más adelante).
`void`	**`putAll`**`(Map<? extends K, ? extends V> m)` Añade al Map todos los pares contenidos en `m`. Tiene el mismo efecto que hacer put de todos los elementos de `m`.

Tanto los conjuntos devueltos por keySet y entrySet como la colección devuelta por va,lues son vistas del Map original, por lo que las modificaciones que se realicen sobre estos repercuten en los pares almacenados en el Map original (con las posibles repercusiones negativas) y viceversa. Hay que destacar que esos tres objetos son iterables. El orden en el que el iterador recorre los elementos de los conjuntos (keySet y entrySet) o la colección (values) es impredecible.

La clase HashMap tiene dos constructores: el constructor sin argumentos, que construye un Map vacío, y el constructor con un argumento de tipo Map (constructor copia), que construye un Map con los mismos pares que el Map que se le pasa como argumento (equivale a crear un Map vacío y aplicar putAll). Existe una clase LinkedHashMap, que se comporta igual que HashMap excepto en que los iteradores sobre las claves, los valores o los pares los devuelven en el orden en que se insertaron.

1.3 El tipo Map.Entry

Los pares de elementos (también llamados entradas) de los que está compuesto un Map<K, V> son de un tipo que viene implementado por la interfaz Map.Entry<K, V>[2]. Esta interfaz, aparte de la operación equals que permite comparar pares para comprobar su igualdad (y el correspondiente hashCode), tiene tres operaciones:

`K`	**`getKey`**`()` Devuelve la clave del par.
`V`	**`getValue`**`()` Devuelve el valor del par.
`V`	**`setValue`**`(V value)` Modifica el valor del par, dándole como nuevo valor `value`. Devuelve el valor (segunda componente) previo del par.

[2] Más información en: http://docs.oracle.com/javase/7/docs/api/java/util/Map.Entry.html

Por ejemplo, si tenemos una entrada (par) p que asocie "II" con 2, p.getKey() devuelve la cadena "II" y p.getValue() devuelve el entero 2; si se escribe p.setValue(3), devuelve el entero 2 y modifica el valor asociado con la clave dándole el valor 3. El toString de las entradas es de la forma clave=valor, de modo que la representación como cadena del par resultante sería II=3.

1.4 Inicialización de un objeto de tipo Map

La algoritmia para inicializar un objeto de tipo Map siempre sigue la misma estructura, que se puede ver en el esquema siguiente:

a. Creación del objeto (*HashMap*)
b. Para cada *clave*
    c. Si la *clave* ya está en el Map (*containsKey*)
        d. Obtener el *valor* asociado a esa clave (*get*)
        e. Actualizar el *valor* u obtener *nuevovalor* a partir de *valor*
        f. Si es necesario, añadir al Map el par *clave*, *nuevovalor* (*put*)
    g. Si la *clave* no está en el Map
        h. Inicializar *valor*
        i. Añadir al Map el par *clave, valor* (*put*)

En el paso a. se construye el objeto tipo Map invocando al constructor de la clase HashMap. El paso b. normalmente incluye un recorrido sobre el conjunto de claves. Para cada clave se calculará el valor correspondiente, normalmente mediante algún tipo de cálculo u operación de acceso a una colección de datos a partir de la clave. Una vez tenemos el par clave-valor a insertar en el Map, nos preguntamos en el paso c. mediante el método containsKey si la clave ya estaba anteriormente en el Map.

Si la clave ya estaba, debemos obtener el valor asociado a esa clave en el Map mediante el método get. Dependiendo de si la actualización consiste en sustituir un nuevo valor por el anterior o actualizar el ya existente, se ejecutan los pasos e. y/o f. Por ejemplo, si el Map asocia a cada palabra una lista con los números de las páginas de un libro donde aparece esa palabra, la actualización será añadir a la vista de la lista que conforma el valor un nuevo elemento y, por tanto, no hace falta invocar al método put (ya que List es un tipo mutable, y podemos hacerle modificaciones). Sin embargo, si el Map es una asociación entre el código de un producto y sus ventas, para actualizar las ventas, al invocar al método get obtendríamos el valor de las ventas pasadas, y a este dato se le deberían sumar las nuevas ventas. Como Double es un tipo inmutable, la operación suma devolverá un objeto nuevo y, por tanto, deberíamos actualizar, mediante el método put, la asociación de ese nuevo objeto de tipo Double con las ventas realizadas, y el código del producto como clave.

Si la clave no estaba en el Map, se calcula el valor inicial en el paso h, para añadir el par clave-valor mediante el método put en el paso i.

Ejemplo 1

Supongamos que se quiere calcular la frecuencia absoluta de aparición o número de veces que aparecen los caracteres en un String. Para ello se define un Map cuyas claves son de tipo Character y cuyos valores son de

tipo Integer. El método recibirá un String y devolverá un objeto de tipo Map<Character, Integer>. El código del método sería:

```
public static Map<Character, Integer> contadorCarac(String frase) {
	Map<Character,Integer> contador = new HashMap<Character,Integer>();

	frase = frase.toUpperCase(); // todos los caracteres en mayúsculas

	for (int i = 0; i < frase.length(); i++) {
		Character caracter = frase.charAt(i);
		if (contador.containsKey(caracter)) {
			Integer valor = contador.get(caracter);
			valor++;
			contador.put(caracter, valor);
		} else {
			contador.put(caracter, 1);
		}
	}
	return contador;
}
```

Nótese como el método sigue fielmente el esquema antes indicado. Se recorren los caracteres del String de entrada mediante un for clásico. Para cada carácter se pregunta si ya está en el Map, de forma que si está, se obtiene el valor correspondiente al número de veces que ha aparecido anteriormente, se incrementa en uno y se vuelve a actualizar la asociación entre el carácter y el nuevo valor (un nuevo objeto, puesto que Integer es un tipo inmutable). Al ser Integer un tipo inmutable, la llamada al método put es obligatoria. En el caso de que el carácter no estuviera en el Map, se inicializa la frecuencia a 1.

Ejemplo 2

Supongamos que tenemos una lista de String que denominamos *palabras* y se quiere obtener un índice de las posiciones que ocupan las cadenas de la lista *palabras* mediante una lista de enteros. El argumento de entrada será por tanto de tipo List<String> y el argumento de salida un Map<String, List<Integer>>. Por ejemplo, para una lista que contuviera las palabras de la cadena “la palabra que más aparece en este texto es la palabra palabra”, la salida sería

```
{más=[3], texto=[7], aparece=[4], palabra=[1, 10, 11], la=[0, 9], en=[5], que=[2],
este=[6], es=[8]}
```

Indicando que ‘la’ está en las posiciones 0 y 9, ‘palabra’ en la 1, 10 y 11, etc. El método tendría el siguiente código:

```
public static Map<String, List<Integer>> indicePal(List<String> palabras) {
	Map<String,List<Integer>> indice = new HashMap<String,List<Integer>>();
	int pos = 0;

	for (String palabra: palabras) {
		if (indice.containsKey(palabra)) {
			indice.get(palabra).add(pos);
		} else {
			List<Integer> lis = new ArrayList<Integer>();
			lis.add(pos);
```

```
                    indice.put(palabra, lis);
                }
                pos++;
        }
        return indice;
}
```

Nótese que la principal diferencia con el ejercicio anterior es que no es necesario invocar al método put en el caso de que la palabra ya esté en el índice, puesto que List es un tipo mutable. Observe que la sentencia

```
indice.get(palabra).add(pos);
```

es equivalente a

```
List<Integer> lis = indice.get(palabra);
lis.add(pos);
```

Igualmente, debe observar que incluso en este caso, tampoco es necesario invocar a put ya que lis es una vista de la lista correspondiente y por tanto al añadir un elemento a lis, ya lo estamos haciendo a la lista guardada en el conjunto imagen del Map.

Este problema se puede restringir a que aparezcan sólo las páginas en las que aparecen las palabras importantes de un texto: las que se denominan palabras clave. (Por ejemplo, en este tema nos interesaría saber en qué páginas aparecen las palabras “Map”, “HashMap”, “TreeMap”, “Guava”, etc., pero no en cuáles aparece “en”, “la”, “las”, etc.). Supongamos que tenemos las palabras claves en un Set de String, ¿cómo modificaría el código del método anterior para que en el Map sólo estuvieran las palabras clave?

2. El tipo SortedMap

2.1 Definición

El tipo SortedMap es un subtipo de Map en el que el conjunto de las claves está ordenado. La clase que implementa SortedMap es TreeMap. Es necesario que el tipo de las claves tenga un orden natural (es decir, que implemente Comparable) o que se proporcione un orden alternativo mediante un Comparator. Por tanto, una inicialización de SortedMap como la siguiente:

```
SortedMap<String, List<Integer>> indice = new TreeMap<String,
```

Crea un SortedMap donde las claves son de tipo String y están ordenadas por el orden natural. Para crear un SortedMap por un orden alternativo debemos invocar al constructor pasándole como argumento un comparador:

```
Comparator<Libro> cmpLibroNumPag = new ComparadorLibroNumPaginas();
SortedMap<Libro, List<Integer>> mp = new TreeMap<Libro, List<Integer>>(cmpLibroNumPag);
```

Las reflexiones que se hicieron para el tipo SortedSet respecto del orden definido por un Comparator siguen siendo válidas para SortedMap. Esto es, dos claves c1 y c2 son iguales para un SortedMap si compare(c1,c2) == 0. Por tanto, si el Comparator no rompe los empates por el orden natural, en el SortedMap anterior sólo habría un Libro clave por cada número de páginas.

2.2 Métodos

SortedMap hereda de Map. Por tanto, todos los métodos de Map son válidos también para un objeto SortedMap. Además, proporciona algunos métodos para manejar el orden en el conjunto de las claves como podemos ver a continuación[3]:

`K`	**`firstKey`**`()` Devuelve la primera clave del Map, según el orden que tenga inducido.
`K`	**`lastKey`**`()` Devuelve la última clave del Map.
`SortedMap<K,V>`	**`headMap`**`(K toKey)` Devuelve una vista de Map con los pares cuyas claves son estrictamente inferiores a `toKey`.
`SortedMap<K,V>`	**`subMap`**`(K fromKey, K toKey)` Devuelve una vista de la porción del Map cuyas claves están en el rango `fromKey`, incluida, a `toKey`, excluida.
`SortedMap<K,V>`	**`tailMap`**`(K fromKey)` Devuelve una vista del Map con los pares que son posteriores o iguales a `fromKey`.
`Comparator<? super K>`	**`comparator()`** Devuelve el comparador utilizado para ordenar las claves, o `null` si se usa el orden natural.

Los métodos keySet, entrySet y values tienen el mismo perfil que en Map, esto es, devuelven Set, Set y Collection, respectivamente. Los iteradores sobre ellos recorren los elementos en el orden en el que están ordenadas las claves.

La clase TreeMap tiene cuatro constructores:

- Sin argumentos: construye un conjunto ordenado vacío que utilizará el orden natural de las claves.
- Con un argumento de tipo Comparator sobre las claves: construye un conjunto ordenado vacío que utiliza para ordenar las claves el orden inducido por el comparador.
- Con un argumento de tipo Map: onstruye un SortedMap con los mismos pares que el Map que recibe como argumento, pero donde las claves estarán ordenadas según el orden natural de eéstas.
- Con un argumento de tipo SortedMap: construye un SortedMap con los mismos elementos que el que recibe como argumento y ordenado según el mismo criterio (sea el natural o uno inducido). Es el constructor copia.

[3] http://docs.oracle.com/javase/7/docs/api/java/util/SortedMap.html