Tema 3: Colecciones y Map

Autor: Miguel Toro, Mariano González. **Revisor**: Fermín Cruz, Pepe Riquelme. **Última modificación**: 3/5/2019

Tabla de contenidos

1. Introducción

En el Bloque 1 presentamos el concepto de agregados de datos y vimos el manejo básico de algunos de ellos, concretamente los tipos List, Set y Map. La adecuada utilización de tipos de datos agregados predefinidos en la API de Java, como los citados, simplifica mucho el desarrollo de programas correctos en Java y tiene muchas otras ventajas:

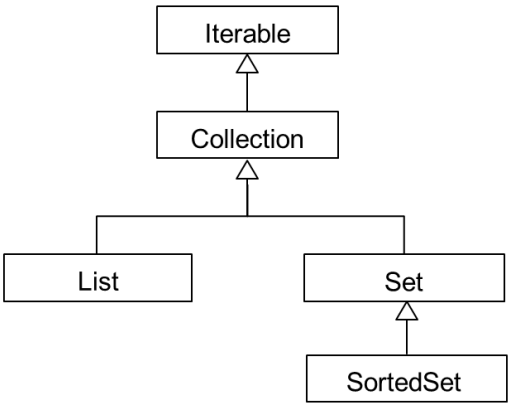
- Reduce el esfuerzo de programación puesto que proporciona estructuras de datos y algoritmos útiles.
- Incrementa la velocidad y la calidad de los programas, puesto que Java ofrece implementaciones optimizadas y libres de errores.
- Simplifica la interoperabilidad y la reemplazabilidad entre aplicaciones, puesto que facilita estructuras que se pueden intercambiar entre distintos componentes.
- Reduce esfuerzos de aprendizaje y diseño.

En este capítulo vamos a ver las colecciones, modeladas mediante la interfaz *Collection* y dos de las interfaces que heredan de ella: *List*, que modela las listas, y *Set*, que modela los conjuntos. También veremos el tipo *Map*, que modela el concepto matemático de Aplicación.

2. La interfaz Collection

La interfaz *Collection*, que define un tipo que podemos denominar Colección, está definida en el paquete *java.util*. Una colección es un tipo muy general, que agrupa objetos (elementos) de un mismo tipo; su comportamiento específico viene determinado por sus subinterfaces, que pueden admitir elementos duplicados o no, y cuyos elementos pueden estar ordenados o no según determinado criterio. No existe una implementación específica de la interfaz *Collection*; sí la tienen sus subinterfaces. El tipo *Collection* se utiliza para declarar variables o parámetros donde se quiere la máxima generalidad posible, esto es, que puedan ser utilizados tanto por listas como por conjuntos (*Collection* tiene otros subtipos que se tratarán más adelante).

La interfaz *Collection* es genérica, por lo que hablaremos de *Collection<E>*. Hereda de *Iterable<E>*, lo que implica que las colecciones, así como las listas y los conjuntos, subtipos suyos, son iterables y se puede utilizar con ellas el *for* extendido. La siguiente figura representa la jerarquía de interfaces de las colecciones.



Las operaciones del tipo *Collection* se especifican a continuación[1]. En general, el valor devuelto por los métodos add, addAll, remove, removeAll y retainAll será *true* si la colección queda modificada por la aplicación del método y *false* en caso contrario.

`boolean`	**`add`**`(E e)` Añade un elemento a la colección, devuelve *false* si no se añade.
`boolean`	**`addAll`**`(Collection<? extends E> c)` Añade todos los elementos de c a la colección que invoca. Es el operador unión. Devuelve *true* si la colección original se modifica.
`void`	**`clear`**`()` Borra todos los elementos de la colección.
`boolean`	**`contains`**`(Object o)` Devuelve *true* si o está en la colección invocante.
`boolean`	**`containsAll`**`(Collection<?> c)` Devuelve *true* si la colección que invoca contiene todos los elementos de c.
`boolean`	**`isEmpty`**`()` Devuelve *true* si la colección no tiene elementos.
`boolean`	**`remove`**`(Object o)` Borra el objeto o de la colección que invoca; si no estuviera se devuelve *false*.
`boolean`	**`removeAll`**`(Collection<?> c)` Borra todos los objetos de la colección que invoca que estén en c. Devuelve *true* si la colección original se modifica.
`boolean`	**`retainAll`**`(Collection<?> c)` En la colección que invoca sólo se quedarán aquellos objetos que están en c. Por tanto, es la intersección entre ambas colecciones. Devuelve *true* si la colección original se modifica.
`int`	**`size`**`()` Devuelve el número de elementos.

3. El tipo List

Conceptualmente, las listas representan colecciones de elementos en los que importa cuál es el primero, el segundo, etc. Cada elemento está referenciado mediante un índice; el índice del primer elemento es el 0. Las listas pueden contener elementos duplicados.

Desde el punto de vista de la API de Java, las listas son un subtipo de *Collection*: la interfaz *List* hereda de la interfaz *Collection*. Por tanto, tienen todas las operaciones vistas en el apartado anterior, aunque debemos prestar atención a la semántica de algunas operaciones: La operación *addAll* añade los elementos de la lista que se pasa al final de la lista sobre la que se invoca. Si en la lista sobre la que se invoca hay elementos duplicados, la operación *removeAll* elimina de esta todas las instancias de los elementos que aparecen en la lista que se pasa. De manera análoga se comporta *retainAll*: si en la lista sobre la que se invoca un elemento

[1] Puede obtener más información de la interfaz *Collection* en la documentación de la API de Java: http://docs.oracle.com/javase/8/docs/api/java/util/Collection.html

aparece *n* veces, y este aparece en la lista que se pasa (independientemente del número de veces que aparezca), en la lista resultado permanecerán las *n* apariciones del elemento.

Veamos esto con un ejemplo. Si se ejecuta el siguiente código,

```
List<String> l1 = new LinkedList<String>();
List<String> l2 = new LinkedList<String>();
l1.add("A");
l1.add("B");
l1.add("C");
l2.add("B");
l2.add("B");
l1.removeAll(l2);
System.out.println("l1 después de l1.removeAll(l2): " + l1);

l1.clear();
l2.clear();
l1.add("A");
l1.add("B");
l1.add("C");
l2.add("B");
l2.add("B");
l2.removeAll(l1);
System.out.println("l2 después de l2.removeAll(l1): " + l2);

l1.clear();
l2.clear();
l1.add("A");
l1.add("B");
l1.add("C");
l2.add("B");
l2.add("B");
l2.retainAll(l1);
System.out.println("l2 después de l2.retainAll(l1): " + l2);
```

Se obtiene como salida lo siguiente:

```
l1 después de l1.removeAll(l2): [A, C]
l2 después de l2.removeAll(l1): []
l2 después de l2.retainAll(l1): [B, B]
```

La interfaz *List*[2] aporta varias operaciones específicas, que no aparecen en *Collection*; en las operaciones que tienen índice, si no se cumple la restricción sobre este se eleva la excepción *IndexOutOfBoundsException*.

`void`	**`add`**`(int index, E element)` Inserta el elemento especificado en la posición especificada. El que estaba previamente en la posición `index` pasará a la posición `index + 1`, el que estaba en la posición `index + 1` pasará a la posición `index + 2`, etc. Los valores lícitos para `index` son `0 ≤ index ≤ size()`.

[2] Puede encontrarse más información en
http://docs.oracle.com/javase/8/docs/api/java/util/List.html

boolean	**addAll**(int index, **Collection**<? extends E> c) Inserta todos los elementos de c en la posición especificada, desplazando el que estaba previamente en la posición index a la posición index + c.size(), etc. Los valores lícitos para index son 0 ≤ index ≤ size().
E	**get**(int index) Devuelve el elemento de la lista en la posición especificada. Los valores lícitos para index son 0 ≤ index < size().
int	**indexOf**(**Object** o) Devuelve el índice donde se encuentra por primera vez el elemento o (si no está devuelve -1).
int	**lastIndexOf**(**Object** o) Devuelve el índice donde se encuentra por última vez el elemento o (si no estuviera devuelve -1).
E	**remove**(int index) Borra el elemento de la posición especificada. Los valores lícitos para index son 0 ≤ index < size().
E	**set**(int index, **E** element) Reemplaza el elemento de la posición indicada por el que se da como argumento. Los valores lícitos para index son 0 ≤ index < size().
List<E>	**subList**(int fromIndex, int toIndex) Devuelve una **vista** de la porción de la lista entre fromIndex, inclusive, and toIndex, sin incluir. Los valores lícitos de fromIndex y toIndex son 0 ≤ fromIndex ≤ toIndex ≤ size().

La operación *subList* devuelve una vista de la lista original. Esto quiere decir que las operaciones que se realicen sobre la sublista se verán reflejadas en la lista original y viceversa. Hay que prestar atención al hecho de que si se produce un cambio estructural que afecte al tamaño de la lista original, pueden ocurrir comportamientos extraños, en particular la elevación de una excepción cuando se utiliza la sublista.

Veamos el comportamiento de *subList* con un ejemplo. Supongamos que se ejecuta el código:

```
List<String> ls = new LinkedList<String>();
ls.add("A");
ls.add("B");
ls.add("C");
ls.add("D");
ls.add("E");
List<String> subLs = ls.subList(1, 4);
System.out.println("Sublista: " + subLs);
ls.set(2, "F");
System.out.println(
      "Sublista después de modificar el elemento 2 de la lista: "
      + subLs);
subLs.remove(1);
System.out.println("Sublista después de eliminar el elemento 1: "
      + subLs);
System.out.println(
      "Lista original después de modificar la sublista: " + ls);
```

```
subLs.add("X");
subLs.add("Y"); // añade "X" e "Y" al final de la sublista
System.out.println("Sublista después de añadirle X e Y: " + subLs);
System.out.println("Lista después de la modificación de la sublista: "
      + ls);
ls.remove(0);
System.out.println("Lista después de eliminar el primer elemento: "
      + ls);
```

El resultado que se obtiene como salida será el siguiente:

```
Sublista: [B, C, D]
Sublista después de modificar el elemento 2 de la lista: [B, F, D]
Sublista después de eliminar el elemento 1: [B, D]
Lista original después de modificar la sublista: [A, B, D, E]
Sublista después de añadirle X e Y: [B, D, X, Y]
Lista después de la modificación de la sublista: [A, B, D, X, Y, E]
Lista después de eliminar el primer elemento: [B, D, X, Y, E]
```

En este otro ejemplo, si se ejectuta el código:

```
List<String> ls = new LinkedList<String>();
ls.add("A");
ls.add("B");
ls.add("C");
ls.add("D");
ls.add("E");
List<String> subLs = ls.subList(1, 4);
System.out.println("Sublista: " + subLs);
ls.remove(2);
System.out.println("Sublista después de modificar la lista: "+ subLs);
```

Se obtiene como salida:

```
Sublista: [B, C, D]
Exception in thread "main" java.util.ConcurrentModificationException
```

Las implementaciones de las listas son *ArrayList* y *LinkedList*[3]. La elección de una u otra implementación dependerá del tipo de operaciones que realicemos sobre ellas. Si se va a acceder preferentemente a los elementos mediante índice o a realizar búsquedas, debe usarse *ArrayList*. Si preferentemente se van a realizar operaciones de inserción o borrado al principio o al final debe usarse *LinkedList*. Las dos clases tienen dos constructores cada una: uno sin argumentos, que construye una lista vacía y otro que recibe una *Collection* y construye una lista con los elementos de la colección, en el orden en el que un *for* extendido devolvería sus elementos.

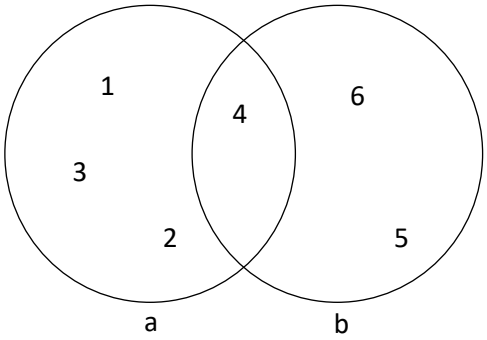
[3] La información completa se puede encontrar en http://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html y http://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html

4. El tipo Set

El tipo Set se corresponde con el concepto matemático de conjunto: un agregado de elementos en el que no hay orden (no se puede decir cuál es el primero, el segundo, el tercero, etc.) y donde no puede haber elementos repetidos.

Dados dos conjuntos (de Integer) a y b:

a = {1, 2, 3, 4} b = {4, 5, 6}



recordemos que:

a ∪ b = {1,2,3,4,5,6}

a – b = {1,2,3}

a ∩ b = {4}

La interfaz *Set*[4] no aporta ningún método extra a los que ya tiene *Collection*; por tanto, sus métodos son los de *Collection* y solo estos. Las operaciones *addAll*, *retainAll* y *removeAll* se pueden identificar con la unión, intersección y diferencia de conjuntos, respectivamente; la operación *contains* equivale a la pertenencia en conjuntos (∈); la operación *containsAll* se corresponde con la de subconjunto (⊆).

La implementación más habitual del tipo *Set* es la clase *HashSet*[5]. Tiene dos constructores: uno vacío, que construye un conjunto vacío, y otro que recibe una colección y construye un conjunto con los elementos de la colección (sin duplicados).

El *for extendido* aplicado a un conjunto devuelve todos sus elementos en un orden que no es predecible. Igual sucede si se muestra el conjunto de una vez con println. Por ejemplo, si se ejecuta el código

```
Set<Character> s = new HashSet<Character>();
s.add('A'); s.add('B'); s.add('P'); s.add('Q');
System.out.println(s);
```

Se puede obtener una salida como la siguiente, aunque también podría ser cualquier otra combinación de los cuatro elementos:

```
[P, A, Q, B]
```

[4] Más información en http://docs.oracle.com/javase/8/docs/api/java/util/Set.html

[5] Más información en http://docs.oracle.com/javase/8/docs/api/java/util/HashSet.html

5. El tipo SortedSet

El tipo *SortedSet* es un subtipo de los conjuntos (por tanto los elementos no están indexados y no puede haber elementos repetidos), en el que existe una relación de orden entre los elementos que permite decir cuál va antes y cuál después. Para simplificar hablaremos de menores o mayores, entendiendo que significa "va antes o después", según el criterio de ordenación.

La implementación de los conjuntos ordenados es la clase *TreeSet*. Esta clase tiene varios constructores:

- Un constructor vacío, que crea un conjunto ordenado vacío donde los elementos se ordenarán según su orden natural (los elementos tienen que implementar *Comparable*).
- Un constructor con un argumento de tipo *Comparator*, que crea un conjunto ordenado vacío cuyos elementos se ordenarán según el orden inducido por el comparador.
- Un constructor con un argumento de tipo *Collection*, que crea un conjunto ordenado con los elementos de la colección ordenados según su orden natural (los elementos de la colección tienen que implementar *Comparable*).
- Un constructor que recibe un *SortedSet*, que crea un conjunto ordenado con los mismos elementos que el que recibe como argumento y usando su mismo orden.

El *for extendido* sobre los elementos de un *SortedSet* los devuelve en el orden que tienen inducido. Por ejemplo, dado el conjunto ordenado anterior, el código

```
SortedSet<Character> ss = new TreeSet<Character>();
ss.add('X');
ss.add('C');
ss.add('F');
ss.add('P');
ss.add('R');
ss.add('Q')
for (Character ch: ss) {
      System.out.println(ch);
}
```

Produce la siguiente salida:

```
C
F
P
Q
R
X
```

6. La clase de utilidad Collections

El paquete *java.util* contiene la clase de utilidad *Collections*[6]. Esta clase está formada por métodos estáticos que permiten realizar operaciones sofisticadas sobre las colecciones, como invertir una lista, barajarla, ordenarla, buscar una sublista dentro de una lista, encontrar el máximo o el mínimo de los elementos de una colección, contar las veces en las que aparece un elemento, etc.

Algunos de sus métodos (los más usados) son:

`static <T> boolean`	**`addAll`**`(Collection<? super T> c, T... elements)` Añade a la colección los elementos indicados en `elements`.
`static void`	**`fill`**`(List<? super T> l, T o)` Reemplaza todos los elementos de la lista `l` por `o`.
`static <T extends Object & Comparable <? super T>> T`	**`max`**`(Collection<? extends T> coll)` Devuelve el elemento máximo de la colección `coll` según el orden natural de sus elementos.
`static <T extends Object & Comparable<? super T>> T`	**`min`**`(Collection<? extends T> coll)` Devuelve el elemento mínimo de la colección `coll` según el orden natural de sus elementos.
`static void`	**`reverse`**`(List<?> list)` Invierte los elementos de la lista `list`.
`static void`	**`shuffle`**`(List<?> list)` Mezcla aleatoriamente los elementos de la lista `list.`
`static <T extends Comparable<? super T>> void`	**`sort`**`(List<T> list)` Ordena la lista según el orden natural del tipo.

Veamos un ejemplo de uso de la clase Collections. Si se ejecuta el código

```
List<String> l = new LinkedList<String>();
l.add("R");
l.add("T");
l.add("B");
l.add("A");
l.add("M");
System.out.println(l);
Collections.reverse(l);
System.out.println(l);
Collections.sort(l);
System.out.println(l);
Collections.fill(l, "X");
System.out.println(l);
```

La salida será:

```
[R, T, B, A, M]
[M, A, B, T, R]
[A, B, M, R, T]
[X, X, X, X, X]
```

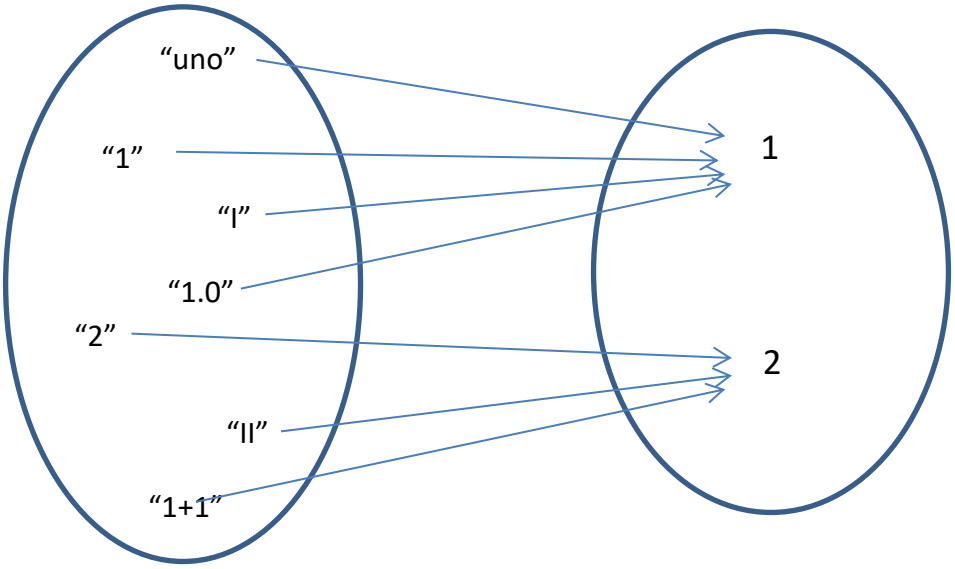
[6] Se puede encontrar una descripción exhaustiva en http://docs.oracle.com/javase/8/docs/api/java/util/Collections.html

7. El tipo Map

7.1 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 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.



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 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. En general, deben usarse claves que sean tipos inmutables.

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>();
```

7.2 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[7].

`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.

[7] Más información en http://docs.oracle.com/javase/8/docs/api/java/util/Map.html

`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.
`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 *values* 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.

La operación *equals* entre dos maps devuelve true si y solo si los conjuntos devueltos por *entrySet* en ambos conjuntos (sus pares) son iguales.

7.3 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>[8]. 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.

[8] Más información en http://docs.oracle.com/javase/8/docs/api/java/util/Map.Entry.html

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

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.

7.4 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 de tipo Map
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*, en el paso *c* nos preguntamos 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

```
{que=[2], aparece=[4], este=[6], texto=[7], palabra=[1, 10, 11],
la=[0, 9], en=[5], más=[3], 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> lista = new ArrayList<Integer>();
                    lista.add(pos);
                    indice.put(palabra, lista);
              }
              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> lista = indice.get(palabra);
lista.add(pos);
```

Igualmente, debe observar que incluso en este caso, tampoco es necesario invocar a *put* ya que *lista* es una vista de la lista correspondiente y por tanto al añadir un elemento a *lista*, 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 documento nos interesaría saber en qué páginas aparecen las palabras “Map”, “HashMap”, “TreeMap”, 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?

8. El tipo SortedMap

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 bien que se indique el orden mediante un comparador. Por ejemplo, una inicialización de SortedMap como la siguiente:

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

crea un SortedMap donde las claves son de tipo String y están ordenadas por su orden natural.

La clase *TreeMap* tiene varios constructores:

- Sin argumentos: construye un *SortedMap* vacío que utilizará el orden natural de las claves.
- Con un argumento de tipo *Comparator*: construye un *SortedMap* vacío que utilizará el orden inducido por el comparador.
- Con un argumento de tipo *Map*: construye un *SortedMap* con los mismos pares que el *Map* que recibe como argumento, pero donde las claves estarán ordenadas según el orden natural de estas.

- 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.