Capítulo 12

Collections y Maps

Estructuras de datos dinámicas

Hay tres tipos fundamentales de estructuras ligadas a la interfaz Collection:

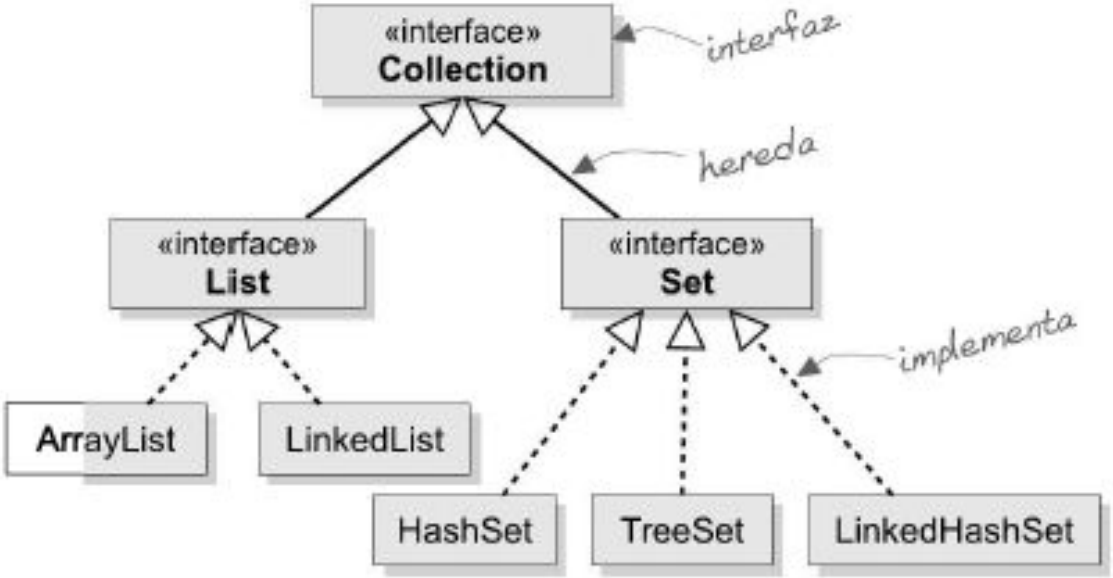
- Listas: manejan sucesiones de datos que pueden estar repetidos y cuyo orden puede ser importante.
- Conjuntos: el orden de los datos no es relevante y las repeticiones tampoco, simplemente si pertenece o no a la estructura.
- Mapas o diccionarios: sirven para guardar datos identificados por claves que no se repiten.

De estas estructuras la más conocida y usada es la lista.

Cuando declaremos una de estas estructuras, especificaremos la clase de objetos que se pueden insertar en ellas, lo que permite hacer un control de tipos más eficiente. Además, como ya veremos, la interfaz Collection permite una gran flexibilidad para intercambiar datos entre distintas colecciones.

Interfaz Collection

Las colecciones sirven para almacenar objetos. A diferencia de los arrays, las colecciones son estructuras dinámicas, es decir, pueden almacenar un número variable de elementos, permitiendo añadir o quitar elementos.
Disponemos de una serie de estructuras dinámicas que comparten un conjunto de métodos declarados en la interfaz Collection.



Interfaz List - Clases de Listas

Las listas nos servirán para almacenar datos que se pueden repetir y cuyo orden de inserción puede ser relevante.

Hay dos implementaciones de lista, las clases ArrayList y LinkedList. Ambas clases implementan todos los métodos de la interfaz List.

Las dos clases proporcionan los mismos métodos y funcionalidades, la diferencia radica en la implementación interna y solo afecta levemente al rendimiento. La primera es más rápida en las operaciones que impliquen recorrer la lista para la lectura o modificación de elementos, mientras que la segunda tiene mejor rendimiento en las operaciones de inserción y eliminación de nodos. Generalmente usaremos la primera en nuestros ejemplos, aunque la segunda valdría exactamente igual.

La Clase ArrayList

ArrayList se conoce como una clase genérica con un tipo genérico E. Puede especificar un tipo concreto para reemplazar E (sea E por ejemplo Cliente, Empleado, Integer...) al crear un ArrayList.

```
ArrayList <E> lista = new ArrayList<>();
```

También podemos definirla de forma más general pudiendo utilizar una variable de tipo List que pueda referenciar objetos de las clases ArrayList y LinkedList:

```
List<E> lista = new ArrayList<>();
```

Ej: `List<Cliente> listaClientes = new ArrayList<>();`

Por ejemplo, la siguiente declaración crea un ArrayList y asigna su referencia la variable cities. Este objeto ArrayList se puede usar para almacenar cadenas.

```
ArrayList<String> cities = new ArrayList<String>();
ArrayList<String> cities = new ArrayList<>();
```

La Clase ArrayList

A diferencia de los arrays, Java proporciona la clase ArrayList que se puede usar para almacenar un número ilimitado de objetos, permitiendo añadir o quitar elementos en cualquier momento.

java.util.ArrayList<E>	
`+ArrayList()`	Crea una lista vacía.
`+add(o: E) : void`	Añade un nuevo elemento **o** al final de esta lista.
`+add(index: int, o: E) : void`	Agrega un nuevo elemento **o** en el índice especificado.
`+clear(): void`	Elimina todos los elementos de esta lista.
`+contains(o: Object): boolean`	Devuelve true si esta lista contiene el elemento **o**.
`+get(index: int) : E`	Devuelve el elemento de la lista en el índice especificado.
`+indexOf(o: Object) : int`	Devuelve el índice del primer elemento coincidente en la lista.
`+isEmpty(): boolean`	Devuelve true si esta lista no contiene elementos.
`+lastIndexOf(o: Object) : int`	Devuelve el índice del último elemento coincidente en la lista.
`+remove(o: Object): boolean`	Elimina el elemento **o** de esta lista.
`+size(): int`	Devuelve la cantidad de elementos en esta lista.
`+remove(index: int) : boolean`	Elimina el elemento en el índice especificado.
`+set(index: int, o: E) : E`	Establece el elemento en el índice especificado.

La Clase ArrayList

Ejemplo de uso de una lista sin tipos genéricos:

```
private ArrayList lista = new ArrayList();
String cadena = “elemento”;
lista.add(cadena);
String cadena1 = (String)lista.get(0);  //Obtiene el 1er elem.
lista.add(new Integer(77));  //Se puede hacer
```

Ejemplo con tipos genéricos:

```
private ArrayList<String> lista = new ArrayList();
String cadena = “elemento”;
lista.add(cadena);
String cadena1 = lista.get(0);  //Obtiene el 1er elem.
lista.add(new Integer(77));  //Genera un error
```

Métodos básicos de la interfaz Collection

Los métodos son de 2 tipos: afectan a elementos individuales (básicos) y otros a grupos de elementos (globales).

Métodos de inserción

boolean add(E elem): insertar un elemento al final de la lista. Se le pasa el objeto a insertar. Si tiene éxito devuelve true y si no false.

Métodos de eliminación

boolean remove(Object ob): elimina un nodo ob de una lista. Devuelve true si la eliminación ha tenido éxito y false en caso contrario. Por otra parte, no se exige a ob que sea del tipo genérico E con el que se ha declarado la lista, ya que el método no va a añadir ningún nodo, y no hay peligro de que se inserte un objeto de una clase no permitida.

void clear(): nos permite eliminar todos los nodos de una lista y dejarla vacía. La lista queda vacía. Después podremos volver a insertar nuevos elementos.

Métodos básicos de la interfaz Collection

Métodos de comprobación

int size(): nos permite saber en cada momento el número de elementos (o nodos) insertados en una lista.

boolean isEmpty(): permite saber si una lista está vacía. Devuelve true si la lista está vacía y false en caso contrario.

boolean contains(Object ob): nos dice si un elemento ob determinado está en una lista. Devuelve true si ob pertenece a la lista y false en caso contrario.

Otros métodos

String toString(): devuelve una cadena que representa la colección.

Iterator<E> iterator(): devuelve un iterador asociado a la colección, siendo E del mismo tipo que el de la colección.

Todas las colecciones se pueden recorrer usando un iterador.

Métodos básicos de la interfaz Collection

Un iterador es un objeto que nos permite iterar sobre los elementos y por tanto, recorrerlos secuencialmente. El orden en que se recorren será el orden definido por la colección. Si la colección no define ningún orden, el orden de iteración será indeterminado, aunque, normalmente coincidirá con el orden de inserción.

Los métodos más importantes de la interfaz Iterator

Los que nos permiten recorrer una colección utilizándolos conjuntamente:

boolean hasNext(): devuelve boolean indicando si quedan elementos por recorrer

Object next(): devuelve el siguiente elemento y avanza la posición del recorrido

El que nos permite eliminar elementos de una colección:

void remove(): elimina el elemento de la lista. No todos los iteradores permiten la eliminación.

Métodos básicos de la interfaz Collection

Ejemplo iterador para un arrayList de Empleados

```
class Empleado{

    public Empleado(String nombre, int edad, double salario){

        this.nombre=nombre;

        this.edad=edad;

        this.salario=salario;


    }

    public String dameDatos(){

        return "El empleado se llama " + nombre + ". Tiene " + edad + " años." + " Y un salario de " + salario;
    }

    private String nombre;

    private int edad;

    private double salario;

}

Iterator <Empleado> mi_iterador=listaEmpleados.iterator();

while(mi_iterador.hasNext()){

    System.out.println(mi_iterador.next().dameDatos());

}
```

Ejercicios básicos de ArrayList

Ejercicio 1: Crear y Manipular un ArrayList

Crea un programa que utilice un ArrayList para almacenar nombres de personas. Debe permitir agregar nombres, mostrar la lista, modificar un nombre existente, eliminar un nombre y mostrar el número total de nombres almacenados.

Ejercicio 2: Recorrer un ArrayList con un bucle

Escribe un programa que almacene una lista de números enteros en un ArrayList. El programa debe llenar la lista con valores del 10 al 50 (de 10 en 10) y mostrar su contenido.

Ejercicio 3: Uso de ArrayList con una Clase

Crea un programa en Java que maneje una lista de estudiantes(nif, nombre, nota) usando ArrayList. Cada estudiante debe tener un nombre y una edad. El programa debe permitir agregar estudiantes y luego mostrar la lista con sus datos.

Ejercicios básicos de Collection

Ejercicio 4: Uso de Collection con ArrayList

Crea un programa en Java que use la interfaz Collection con un ArrayList para gestionar una colección de nombres. Permite agregar nombres, eliminar uno y mostrar los nombres con un bucle for-each.

Ejercicio 5: Recorrer una Collection con un Iterator

Crea un programa en Java que almacene una lista de tareas en una Collection y las recorra usando un Iterator, permitiendo eliminar tareas que ya han sido completadas.

Métodos globales de la interfaz Collection

boolean containsAll(Collection<?> c): se le pasa como parámetro otra colección, por ejemplo una lista (<?> significa que la colección que se le pasa es de un tipo genérico cualquiera) y devuelve true si todos los elementos de c están en la colección que hace la llamada y false si al menos un elemento no está en ella.

boolean addAll (Collection <? extends E> c): inserta al final de la colección que hace la llamada todos los nodos de la colección c. La expresión <? extends E>, donde aparece de nuevo el comodín, significa un tipo genérico cualquiera con la condición de que herede del tipo E de la colección invocante.

boolean removeAll (Collection <?> e): elimina de la colección invocante todos los nodos que estén contenidos en c. Después de ejecutar el método no habrá elementos comunes a las dos colecciones.

boolean retainAll (Collection <?> e): elimina de la colección invocante todos los nodos salvo aquellos que también estén contenidos en c.

Métodos de la tabla de la interfaz Collections

Métodos que sirven para volcar los datos de una colección en tabla:
Object[] toArray(): devuelve una tabla de tipo Object con los mismos elementos de la colección. En el caso de las listas, como en estas el orden importa, la tabla albergará los mismos elementos, incluyendo las repeticiones, en el mismo orden.
<T>T[] toArray (T[] t): que es igual que el anterior, pero devuelve una tabla de tipo genérico T. El método es invocado por la colección y, como parámetro, le pasamos una tabla del tipo genérico con que se ha definido. No hay que inicializar la tabla ni importa su tamaño. El método la devuelve redimensionada con el tamaño necesario para albergar todos los elementos de la colección. De hecho, es costumbre definirla con tamaño 0.

Métodos específicos de la interfaz List

Además de los métodos de la interfaz Collections, las listas implementan la interfaz List añadiendo una serie de métodos y funcionalidades específicas:
E get(int indice): devuelve el elemento que ocupa el lugar índice en la lista.
E set(int indice, E elem): guarda el elemento elem en la posición índice en la lista, machacando el valor que hubiera previamente en esa posición, que es devuelto por el método.
void add(int indice, E elem): inserta el valor elem en la posición indice, añadiéndose sin machacar el valor previo. Todos los nodos que ocupan una posición mayor o igual que indice se desplazan una posición hacia el final de la lista, dejando el hueco para el nuevo elemento.
boolean addAll (int indice, Collection<? extends E> c): inserta todos los elemen tos de la colección c en la lista que invoca al método, empezando por el lugar indice y desplazando hacia el final todos los nodos de la lista original a partir de indice, incluido este.

Métodos específicos de la interfaz List

La colección c debe ser de un tipo compatible con nuestra lista, es decir, de la clase genérica E, declarada en la lista, o de una subclase de E.

E remove (int indice): elimina el nodo que ocupa el lugar indice y devuelve el elemento eliminado.

int indexOf(Object ob): devuelve el índice de la primera ocurrencia de ob en la lista. Si no está, devuelve -1.

int lastindexOf(Object ob): hace lo mismo que indexOf(), pero empezando la búsqueda por el final.

boolean equals (Object otraLista): las listas, tanto ArrayList como LinkedList, se pueden comparar por medio del método equals(), que devuelve true si ambas tienen exactamente los mismos elementos en el mismo orden.

void sort (Comparator<? super E> c): ordena la lista invocante con el criterio de c, que compara objetos de la clase E o una subclase.

Diferencias entre Interfaces Collection y List

Diferencias clave entre `Collection` y `List`

Característica	`Collection<E>` (Interfaz General)	`List<E>` (Subinterfaz de Collection)
Orden de los elementos	No garantiza un orden específico	Mantiene el orden de inserción
Acceso por índice	No permite acceso por índice	Permite acceder con `get(int index)`
Métodos específicos	Métodos básicos (`add()`, `remove()`, `contains()`)	Métodos adicionales (`get()`, `set()`, `indexOf()`, `subList()`)
Implementaciones comunes	`ArrayList`, `HashSet`, `TreeSet`, etc.	`ArrayList`, `LinkedList`, `Vector`

Ejercicios básicos de List

Ejercicio 6: Uso de la interfaz List

Escribe un programa en Java que utilice la interfaz List para gestionar una lista de números enteros. Usa ArrayList como implementación y muestra cómo acceder a elementos por su índice, modificar elementos y recorrer la lista con diferentes métodos.

Ejercicio 7: Uso de List con ArrayList

Escribe un programa en Java que utilice la interfaz List con un ArrayList para gestionar una lista de números enteros. El programa debe permitir agregar números, modificar uno, eliminar otro y mostrar la lista con get() y un bucle for.

Ejercicio 8: Uso de List con ArrayList y subList()

Crea un programa en Java que almacene una lista de palabras en un List (ArrayList) y permita extraer una sublista con solo algunas palabras.

Interfaz Set

La interfaz *Set* trata los datos como un conjunto matemático, sin un orden preestablecido y eliminando las repeticiones, aunque tiene una implementación que permite introducir un orden. Todos sus métodos los hereda de Collection. Lo único que añade es la restricción de no permitir duplicados. Esto significa que si intentamos insertar un nodo que ya existe, no lo hará.

El conjunto de métodos disponibles es el mismo que vimos en los apartados de métodos básicos y globales de las colecciones:

int size()
boolean isEmpty()
boolean contains(Object element)
boolean add(E element)
boolean remove(Object element)
Iterator<E> iterator()

Interfaz Set

boolean containsAll(Collection<?> e) boolean addAll(Collection<? extends E> e)
boolean removeAll(Collection<?> e) boolean retainAll(Collection<?> e)
void clear()
Object[] toArray()
<T> T[] toArray(T[])
Asimismo, podemos recorrer un conjunto con una estructura for-each, igual que las listas.
Las diferencias más importantes son el orden en que se van insertando los nodos nuevos y que un nodo que ya está en el conjunto no se puede volver a insertar, ya que no son posibles los nodos repetidos. Cuando intentemos insertar un nodo repetido con el método add () la función devuelve false.
Sin embargo no tendremos los métodos propios de listas, que vienen declarados en la interfaz List. En particular, no es posible el acceso posicional por medio de índices.

Interfaz Set

HashSet: tiene un buen rendimiento, aunque no garantiza ningún orden en la in serción.
TreeSet: a pesar de tener peor rendimiento, garantiza el orden basado en el valor de los elementos insertados. El criterio de ordenación se lo proporciona un com parador en el constructor, o bien es el natural (el proporcionado por el método compareTo()), si no se especifica nada en el constructor.
LinkedHashSet: garantiza el orden basado en la inserción, ya que siempre inserta los nodos al final.

Al contrario de lo que pasa con las listas, las implementaciones de Set tienen diferencias en su comportamiento. Es muy común utilizar variables de tipo Set para referenciarlos. Esto permite aprovechar el polimorfismo de las distintas implementaciones y, como veremos, hacer transformaciones de unas en otras.

Ejercicios básicos Collection con Set

Ejercicio 9: Uso de Collection con un HashSet

Crea un programa que utilice la interfaz Collection con un HashSet para almacenar una lista de productos sin permitir duplicados. Realiza las operaciones agregar productos, eliminarlos y verificar si un producto existe en la colección.

Ejercicio 10: Uso de la interfaz Set con HashSet

Crea un programa que almacene un conjunto de ciudades en un Set (HashSet). El programa debe permitir agregar ciudades, intentar agregar duplicados, eliminar una ciudad y mostrar todas las ciudades almacenadas.

Ejercicio 11: Uso de la interfaz Set con TreeSet

Escribe un programa que almacene una lista de números enteros en un Set (TreeSet). El programa debe mostrar los números ordenados de menor a mayor y permitir eliminar un número.

Conversiones entre colecciones

Una característica interesante de los conjuntos y de todas las colecciones en general es la posibilidad de transformar unas en otras por medio de los constructores.

Por ejemplo, para obtener un conjunto ordenado (un TreeSet) a partir de otro que no lo está (un HashSet o LinkedHashSet) o, dicho de otra forma, si queremos ordenar un conjunto.

Se pueden crear listas pasando conjuntos a su constructor y viceversa; también se pueden añadir a una lista todos los elementos de un conjunto.

Un ejemplo útil es la creación de un conjunto a partir de una lista para eliminar elementos repetidos.

Cuando se trabaja con colecciones de distinto tipo, siempre que se usen constructores o métodos comunes a listas y conjuntos, es costumbre definir las variables de tipo *Collection* para permitir la referencia a diferentes tipos de colección con una misma variable en caso de conversiones.

Métodos específicos de la Clase Collection

Además de los métodos aportados por la interfaces Collection, List y Set con sus implementaciones, la clase Collections (DISTINTA a la interfaz Collection) reúne una serie de utilidades en forma de métodos estáticos de búsqueda, ordenación y manipulación de datos, que trabajan con tipos genéricos.

Métodos de ordenación

La clase Collections posee métodos sobrecargados para ordenar.

static void sort (List<T> lista): ordena la lista que se le pasa como argumento. Esta será de un tipo genérico T, es decir, podemos pasar una lista de cualquier tipo. El criterio de ordenación será el criterio «natural», que es el que establecerá el método compareTo() de la interfaz Compare para la clase T (el tipo de los nodos de la lista, sea el que sea).

Ej Ordenar una lista por su orden natural: Collections.sort(lista);

Métodos específicos de la Clase Collection

Métodos de búsqueda

binarySearch(): hace una búsqueda binaria de un objeto, clave de búsqueda, en una lista ordenada, que se le pasan como argumento y devuelve el índice en la lista del elemento buscado si lo encuentra o un entero negativo si no.
Es muy eficiente, pero requiere que la lista esté ordenada. Merece la pena ordenar la lista si se van a hacer muchas búsquedas con el mismo criterio si no es más eficiente hacer una búsqueda secuencial.

Métodos para manipulación de datos

static void swap(List<?> lista, int i, int j): intercambia los dos nodos de índices i y j entre sí.

static boolean replaceAll(List<T> lista, T valorAntiguo, T valorNuevo): reemplaza todas las ocurrencias de un nodo determinado por otro.

Métodos específicos de la Clase Collection

static void fill(List<? super T> lista, T valorDeRelleno): sustituye todos los valores de los nodos de una lista por el de valorDeRelleno.
static void copy(List<? super T> destino, List<? extends T> origen): copia los elementos de la lista origen en la lista destino, sobreescribiendo los valores previos. La lista destino deberá ser, como mínimo del tamaño de la lista origen. Los nodos de la lista origen a insertar deben ser de clase compatible con la lista destino.

Otras utilidades
static void shuffle(List<?> lista): mezcla, desordena, los elementos de lista.
static int frequency(Collection<?> col, Object ob): devuelve el número de veces que aparece un nodo en una colección. Tiene sentido con listas, ya que en los conjuntos no hay repeticiones.

Métodos específicos de la Clase Collection

static T max(Collection<? extends T> col): devuelve el valor máximo de una colección (no tiene que ser una lista), no tiene que estar ordenada, basándose en el orden natural (la clase debe tener implementada la interfaz Comparable).
static T max(Collection<? extends T> col, Comparator<? super T> comp): devuelve el valor máximo de una colección (no tiene que ser una lista), basándose en un criterio de ordenación distinto al natural (comp).
Y de igual forma serían para el mínimo con el método ***min***.
void reverse (List<?> lista): invierte lista, colocando los nodos en orden inverso. La función no devuelve una nueva lista invertida; invierte la lista original.
Set<T> singleton(T elem): devuelve un conjunto con el tipo genérico T del nodo. Es un conjunto inmutable, es decir, no podemos añadir más nodos ni eliminar el que ya está. Se suele emplear para eliminar un nodo repetido de una lista sin necesidad de usar un bucle.

Interfaz Map

Los mapas o diccionarios son estructuras dinámicas cuyos nodos, llamados entradas, son pares Clave/Valor en vez de valores individuales como en las colecciones.

La interfaz Map, *no hereda de Collection*. Por tanto, los mapas no son colecciones, aunque están íntimamente relacionados con ellas y funcionan dentro del mismo entorno de trabajo.

En un mapa se insertan nodos, llamados entradas, que constan de una clave, que no se puede repetir, y un valor asociado con ella, que sí puede estar repetido.

Las operaciones fundamentales en un mapa son la inserción, la lectura y la eliminación de entradas, aunque veremos algunas más.

Implementaciones de la Interfaz Map

Usaremos tres implementaciones de Map: HashMap, TreeMap y LinkedHashMap, que se diferencian entre sí de forma similar a HashSet, TreeSet y LinkedHashSet.

La implementación ***HashMap***, es muy eficiente en cuanto a la velocidad de acceso a los datos, aunque no garantiza ningún orden de inserción en las entradas.
El constructor más sencillo es de la forma,

```
Map<K, V> m = new HashMap<>()
```

donde K es el tipo de las claves y V el de los valores. Son tipos genéricos que, necesariamente, serán clases y no tipos primitivos. El comportamiento de m estará determinado por la clase de objeto referenciado.

La implementación ***TreeMap***, a semejanza de TreeSet, tiene una estructura de árbol que permite una inserción ordenada y una búsqueda rápida y eficiente de los nodos. Las entradas se insertan por orden natural creciente de las claves.

Implementaciones de la Interfaz Map

Podemos hacer que el orden de un TreeMap sea distinto. Para ello le pasamos un comparador al constructor como parámetro de entrada, igual que hacíamos con TreeSet. En cualquier caso, el orden siempre se refiere a las claves, nunca a los valores.

La implementación ***LinkedHashMap*** mantiene el orden en que se van insertando los nodos, de forma similar a lo que ocurre con LinkedHashSet. Es muy eficiente en las operaciones de inserción y eliminación de entradas y algo más lento en las búsquedas.

Métodos de la Interfaz Map

Algunos de los métodos más importantes de la clase HashMap son:

V get(K clave): Obtiene el valor correspondiente a una clave o devuelve null si no existe esa clave en el diccionario.

V put(K clave, V valor): Añade un par (clave, valor) al diccionario, si no había un valor para esa clave y devuelve null. Y si ya había, se machaca y devuelve el valor antiguo.

V remove(K clave): Elimina la entrada de clave K, si existe, devolviendo el valor asociado con esa clave. En caso contrario, devuelve null.

void clear(): Elimina todas las entradas, dejando el mapa vacío.

boolean containsKey(K clave): Devuelve true si el diccionario contiene alguna entrada con la clave indicada y false en caso contrario.

boolean containsValue(V valor): Devuelve true si el diccionario contiene alguna entrada con el valor indicado y false en caso contrario.

Métodos de la Interfaz Map

K getKey(): Devuelve la clave de la entrada. Se aplica a una sola entrada del diccionario (no al diccionario completo), es decir a una pareja (clave, valor).
V getValue(): Devuelve el contenido de la entrada. Se aplica a una entrada del diccionario (no al diccionario completo), es decir a una pareja (clave, valor).
V setValue(V nuevoValor): asigna nuevoValor a la entrada y devuelve el valor del antiguo.
Vistas Collection de los mapas
A través de las siguientes vistas con estructura de colección, se puede trabajar simultáneamente con ambas interfaces, Map y Collection:
Set<K> keySet(): Devuelve una vista, un conjunto (set), con todas las claves.
Collection<V> values(): Devuelve una colección con todos los valores (los valores pueden estar duplicados a diferencia de las claves).
Set<Map.Entry<K, V>> entrySet(): Devuelve una colección con todos los pares (clave, valor). Los métodos que podemos usar sobre la vista de entradas son:

Métodos de la Interfaz Map

- *K getKey()*: devuelve la clave de la entrada.
- *V getValue()*: devuelve el valor de la entrada.
- *V setValue(V newValor)*: asigna el nuevo valor y devuelve el valor antiguo.

Al usar las vistas podemos solventar inconvenientes debido a que los mapas no son iterables y por tanto que tampoco el uso de la estructura for-each…
Los cambios que hagamos sobre en las collections se reflejarán en el mapa.

Podemos eliminar elementos del conjunto de claves devuelto por keySet() por medio de los métodos remove() de Iterator, remove() de Set, removeAll() o retainAll(), con los cuales se eliminarán las entradas correspondientes en el mapa.

Sin embargo, no podemos añadir entradas a un mapa por medio de add() o addAll() a través de ninguna de sus vistas de tipo colección.

Ejercicios básicos Interfaz Map

Ejercicio 12: Uso de la interfaz Map con HashMap
Crea un programa que almacene el nombre y la edad de varias personas en un Map (HashMap). Debe permitir agregar personas, consultar la edad de una persona específica, eliminar una persona y mostrar todos los elementos.

Ejercicio 13: Uso de Map con TreeMap (Ordenado por Clave)
Escribe un programa que almacene los nombres y precios de productos en un TreeMap, asegurando que los productos estén ordenados alfabéticamente. El programa debe permitir agregar productos, consultar el precio de un producto y eliminar uno.

Ejercicio 14: Recorrer un Map con entrySet()
Crea un programa que almacene nombres y teléfonos en un Map (HashMap). Luego, recorre el mapa para mostrar cada nombre con su número de teléfono.

Diferencias entre ArrayList, Set y Map

Característica	`ArrayList<E>` (Lista)	`Set<E>` (Conjunto)	`Map<K, V>` (Diccionario)
Estructura	Lista ordenada de elementos.	Conjunto de elementos únicos.	Pares clave-valor.
Permite duplicados?	✅ Sí, permite duplicados.	❌ No, no permite duplicados.	✅ Sí en valores, ❌ No en claves.
Orden de elementos	Mantiene el orden de inserción.	Depende de la implementación: `HashSet` no garantiza orden, `TreeSet` sí.	Depende de la implementación: `HashMap` no garantiza orden, `TreeMap` sí (ordenado por clave).
Acceso por índice?	✅ Sí, con `get(int index)`.	❌ No, solo se puede recorrer.	❌ No, se accede por clave con `get(K key)`.
Métodos clave	`add()`, `get()`, `set()`, `remove()`, `size()`.	`add()`, `remove()`, `contains()`.	`put()`, `get()`, `remove()`, `containsKey()`.
Implementaciones comunes	`ArrayList` (más usada), `LinkedList`.	`HashSet`, `TreeSet`, `LinkedHashSet`.	`HashMap`, `TreeMap`, `LinkedHashMap`.
Uso recomendado	Cuando se necesita una lista ordenada con acceso rápido por índice.	Cuando se necesitan elementos únicos sin duplicados.	Cuando se necesita una estructura clave-valor para búsqueda rápida.

Cuándo usar ArrayList, Set o Map

¿Cuándo usar cada interfaz?

ArrayList → Cuando necesitas una lista ordenada con acceso rápido por índice.

Set → Cuando necesitas evitar duplicados y no te importa el orden.

Map → Cuando necesitas almacenar pares clave-valor y acceder rápidamente por clave.

Cuándo usar ArrayList, Set o Map