UNIDAD 6: ESTRUCTURAS DE DATOS DINÁMICAS

Profesor: José Ramón Simó Martínez

Contenido

1. Introducción

Empecemos recordando que un dato de tipo simple no está compuesto de otras estructuras que no sean los bits, y que por tanto su representación sobre el ordenador es directa, sin embargo, existen unas operaciones propias de cada tipo, que en cierta manera los caracterizan.

Una estructura de datos es, a grandes rasgos, una colección de datos (normalmente de tipo simple) que se caracterizan por su organización y las operaciones que se definen en ellos.

Llamaremos dato de tipo estructurado a una entidad, con un solo identificador, constituida por datos de otro tipo, de acuerdo con las reglas que definen cada una de las estructuras de datos.

Los datos estructurados se pueden clasificar según la variabilidad de su tamaño durante la ejecución del programa en:

- **Estructuras de datos estáticas:** Las estructuras estáticas son aquellas en las que el tamaño ocupado en memoria se define con anterioridad a la ejecución del programa que los usa, de forma que su dimensión no puede modificarse durante la misma (p.e., un vector o una matriz) aunque no necesariamente se tenga que utilizar toda la memoria reservada al inicio (en todos los lenguajes de programación las estructuras estáticas se representan en memoria de forma contigua).
- **Estructuras de datos dinámicas:** Por el contrario, ciertas estructuras de datos pueden crecer o decrecer en tamaño, durante la ejecución, dependiendo de las necesidades de la aplicación, sin que el programador pueda o deba determinarlo previamente: son las llamadas estructuras dinámicas. Las estructuras dinámicas no tienen teóricamente limitaciones en su tamaño, salvo la única restricción de la memoria disponible en el computador.

Las estructuras estáticas las estudiamos en anteriores unidades. En esta unidad introduciremos las estructuras dinámicas más utilizadas en Java:

- ArrayList (interfaz List)
- Stack
- Queue
- HashMap
- HashSet

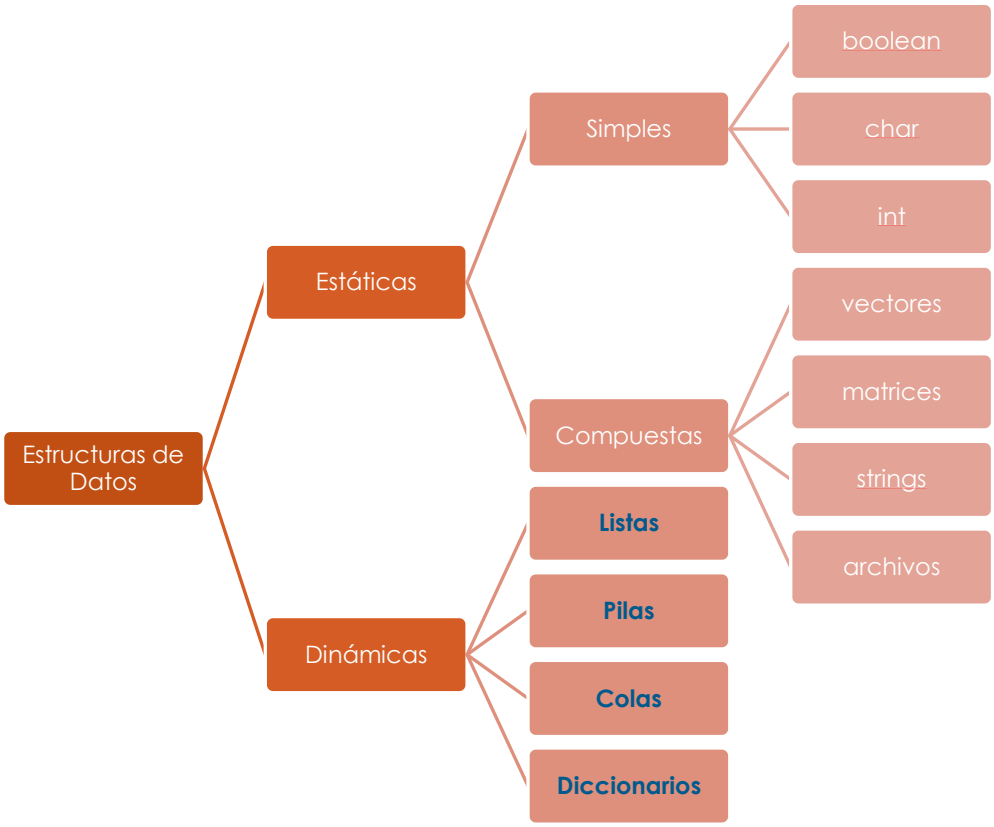
A estas estructuras también las conocemos en Java como **Colecciones**.

Nota

En este tema aparecerán conceptos de programación orientada a objetos que se estudiará en unidades posteriores. Al igual que hemos hecho con otros objetos de Java presentados hasta el momento (String, Arrays, Date, etc), nos centraremos en el uso de estos objetos.

2. Estructuras de datos dinámicas

A continuación, se presenta un esquema resumen de los tipos de datos estáticos y dinámicos disponibles en el lenguaje Java:



Las estructuras de datos dinámicas son conceptos abstractos conocidos como Tipos Abstractos de Datos (TAD). No definen cómo se guardan los datos sino como estos se comportan a la hora, principalmente, de:

- Crear
- Leer o recuperar datos
- Escribir o actualizar datos
- Eliminar datos

Estas acciones también son conocidas como CRUD (Create, Read, Update, Delete).

A continuación, se hará una breve definición de cada uno de los TAD principales.

2.1. Listas

Una lista es un TAD que representa un número de valores ordenados, donde el mismo valor puede aparecer más de una vez. Las operaciones que se pueden hacer sobre una lista son muy variadas, pero principalmente son las siguientes:

- **Crear:** crear una lista vacía.
- **Insertar:** añadir elementos a la lista.
- **Eliminar:** eliminar elementos de la lista.
- **Consultar:** consultar un elemento de la lista.
- **Vacía:** consultar si la lista está vacía.

La inserción/eliminación/consulta de una lista se podrá hacer sobre cualquier posición del elemento en la lista.

Como analogía, tenemos los siguientes ejemplos de pilas en el mundo real:

- Lista de la compra: creamos una lista de la compra con diferentes productos que vamos consultando y/o eliminando de la lista conforme compramos.
- Lista de tareas.
- Etc.

Actualmente las principales clases que utilizan la **interfaz List**[1] para implementar una lista son:

- ArrayList
- LinkedList
- Vector (se utiliza ArrayList en vez de esta)
- Stack

En esta unidad nos centraremos en el uso de la clase ArrayList. Decir que la clase LinkedList también se usa para la creación de listas, sin embargo en este tema sólo la veremos para su uso en la creación Colas.

> **Nota**
>
> (1) El concepto de interfaz se estudiará en unidades posteriores. Por ahora, entenderemos interfaz algo que define el comportamiento de una clase, pero no podemos usar directamente. Por ejemplo, la clase ArrayList define el comportamiento de la interfaz List.
>
> Recordad que en unidades anteriores hemos utilizado clases de Java como String, Random, etc.

2.1.1. Uso del TAD Lista: la clase ArrayList

La clase ArrayList es un tipo de Lista e implementa la interfaz List. Destacar que ArrayList será una de las clases más utilizadas para la creación de listas en Java. A continuación, un ejemplo de uso de la clase ArrayList en Java:

```java
import java.util.ArrayList;

public class EjemploArrayList {

    public static void main(String[] args) {

        // Creo una lista de cadenas de texto...
        ArrayList<String> listaCompra = new ArrayList<String>();

        // Muestro el contenido de la lista (de golpe)
        System.out.println("Mostrar lista: " + listaCompra);

        // Añade un elemento al final de la lista
        listaCompra.add("Fruta");
        System.out.println("Mostrar lista: " + listaCompra);

        // Añade un elemento al final de la lista
        listaCompra.add("Pan");
        System.out.println("Mostrar lista: " + listaCompra);

        // Añade un elemento en la primera posición de la lista
        listaCompra.add(0, "Arroz");
        System.out.println("Mostrar lista: " + listaCompra);

        // Consulto el primer elemento de la lista...
        System.out.println("Consultar elemento de la lista: " + listaCompra.get(0));
        // Consulto el segundo elemento de la lista...
        System.out.println("Consultar elemento de la lista: " + listaCompra.get(1));

        // Muestra el número de elemento que tiene la lista...
        System.out.println("Tamaño de la lista: " + listaCompra.size());

        while (!listaCompra.isEmpty()) {
            // Elimina el primer elemento de la lista...
            System.out.println("Elimino elemento de la lista: " + listaCompra.remove(0));
            System.out.println("Mostrar lista: " + listaCompra);
        }

        System.out.println("Tamaño de la lista: " + listaCompra.size());
    }

}
```

Para crear una lista debemos decir qué tipo de datos[1] va a contener dicha lista. En el ejemplo anterior se puede ver que el ArrayList sólo va a contener datos de tipo String. La sintaxis es la siguiente:

TipoDatoDinamico<Objeto> nombreVariable = new TipoDatoDinamico<Objeto>();

Esta sintaxis se puede aplicar al resto de tipos de datos dinámicos que estudiaremos en esta unidad.

Para recorrer una lista también podemos hacer uso de la variante for (conocida como foreach) que estudiamos en la unidad anterior. Por ejemplo, podríamos recorrer el ArrayList del ejemplo así:

```
for (String producto : listaCompra)
    System.out.println(producto);
```

Por otra parte, observa que necesitamos importar la clase ArrayList del paquete **java.util**.

La clase *ArrayList* utiliza varios métodos para su uso. Los principales son:

- **add(elemento):** añade un elemento al final de la lista.
- **add(X, elemento):** añade un elemento en la posición X.
- **get(X):** consulta un elemento de la posición X.
- **remove(X):** elimina un elemento de la posición X.
- **size():** devuelve el número de elemento que tiene la lista.
- **isEmpty():** devuelve verdadero si la lista está vacía, en caso contrario falso.
- **clear():** elimina todos los elementos pero no borra la lista.
- **indexOf(elemento):** devuelve la posición de la primera ocurrencia del elemento que se indica entre paréntesis.
- **contains(elemento):** devuelve *true* si el elemento se encuentra en la lista y *false* en caso contrario.
- **addAll(nuevaLista):** añade todos los elementos de nuevaLista al final de la lista.

> **Nota**
>
> Hay que tener en cuenta que la clase ArrayList sólo admite tipos de datos objeto como elementos de la lista. Dicho de otra manera, no podemos crear listas directamente con tipos de datos primitivos como int, float, double, etc. Para ello, deberíamos usar las clases envoltorio estudiadas en unidades anteriores (Integer, Double, Float, etc).

Más información de la interfaz List y sus clases que la implementan:

https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/List.html

https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/ArrayList.html

https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/LinkedList.html

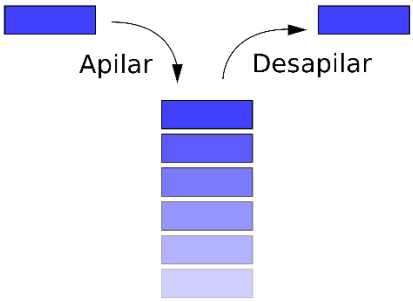
A continuación, estudiaremos dos casos especiales de listas: Pilas y Colas. Estos TAD definirán el comportamiento que deberá tener la lista.

2.2. Pilas

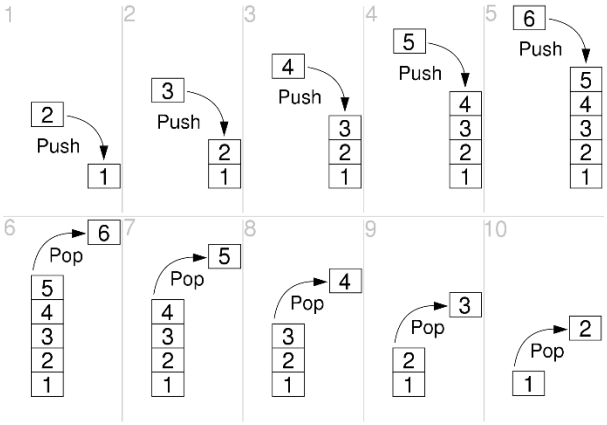
Es una colección de elementos sobre los que se tiene dos principales operaciones:

- **Apilar:** añades un elemento en la cima de la colección (o array).
- **Desapilar:** elimina un elemento de la cima de la colección (o array).

El comportamiento de la pila se conoce como **LIFO** (Last In, First Out), es decir, el último elemento que se añade será el primero en eliminarse/consultarse. Esta forma de actuar gráficamente se vería así:



Otro ejemplo gráfico de su funcionamiento se vería así (push es apilar, pop es desapilar):



Puedes pensar que la colección de elementos del anterior ejemplo es un array.

Como analogía, tenemos los siguientes ejemplos de pilas en el mundo real:

- **Pila de platos:** en una pila de platos, vamos desapilando para lavar cada plato y por otro lado vamos apilándolos cuando ya están secos.
- **Deshacer/Rehacer:** en software, se van Apilando las acciones que se realizan en un programa. Por otro lado, cuando hacemos click en Deshacer lo que estamos es desapilando la última acción, o en Rehacer estamos apilando otra vez la última acción.

2.2.1. Uso del TAD Pila en Java: la clase Stack

En Java tenemos la clase **Stack** para hacer implementa el TAD pila. Esta clase también implementa la interfaz List. En el siguiente código podemos ver su uso:

```
1  import java.util.Stack;
2
3  public class EjemploStack {
4
5      public static void main(String[] args) {
6
7          Stack<Integer> pila = new Stack<Integer>();
8
9          pila.push(1);
10         pila.push(2);
11         pila.push(3);
12
13         System.out.println("Cima pila: " + pila.peek());
14
15         while (!pila.isEmpty())
16         {
17             System.out.println("Desapilar: " + pila.pop());
18         }
19     }
20 }
```

Observa que necesitamos importar la clase Stack del paquete **java.util**.

La clase *Stack* utiliza varios métodos para su uso. Los principales son:

- **push(elemento):** para añadir un elemento a la pila.
- **peek():** para mostrar el elemento que está en la cima de la pila (no lo elimina).
- **pop():** elimina el elemento de la cima de la pila.
- **size():** devuelve el números de elementos que contiene la pila.
- **clear():** vacía los elementos que contiene la pila.
- **isEmpty():** comprueba que la pila está vacía.

Se puede implementar el concepto de Pila con estructuras estáticas en Java, como los arrays. Sin embargo, está fuera de este curso presentar el estudio de su implementación (se deja como ampliación).

Más información de la clase Stack:

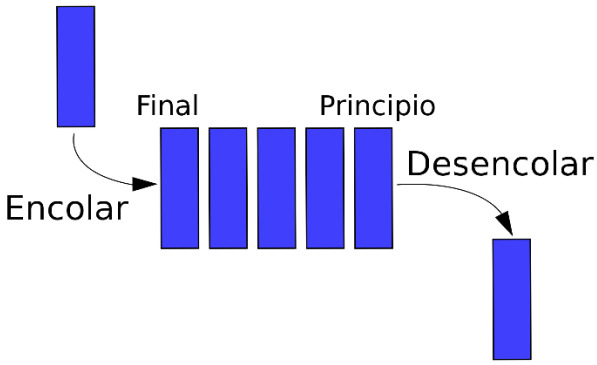
https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Stack.html

2.3. Colas

Es una colección de elementos sobre los que se tiene dos principales operaciones:

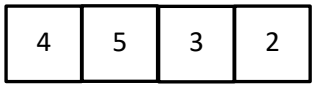
- **Encolar:** añades un elemento a la cola de la colección (o array).
- **Desencolar:** elimina un elemento de la cola de la colección (o array).

El comportamiento de la pila se conoce como **FIFO** (First In, First Out), es decir, el primer elemento que se añade será el primero en eliminarse/consultarse. Esta forma de actuar gráficamente se vería así:

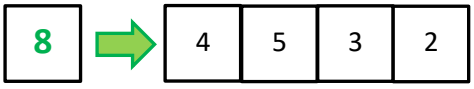


Otro ejemplo gráfico sería el siguiente:

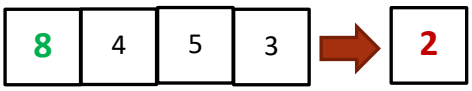
1. Tenemos originalmente este contenido dentro de una Cola:



2. Añadimos el elemento 8 a la cola y este se añade en la cabeza de la cola:



3. Si ahora eliminamos un elemento de la cola:



Como analogía, tenemos los siguientes ejemplos de pilas en el mundo real:

- Cola del cine, discoteca, teatro, etc.
- Cola de procesos en una CPU: el primer proceso en llegar será el primero en ejecutarse.

2.3.1. Uso del TAD Cola: la clase LinkedList

En Java tenemos la interfaz (que NO clase) **Queue** para hacer uso del TAD Cola. Esto significa que no podemos instanciar (hacer un new) de una clase Queue, pero si que podremos usar en este caso la clase **LinkedList** (tipo de lista) que implementa a la interfaz Queue.

En este punto del curso insisto en que no importa tanto que no sepas la diferencia entre la clase y la interfaz, sino en la creación y uso de TADs en Java. Fíjate en el siguiente código de ejemplo para saber cómo se hace:

```
1  import java.util.Queue;
2  import java.util.LinkedList;
3
4  public class EjemploQueueLinkedList {
5
6      public static void main(String[] args) {
7
8          Queue<Integer> cola = new LinkedList<Integer>();
9
10         System.out.println("Mostrar cola: " + cola);
11         cola.add(1);
12         System.out.println("Mostrar cola: " + cola);
13         cola.add(2);
14         System.out.println("Mostrar cola: " + cola);
15         cola.add(3);
16         System.out.println("Mostrar cola: " + cola);
17
18         System.out.println("Consultar elemento de la cola: " + cola.peek());
19
20         while (!cola.isEmpty())
21         {
22             System.out.println("Elimino elemento de la cola: " + cola.remove());
23             System.out.println("Mostrar cola: " + cola);
24
25         }
26     }
27 }
```

Observa que en la creación de la Cola participan tanto la interfaz Queue, que define lo que podemos hacer con la cola, y la clase LinkedList para instanciar (crear) la cola.

Por otra parte, necesitamos importar tanto Queue como LinkedList del paquete **java.util**.

La interfaz *Queue* utiliza varios métodos para su uso. Los principales son:

- **add(elemento):** para añadir un elemento a la cola de la cola.
- **peek():** mostrar primer elemento que llegó a la cola (la cabeza de la cola).
- **remove():** elimina el primer elemento que llegó a la cola (la cabeza de la cola).
- **clear():** elimina todos los elementos de la cola.
- **isEmpty():** comprueba que la cola está vacía.

También observamos que hacemos uso de la clase **LinkedList**. Esto es necesario para poder implementar la interfaz **Queue**. Otra opción es usar, en vez de LinkedList, la clase **PriorityQueue**. Se deja como ejercicio de ampliación al estudiante. Más información de la interfaz Queue:

https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Queue.html

https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/LinkedList.html

https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/concurrent/PriorityBlockingQueue.html

2.4. Conjuntos

Matemáticamente un conjunto es una colección no ordenada de elementos no repetidos. Por una parte, es no ordenada porque no podemos acceder a los elementos a través de un índice. Y por otra, no repetidos porque cada elemento de del conjunto es único.

Por tanto, en este punto cabe recalcar la diferencia entre:

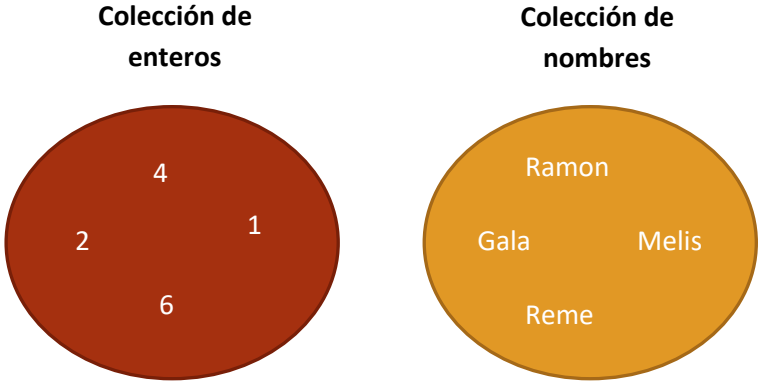
- **Lista:** colección de elementos ordenados y pueden estar repetidos. Se accede a los elementos a través de un índice.
- **Conjunto:** colección de elementos no ordenados y no duplicados. No se puede acceder a través de un índice.

Principales operaciones:

- **Añadir:** añades un elemento al conjunto SI este no existe ya en dicho conjunto.
- **Consultar:** consulta si un elemento concreto está en el conjunto.
- **Eliminar:** elimina un elemento del conjunto SI este existe en dicho conjunto.

A diferencia de las Listas, Pilas y Colas, los conjuntos no tienen comportamiento LIFO o FIFO. ¡Es decir, no nos importa cómo ni dónde se añaden los elementos en el conjunto… porque es un conjunto!

Por tanto, gráficamente podemos ver un Conjunto como un "saco" de elementos: los conjuntos matemáticos de toda la vida:



Como analogía, tenemos los siguientes ejemplos de pilas en el mundo real:

- Conjunto de números enteros, reales, etc.
- Conjunto de todos los DNI de España.
- Conjunto de todas las matrículas de coche.
- Etc.

2.4.1. Uso del TAD Conjunto: la clase HashSet

En Java tenemos la interfaz (que NO clase) **Set** para hacer uso del TAD Conjunto. En este caso, Java implementa la interfaz Set en las siguientes clases:

- **HashSet**
- TreeSet
- LinkedHashSet

En esta unidad estudiaremos el uso de la clase HashSet para el uso de conjuntos en Java. A continuación, un ejemplo de código para crear conjuntos de elementos con la clase HashSet:

```
1  import java.util.HashSet;
2
3  public class EjemploHashSet {
4
5      public static void main(String[] args) {
6
7          // Creo el conjunto...
8          HashSet<String> nombres = new HashSet<String>();
9
10         // Añado elementos al conjunto...
11         nombres.add("Leia");
12         nombres.add("Anakin");
13         nombres.add("Darth Vader");
14         nombres.add("Obijuan");
15         System.out.println(nombres);
16
17         // Añado y muestro si se ha podido añadir
18         System.out.println(nombres.add("Leia"));
19
20         // Pregunto si existe
21         System.out.println(nombres.contains("Anakin"));
22
23         // Elimina si existe
24         System.out.println(nombres.remove("Darth Vader"));
25
26         System.out.println(nombres.remove("Jabba"));
27     }
28 }
```

Salida por pantalla:

```
[Leia, Anakin, Darth Vader, Obijuan]
false
true
true
false
```

Observa que necesitamos importar las clase *HashSet* del paquete **java.util**.

La clase *HashSet* utiliza varios métodos para su uso. Los principales son:

- **add(elemento):** añade un elemento al conjunto, si este no existe todavía.
- **contains(elemento):** nos dice si un elemento en concreto existe en el conjunto.
- **remove(elemento):** elimina el elemento del conjunto si este existe en dicho conjunto.
- **clear():** elimina todos los elementos del conjunto.
- **size():** devuelve el número de elementos que contiene el conjunto.
- **isEmpty():** comprueba que no hay elementos en el conjunto.

Más información de la interfaz Set y la clase HashSet:

https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Set.html

https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/HashSet.html

2.5. Diccionarios

Hasta ahora hemos estudiado dos tipos de datos dinámicos: listas y conjuntos. En las listas cada elemento esta indexado y podemos acceder a este indicando un número entero como índice. Los Diccionarios, al igual que las listas, los datos están indexados. Sin embargo, sus índices (claves) pueden ser cualquier tipo de valor. Por tanto, a los Diccionarios se les conoce como dato dinámico de tipo **CLAVE-VALOR**.

Ejemplos de su uso pueden ser:

- Almacenar nombres de capitales (VALOR) y acceder a ellas mediante el nombre de su país (CLAVE).

UK	ES	DE	PT
Londres	Madrid	Berlín	Lisboa

- En un juego de Star Wars, para cada personaje (CLAVE) se puede almacenar su fuerza (VALOR).

Anakin	Luke	Leia	Yoda	C3PO
1000.0	500.0	500.0	5000.0	0.0

Como podrás intuir, no puede haber claves duplicadas en el diccionario (pero sí valores).

2.5.1. Uso del TAD Diccionario: la clase HashMap

En Java los diccionarios se implementan a partir de la interfaz **Map**. Y recuerdo que no importa tanto ahora que sepamos que es la interfaz. Lo importante es que sepamos que existen las siguientes tres clases para usar Diccionarios:

- **HashMap**
- TreeMap
- LinkedHashMap

En esta unidad sólo estudiaremos el uso de la clase HashMap para la creación de diccionarios. A continuación, un ejemplo de código para crear conjuntos de elementos con la clase HashMap:

```java
import java.util.HashMap;

public class EjemploHashMap {

    public static void main(String[] args) {

        // Crear el diccionario
        HashMap<String,Double> fuerzaPersonajes = new HashMap<String,Double>();

        // Añadir elementos al diccionario en forma de clave-valor
        fuerzaPersonajes.put("Anakin", 1000.0);
        fuerzaPersonajes.put("Luke", 500.0);
        fuerzaPersonajes.put("Leia", 500.0);
        fuerzaPersonajes.put("Yoda", 5000.0);
        fuerzaPersonajes.put("C3PO", 0.0);

        // Mostrar contenido del diccionario
        System.out.println(fuerzaPersonajes);

        // Acceder a dato del diccionario
        Double fuerza = fuerzaPersonajes.get("Anakin");
        System.out.println("Fuerza Anakin: " + fuerza);

        // Eliminar dato del diccionario
        fuerzaPersonajes.remove("Anakin");

        System.out.println(fuerzaPersonajes);

        // Actualizar dato del diccionario
        fuerzaPersonajes.replace("Leia", 800.0);

        // Recorrer datos del diccionario
        for (String clave : fuerzaPersonajes.keySet()) {
            Double valor = fuerzaPersonajes.get(clave);
            System.out.println("Personaje: " + clave + ", Fuerza: " + valor);
        }
    }
}
```

Salida por pantalla:

```
{C3PO=0.0, Leia=500.0, Luke=500.0, Yoda=5000.0, Anakin=1000.0}
Fuerza Anakin: 1000.0
{C3PO=0.0, Leia=500.0, Luke=500.0, Yoda=5000.0}
Personaje: C3PO, Fuerza: 0.0
Personaje: Leia, Fuerza: 800.0
Personaje: Luke, Fuerza: 500.0
Personaje: Yoda, Fuerza: 5000.0
```

Antes que nada, observa que necesitamos importar las clase *HashMap* del paquete **java.util**.

La clase *HashMap* utiliza varios métodos para su uso. Los principales son:

- **put:** añade un elemento al diccionario. Necesita la clave y el valor.
- **get:** devuelve un elemento del diccionario a partir de su clave.
- **remove:** elimina un elemento del diccionario a partir de su clave.
- **replace:** actualiza un elemento del diccionario a partir de su clave y el nuevo valor
- **keySet:** devuelve un Set, que será el conjunto de claves de nuestro HashMap.
- **values():** devuelve una colección con todos los valores (los valores pueden estar duplicados a diferencia de las claves).
- **entrySet():** devuelve un Set con todos los pares CLAVE-VALOR.
- **containsKey(clave):** devuelve true si el diccionario contiene la clave indicada y false en caso contrario.
- **size():** devuelve el número de elementos que hay en el HashMap.
- **clear():** elimina todos los elementos del HashMap.
- **isEmpty():** comprueba si el HashMap ya no contiene elementos CLAVE-VALOR.

Ahora estudiaremos dos peculiaridades del HashMap: creación y recorrido.

Creación de HashMap

Para empezar, en la creación de Diccionarios hay una diferencia a destacar respeto a las Listas o Conjuntos. Los diccionarios necesitan dos tipos de datos para poder crearse: la clave y el valor. Además, estos deben ser de tipo objeto (no datos primitivos int, float,etc). La sintaxis para crear HashMap será:

```
HashMap<Objeto1, Objeto2> nombreVariable = new HashMap<Objeto1, Objeto2>();
```

En el ejemplo anterior se puede ver que el HashMap tendrá como clave un String y como valor un Double.

Recorrer el HashMap

Por otra parte, hay que destacar la manera de recorrer los datos del HashMap. Para ello, como se puede observar, utilizaremos una variante del bucle for (conocida en otros lenguajes como *foreach*) que permite recorrer de forma óptima el HashMap.

```
for (String clave : fuerzaPersonajes.keySet()) {
    Double valor = fuerzaPersonajes.get(clave);
    System.out.println("Personaje: " + clave + ", Fuerza: " + valor);
}
```

Para ello usamos el método *keySet()* que devuelve el conjunto de CLAVES de nuestro HashMap en forma de tipo de dato Set. Luego ya podemos acceder al VALOR de cada uno de nuestros datos en el HashMap a partir de su CLAVE.

Más información de la interfaz Map y sus clases:

https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Map.html

https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/HashMap.html

https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/TreeMap.html

https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/LinkedHashMap.html

3. La clase Collections

Una vez estudiados los principales TAD y las estructuras dinámicas (Colecciones) que ofrece Java para su uso, es conveniente presentar una clase que resultará bastante útil para manipular dichas estructuras. Esta clase se conoce como **Collections**.

La clase Collections está formada por un conjunto de métodos (recuerda, funciones en Java) que son estáticos (al igual por ejemplo que la clase Math). Por tanto, al ser estáticos no hace falta instanciar (crear con new) la clase Collections.

Los métodos más destacados de la clase Collections son:

- **sort():** recibe como parámetro un objeto de tipo List (ArrayList por ejemplo) y ordena sus elementos.
- **reverse():** recibe como parámetro un objeto de tipo List (ArrayList por ejemplo) e invierte el orden de sus elementos.
- **shuffle():** recibe como parámetro un objeto de tipo List (ArrayList por ejemplo) y mezcla aleatoriamente sus elementos, es decir, como en una baraja de cartas.
- **max():** recibe como parámetro cualquier objeto de tipo Collection (cualquiera de los que hemos estudiado) y devuelve el máximo del orden natural de los valores que contiene dicha colección.
- **min():** recibe como parámetro cualquier objeto de tipo Collection (cualquiera de los que hemos estudiado) y devuelve el mínimo del orden natural de los valores que contiene dicha colección.

> Nota
>
> No confundir la clase Collection**s** con Collection.

3.1. Uso de la clase Collections

A continuación, un ejemplo de uso de la clase Collections:

```
1  import java.util.ArrayList;
2  import java.util.Collections;
3
4  public class EjemploCollections {
5      public static void main(String[] args) {
6          ArrayList<Integer> lista = new ArrayList<Integer>();
7
8          lista.add(4);
9          lista.add(1);
10         lista.add(3);
11         lista.add(5);
12         lista.add(2);
13
14         System.out.println(lista);
15
16         // Ordeno la lista creada...
17         Collections.sort(lista);
18         System.out.println(lista);
19
20         // Obtengo el máximo...
21         Integer maximo = Collections.max(lista);
22         System.out.println(maximo);
23     }
24 }
```

Observa que necesitamos importar las clase *Collections* del paquete **java.util**.

Por otra parte, fíjate que en el caso del método *sort* no hace falta guardar la lista ordenada en otra lista, es decir, la lista que recibe como parámetro la modifica directamente. Sin embargo, el método *max* sí que devuelve un objeto: en este caso como tenemos tipos de datos Integer dentro de la lista, debemos guardar el valor devuelto en una variable de tipo Integer.

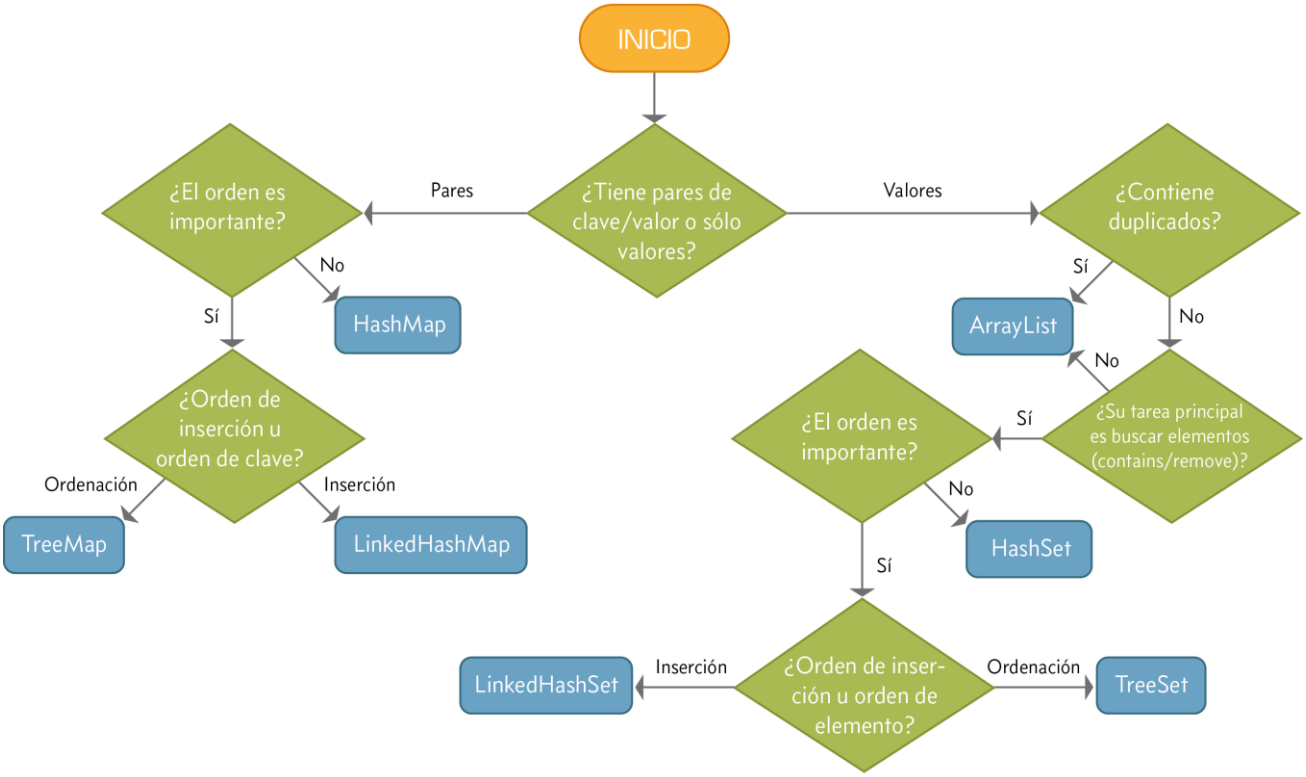
De todas formas, habrá que leer la documentación oficial para consultar los parámetros que recibe y lo que devuelve cada método.

Más información de la clase Collections y sus métodos:

https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Collections.html

4. Diagrama de decisión para el uso de Colecciones de Java

Diagrama de decisión para uso de Colecciones en Java:



5. Bibliografía

Documentación oficial: https://docs.oracle.com/en/java/javase/17/docs/api/index.html

Web w3schools.com

Librerías de clases útiles: Apuntes de José Chamorro del CFGS DAW del IES Sant Vicent Ferrer (Algemesí).