DESARROLLO DE APLICACIONES WEB

Unidad 7

Estructuras de datos dinámicas

1r DAW

IES La Mola de Novelda

Departament d'informàtica

Profesor: Joan Carbonell Picó

ÍNDEX

Unidad 7: Estructuras de Datos Dinámicas

1.- INTRODUCCIÓN

A menudo necesitamos guardar información, pero no sabemos de antemano el espacio que va a ocupar en la memoria. En estos casos, los *arrays* no es la solución más adecuada, ya que su tamaño debe de permanecer fijo una vez declarados. En su lugar, necesitaremos estructuras dinámicas de datos, es decir, objetos que alberguen datos que se crean y se destruyen en tiempo de ejecución, según las necesidades de la aplicación. Para trabajar con estas estructuras dinámicas se usan "**colecciones**".

2.- COLECCIONES

Llamaremos "colecciones" a cualquier conjunto de objetos. Por ejemplo, un *String[ ]* es una colección de cadenas, un *Integer[ ]* es una colección de objetos *Integer* y un *Object [ ]* es una colección de objetos de cualquier clase porque, como ya sabemos, todas las clases heredan de la clase base *Object*. Si además, le añadimos la característica de que sean estructuras dinámicas, diremos que una colección es un almacén de objetos dinámicos que puede crecer o menguar durante la ejecución de un programa.

Para este tipo de estructuras dinámicas, Java proporciona la interfaz ***Collection***. En general, cuando hablamos de "**colecciones**" nos estamos refiriendo a un objeto cuya clase implementa la interfaz ***Collection***. De esa forma, todas estas colecciones (estructuras dinámicas) comparten un conjunto de métodos declarados en la interfaz ***Collection***. Todas ellas implementan dicha interfaz, aunque de diferente forma. Además, estas estructuras dinámicas permiten añadir y quitar unidades de información (objetos llamados nodos o elementos) en tiempo de ejecución, cambiando el tamaño total. En general, llamaremos colecciones a todas las clases que implementan la interfaz ***Collection***, aunque también usaremos el nombre particular de la clase implementada (por ejemplo *ArrayList*).

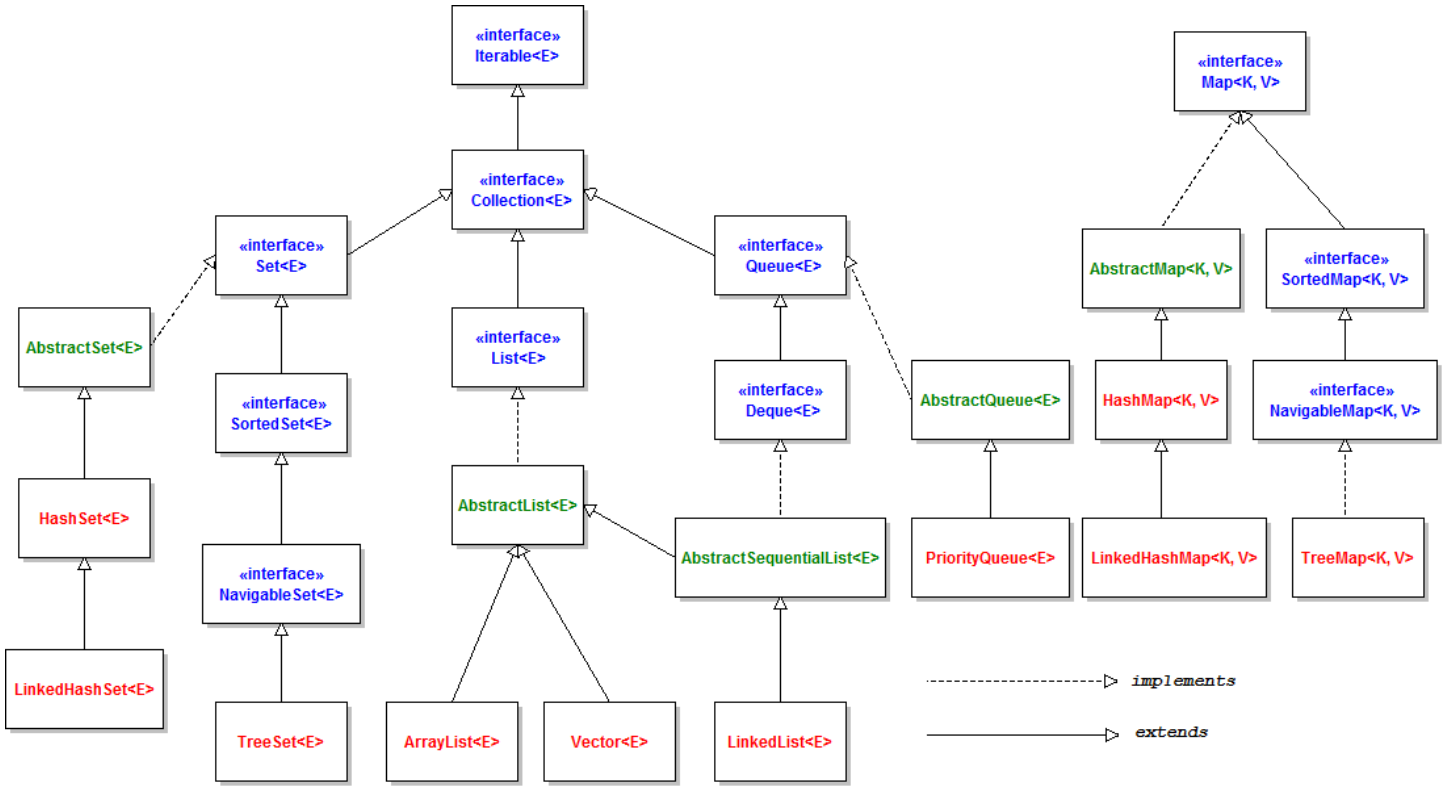
El conjunto de interfaces y clases del marco de trabajo (*framework*) *Collection* se encuentran en el paquete **java.util**.

3.- JAVA COLLECTIONS FRAMEWORK

Java Collections Framework es un conjunto de estructuras y algoritmos de datos reutilizables diseñados para liberar los programadores de la implementación de estructuras de datos, para que puedan centrarse en la lógica de la aplicación o programa. También se encarga de gestionar todos

los tipos de colecciones que el lenguaje Java soporta. En sus primeras versiones tenía una estructura bastante sencilla pero con el paso del tiempo se han ido ampliando los elementos.

Además, *Java Collections Framework* nos proporciona estructuras de datos dinámicas (colecciones) de un gran rendimiento, de alta calidad y fáciles de utilizar; así como algoritmos útiles y sólidos (buscar y ordenar, operar entre colecciones y matrices, etc.) que nos van a ayudar en la implementación de nuestras aplicaciones.



El *Java Collections Framework* es una API estándar que se utiliza ampliamente en la programación de Java. Es la API estándar y perfecta para manipular colecciones. Sus ventajas principales son:

- **Reduce la dedicación y tiempo que se dedica a la programación**. Las estructuras de datos que ofrece el *Java Collections Framework* ayuda a que el programador pueda dedicarse íntegramente al desarrollo del negocio de la aplicación.

- **Aumenta la velocidad y la calidad del programa**. Las estructuras implementadas en el *Java Collections Framework* nos ofrecen un alto rendimiento y una alta cualidad. Los programadores aprovechan esta ventaja para mejorar sus aplicaciones.

- **Fomenta la reutilización del código**. Debido a que el *Java Collections Framework* se encuentra integrado en el *JDK*, todo el código implementado con este *framework* se podrá reutilizar entre las aplicaciones, bibliotecas y API. De esa forma reducimos costos de

desarrollo y aumentamos la interoperabilidad entre los programas de Java.

4.- TIPOS DE COLECCIONES

Vamos a analizar los principales tipos de colecciones que se encuentran en el *Java Collections Framework*, así como sus implementaciones más importantes. Hemos de tener en cuenta que **las colecciones solo trabajan con objetos**, por lo que no podemos hacer uso de los tipos primitivos (*int*, *char*, *double*, etc). Tendremos que hacer uso de las clase envolventes o *wrappers* (*Integer*, *Character*, *Double*, etc).

4.1.-LISTAS

La interfaz ***List*** responde a la necesidad de manejar sucesiones de datos que se pueden repetir y cuyo orden puede ser importante. De alguna forma sustituyen a los *arrays,* con la diferencia de que podemos insertar o eliminar datos en las listas sin que sobre ni falte sitio.

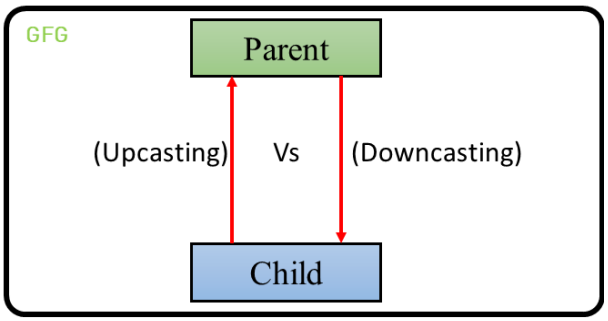
Dentro de la interfaz ***List*** existen los siguientes tipos de implementaciones realizadas dentro de la plataforma Java: *ArrayList*, *Vector* y *LinkedList*. Las diferencias entre ellas radica en sus implementaciones internas (la forma en que se han creado) y solo afecta levemente al rendimiento. La implementación más utilizada es la de ***ArrayList*** ya que es la que mejor rendimiento tiene sobre la mayoría de las situaciones. En los siguientes ejemplos utilizaremos la clase *ArrayList*, aunque las otras valdrían igualmente.

4.1.1.- Sintaxis

La sintaxis general para construir un ***ArrayList*** con un tipo genérico de datos **E** (puede ser Cliente, Empleado, Integer, etc) es:

Utilizando la clase ''hija'' de List	Utilizando polimorfismo (se aplica upcasting)
ArrayList <E> lista = new ArrayList<>();	**List <E> lista = new ArrayList<>();**

En la declaración y creación de la lista estamos indicando que solo se podrán insertar objetos (nodos) del tipo **E**. Si **E** es **Cliente**, solo se podrán insertar objetos **Cliente** o de una subclase de **Cliente** (recordar que un objeto de una subclase de **Cliente** también es un **Cliente**). **E** debe ser siempre el nombre de una clase, nunca un tipo primitivo. La declaración y creación de un objeto de la clase ***ArrayList*** que nos sirva para guardar objetos del tipo **Cliente** sería:



```
ArrayList<Cliente> clie1 = new ArrayList<Cliente>(); // List<Cliente> clie1 = new ArrayList<Cliente>();
// Aunque no hace falta escribir Cliente en el constructor.
// El operador<> (operador diamante) se deja vacío
ArrayList<Cliente> clie1 = new ArrayList<>(); // List<Cliente> clie1 = new ArrayList<>();
```

- Ejemplo: A partir de la siguiente clase sencilla de **Cliente**, vamos a comprobar los distintos métodos con ejemplos.

```
package list;

public class Cliente implements Comparable{
  String dni;
  String nombre;
  int edad;

  @Override
  public boolean equals(Object obj) {
    return this.dni.equals(((Cliente)obj).dni);
  }

  @Override
  public int compareTo(Object o) {
    return this.dni.compareTo(((Cliente)o).dni));
  }

  @Override
  public String toString() {
    return  "Dni = " + dni + "\nNombre = " + nombre + "\nEdad = " + edad + "\n";
  }
}
```

4.1.2.- Métodos básicos de la interfaz *Collection*.

Los métodos de la interfaz *Collection* se pueden consultar a la API de Java. Algunos de ellos son los siguientes:

java.util

Interface Collection<E>

MÉTODOS DE INSERCIÓN Y ELIMINACIÓN	
boolean add(E e)	Inserta un nuevo elemento o nodo al final de la lista. Si la inserción tiene éxito se devuelve un *true*, en otro caso un *false*. Cliente cliente = new Cliente("111","Marta",20); clie1.add(cliente);
boolean remove(Object ob)	Elimina el nodo ob de la lista. Devuelve *true* si se ha eliminado con éxito y *false* si el objeto no estaba en la lista. No se exige que ob sea del tipo genérico E declarado en la lista, ya que al no añadirlo no hay peligro de que se

inserte un objeto de una clase no permitida

clie1.remove(cliente);

void clear() Elimina todos los nodos de una lista y la deja vacía, pero no elimina la propia lista, simplemente queda vacía.

MÉTODOS DE COMPROBACIÓN

int size() Número de elementos o nodos que tiene la lista.

boolean isEmpty() Comprobar si la lista está vacía.

boolean contains(Object ob) Nos dice si el objeto ob está en la lista.

clie1.add(new Cliente("115","Jorge",21)); // devuelve *true*

clie1.add(new Cliente("115","",0)); // devuelve *true*

Si nos fijamos en el atributo que usa el método *equals()*, veremos que es el dni del **Cliente**.

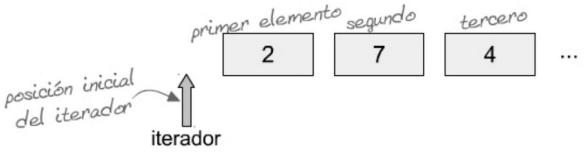
OTROS MÉTODOS

String toString() Devuelve una cadena que representa la colección. Cada nodo se muestra según la implementación de *toString()* que tenga la clase genérica E.

Iterator<E> iterator() Para recorrer una lista nodo a nodo utilizaremos los iteradores. Éstos son objetos que van apuntando sucesivamente a los nodos de la lista empezando por el primero.

El método *iterador()*, invocado por una lista, nos devuelve un iterador asociado a ella:

Iterator it = clie1.iterator(); // Creación del iterador asociado a la lista clie1

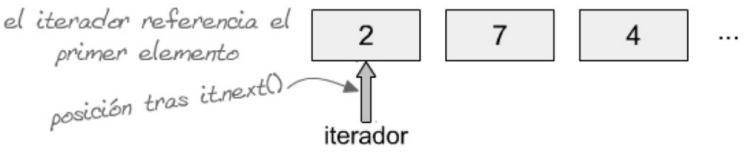


java.util

Interface Iterator<E>

it es el iterador que sirve para recorrer la lista. Las funciones *hasNext()* y *next()* se complementan y se usan con el iterador. Inicialmente it apunta al principio de la lista, justo antes del primer elemento.

- **boolean hasNext()** → Comprueba si quedan elementos por visitar.
- **Object next()** -→Nos devuelve una copia del siguiente elemento, si existe, y avanza una posición en la lista, apuntando al nodo siguiente. Por ejemplo, la posición del iterador después del primer *next()* es:



En el caso de que no haya siguiente porque estamos al final de la lista o porque ésta estuviera vacía, *next()* lanzará la excepción *NoSuchElementException*. Como vemos en la figura, la primera llamada a *next()* nos

devuelve el primer elemento de la colección.

Para evitar la excepción, los métodos *hasNext()* y *next()* se usan conjuntamente.

Ejemplo: Con un bucle *for*

```
Iterator it = clie1.iterator(); // Creación del iterador asociado a la lista clie1
for(;it.hasNext();){
        Cliente p = (Cliente)it.next();
        System.out.println(p.toString());
}
```

También sería válido poner la declaración e inicialización del iterador en la parte de inicialización del bucle *for*

```
for(Iterator it = clie1.iterator();it.hasNext();){
        Cliente p = (Cliente)it.next();
        System.out.println(p.toString());
}
```

La parte del incremento se deja vacia porque el método *next()* , además de devolver el nuevo nodo al que apunta, incrementa el iterador para que apunte al siguiente nodo.

El *cast* se puede evitar si definimos el iterador con un tipo genérico, en este caso Cliente.

```
Iterator<Cliente> it = clie1.iterator(); // Creación del iterador asociado a la lista clie1
for(;it.hasNext();){
        Cliente p = it.next();
        System.out.println(p.toString());
}
```

Los iteradores tienen un tercer método que permite eliminar nodos de una lista:

- **void remove()** → Elimina el último elemento devuelto con la instrucción *next()*. Por ejemplo, podemos eliminar un nodo de una lista a partir de una condición mientras la estamos recorriendo con un iterador, de forma que se preserve la integridad de la colección. No hay que confundirlo con el método *remove()* de la interfaz *Collection* que hemos visto anteriormente, ya que éste se llama desde un objeto lista (u otra colección).

Ejemplo: Queremos eliminar de nuestra lista aquellos clientes con edad mayor a 20:

```
it = clie1.iterator();
while(it.hasNext()){
        Cliente p = it.next();
        if (p.getEdad()>20)
```

```
                it.remove(); // elimina el último cliente devuelto por next()
                //!!!NO USAR clie1.remove(p)!!!!!!!!
        }
```

Una forma más simple de recorrer una colección es por medio de la estructura ***for-each***:

```
        for(Cliente c: clie1){
                System.out.println(c);
        }
```

Hay operaciones que no podemos hacer con un *for-each*. Por ejemplo, no podemos eliminar nodos, ya que c es siempre la referencia a una copia de un elemento de la colección (en este caso de una lista). Para eso tendremos que usar un iterador.

4.1.3.- Métodos globales de la interfaz *Collection*.

boolean containsAll(Collection<?> c)	Se le pasa como parámetro otra colección de tipo genérico (el carácter ? se le conoce como comodín). Devuelve true si todos los elementos de c están en la colección que hace la llamada, y *false* si hay al menos un elemento de c que no está en ella.

```
//Insertamos dos clientes que ya estaban en la lista
clie2.add(new Cliente("111","Marta",20));
clie2.add(new Cliente("112","Carlos",19));
if(clie1.containsAll(clie2))
        System.out.println("clie1 contiene todos los elementos de la lista clie2");
else
        System.out.println("clie1 no contiene todos los elementos de la lista clie2");
clie2.add(new Cliente("211","Ana",19));
if(clie1.containsAll(clie2))
        System.out.println("clie1 contiene todos los elementos de la lista clie2");
        else
        System.out.println("clie1 no contiene todos los elementos de la lista clie2");

//PARA SABER QUE SE COMPARA PARA SABER SI SON IGUALES SE MIRA EL MÉTODO equals()
```

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> significa un tipo genérico cualquiera con la condición de que herede del tipo E de la colección invocante.

```
//Podemos pasarle como parámetro cualquier colección que sea del tipo Cliente
// o de una subclase de Cliente
clie1.addAll(clie2);
System.out.printf(clie1.toString());
// Vemos que se han añadido al final, aunque estén repetidos. Al ser una lista se pueden repetir.
```

boolean removeAll(Collection<?> c)	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.

```
//El método removeAll(Collection<?> c) elimina de la colección que hace
// la llamada a la función todas las ocurrencias de los nodos de c
clie1.add(new Cliente("555","Joan",53));
System.out.println("============================");
System.out.printf(clie2.toString());
System.out.println("============================");
System.out.printf(clie1.toString());
System.out.println("============================");
clie1.removeAll(clie2);
System.out.printf(clie1.toString());
```

boolean retainAll(Collection<?> c) Elimina todos los nodos de la colección invocante, salvo aquellos que también estén en c.

4.1.4.- Métodos de *arrays* de la interfaz *Collection*.

Sirven para volcar los datos de una colección en un *array*.

Object[ ] toArray( ) Devuelve un *array* de tipo *Object* con los mismos elementos de la colección. En el caso de las listas, el *array* contendrá los mismos elementos, incluyendo las repeticiones, en el mismo orden

```
Object t1[] = clie2.toArray();

System.out.println("Cliente 2 - posicion 0: "+((Cliente)t1[0]).getNombre());
```

El inconveniente que tiene este método es que devuelve un *array* de tipo *Object*, aunque sabemos que son clientes. Esto nos obliga a emplear un *cast* para acceder a los miembros de la clase a la que pertenece.

<T> T[ ] toArray(T[ ] a) Para evitar el problema de tener que hacer un *cast* utilizaremos este método. En este caso devuelve un *array* de tipo genérico T. El método es invocado por la colección y como parámetro le pasamos un *array* del tipo genérico con que se ha definido. No hay que inicializar el *array* 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.

```
Cliente t2[] = clie2.toArray(new Cliente[0]);

    System.out.println("Cliente 2 - posicion 0: "+(t2[0].getNombre()));
```

Ahora t2 es de tipo Cliente y recoge los elementos de clie2. Con este procedimiento no es necesario el *cast* delante de t2[0].

```
 int i=0;
for (Cliente c:t2) { // Imprimimos el array t2 de Cliente
      System.out.println("Cliente 2 - posición "+i+" : "+ c.toString());
      i++;
```

```
}
```

4.1.5.- Métodos específicos de la interfaz List

Todo los métodos vistos anteriormente pertenecen a la interfaz *Collection* y son implementados por todas las clases que la implementan, aunque los hemos probado con listas. En verdad, las listas implementan la interfaz ***List***, que hereda de *Collection* y añade una serie de métodos y funcionalidades específicas de la lista, no compartidas por otras colecciones, como por ejemplo los conjuntos (*sets*).

Veamos los métodos más importantes aportados por la interfaz ***List*** a partir del siguiente ejemplo:

- Ejemplo: Vamos a trabajar con una lista de enteros. En este caso lo enteros no pueden ser primitivos y deberemos de utilizar las clases envoltorios o *wrappers*.

```
//Definición de la lista con el tipo genérico Integer
ArrayList<Integer> num1 = new ArrayList<>();
//Insertamos elementos
num1.add(3);// auboxing. No hace falta poner num1.add(new Integer(3));
num1.add(1);
num1.add(-2);
num1.add(0);
num1.add(3);
num1.add(7);
System.out.println(num1);
/************************************** RESULTADO ***********************************************/
[3, 1, -2, 0, 3, 7]
/************************************************************************************************/
```

E get(int index)	Devuelve el elemento que ocupa *index* en la lista. System.out.println(num1.get(2)); // devolverá -2
E set(int index, E element)	Guarda el elemento *element* en la posición *index*, machacando el valor que hubiera previamente en esa posición, el cual es devuelto por el método. System.out.println(num1.set(2,23));// devolverá -2 y guarda el 23 en la posición 2

void add(int index, E element)	Inserta el valor *element* en la posición *index*, añadiéndose, sin machacar el valor previo. Todos los nodos que ocupaban una posición mayor o igual que *index* se desplazan una posición hacia el final de la lista, para dejar hueco al nuevo elemento.
	num1.add(3,40); // Inicialmente: [3, 1, 23, 0, 3, 7]
	System.out.println(num1); // Resultado: [3, 1, 23, 40, 0, 3, 7]

boolean addAll(int index, Collection<? extends E> c)	Inserta todos los elementos de la colección c en la lista que invoca al método, empezando por el lugar *index*

y desplazando hacia el final todos los nodos de la lista original a partir de *index*, incluido éste. 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 E.

```
System.out.println(num1); // Inicialmente: [3, 1, 23, 40, 0, 3, 7]

ArrayList<Integer> num2 = new ArrayList<>();
num2.add(100);
num2.add(101);
num2.add(102);
num1.addAll(2,num2);
System.out.println(num1);// RESULTADO: [3, 1, 100, 101, 102, 23, 40, 0, 3, 7]
```

E remove(int index) Es una versión sobrecargada del método *remove( )* visto anteriormente. Elimina el nodo que ocupa la posición *index* y devuelve el elemento eliminado.

Si lo usamos en una lista de objetos *Integer*, la sentencia num1.remove(5) se interpreta que queremos eliminar el elemento que hay en el índice 5 ya que al haber una versión de *remove( )* que admite un *int* como argumento, Java no hace autoboxing. En el caso de que quisiéramos eliminar un nodo Integer cuyo valor es 5, escribiríamos:

```
num1.remove(new Integer(5)); // Aquí pasamos como argumento un objeto y no un int, por lo que Java ejecuta la versión primera del método y elimina el nodo de valor 5
```

ADEMÁS DE LOS MÉTODOS DE LECTURA, ESCRITURA, INSERCIÓN Y ELIMINACIÓN DE NODOS, LA INTERFAZ *LIST* AÑADE UN PAR DE FUNCIONES DE BÚSQUEDA.

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

int lastIndexOf(Object o) Lo mismo que *indexOf* pero empezando la búsqueda por el final.

boolean equals(Object o) Compara la lista que invoca el método con la que se le pasa como parámetro. Devuelve *true* si ambas tienen exactamente los mismos elementos en el mismo orden.

4.1.6.- Ejercicios 1

1. Crea un *ArrayList* con los nombres de 6 compañeros de clase. A continuación, muestra esos nombres por pantalla.

2. Realiza un programa que introduzca valores aleatorios (entre 0 y 100) en un *ArrayList* y que luego calcule la suma, la media, el máximo y el mínimo de esos números. El tamaño de la lista también será aleatorio y podrá oscilar entre 10 y 20 elementos ambos inclusive.

3. Escribe un programa que ordene 5 números enteros introducidos por teclado y almacenado en un objeto de la clase *ArrayList*.

4. Realiza un programa equivalente al anterior pero en esta ocasión, el programa debe ordenar palabras en lugar de números.

5. Deseamos crear una agenda en la cual guardar todos nuestros contactos. Utilizando un *ArrayList*, genera una aplicación con la que podamos generar nuestros contactos, visualizarlos, buscarlos y eliminarlos cuando ya no los necesitemos.

4.1.7.- Ejercicios 2

1. Ordenar una lista de empleados por su cargo, luego por su edad y luego por su salario. Primero comparando uno a uno con los comparadores creados y luego utilizando un comparador encadenado (OrdenarUnaListaPorAtributos).

4.2.-CONJUNTOS

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. Por lo tanto, si intentamos insertar un nodo que ya existe, no lo hará.

El conjunto de métodos disponibles es el mismo que vimos en el apartado 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()
boolean containsAll(Collection<?> c)
boolean addAll(Collection<? extends E> c)
boolean removeAll(Collection<?> c)
boolean retainAll(Collection<?> c)
void clear()
Object[] toArray()
<T> T[] toArray(T[])
```

Como en las Listas, para recorrer un conjunto también utilizaremos la estructura ***for-each***. Las diferencias importante 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 posible los nodos repetidos. Si intentamos insertar un nodo repetido con el método ***add()***, no se producirá ningún error ni se arrojará

ninguna excepción; sencillamente, el nodo no se inserta y 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.

Las implementaciones de ***Set*** son las clases:

- ***HashSet*** → Tiene un buen rendimiento, aunque no garantiza ningún orden en la inserción.
- ***TreeSet*** → A pesar de tener el peor rendimiento, garantiza el orden basado en el valor de los elementos insertados. El criterio de ordenación se lo proporciona un comparador en el constructor, o bien es natural (el proporcionado por el método ***compareTo()***), si no se especifica en el constructor.
- ***LinkedHashSet*** → Garantiza el orden basado en la inserción, ya que siempre inserta los nodos al final.

Las implementaciones de ***Set*** tienen diferencias en su comportamiento. Por eso, es muy común utilizar variables de tipo ***Set*** para referenciarlos. Esto permite aprovechar el polimorfismo de las distintas implementaciones y así poder hacer transformaciones de unas a otras.

```
Set<Integer> s1 = new TreeSet<>();
```

Usamos el tipo Set para la variable s1. De esa manera, con la misma variable podemos referenciar distintas implementaciones (HashSet o LinkedGHashSet)

- Ejemplo: Vamos a declarar un conjunto con un orden natural, basado en su implementación de la interfaz ***Comparable*** del tipo genérico:

```
public class Cliente implements Comparable{
  String dni;
  String nombre;
  int edad;
       . . . .
  @Override
    public int compareTo(Object o) {
      return this.dni.compareTo(((Cliente)o).dni);
    }
}
```

```
TreeSet<Cliente> c1 = new TreeSet<>();
```

La clase Cliente implementa el método ***compareTo()*** basado en el dni. Los nodos se insertaran ordenados por el atributo dni, ordenación natural de los objetos Cliente.

```
c1.add(new Cliente("111","Marta",20));
c1.add(new Cliente("115","Jorge",21));
c1.add(new Cliente("112","Carlos",18));
System.out.println(c1);
```

```
[Dni = 111  Nombre = Marta  Edad = 20
, Dni = 112  Nombre = Carlos  Edad = 18
, Dni = 115  Nombre = Jorge  Edad = 21
]
```

Si modificamos el método ***compareTo()*** para la edad el resultado sería diferente

```
public int compareTo(Object o) {
    Integer e1 = this.edad;
    Integer e2 = ((Cliente)o).edad;
    return e1.compareTo(e2);
}
```

```
[Dni = 112  Nombre = Carlos  Edad = 18
, Dni = 111  Nombre = Marta  Edad = 20
, Dni = 115  Nombre = Jorge  Edad = 21
]
```

En los dos ejemplos podemos ver que el orden en el que aparecen, que no coincide con el orden de inserción, corresponde a la implementación hecha para la interfaz ***Comparable*** de la clase ***Cliente***.

A continuación vamos a ver que dependiendo de la implementación del ***compareTo()***, considerará si el cliente a insertar está repetido o no (los ***Set*** no permiten duplicados y que sean duplicados depende del código escrito en el método ***compareTo()***).

Ahora insertamos un nodo que ya existe (tanto con el método ***compareTo()*** para dni como para edad el resultado es el mismo)

```
if(c1.add(new Cliente("111","Marta",21)))
        System.out.println("Cliente insertado");
else
        System.out.println("Cliente no insertado");
System.out.println(c1);
```

```
[Dni = 112  Nombre = Carlos  Edad = 18
, Dni = 111  Nombre = Marta  Edad = 20
, Dni = 115  Nombre = Jorge  Edad = 21
]
Cliente no insertado
[Dni = 112  Nombre = Carlos  Edad = 18
, Dni = 111  Nombre = Marta  Edad = 20
, Dni = 115  Nombre = Jorge  Edad = 21
]
```

Ahora insertamos el mismo nodo pero cambiando la edad i con el ***compareTo()*** de la edad

```
if(c1.add(new Cliente("111","Marta",21)))
```

La salida por pantalla es la misma que la anterior debido a que ya existe un cliente con esa edad

```
En este caso no hay ningún cliente con la edad 27
[Dni = 112  Nombre = Carlos  Edad = 18
, Dni = 111  Nombre = Marta  Edad = 20
, Dni = 115  Nombre = Jorge  Edad = 21
]
```

```
if(c1.add(new Cliente("111","Marta",27)))
```

```
Cliente insertado
[Dni = 112  Nombre = Carlos  Edad = 18
, Dni = 111  Nombre = Marta  Edad = 20
, Dni = 115  Nombre = Jorge  Edad = 21
, Dni = 111  Nombre = Marta  Edad = 27
]
```

Volvemos a hacer la misma prueba de cambiar la edad pero con el ***compareTo()*** del dni

```
if(c1.add(new Cliente("111","Marta",27)))
```

```
En este caso si que hay un cliente con el dni 111 y por eso no lo inserta, ya que un conjunto no admite duplicados
[Dni = 111  Nombre = Marta  Edad = 20
, Dni = 112  Nombre = Carlos  Edad = 18
, Dni = 115  Nombre = Jorge  Edad = 21
]
Cliente no insertado
[Dni = 111  Nombre = Marta  Edad = 20
, Dni = 112  Nombre = Carlos  Edad = 18
, Dni = 115  Nombre = Jorge  Edad = 21
]
```

Imaginemos que queremos otro conjunto de clientes ordenados por nombre. En este caso, lo podemos crear pasando al constructor, como argumento, un comparador basado en el atributo nombre. Antes de nada, vamos a recordar la interfaz ***Comparator***:

- Definíamos el método ***compareTo()***.
- Ponemos sendos cast a los objetos que queremos comparar para acceder a los atributos.

```
public class ComparaEdades implements Comparator {
  @Override
  public int compare(Object o1, Object o2) {
    Persona p1 = (Persona)o1;
    Persona s2 = (Persona)o2;
    return p1.edad-p2.edad;
  }
}
```

- Sin embargo la interfaz ***Comparator*** puede recibir tipos genéricos. Si especificamos la clase genérica, ya no hacen falta los cast.

```
public class ComparaEdades implements Comparator <Persona>{
  @Override
  public int compare(Persona p1, Persona p2) {
    return p1.edad-p2.edad;
  }
}
```

- Otro ejemplo sería un comparador de nombres para la clase Cliente.

```
public class ComparaNombres implements Comparator <Cliente>{
  @Override
  public int compare(Cliente cli1, Cliente cli2) {
```

```
        return cli1.nombre.compareTo(cli2.nombre);
      }
    }
```

Ahora vamos a crear un conjunto con un orden basado en los nombres de los clientes. Esto se lleva a cabo usando un ***TreeSet***, a cuyo constructor le pasamos un comparador de la clase **ComparaNombres**.

```
TreeSet<Cliente> c2 = new TreeSet<>(new ComparaNombres());
c2.add(new Cliente("111","Marta",20));
c2.add(new Cliente("115","Jorge",21));
c2.add(new Cliente("112","Carlos",18));
System.out.println(c2);
```

Independientemente del orden de inserción de cada Cliente, la inserción se realiza a partir del criterio ***ComparaNombres***, que los inserta en orden alfabético de sus nombres.

```
[Dni = 112  Nombre = Carlos  Edad = 18
, Dni = 115  Nombre = Jorge  Edad = 21
, Dni = 111  Nombre = Marta  Edad = 20
]
```

Si queremos que los nodos se vayan colocando por orden de inserción, como en las listas, en vez de un ***TreeSet*** crearemos un objeto ***LinkedHashSet***. Si el orden nos es indiferente, podemos usar un ***HashSet***, que tiene un mejor rendimiento

- Ejemplo: A continuación se muestra otro ejemplo del conjunto ***TreeSet***.

```
TreeSet<String> miConjunto = new TreeSet<>(); // CREAMOS un TreeSet
miConjunto.add("HOLA");// AÑADIMOS varios elementos al TreeSet
miConjunto.add("ADIOS");
System.out.println(miConjunto.size()); // CONSULTAMOS que TAMAÑO tiene
miConjunto.remove("HELLO");// ELIMINAMOS el elemento con valor "HELLO" (solo puede haber uno)
for (String cadaElemento: miConjunto){//RECORREMOS el TreeSet para ver sus valores.
   System.out.println(cadaElemento);//Modo bucle for-each
}
//COMPROBAMOS si existe un valor en el TreeSet
boolean esta = miConjunto.contains("HOLA");
System.out.println(esta);
//BORRAMOS TODOS los elementos del TreeSet
miConjunto.clear();
//ITERATOR: crear un iterador
Iterator<String> it = miConjunto.iterator();
//RECORRER el conjunto con un iterador
while(it.hasNext()){
   String s = it.next();//extraer elemento dentro del bucle
   if(s.startsWith("A"))
      it.remove();//eliminar el elemento que actualmente se está iterando
```

```
    }
    for (String cadaElemento: miConjunto){//RECORREMOS el TreeSet para ver sus valores.
        System.out.println(cadaElemento);//Modo bucle for-each
    }
}
```

Un ***TreeSet*** tiene siempre sus elemento ordenados. Si el elemento tiene un orden natural, como ***Integer*** o ***String***, funciona sin más, pero como añadamos objetos de clases propias a un ***TreeSet***, hemos de decirle cómo debe ordenar los objetos. Fijaros en el siguiente ejemplo:

- Ejemplo: Su ejecución produce una excepción debido a que no sabe cómo comparar los humanos para introducirlos en el conjunto ordenado TreeSet.

```
public class Humano {
    private String nombre;
    private String apellidos;

    public Humano(String nombre, String apellidos) {
        this.nombre = nombre;
        this.apellidos = apellidos;
    }
}
```

```
void start(){
    TreeSet<Humano> p = new TreeSet<>();
    p.add(new Humano("Joan","Carbonell"));
}
```

En este caso sería necesario decirle cómo tiene que ordenar los nodos cuando se introduzcan en un conjunto ordenado como ***TreeSet***:

```
public class CompararHumanosApellidos implements Comparator<Humano>{
    @Override
    public int compare(Humano o1, Humano o2) {
        return o1.getApellidos().compareTo(o2.getApellidos());
    }

TreeSet<Humano> p = new TreeSet<>(new CompararHumanosApellidos());
    p.add(new Humano("Joan","Carbonell"));
    p.add(new Humano("Anna","Amat"));
    p.add(new Humano("Joan","Carbonell"));
    p.add(new Humano("Joan","Pico"));
    System.out.println(p);
```

El humano Joan Carbonell ya no lo introduce porque existe. En cambio, Joan Pico si porque la instancia al criterio de ordenación **CompararHumanosApellidos** pasada al constructor del ***TreeSet***, comprueba que si un humano tiene el mismo apellido lo considera como duplicado y por eso no incluye ni Joan Carbonell ni Pepito Carbonell, ya que ambos existen aplicando el

criterio de ordenación incluido en el constructor.

4.2.1.- Conversiones entre colecciones

Una de las características más importantes 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 (***TreeSet***) a partir de otro que no lo está (***HashSet*** o ***LinkedHashSet***) o, dicho de otra forma, si queremos ordenar un conjunto, disponemos de dos caminos a seguir:

1. Construimos un ***TreeSet*** con el criterio de ordenación que deseamos y luego le añadimos con ***addAll()*** al conjunto a ordenar.

2. Si el criterio de ordenación va a ser el natural (el de la interfaz ***Comparable***), podemos construir un ***TreeSet*** pasando como argumento al constructor el conjunto que queremos ordenar.

- Ejemplo: Vamos a crear un ***LinkedHashSet*** (los elementos se ordenan por el orden en que fueron añadidos) de números enteros.

```
Set<Integer> s1 = new LinkedHashSet<>();
s1.add(4);
s1.add(1);
s1.add(5);
s1.add(10);
s1.add(3);
System.out.println(s1);
```

Utilizamos en nombre de la interfaz ***Set*** para la variable **s1**. Esto será necesario si queremos tener la posibilidad de referenciar, con la misma variable, conjuntos con distintas implementaciones. Si la variables **s1** fuera del tipo ***LinkedKHashSet***, solo serviría para referenciar objetos de esa clase, pero no un ***TreeSet***, por ejemplo. Es una forma más de **polimorfismo de Java**.

Ahora vamos a obtener un conjunto ordenado a partir de los elementos de s1. La primera opción será crear un ***TreeSet***:

```
Set<Integer> s2 = new TreeSet<>();
s2.addAll(s1);
System.out.println(s2);
```

El resultado en pantalla es:

[1, 3, 4, 5, 10]

En estos momentos tenemos dos variables s1 y s2 referenciando cada una de ellas a dos tipos distintos de conjunto

Si referenciamos **s1** con el nuevo conjunto creado, es decir **s1 = s2**, habríamos ordenado s1. A partir de ese momento s1 sería un ***TreeSet*** y mantendría el orden con la inserción y eliminación de nodos.

```
Set<Integer> s2 = new TreeSet<>();
s2.addAll(s1);                  El resultado en pantalla es:
s1 = s2;                        [4, 1, 5, 10, 3] //s1 era un LinkedHashSet
System.out.println(s1);         [1, 3, 4, 5, 10] //Aquí s1 pasa a ser un TreeSet
s1.add(6);                      [1, 3, 4, 5, 6, 10] //Vemos que s1 es un TreeSet y no un LinkedHashSet
System.out.println(s1);
```

La segunda forma de ordenar un conjunto es pasarlo al constructor de un ***TreeSet***.

```
Set<Integer> s2 = new TreeSet<>(s1);      Obtendríamos el resultado de lo que hemos hecho anteriormente
```

Este segundo método solo sirve si queremos un conjunto con el orden natural de los nodos. Si, en vez de enteros, tuviéramos un conjunto de **Clientes** sin ordenar y quisiéramos ordenarlos por nombre, tendríamos que usar el primer procedimiento, construyendo un ***TreeSet*** con un comparador **CompararNombres**.

```
//Conjunto sin orden
Set<Cliente> c1 = new LinkedHashSet<>();
c1.add(new Cliente("111","Marta",20));
c1.add(new Cliente("115","Jorge",21));
c1.add(new Cliente("112","Carlos",18));
System.out.println(c1);
Set<Cliente> c2 = new TreeSet<>(new CompararNombres());
//Tendremos en mismo conjunto pero ordenado por nombre
c2.addAll(c1);
System.out.println(c2);
```

```
RESULTADO:
[Dni = 111  Nombre = Marta  Edad = 20
, Dni = 115  Nombre = Jorge  Edad = 21
, Dni = 112  Nombre = Carlos  Edad = 18
]
[Dni = 112  Nombre = Carlos  Edad = 18
, Dni = 115  Nombre = Jorge  Edad = 21
, Dni = 111  Nombre = Marta  Edad = 20
]
```

```
public class CompararNombres implements Comparator <Cliente> {
  @Override
  public int compare(Cliente cli1, Cliente cli2) {
    return cli1.getNombre().compareTo(cli2.getNombre());
  }
}
```

Podemos ver con estos ejemplos que utilizando los constructores, es posible hacer conversiones entre todo tipo de colecciones. Se pueden crear listas pasando conjuntos a su constructor y viceversa; también se pueden añadir a una lista los elementos de un conjunto.

- Ejemplo: Un ejemplo útil es la creación de un conjunto a partir de una lista para eliminar elementos duplicados.

```
List<Integer> lista = new ArrayList<>();
lista.add(5);
lista.add(3);
lista.add(5);//elemento repetido
lista.add(2);
lista.add(5); //elemento repetido
System.out.println(lista);
//Conjunto si elementos repetidos
Set<Integer> s = new TreeSet<>(lista);
System.out.println(s);
```

RESULTADO:

[5, 3, 5, 2, 5]

[2, 3, 5]

Podemos ver que se han eliminado las repeticiones y al ser un TreeSet se han ordenado los elementos.

Cuando se trabaja con colecciones de distintos 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 encaso de conversiones. Por ejemplo, el código anterior podría ser:

```
Collection<Integer> col = new ArrayList<>();
col.add(5);
col.add(3);
col.add(5);//elemento repetido
col.add(2);
col.add(5); //elemento repetido
System.out.println(col);
//Conjunto si elementos repetidos
col = new TreeSet<>(col);
System.out.println(col);
```

El resultado sería el mismo:

[5, 3, 5, 2, 5]

[2, 3, 5]

En este caso, con **col** podemos referenciar cualquier lista o conjunto. El comportamiento dependerá del objeto referenciado: otro caso de **polimorfismo**.

4.2.2.- Clase Collections

Además de los métodos aportados por las interfaces ***Collection***, ***List*** y ***Set*** con sus implementaciones, la clase ***Collections*** (no confundirla con la interfaz ***Collection***) reúne una serie de utilidades en forma de métodos estáticos que trabajan con tipos genéricos. En ellos, el primer parámetro de entrada es la colección sobre la que deseamos operar y comprenden métodos de búsqueda, ordenación y manipulación de datos entre otros. Casi todos ellos operan sobre listas, aunque algunos valen para cualquier colección. El símbolo **T** es el tipo genérico de las colecciones y los nodos involucrados.

MÉTODOS DE ORDENACIÓN

La clase ***Collections*** posee métodos sobrecargados para ordenar

static void sort(List<T> lista)

Ordena la lista que se la pasa como argumento. El criterio de ordenación será el llamado criterio "natural", que es el que establecerá el método ***compareTo()*** de la interfaz ***Comparable*** para la clase T (el tipo de nodos de la lista, sea el que sea). Las clases envoltorio (***wrapper***) proporcionadas por Java como ***Integer***, ***Character*** o ***Double***, así como la clase ***String*** lo traen implementado, de manera que ordenan los números de menor a mayor. Los caracteres según el orden Unicode y los String por orden alfabético.

```
List<Cliente> l = new ArrayList<>();
l.add(new Cliente("111","Marta",20));
l.add(new Cliente("115","Jorge",21));
l.add(new Cliente("112","Carlos",18));
System.out.println(l);
Collections.sort(l);
System.out.println(l);
```

```
RESULTADO:
[Dni = 111  Nombre = Marta  Edad = 20
, Dni = 115  Nombre = Jorge  Edad = 21
, Dni = 112  Nombre = Carlos  Edad = 18
]
[Dni = 111  Nombre = Marta  Edad = 20
, Dni = 112  Nombre = Carlos  Edad = 18
, Dni = 115  Nombre = Jorge  Edad = 21
]
```

La lista se ordenará con el criterio natural del tipo con el que se declaró. Podemos ver que en la clase Cliente ha sido por **dni**. Si queremos ordenar por otro criterio, tendremos que usar comparadores. Para ello disponemos de una segunda versión sobrecargada de ***sort()***, en la que se añade como segundo parámetro el comparador con el criterio deseado. Si queremos ordenar por el nombre deberíamos escribir:

```
Collections.sort(l, new ComparaNombres())
```

MÉTODOS DE BÚSQUEDA

La clase ***Collections*** posee métodos sobrecargados para buscar

static int binarySearch()
(consultar API de Java)

Hace una búsqueda binaria de un objeto (llamado clave de búsqueda) en una lista que debe estar ordenada previamente. Todo ello necesita un criterio de ordenación que, por defecto, es el natural del tipo genérico de la lista. No obstante, se exige que la implementación de Comparable sea también genérica

En la clase Cliente sería de la forma:

```
public class Cliente implements Comparable<Cliente>{
        . . .
        public int compareTo(Cliente ob) {
                return dni.compareTo(ob.dni);
        }
}
```

Aquí es una exigencia de la sintaxis de ***binarySearch()***

- Ejemplo:

```
List<Cliente> l = new ArrayList<>();
l.add(new Cliente("111","Marta",20));
l.add(new Cliente("115","Jorge",21));
l.add(new Cliente("112","Carlos",18));
System.out.println(l);
Collections.sort(l);
System.out.println(l);
//Buscar a Carlos cuyo dni es 112
//Devuelve el índice si lo encuentra
int indice = Collections.binarySearch(l,new Cliente("112",null,0));
System.out.println("Indice: "+indice);
//Si no lo encuentra devuelve un entero negativo que nos indicará
//el índice de inserción que le correspondería al nodo si lo
//insertamos manteniendo la lista ordenada:
indice = Collections.binarySearch(l,new Cliente("114",null,0));
System.out.println("Indice: "+indice);
```

RESULTADO:

```
[Dni = 111  Nombre = Marta  Edad = 20
, Dni = 115  Nombre = Jorge  Edad = 21
, Dni = 112  Nombre = Carlos  Edad = 18
]
[Dni = 111  Nombre = Marta  Edad = 20
, Dni = 112  Nombre = Carlos  Edad = 18
, Dni = 115  Nombre = Jorge  Edad = 21
]
Indice: 1
Indice: -3
```

Ese ínice -3 indica que no existe i que si lo insertamos debería de ocupar la posición 2 (-indice-1) para mantener la lista ordenada.

Si decidiéramos insertar un cliente que no existe, y queremos mantener la lista ordenada, pondríamos lo siguiente:

```
Cliente nuevo = new Cliente("114","Vicent",23);
indice = Collections.binarySearch(l,nuevo);
if(indice < 0){//no está en la lista
        l.add(-indice-1,nuevo);//la lista sigue ordenada
}
System.out.println(l);;
```

RESULTADO:

```
[Dni = 111  Nombre = Marta  Edad = 20
, Dni = 112  Nombre = Carlos  Edad = 18
, Dni = 114  Nombre = Vicent  Edad = 23
, Dni = 115  Nombre = Jorge  Edad = 21
]
```

Si queremos hacer una búsqueda en una lista ordenada con un criterio distinto al natural, tendremos que pasar a ***binarySearch()*** como tercer parámetro, el mismo comparador que se usó para ordenarla.

- Ejemplo:

```
//Ordenamos la lista por orden alfabético de los nombres
Collections.sort(l, new CompararNombres());
//Ahora, para buscar a Carlos, debemos hacerlo por nombre
indice = Collections.binarySearch(l, new Cliente(null, "Carlos",0), new CompararNombres() )
```

Devolverá 0, ya que Carlos es el primero de la lista por orden alfabético de nombres

El único inconveniente que tiene el método binarySearch() es que precisa que la lista esté ordenada previamente. Merece la pena ordenar la lista si vamos a tener que hacer muchas búsquedas con el mismo criterio. En caso contrario, es más eficiente hacer una búsqueda

secuencial, por medio de un bucle

MÉTODOS PARA LA MANIPULACIÓN DE DATOS

La clase ***Collections*** posee métodos sobrecargados para manipular

static void swap(List<?> lista, int i, int j) (consultar API de Java) Sirve para intercambiar dos nodos en una lista. Intercambia los nodos con índice i y j entre sí.

Ejemplo con una lista de enteros

```
List<Integer> num1 = new ArrayList<>();
    num1.add(1);//índice 0
    num1.add(2);
    num1.add(3);
    num1.add(4);//índice 3
    num1.add(5);
    System.out.println(num1);
    //Cambiar elementosde índices 0 y 3
    Collections.swap(num1,0,3);
    System.out.println(num1);
```

[1, 2, 3, 4, 5]

[4, 2, 3, 1, 5]

static <T> boolean replaceAll(List<T> list, T oldVal, T newVal) Reemplaza el nodo *oldVal*, en todos los lugares en que aparezca en la lista, por el nodo *newVal.*

Queremos reemplazar los nodos que valgan 4 por 100

```
System.out.println(num1);
 //Reemplazar nodos 4 por 100
 Collections.replaceAll(num1,4,100);
 System.out.println(num1);
```

[4, 2, 3, 1, 5]

[100, 2, 3, 1, 5]

static <T> void fill(List<? super T> list, T obj) Sustituye todos los valores de los nodos de la lista por el de *obj*. La lista debe de ser de tipo <? super T>, es decir, de la clase T o cualquier subclase de T. Dicho de otro modo, el nodo de relleno debe ser de la clase de la lista o de una subclase. Esto garantiza que el elemento de relleno de la clase T se pueda insertar en ella.

Rellenamos num1 con el valor 7

```
System.out.println(num1);
Collections.fill(num1,7);
System.out.println(num1);
```

[100, 2, 3, 1, 5]

[7, 7, 7, 7, 7]

static <T> void copy(List<? super T> dest, List<? extends T> src) Copia los elementos de la lista *src* en la lista *dest*, empezando por el principio y 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 a lista destino. Por eso, esta última debe ser de clase T o superclase de T (<? super T>) y la lista origen debe ser clase T o subclase T (<? extends T). En particular, si las dos listas son de la misma clase genérica, se podrán copiar sin problema.

Construimos una segunda lista de enteros:

```
List<Integer> num2 = new ArrayList<>();
    num2.add(9);
    num2.add(10);
    num2.add(11);
    //Copiamos num2 en num1
    System.out.println("Num1: "+num1);
    System.out.println("Num2: "+num2);
    Collections.copy(num1,num2);
    System.out.println("Num1: "+num1);
    System.out.println("Num2: "+num2);
```

```
Num1: [7, 7, 7, 7, 7]
Num2: [9, 10, 11]
Num1: [9, 10, 11, 7, 7]
Num2: [9, 10, 11]
```

OTRAS UTILIDADES

static void shuffle(List<?> list)	Barajar, es decir, desordenar los elementos de la lista
static int frequency(Collection<?> c, Object o)	Número de veces que aparece un nodo en una colección. Tiene sentido en las listas, ya que en los conjuntos no hay repeticiones
static T max(Collection<? extends T> coll)	Nos devuelve el valor máximo de una colección. El máximo lo busca basándose en el orden natural. Eso exige que la clase genérica tenga implementada la interfaz ***Comparable***.

Si queremos que nos devuelva el valor máximo atendiendo a un criterio de ordenación distinto al natural, le pasaremos a ***max()*** un segundo parámetro con un comparador adecuado.

static T max(Collection<? extends T> coll, Comparator <? super T> comp)

En el conjunto Cliente s1, con Marta, Carlos y Jorge, si queremos obtener el máximo, es decir, el nodo que ocuparía el último lugar si el conjunto estuviera ordenado por orden alfabético de nombres, pondremos.

```
Cliente ultimo = Collections.max(s1, new ComparaNombres());
```

Para llamar al método ***max()*** hace falta un criterio de ordenación, pero eso no implica que la colección tenga que estar ordenada.

Hay métodos análogos para calcular el mínimo de una colección, que funcionan exactamente igual:

```
Integer minimo = Collections.min(num1);
```

Cliente primero = Collections.min(s1, new ComparaNombres());

static void reverse(List<?> list)	Invierte la lista, colocando los nodos en orden inverso. Invierte la lista original.
Set<T> singleton(T o)	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.

- Ejemplo: Para eliminar el 7, que aparece dos veces en num1, escribimos:

  num1.removeAll(Collections.singleton(7));

4.3.-MAPAS

Los mapas o diccionarios son estructuras dinámicas cuyos nodos, que aquí se llaman entradas (las entradas son un tipo que viene implementado por la interfaz ***Map.Entry***) , son pares **Clave / Valor** en vez de valores individuales como en las colecciones. Todas ellas implementan la interfaz ***Map*** que no hereda de Collection. Por lo tanto los mapas no son colecciones, aunque están muy relacionados con ellas y funcionan dentro del mismo entorno de trabajo. Existen tres implementaciones de ***Map***: ***HashMap***, ***TreeMap*** y ***LinkedHashSet***, cuyas diferencias son muy similares a las implementaciones de ***Set***: ***HashSet***, ***TreeSet*** y ***LinkedHashSet***.

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

Las operaciones fundamentales en una mapa son la inserción, la lectura y la eliminación de entradas (aunque hay algunas más). Para ilustrar el uso de mapas vamos a empezar utilizando la **implementación *HashMap***, que no garantiza ningún orden de inserción en las entradas, aunque es muy eficiente en cuanto a la velocidad de acceso a los datos. El constructor más sencillo es:

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

K es el tipo de las claves y **V** el de los valores. Son tipos genéricos que necesariamente deben de ser clases y no tipos primitivos. En el constructor anterior hemos elegido ***Map*** como tipo de la variable **m** (podríamos haber puesto ***HashMap***), con objeto de garantizar la posibilidad de un mayor polimorfismo. El comportamiento de **m** estará determinado por la clase del objeto referenciado.

- Ejemplo: Queremos mantener la información de la estaturas de un grupo de escolares, con entradas en las que figura el nombre del alumno (***String***) como clave y la estatura (***Double***) como valor.

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

V put(K key, V value) Inserta una entrada. Le pasamos la clave y el valor asociado a ella. Si no había ninguna entrada previa con la misma clave, se inserta en el mapa la nueva entrada con esa clave y ese valor, y el método devuelve ***null***. Si ya había una entrada con la misma clave, se sustituye el valor antiguo por el nuevo y la función devuelve el valor antiguo.

```
m.put("Anna",1.65);
m.put("Marta",1.60);
m.put("Luis",1.73);
m.put("Pedro",1.69);
System.out.println(m.toString());
```

RESULTADO:

{Marta=1.6, Luis=1.73, Pedro=1.69, Anna=1.65}

Los mapas, igual que los conjuntos, también disponen de una implementación ***toString()*** para visualizarlos.

```
//La ejecución de esta instrucción,
//a parte de actualizar la estatura de Pedro,
//nos devuelve la estatura antigua.
System.out.println(m.put("Pedro",1.71))
System.out.println(m.toString());
```

RESULTADO:

1.69

{Marta=1.6, Luis=1.73, Pedro=1.71, Anna=1.65}

V remove(Object key) Elimina la entrada cuya clave es ***key***, si existe. En este caso, devuelve el valor asociado con esa clave. En caso contrario, devuelve ***null***.

void clear() Elimina todas las entradas y deja el mapa vacío.

V get(Object key) Devuelve el valor asociado con la clave ***key*** o ***null*** si no hay ninguna entrada con esa clave

```
System.out.println(m.get("Anna"));
```

1.65

boolean containsKey(Object key) Devuelve ***true*** si hay una entrada con la clave ***key***.

```
System.out.println(m.containsKey("Anna"));
```

true

boolean containsValue(Object value) Devuelve ***true*** si hay alguna entrada con valor ***value***

```
System.out.println(m.containsValue(1.56));
```

false

Dos mapas se pueden comparar entre sí con el método ***equals()***, que devuelve ***true*** si ambos tienen exactamente las mismas entradas.

4.3.1.- Vistas Collections de los mapas

Aunque ***Map*** no hereda de ***Collection***, los mapas están íntimamente ligados a las colecciones,

de forma que se trabaja simultáneamente con ambas interfaces a través de distintas vistas con estructura de colección. Por vista entendemos una colección respaldada por el mapa original, de forma que cuando accedemos a un nodo de la vista estamos accediendo a la entrada original en el mapa, con lo que los cambios que se hagan en aquella se reflejarán en este. Hay tres tipos de vistas de una mapa.

1. Una vista de las claves del mapa.

   Set<K> keySet() Nos devuelve una vista, con estructura ***Set***, de las claves presentes en un mapa

   ```
   Set<String> claves = m.keySet();
   System.out.println(claves);
   ```
   [Marta, Luis, Pedro, Anna]

2. Una vista de los valores del mapa.

   Collection<V> values() Devuelve un vista ***Collection*** de los valores. Si alguno se encuentra más de una vez en el mapa, también aparece repetido en la colección devuelta.

   ```
   Collection<Double> valores1 = m.values();
   System.out.println(valores1);
   ```
   [1.6, 1.73, 1.71, 1.65]

3. Una vista de las entradas del mapa.

   Set<Map.Entry<K,V>> entrySet() Devuelve una vista conjunto de las entradas, objetos del tipos que implementa ***Map.Entry***, de las que se puede obtener la clave con ***getKey()*** o el valor con ***getVAlue()***.

   ```
   Set<Map.Entry<String,Double>> e1 = m.entrySet();
   System.out.println(e1);
   ```
   [Marta=1.6, Luis=1.73, Pedro=1.71, Anna=1.65]

   Se puede usar la vista de entradas para acceder a las entradas individuales y obtener la clave o el valor, o bien para cambiar su valor, con los métodos de la interfaz ***Map.Entry***:

   - **K getKey()** → Devuelve la clave de la entrada.
   - **V getValue()** → Devuelve el valor de la entrada.
   - **V setValue(V value)** → Asigna ***value*** a la entrada y devuelve el valor antiguo.

Uno de los inconvenientes de los mapas es que no son iterables. Esto supone que, además de no poder usar iteradores para recorrerlos ni eliminar nodos, tampoco es posible el uso de la estructura ***for-each***, cosa que sí podemos hacer con las vistas obtenidas, ya que son colecciones. Como hemos visto, los cambios que hagamos en esas vistas se reflejarán en el mapa. En particular, 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.

- Ejemplo: Eliminamos la clave "Marta" del conjunto de claves. Eliminar de la vista supone la eliminación del mapa

```
System.out.println(claves);
System.out.println(m.toString());
claves.remove("Marta");
System.out.println(claves);
System.out.println(m.toString());
```

```
RESULTADO:
[Marta, Luis, Pedro, Anna]
{Marta=1.6, Luis=1.73, Pedro=1.71, Anna=1.65}
[Luis, Pedro, Anna]
{Luis=1.73, Pedro=1.71, Anna=1.65}
```

La única forma segura de eliminar entradas durante un proceso de iteración sobre cualquiera de las tres vistas es el método ***remove()*** de la interfaz ***Iterator***.

- Ejemplo: Añadimos más entradas al mapa. Filtramos el mapa eliminando todos aquellos alumnos con estatura mayor que 1.71. Para ello iteramos sobre el conjunto de las entradas.

```
//Filtrar mapa con estaturas mayores a 1.71.l
m.put("Lucas",1.8);
m.put("Marta",1.6);
m.put("Jorge",1.75);
System.out.println(m.toString());
//OBTENEMOS EL CONJUNTO DE ENTRADAS
Set<Map.Entry<String,Double>> e2 = m.entrySet();
//ITERAMOS SOBRE EL CONJUNTO DE ENTRADAS
for (Iterator<Map.Entry<String,Double>> it = e2.iterator();it.hasNext();){
  Map.Entry<String,Double> o = it.next();
  if (o.getValue() > 1.71)
    it.remove();
}
System.out.println(m.toString());
            RESULTADO:
            {Marta=1.6, Luis=1.73, Lucas=1.8, Pedro=1.71, Jorge=1.75, Anna=1.65}
            {Marta=1.6, Pedro=1.71, Anna=1.65}
 //TAMBIÉN SE HUBIESE PODIDO HACER ITERANDO SOBRE LA VISTA DE LOS VALORES
Collection<Double> valores2 = m.values();
for (Iterator<Double> it = valores1.iterator();it.hasNext();){
  Double v = it.next();
  if (v > 1.71)
    it.remove();
}
```

En cambio, no podemos añadir entradas a un mapa por medio de ***add()*** o ***addAll()*** a través de

ninguna de sus vistas de tipo colección.

4.3.2.- Implementaciones de Map

En todos los ejemplos de mapas hemos usado la implementación ***HashMap***, que destaca por su eficiencia, pero que no garantiza ningún orden en la inserción de los nodos. La interfaz ***Map*** tiene otras dos implementaciones: ***TreeMap*** y ***LinkedHashMap***.

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. Por ejemplo:

```
TreeMap<String, Double> tm = new TreeMap<>();
tm.put("Pedro",1.71);
tm.put("Anna",1.65);
tm.put("Marta",1.60);
tm.put("Luis",1.73);
System.out.println(tm.toString());
```

```
RESULTADO:
{Anna=1.65, Luis=1.73, Marta=1.6, Pedro=1.71}
```

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.

Por último, 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.

4.3.3.- Ejercicios de Maps

1. Queremos calcular la frecuencia absoluta de aparición o número de veces que aparecen los caracteres en un String.

2. Supongamos que tenemos frase y se quiere obtener las posiciones que ocupan las palabras de esa frase en la frase. Por ejemplo, para la siguiente frase “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]}

3. Imaginemos que necesitamos una aplicación para una tienda mediante la que queremos almacenar los distintos productos que venderemos y el precio que tendrán. Y se quiere que tenga las funciones básicas, introducir un elemento, modificar su precio, eliminar un producto y mostrar los productos que tenemos con su precio.

5.- EJERCICIOS 1

1. Crear una colección de 20 números enteros aleatorios menores que 100 y guárdalos en el orden en que se vayan generando; mostrar por pantalla dicha lista una vez creada. Ordenarla en sentido creciente y volverla a mostrar por pantalla.

2. Repetir el ejercicio anterior, pero ordenar la lista en sentido decreciente.

3. Crear una colección de 20 números enteros aleatorios distintos menores que 100 y guárdalos por orden decreciente a medida que se vayan generando. Mostrar la colección por pantalla.

4. Repetir el ejercicio anterior, pero esta vez permitir números repetidos y utilizar números aleatorios menores que 10

5. Introducir por teclado, hasta que se introduzca ‘’fin’’, una serie de nombres que se insertarán en una colección, de forma que se conserve el orden de inserción y que no puedan repetirse. Mostrar la lista por pantalla.

6. Introducir por teclado, hasta que se introduzca ‘’fin’’, una serie de nombres que se insertarán por orden alfabético en una colección que no permita repeticiones. Mostrar la lista de nombres por pantalla.

7. Implementar una función a la que se pase una lista de nombres y devuelva una copia sin elementos repetidos (sin modificar la original), con el prototipo:

   List eliminaRepeticiones(List c)

8. Introducir por consola una frase que conste exclusivamente de palabras separadas por espacios. Almacenar en una lista las palabras de la frase, una en cada nodo y mostrar por pantalla las palabras que estén repetidas. A continuación, mostrar las que no lo estén.

9. Implementar el método unión de dos conjuntos, que devuelva un nuevo conjunto con todos loes elementos que pertenezcan, al menos, a uno de los dos conjuntos, con el prototipo:

   Set union(Set conjunto1, Set conjunto2)

10. Hacer los mismo que en el ejercicio anterior con la intersección, formada por los elementos comunes a los dos conjuntos, con el prototipo:

    Set interseccion(Set conjunto1, Set conjunto2)

11. Diseñar un método que nos devuelva la diferencia de dos conjuntos (elementos que pertenecen al primero, pero no al segundo), con la sintaxis:

    Set diferencia(Set conjunto1, Set conjunto2)

12. Escribir el método **incluido()**, que devuelve *true* si todos los elementos del primer conjunto pertenecen al segundo y *false* si hay algún elemento del primero que no pertenezca al segundo. Su sintaxis es:

    boolean incluido (Set conjunto1, Set conjunto2)

13. Implementar una función a la que se les pasen dos listas ordenadas y nos devuelva una única lista, fusión de las dos anteriores. Desarrollar el algoritmo de forma no destructiva, es decir, que las listas utilizadas como parámetros de entrada se mantengan intactas.

14. Implementar la función **leeCadena**, con el siguiente prototipo:

    List<Character> leeCadena()

    Dicha función lee una cadena por teclado y nos la devuelve en una lista con un carácter en cada nodo.

15. Implementar la función **uneCadenas**, con el prototipo:

    List<Character> uneCadenas(List<Character> cad1, List<Character> cad2)

    que devuelva una lista con la concatenación de cad1 y cad2.

16. Implementar la función:

    List clonaList(List)

17. Diseñar la clase ListaOrdenada que hereda de LinkedList y que permita la inserción ordenada. Codificar un método que inserte un nuevo elemento con el prototipo:

    void insertarOrdenado(E elemento)

6.- EJERCICIOS 2

1. Crea un ArrayList con los nombres de 6 compañeros de clase. A continuación, muestra esos nombres por pantalla. Utiliza para ello un bucle for que recorra todo el ArrayList sin usar ningún índice.

2. Realiza un programa que introduzca valores aleatorios (entre 0 y 100) en un ArrayList y que luego calcule la suma, la media, el máximo y el mínimo de esos números. El tamaño de la lista también será aleatorio y podrá oscilar entre 10 y 20 elementos ambos inclusive.

3. Escribe un programa que ordene 10 números enteros introducidos por teclado y almacenados en un objeto de la clase ArrayList.

4. Realiza un programa equivalente al anterior pero en esta ocasión, el programa debe ordenar palabras en lugar de números.

5. Realiza una gestión típica – alta, baja, listado y modificación - de una colección de discos. Este tipo de programas se suele denominar CRUD (Create Read Update Delete). Un disco puede tener un código, autor, título, género y duración en minutos.

6. Implementa el control de acceso al área restringida de un programa. Se debe pedir un nombre de usuario y una contraseña. Si el usuario introduce los datos correctamente, el programa dirá “Ha accedido al área restringida”. El usuario tendrá un máximo de 3 oportunidades. Si se agotan las oportunidades el programa dirá “Lo siento, no tiene acceso al área restringida”. Los nombres de usuario con sus correspondientes contraseñas deben estar almacenados en una estructura de la clase HashMap.

7. La máquina Eurocoin genera una moneda de curso legal cada vez que se pulsa un botón siguiendo la siguiente pauta: o bien coincide el valor con la moneda anteriormente generada - 1 céntimo, 2 céntimos, 5 céntimos, 10 céntimos, 25 céntimos, 50 céntimos, 1 euro o 2 euros - o bien coincide la posición – cara o cruz. Simula, mediante un programa, la generación de 6 monedas aleatorias siguiendo la pauta correcta. Cada moneda generada debe ser una instancia de la clase Moneda y la secuencia se debe ir almacenando en una lista.

```
Ejemplo:
2 céntimos – cara
2 céntimos – cruz
50 céntimos – cruz
1 euro – cruz
1 euro – cara
10 céntimos – cara
```

8. Realiza un programa que escoja al azar 10 cartas de la baraja española (10 objetos de la clase Carta). Emplea un objeto de la clase ArrayList para almacenarlas y asegúrate de que no se repite ninguna.

9. Modifica el programa anterior de tal forma que las cartas se muestren ordenadas. Primero se ordenarán por palo: bastos, copas, espadas, oros. Cuando coincida el palo, se ordenará por número: as, 2, 3, 4, 5, 6, 7, sota, caballo, rey.

10. Crea un mini-diccionario español-inglés que contenga, al menos, 20 palabras (con su correspondiente traducción). Utiliza un objeto de la clase HashMap para almacenar las parejas de palabras. El programa pedirá una palabra en español y dará la correspondiente traducción en inglés.

11. Realiza un programa que escoja al azar 5 palabras en español del mini-diccionario del ejercicio anterior. El programa irá pidiendo que el usuario teclee la traducción al inglés de cada una de las palabras y comprobará si son correctas. Al final, el programa deberá mostrar cuántas respuestas son válidas y cuántas erróneas.

12. Escribe un programa que genere una secuencia de 5 cartas de la baraja española y que sume los puntos según el juego de la brisca. El valor de las cartas se debe guardar en una estructura HashMap que debe contener parejas (figura, valor), por ejemplo (“caballo”, 3). La secuencia de cartas debe ser una estructura de la clase ArrayList que contiene objetos de la clase Carta. El valor de las cartas es el siguiente: as → 11, tres → 10, sota → 2, caballo → 3, rey → 4; el resto de cartas no vale nada.

```
Ejemplo:
as de oros
cinco de bastos
caballo de espadas
sota de copas
tres de oros
Tienes 26 puntos
```

13. Un supermercado de productos ecológicos nos ha pedido hacer un programa para vender su mercancía. En esta primera versión del programa se tendrán en cuenta los productos que se indican en la tabla junto con su precio. Los productos se venden en bote, brick, etc. Cuando se realiza la compra, hay que indicar el producto y el número de unidades que se compran, por ejemplo **guisantes** si se quiere comprar un bote de guisantes y la cantidad, por ejemplo **3** si se quieren comprar 3 botes. La compra se termina con la palabra **fin**. Suponemos que el usuario no va a intentar comprar un producto que no existe. Utiliza un diccionario para almacenar los nombres y precios de los productos y una o varias listas para almacenar la compra que realiza el usuario.

A continuación se muestra una tabla con los productos disponibles y sus respectivos precios:

avena	garbanzos	tomate	jengibre	quinoa	guisantes
2,21	2,39	1,59	3,13	4,50	1,60

Ejemplo:

```
Producto: tomate
Cantidad: 1
Producto: quinoa
Cantidad: 2
Producto: avena
Cantidad: 1
Producto: tomate
Cantidad: 2
Producto: fin

Producto Precio Cantidad Subtotal
---------------------------------
tomate    1,59     1       1,59
quinoa    4,50     2       9,00
avena     2,21     1       2,21
tomate    1,59     2       3,18
---------------------------------
TOTAL: 15,98
```

14. Realiza una nueva versión del ejercicio anterior con las siguientes mejoras: Si algún producto se repite en diferentes líneas, se deben agrupar en una sola. Por ejemplo, si se pide primero 1 bote de tomate y luego 3 botes de tomate, en el extracto se debe mostrar que se han pedido 4 botes de tomate. Después de teclear “fin”, el programa pide un código de descuento. Si el usuario introduce el código “ECODTO”, se aplica un 10% de descuento en la compra. Ejemplo:

```
Producto: tomate
Cantidad: 1
Producto: quinoa
Cantidad: 2
Producto: avena
Cantidad: 1
Producto: quinoa
Cantidad: 2
Producto: tomate
Cantidad: 2
Producto: fin
Introduzca código de descuento (INTRO si no tiene ninguno): ECODTO

Producto Precio Cantidad Subtotal
---------------------------------
tomate    1,59     3        4,77
quinoa    4,50     4       18,00
avena     2,21     1        2,21
---------------------------------
Descuento: 2,50
---------------------------------
TOTAL: 22,48
```

15. Realiza un programa que sepa decir la capital de un país (en caso de conocer la respuesta) y que, además, sea capaz de aprender nuevas capitales. En principio, el programa solo conoce las capitales de España, Portugal y Francia. Estos datos deberán estar almacenados en un diccionario. Los datos sobre capitales que vaya aprendiendo el programa se deben almacenar en el mismo diccionario. El usuario sale del programa escribiendo la palabra “salir”.

```
Escribe el nombre de un país y te diré su capital: España
La capital de España es Madrid
Escribe el nombre de un país y te diré su capital: Alemania
No conozco la respuesta ¿cuál es la capital de Alemania?: Berlín
Gracias por enseñarme nuevas capitales
Escribe el nombre de un país y te diré su capital: Portugal
La capital de Portugal es Lisboa
Escribe el nombre de un país y te diré su capital: Alemania
La capital de Alemania es Berlín
Escribe el nombre de un país y te diré su capital: Francia
La capital de Francia es París
Escribe el nombre de un país y te diré su capital: salir
```

16. Una empresa de venta por internet de productos electrónicos nos ha encargado implementar un carrito de la compra. Crea la clase Carrito. Al carrito se le pueden ir agregando elementos que se guardarán en una lista, por tanto, deberás crear la clase Elemento. Cada elemento del carrito deberá contener el nombre del producto, su precio y la cantidad (número de unidades de dicho producto). A continuación se muestra tanto el contenido del programa principal como la salida que debe mostrar el programa. Los métodos a implementar se pueden deducir del main.

Contenido del main:

```
Carrito miCarrito = new Carrito();
miCarrito.agrega(new Elemento("Tarjeta SD 64Gb", 19.95, 2));
miCarrito.agrega(new Elemento("Canon EOS 2000D", 449, 1));
System.out.println(miCarrito);
System.out.print("Hay " + miCarrito.numeroDeElementos());
System.out.println(" productos en la cesta.");
System.out.println("El total asciende a "
  + String.format("%.2f", miCarrito.importeTotal()) + " euros");

System.out.println("\nContinúa la compra...\n");
miCarrito.agrega(new Elemento("Samsung Galaxy Tab", 199, 3));
miCarrito.agrega(new Elemento("Tarjeta SD 64Gb", 19.95, 1));
System.out.println(miCarrito);
System.out.print("Ahora hay " + miCarrito.numeroDeElementos());
System.out.println(" productos en la cesta.");
System.out.println("El total asciende a "
  + String.format("%.2f", miCarrito.importeTotal()) + " euros");
```

Salida:

```
Contenido del carrito
=====================
Tarjeta SD 64Gb PVP: 19,95 Unidades: 2 Subtotal: 39,90
Canon EOS 2000D PVP: 449,00 Unidades: 1 Subtotal: 449,00

Hay 2 productos en la cesta.
El total asciende a 488,90 euros

Continúa la compra...


Contenido del carrito
=====================
Tarjeta SD 64Gb PVP: 19,95 Unidades: 2 Subtotal: 39,90
Canon EOS 2000D PVP: 449,00 Unidades: 1 Subtotal: 449,00
Samsung Galaxy Tab PVP: 199,00 Unidades: 3 Subtotal: 597,00
Tarjeta SD 64Gb PVP: 19,95 Unidades: 1 Subtotal: 19,95

Ahora hay 4 productos en la cesta.
El total asciende a 1105,85 euros
```

17. Mejora el programa anterior (en otro proyecto diferente) de tal forma que al intentar agregar un elemento al carrito, se compruebe si ya existe el producto y, en tal caso, se incremente el número de unidades sin añadir un nuevo elemento. Observa que en el programa anterior, se repetía el producto “Tarjeta SD 64Gb” dos veces en el carrito. En esta nueva versión ya no sucede esto, si no que se incrementa el número de unidades del producto que se agrega. El contenido del main es idéntico al ejercicio anterior pero la salida es:

```
Contenido del carrito
=====================
Tarjeta SD 64Gb PVP: 19,95 Unidades: 2 Subtotal: 39,90
Canon EOS 2000D PVP: 449,00 Unidades: 1 Subtotal: 449,00

Hay 2 productos en la cesta.
El total asciende a 488,90 euros

Continúa la compra...


Contenido del carrito
=====================
Tarjeta SD 64Gb PVP: 19,95 Unidades: 3 Subtotal: 59,85
Canon EOS 2000D PVP: 449,00 Unidades: 1 Subtotal: 449,00
Samsung Galaxy Tab PVP: 199,00 Unidades: 3 Subtotal: 597,00

Ahora hay 3 productos en la cesta.
El total asciende a 1105,85 euros
```

18. Realiza un buscador de sinónimos. Utiliza el diccionario español-inglés que se proporciona a continuación. El programa preguntará una palabra y dará una lista de sinónimos (palabras que tienen el mismo significado). Por ejemplo, si se introduce la palabra “caliente”, el programa dará como resultado: ardiente, candente, abrasador. ¿Cómo sabe el programa cuáles son los sinónimos de “caliente”? Muy fácil, en el diccionario debe existir la entrada (“caliente”,“hot”), por tanto solo tendrá que buscar las palabras en español que también signifiquen “hot”; esta información estará en las entradas (“ardiente”, “hot”) y (“abrasador”, “hot”). Cuando una palabra existe en el diccionario pero no tiene sinónimos, debe mostrar el mensaje “No conozco sinónimos de esa palabra”. Si una palabra no está en el diccionario se mostrará el mensaje “No conozco esa palabra”. El usuario sale del programa escribiendo la palabra “salir”.

Español	caliente	rojo	ardiente	verde	agujetas	abrasador	hierro	grande
Inglés	hot	red	hot	green	stiff	hot	iron	big

Ejemplo:

```
Introduzca una palabra y le daré los sinónimos: caliente
Sinónimos de caliente: ardiente, abrasador
Introduzca una palabra y le daré los sinónimos: rojo
No conozco sinónimos de esa palabra
Introduzca una palabra y le daré los sinónimos: blanco

No conozco esa palabra
Introduzca una palabra y le daré los sinónimos: grande
No conozco sinónimos de esa palabra
Introduzca una palabra y le daré los sinónimos: salir
```

19. Amplía el programa anterior de tal forma que sea capaz de aprender palabras y sinónimos. Cuando una palabra no tiene sinónimos, es decir, cuando aparece la palabra en español con su traducción y esa traducción no la tiene ninguna otra palabra española, se le preguntará al usuario si quiere añadir uno (un sinónimo) y, en caso afirmativo, se pedirá la palabra y se añadirá al diccionario. Se puede dar la circunstancia de que el usuario introduzca una palabra en español que no está en el diccionario; en tal caso, se mostrará el consiguiente mensaje y se dará la posibilidad al usuario de añadir la entrada correspondiente en el diccionario pidiendo, claro está, la palabra en inglés.

Ejemplo:

```
Introduzca una palabra y le daré los sinónimos: caliente
Sinónimos de caliente: ardiente, abrasador
Introduzca una palabra y le daré los sinónimos: rojo
No conozco sinónimos de esa palabra ¿quiere añadir alguno? (s/n): s
Introduzca un sinónimo de rojo: colorado
Gracias por enseñarme nuevos sinónimos.
Introduzca una palabra y le daré los sinónimos: blanco
No conozco esa palabra ¿quiere añadirla al diccionario? (s/n): s
Introduzca la traducción de blanco en inglés: white
Introduzca una palabra y le daré los sinónimos: rojo
Sinónimos de rojo: colorado
Introduzca una palabra y le daré los sinónimos: blanco
No conozco sinónimos de esa palabra ¿quiere añadir alguno? (s/n): s
Introduzca un sinónimo de blanco: albino
Gracias por enseñarme nuevos sinónimos.
Introduzca una palabra y le daré los sinónimos: blanco
Sinónimos de blanco: albino
Introduzca una palabra y le daré los sinónimos: salir
```

20. La asociación “Amigos de los anfibios” nos ha encargado una aplicación educativa sobre estos animalitos. Crea un programa que pida al usuario el tipo de anfibio y que, a continuación, nos muestre su hábitat y su alimentación. Si el tipo de anfibio introducido no existe, se debe mostrar el mensaje “Ese tipo de anfibio no existe”.

Ejemplo 1:

```
Introduzca el tipo de anfibio: salamandra
Hábitat: Ecosistemas húmedos.
Alimentación: Pequeños crustáceos e insectos.
```

Ejemplo 2:

```
Introduzca el tipo de anfibio: gato
Ese tipo de anfibio no existe.
```

La información se debe guardar en dos diccionarios (dos HashMap). Uno de ellos tendrá parejas clave-valor del tipo **(tipo de anfibio, hábitat)** y otro **(tipo de anfibio, alimentación)**. A continuación se muestra la información en una tabla:

				
Tipo de anfibio	rana	salamandra	sapo	tritón
Hábitat	En los trópicos y cerca de las zonas húmedas y acuáticas	Ecosistemas húmedos.	En cualquier sitio salvo el desierto y la Antártida.	América y África.
Alimentación	Larvas e insectos.	Pequeños crustáceos e insectos.	Insectos, lombrices y pequeños roedores.	Insectos

21. En ajedrez, el valor de las piezas se mide en peones. Una dama vale 9 peones, una torre 5 peones, un alfil 3, un caballo 2 y un peón vale, lógicamente, 1 peón. Realiza un programa que genere al azar las capturas que ha hecho un jugador durante una partida. El número de capturas será un valor aleatorio entre 0 y 15. Hay que tener en cuenta que cada jugador tiene la posibilidad de capturar algunas de las siguientes piezas (no más): 1 dama, 2 torres, 2 alfiles, 2 caballos y 8 peones. Al final debe aparecer la puntuación total.

Ejemplo:

```
Fichas capturadas por el jugador:
Alfil (3 peones)
Caballo (2 peones)
Peón (1 peones)
Torre (5 peones)
Peón (1 peones)
Puntos totales: 12 peones.
```