1) EjerTresSetInt

Programa que genera un número aleatorio de datos que consisten en números al azar con valores entre 1 y 15 y los va insertando:

- En un objeto de clase **`HashSet`**.
- En un objeto de clase **`LinkedHashSet.`**
- En un objeto de clase **`TreeSet`**.

Una vez hecho hace un recorrido de cada objeto y muestra todos sus elementos.

Muestra en pantalla el resultado de invocar al método toString de cada uno de los objetos.

Ejemplo de ejecución:

```
Se van a generar 6 datos.
Insertando:
13 * 6 * 5 * 1 * 5 * 4 *
Valores de objHashSet:
1 4 5 6 13
Valores de objLinkHashSet:
13 6 5 1 4
Valores de objTreeSet:
1 4 5 6 13

Valores de objHashSet: [1, 4, 5, 6, 13]
Valores de objLinkHashSet: [13, 6, 5, 1, 4]
Valores de objTreeSet: [1, 4, 5, 6, 13]
```

2) EjerTresSetString

Hacer lo mismo que en el ejercicio anterior pero con conjuntos de String y asignando los elementos al azar a partir de un vector de Strings.

Ejemplo de ejecución:

```
Se van a generar 8 datos.
Insertando:
este * montaña * letra * bajo * frio * bajo * bajo * aleatorio *

Valores de objHashSet (no hay orden):
este frio aleatorio bajo letra montaña

Valores de objLinkHashSet (por orden de inserción):
este montaña letra bajo frio aleatorio

Valores de objTreeSet (orden alfabético):
aleatorio bajo este frio letra montaña

Valores de objHashSet: [este, frio, aleatorio, bajo, letra, montaña]
Valores de objLinkHashSet: [este, montaña, letra, bajo, frio, aleatorio]
Valores de objTreeSet: [aleatorio, bajo, este, frio, letra, montaña]
```

3) EjerDosListInt

Programa que genera un número aleatorio de datos que consisten en números al azar con valores entre 1 y 15 y los va insertando:

- En un objeto de clase **`ArrayList`**.
- En un objeto de clase **`LinkedList.`**

Una vez hecho hace un recorrido de cada objeto y muestra todos sus elementos.

Muestra en pantalla el resultado de invocar al método toString de cada uno de los objetos.

Ejemplo de ejecución:

```
Se van a generar 14 datos.
Insertando:
15 * 4 * 9 * 1 * 6 * 12 * 13 * 15 * 1 * 11 * 5 * 8 * 12 * 8 *

Valores de objArrayList:
15 4 9 1 6 12 13 15 1 11 5 8 12 8

Valores de objLinkedList:
15 4 9 1 6 12 13 15 1 11 5 8 12 8

Valores de objArrayList: [15, 4, 9, 1, 6, 12, 13, 15, 1, 11, 5, 8, 12, 8]
Valores de objLinkedList: [15, 4, 9, 1, 6, 12, 13, 15, 1, 11, 5, 8, 12, 8]
```

4) EjerDosListString

Hacer lo mismo que en el ejercicio anterior pero con conjuntos de String y asignando los elementos al azar a partir de un vector de Strings.

Ejemplo de ejecución:

```
Se van a generar 3 datos.
Insertando:
este * este * bajo *

Valores de objArrayList:
este este bajo

Valores de objLinkedList:
este este bajo

Valores de objArrayList: [este, este, bajo]
Valores de objLinkedList: [este, este, bajo]
```

5) ListarNumerosEnOrden

Programa que pide números al usuario hasta que teclea el -9999. Una vez dado ese valor escribe en orden creciente todos los elementos distintos de la secuencia de números introducida por el usuario.

Para almacenar los números que diga el usuario se creará un objeto TreeSet<Integer>.

6) ListarNumPalabrasOrdenadas

Escribe un programa en Java que:

- Pida al usuario que introduzca un número indeterminado de palabras. La introducción de datos terminará cuando introduzca un "*" (que por supuesto no se tendrá en cuenta para hacer lo que se pide en el ejercicio).
- Pida que se diga un número entero positivo num que no puede ser mayor que el número de palabras distintas que se han dado.
- Muestre las num primeras palabras en orden alfabético de la lista de palabras que ha dado el usuario.

7) TreemapNotasAlumnosPorApellido

Escribe un programa en Java que:

- Pida el apellido (se supone que no se repiten) y la calificación obtenida por un alumno. La entrada de datos terminará cuando se introduzca como apellido "*".
- Muestre un listado por orden alfabético de los datos de cada alumno.

8) TreemapNotasAlumnosPorNota

Modifica el programa del ejercicio anterior para que podamos mostrar un listado ordenado por notas, teniendo en cuenta que puede haber repeticiones de notas.

Ejercicio 1 Java – Interface Map – HashMap, TreeMap.

Implementa una clase **Colegio** que almacene las nacionalidades de los alumnos de un colegio. La clase contendrá los siguientes métodos:

```
public void addAlumno(String k)
```

Añade la nacionalidad (Key) a la colección si no existe y si existe incrementará el contador (valor asociado).

```
public void showAll()
```

Muestra en un listado las distintas nacionalidades y el número de alumnos que existen por cada nacionalidad.

pista:
for(String key: nacionalidades.keySet())
System.out.println(key + " – " + nacionalidades.get(key));

```
public void showNacionalidad(String k)
```

Si existe en la colección, muestra la nacionalidad y el número de alumnos de esa nacionalidad.

```
public int cuantos()
```

Muestra cuántas nacionalidades diferentes existen en el colegio.

```
public void borra()
```

Elimina los datos insertados.

Nota: Las nacionalidades se almacenan en **mayúsculas.**

- Implementa un método que instancie una clase Colegio, Seleccione 50 nacionalidades al azar, muestre cuantas nacionalidades hay en total,muestre un listado y borre los datos.

```
final int veces = 50;
String nac[] = {"Francesa","Portuguesa","Espanola","Alemana","Holandesa",
```

```
"Rumana","Danesa","Estadounidense","Colombiana","Peruana","China","Japonesa",
"Inglesa","Italiana","Irlandesa","Ucraniana","Tailandesa","Dominicana"};
```

- Crea también una clase **exceptionVacio** del tipo **Exception** que será lanzada en el caso de que se llame al método **addAlumno**(String nacionalidad) con un parámetro nacionalidad vacío.

Ejercicio 2 Java – Interface List – ArrayList, LinkedList.

Implementa una lista que contenga los días de la semana.

Pista:

```
List listDias = new ArrayList();
```

Inserta en la posición 4 el elemento «Juernes».

Copia esa lista a otra llamada **listaDos**.

Pista:

```
List listaDos = new ArrayList<>(listaUno);
```

Añade a **listDias** el contenido de **listaDos**.

Muestra el contenido de las posiciones 3 y 4 de la lista original.

Muestra el primer elemento y el último de la lista original.

Pista: getFirst() y getLast()

Elimina el elemento que contenga «Juernes» de la lista y comprueba si elimina algo o no.

Pista:

```
if (listaDias.remove(«Juernes»)) {
System.out.println(«Borrado»);
} else {
System.out.println(«No existe»);
}
```

Crea un **iterador** y muestra uno a uno los valores de la lista original.

Busca si existe en la lista un elemento que se denomine «Lunes».
Pista: listaDias.contains(«Lunes»)

Busca si existe en la lista un elemento que se denomine «Lunes». No importa si está en mayúscula o minúscula.

Ordena la lista y muestra su contenido.
Pista: método sort().

Borra todos los elementos de la lista.
Pista: clear()

Ejercicio 3 Java - Interface Set – HashSet, LinkedHashSet.

Crea un conjunto al que se le va a llamar jugadores. Inserta en el conjunto los jugadores del FC Barcelona.

Realiza un bucle sobre los jugadores del conjunto y muestra sus nombres.
Pista:

```
for (String nombre : jugadores) {
System.out.println(nombre);
}
```

Consulta si en el conjunto existe el jugador «Neymar JR». Avisa si existe o no.

Crea un segundo conjunto jugadores2 con los jugadores «Piqué» y «Busquets».

Consulta si todos los elementos de jugadores2 existen en jugadores.

Realiza una unión de los conjuntos jugadores y jugadores2.

Elimina todos los jugadores del conjunto jugadores2 y muestra el número de jugadores que tiene el conjunto jugadores2 (debería ahora ser cero).

EJERCICIO ORDENACIÓN

Utilizando **Comparable** y **compareTo** resuelve el siguiente problema donde debemos partir de una clase Persona con atributos nombre, edad y altura. Queremos ordenar por edad y por altura a las siguientes personas:

Nombre	Altura	Edad
Mario	187	22
Pepe	173	52
Manuel	158	27
David	164	25
Alberto	184	80

Debemos comparar las personas y ordenarlas por **altura** primero (de mayor a menor) y por **edad** (de menor a mayor) después. Por pantalla debe mostrarse la lista de personas sin ordenar, ordenada por altura y ordenada por edad

Ejercicios sobre Collection y Map

Ejercicio 1

Implementar el tipo abstracto de datos `Pila` de enteros (se podrán almacenar un número indeterminado de números enteros y a la hora de extraer información el último elemento que se haya añadido será el primero en salir) con las operaciones de:

- Apilar – pone un nuevo elemento
- Desapilar -- devuelve el valor de la cima y lo elimina de la colección
- Cima -- nos devuelve el valor que tiene la cima
- Elementos -- Número de elementos apilados
- Visualizar – muestra los elementos de la pila.

Ejercicio 2

Este ejercicio pretende desarrollar el programa de cálculo de escaños en unas elecciones, teniendo como datos de entrada: El número de partidos que se presentan (N), el número de escaños a repartir (M) y los votos que ha obtenido cada partido

A partir de aquí, hay que hacer lo siguiente se dividen los votos de cada partido entre 1, 2, 3....M y se obtienen M divisiones y así sucesivamente para el partido 2,3,...N es decir se obtendrán NxM cocientes, y de esos cocientes se eligen los M mayores, puesto que hay M diputados a repartir

Un ejemplo sería el siguiente:

PARTIDOS: 4
ESCAÑOS: 5

	Escaños				
Partidos	1	2	3	4	5
Part1	**10000**	**5000**	**3333**	**2500**	**2000**
Part2	**25000**	**12500**	**8333**	**6250**	**5000**
Part3	**30000**	**15000**	**10000**	**7500**	**6000**
Part4	**12000**	**6000**	**4000**	**3000**	**2400**

Una vez hechos los cálculos habrá que ver a qué partidos le corresponden los **5 mayores cocientes.**

El listado de resultados debe hacerse ordenado por número de votos.

Los partidos que no obtengan un 5% de los votos emitidos deben rechazarse, y no entrarán en el reparto de escaños.

NOTA: Obligatorio el uso de colecciones para resolverlo, se puede usar un TreeMap con clave el cociente y valor asociado un TreeSet que en cada elemento guarda el nombre del partido a quien pertenece ese cociente. El mapa debe estar ordenado por los valores de los cocientes de mayor a menor. Si hay M escaños los M primeros elementos de los TreeSet que se guardan en el mapa nos indican a quien corresponden los escaños. En este ejercicio nos interesa recorrer los primeros nodos del mapa y llevar la cuenta del número de elementos recorridos de los TreeSet que van asociados a cada nodo porque cuando tengamos 5 en total se termina.

Se pueden utilizar otros enfoques con colecciones para resolverlo.

Ejercicio 3.

En este ejercicio se generan los datos a partir de dos vectores de 10 nombres y 10 apellidos combinándolos al azar. Hay que generar un ArrayList de 50 alumnos para guardar el nombre, el apellido y las notas obtenidas en tres exámenes parciales que se puntúan entre 0 y 100.

El programa debe:

1. Generar al azar los datos de **50 alumnos**. Cada alumno se almacenará en un elemento del **ArrayList**. Los datos que queremos guardar de cada alumno son:

   String nombre;
   String apellido;
   int nota1;
   int nota2;
   int nota3;
2. Después generar tres mapas (TreeMap), uno para cada nota, donde la key será la nota por la que ordenamos y el dato asociado un ArrayList con los índices del vector donde aparece esa nota (puede haber varios alumnos que tengan la misma nota).
3. Presentar un menú para poder listar la clase por cualquiera de las tres notas en orden ascendente, es decir ascen1ª, ascen2ª, ascen3ª.
4. Presentar en pantalla el listado requerido

De esta forma no hay que reordenar el ArrayList cada vez que se pide un tipo de listado pues el ArrayList, aunque este desordenado, es accesible en el orden de las notas según el mapa que se recorra.

Ejercicio 4.

Deseamos realizar una aplicación para la gestión de usuarios y claves de acceso a un sistema (red, base de datos, programa de gestión etc.)

Para ello vamos a utilizar mapas (TreeMap) donde el par de datos será:

- <usuario, clave de acceso>, los usuarios no pueden repetirse (TreeMap).
- Además para mayor seguridad las claves no se guardan tal y como la tecleamos sino que se guardan codificadas con un sencillo algoritmo que consiste en desplazar los caracteres un número fijo entero, p. ej. si la clave es ABC y el número fijo es 2 la clave será ABC pero en el mapa guardamos CDE, de forma que si nos pillan el fichero de claves, dicho fichero no sería el real sino el codificado.
- Las claves deben ser de tal forma que solo admitan mayúsculas, minúsculas y números y ningún otro carácter debe ser admitido.

Crear la aplicación para que se pueda:

- ✔ Insertar usuario, clave
- ✔ Borrar usuarios
- ✔ Modificar claves de un usuario
- ✔ Validar a un usuario si su clave es correcta

Ejercicio 5.

Simulación de colas de espera en una oficina de atención a clientes.

En una oficina de atención a clientes se disponen de 5 ventanillas para los mismos, cada una de ellas genera una cola de espera.

Cada nodo de información que se inserta en la cola tiene una tarea a despachar que puede ser de duración 10 min, 20 min ó 30 min , se ha estimado estadísticamente que llegan un 60% de tareas de duración 10 un 25% de tareas de duración 20 y un 15% de tareas de duración 30.

El programa debe generar con los porcentajes descritos un nodo de tipo 15, 20 o 30. Se sugiere generar números aleatorios entre 1 y 100 de manera que si el número generado está entre 1 y 60 la tarea será de duración 15, si está entre 61 y 85 de duración 20 y si está entre 86 y 100 de duración 30.

Cuando se genera un cliente(es decir nodo ó tarea), este se coloca en la cola cuya longitud es mínima.

Se ha estimado estadísticamente también que cada 5 minutos llega un nuevo cliente (nodo/tarea).

El programa debe simular el comportamiento de las 5 colas durante 5 horas es decir se genera un bucle de 1 a 300 porque son trescientos minutos los que transcurren en ese intervalo. Para cada minuto, es decir para cada paso del bucle hay que hacer dos cosas:

1. Si el minuto es múltiplo de 5 generar un cliente (tarea/nodo) y ponerlo en su cola correspondiente.
2. Dar servicio a las 5 ventanillas esto es las tareas que estén en el frente de cada cola descontarles un minuto que ha transcurrido y si quedan en cero suprimirlas de la cola.

El programa debe darnos las longitudes de cada cola de espera (ventanilla) cada 15 minutos. Y al final la longitud media de cada cola, obtenidas como la suma de las longitudes de dicha cola en cada instante dividida por el número de instantes totales

Los métodos que se proponen para este ejercicio de simulación son las siguientes:

ServirColas : su tarea es dar servicio a las 5 colas decrementando en uno los minutos de su gestión , a las cabeceras de dichas colas y en el caso de que los minutos de ese nodo queden a cero suprimirlos de la cola.

MedirCola : su función es darnos el número de nodos de la cola en un momento dado.

AsignarCola : su función es indicar donde debe insertarse la tarea que se genera cada 5 minutos.

InsertarTarea : insertar una tarea en la cola que le corresponda.

ExtraerTarea : eliminar una tarea cuyo número de minutos haya terminado.

Estos métodos serán indispensables para la buena ejecución del programa, pero si se considera oportuno, se pueden crear otros métodos auxiliares.

1. **Escribe un proyecto en Java que tenga:**

Una clase llamada ***Comedor*** en la que estará el método public static void main() y en la que se instanciarán los objetos de las clases que sean necesarias para:

- El restaurante elabora una serie de **platos** que se componen de los productos que tiene en su lista de ingredientes. Cada plato tiene como precio el de la suma de las porciones de **ingredientes** utilizados más un fijo que es de 1 euro (gasto de luz, utensilios, tiempo,…)
- Mostrar la **carta** de cada día de un restaurante que se compondrá de 6 platos de la lista de platos que elabora el restaurante. En cada plato se mostrarán los ingredientes que lo componen y el precio de ese plato.
- Cada **menú** consta de dos platos que es lo que cada cliente solicitará más un fijo de pan, bebida y servicio. Al postre invita la casa.
- A cada **cliente** que vaya al restaurante se le anotarán los platos que han formado su menú y la **mesa** en que está. A la hora de pagar se mostrará **la cuenta** en pantalla: el detalle de los menús que tiene que pagar, indicando el precio de cada plato, el total a pagar, el dinero entregado y el cambio.

2. **Escribe un proyecto en Java que permita gestionar la venta de billetes de tren.**

Crea las clases que se necesiten, los métodos adecuados en cada clase y las colecciones que sean más adecuadas. (crea tres trenes de forma fija)

- Cada **tren** se identifica con un número de tren, una hora de salida y las estaciones de salida y llegada. El número de tren es único.
- El tren para una fecha se compondrá de vagones. Cada **vagón** tiene una capacidad de 10 **asientos**, numerados del 1 al 10. Los vagones se numeran secuencialmente. Al tren se le van añadiendo vagones según se van llenando. (cada vagón se puede guardar en un array de 11 elementos en el que la posición 0 se guarda el número de vagón y de las posiciones 1 a la 10 un 0 si están libres ó un 1 si están ocupados. También se puede guardar el número del billete en el que se ha vendido)
- La venta de **billetes** se realiza para una fecha (utiliza la **clase Fecha**) en un tren concreto, un vagón y un asiento. (atributos de la clase billete: número, fecha, tren, vagón, asiento y precio) Cada billete lleva su precio y su número identificador.
- Los billetes se pueden devolver (borrar el billete), lo que dejará el asiento correspondiente libre que serán los primeros en ser ofertados en la siguiente venta para la fecha correspondiente. Los billetes se guardan en una list o set.
- Los billetes podrán ser ordenada por diferentes criterios:
    - **número de billete**: datos de billetes (de billetes)
    - **fecha, tren, vagón y asiento**: (este puede salir del mapa con **clave** la fecha y **valor** que puede ser el billete).
    - **tren, fecha, vagón** : asientos vendidos (de billetes)
- El menú del programa tiene venta de billetes, anulación de billetes y listados por las tres opciones anteriores.