Master Degree on

Advanced Analytics on **Big Data**

MÓDULO INICIAL. PROGRAMACIÓN ORIENTADA A OBJETOS CON JAVA

Relación de Problemas Nº 4

Módulo mdBusV2 (mdBusV1L, set, orden)(*mandatory*)

Modificar el módulo `mdBusV1` (hacer una copia de sus ficheros) de manera que los datos del servicio se almacenen en un conjunto en lugar de en una lista.

Crear un orden natural para la clase `Bus` de manera que un bus sea menor que otro si lo es su línea y a igualdad de línea, si lo es su código de bus.

Modificar la clase Servicio de la siguiente manera:

- Modificar el método `filtra` para que:
  - devuelva un conjunto de buses en lugar de una lista de buses.
  - Tenga un segundo argumento que sea `Comparator<Bus>`.

  Con esto, el conjunto que devuelve este método estará ordenado por el criterio dado.

- Modificar el método `guarda(String file, Criterio c)` para que tenga un parámetro nuevo. Ahora será el método `guarda(String file, Comparator<Bus> cb, Criterio c)` y utilizará ese comparador para ordenar los buses. Hacer un cambio parecido en el método `guarda(PrintWriter pw, Criterio c)`.

Usar la aplicación `MainPrueba` para probar la aplicación

.

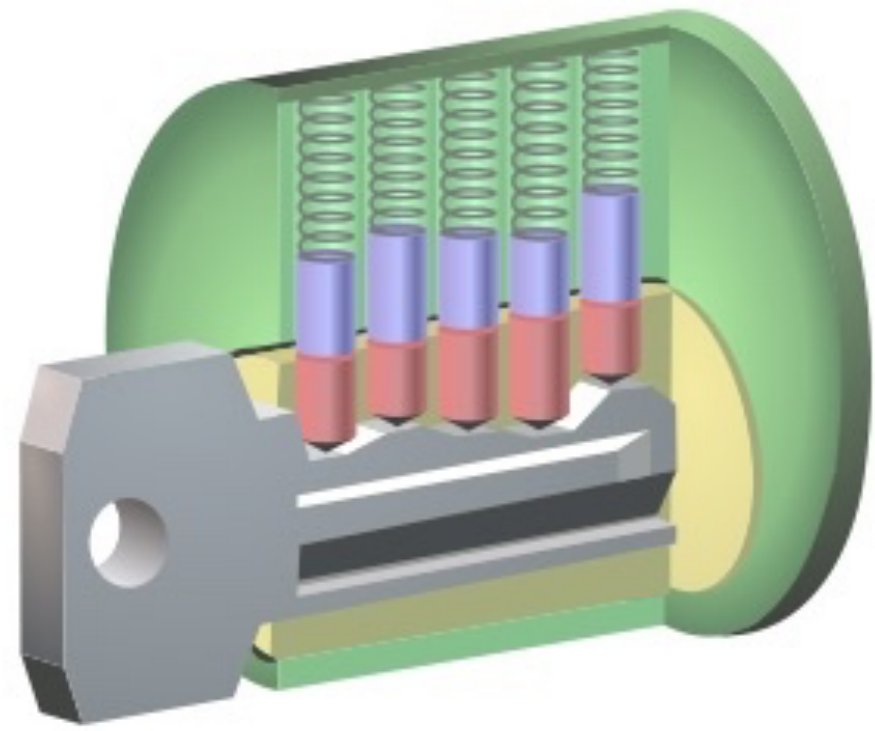
Módulo mdLlaves (colecciones (list y set), excepción)

Una llave está formada por un número determinado de dientes, cada uno de una altura (una llave se representará como una lista de enteros). Inicialmente las llaves tienen sus dientes sin limar a una altura de 10 milímetros, obteniéndose el perfil deseado limando cada uno de estos dientes una altura n_i (entre 0 y 10), de forma que la altura final de cada uno de estos dientes sea $10 - n_i$.

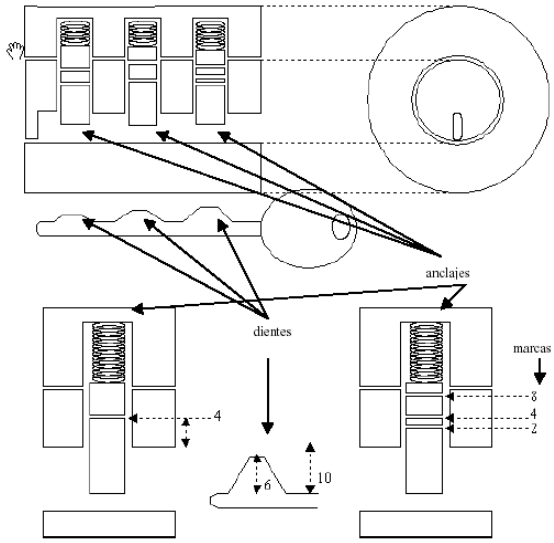
Una cerradura tiene un bombillo con un número determinado de "anclajes". Al introducir una llave en el bombillo de una cerradura cada diente de la llave se corresponderá con un anclaje del bombillo. Cada anclaje puede llevar hasta 4 cortes, a los cuales llamaremos "marcas". Cada marca estará a una altura de entre 0 y 10 milímetros, comenzando desde la base del anclaje. (Una cerradura se representará como una lista de conjuntos de enteros, el conjunto de marcas de cada bombillo.)

Una llave abre una cerradura si el número de dientes de la llave coincide con el número de anclajes de la cerradura y, además, cada diente empuja el émbolo del anclaje correspondiente de manera que nivela una de las marcas con la zona de giro. En definitiva, si existe una marca cuyo valor sumado con el del diente correspondiente sea 10, entonces ese anclaje permite la apertura. Si todos los anclajes permiten la apertura, el bombillo gira y la cerradura se abre.

En la siguiente figura vemos, en la parte superior, el frente y el perfil de un bombillo de tres anclajes, donde se puede observar la altura de la zona de giro y las marcas de cada uno de los anclajes. Vemos también cómo el primer anclaje (el situado más a la izquierda del dibujo) tiene dos marcas, el segundo otras dos y el tercero tres. En el detalle de la parte inferior derecha de la figura se pueden ver las alturas a que se encuentran las marcas de un émbolo. En la parte inferior izquierda observamos cómo un diente de una altura 6 elevaría lo suficiente un émbolo con una marca a una altura 4 para hacer coincidir dicha marca con la zona de giro.



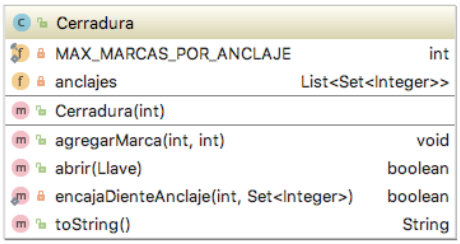
bombillo (visto de perfil)

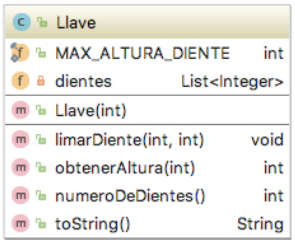


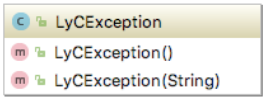
bombillo (visto desde el frente)

Se pide crear las clases siguientes dentro del paquete `llaves`:

- La excepción no comprobada `LyCException` que asegura que tanto los dientes como los anclajes usados sean correctos.
- Las clases `Llave` y `Cerradura` con los métodos especificados en el diagrama de clases mostrado más abajo.
- Una aplicación que pruebe si una cerradura se abre con varias llaves dadas.







Módulo mdNotas. (colecciones (list y set), excepciones, io)

Se va a crear una aplicación para anotar las calificaciones obtenidas por alumnos en una asignatura. Para ello se crearán las clases `Alumno`, `Asignatura`, y `AlumnoException`.

- Crea la excepción comprobada `AlumnoException` para manejar situaciones excepcionales que podrán producirse en las siguientes clases.

- Crea la clase `Alumno` que mantiene información de un alumno del cual se conocen el dni (`String`), el nombre (`String`) y la calificación obtenida en una asignatura (`double`). La clase tendrá dos constructores. En el primer constructor se proporcionan el dni, el nombre y la calificación. En el otro constructor se proporcionan sólo el dni y el nombre, siendo en este caso la calificación igual a cero. Si la calificación dada es negativa se deberá lanzar una excepción `AlumnoException`.
  Dos alumnos son iguales si coinciden sus nombres y sus dni. La letra del dni podrá estar indistintamente en mayúsculas o minúsculas.
  Crear también métodos para conocer el nombre, (`String getNombre()`), el dni (`String getDni()`) y la calificación (`double getCalificacion()`).
  La representación de un alumno debe mostrar el nombre y el dni pero no la calificación.
- Crea una aplicación (clase distinguida `PruebaAlumno`) para probar la clase anterior. En esta clase se crean dos alumnos con los datos siguientes:

  DNI: 22456784F Nombre: Gonzalez Perez, Juan Nota: 5.5

  DNI: 33456777S Nombre: Gonzalez Perez, Juan Nota: 3.4

  Y se muestra por pantalla el nombre de cada alumno, así como sus calificaciones. Además, se comprueba si ambos alumnos son iguales, indicándolo por pantalla. Ten en cuenta que la Excepción `AlumnoException` es de obligado tratamiento a la hora de implementar `PruebaAlumno`. Ejecuta el programa.

  A continuación, modifica los datos del segundo alumno tal y como se indica abajo y ejecuta de nuevo el programa.

  ```
  DNI: 33456777S Nombre: Gonzalez Perez, Juan Nota: -3.4
  ```

  Observa lo que sucede.

- Crea la clase `Asignatura`. Una asignatura mantiene el nombre de esta y dos listas, una contendrá alumnos (se llamará `alumnos`) y otra `String` (se llamará `errores`). Al constructor solo se le pasa el nombre de la asignatura y deberá inicializar las listas.
- El método `void leeDatos(String[])` lee los datos de los alumnos de un array de `String`. Cada elemento del array contendrá toda la información para crear un alumno con el siguiente formato (deben aparecer siempre los tres tokens separados por `;`)

  ```
  <Dni>;<Apellidos, nombre>;<Calificación>
  ```

  Por ejemplo:

```
55343442L;Godoy Molina, Marina;6.3
```
[1]

Para cada elemento en el array el método deberá crear, si es posible, el alumno con el nombre, dni y calificación dadas y almacenarlo en una lista de alumnos. Si no fuera posible crear un alumno, el constructor deberá almacenar esa entrada en otra lista de `String` (llamada `errores`) precedido de un comentario que indique cual ha sido el problema por el que no se ha podido crear el alumno. Por ejemplo, ante la entrada:

```
342424f2J;Fernandez Vara, Pedro;tr
```

Se incluirá en `errores` el siguiente `String`:

```
ERROR. Nota no numérica: 342424f2J;Fernandez Vara, Pedro;tr
```

También incluirá otro método `void leeDatos(String file) throws IOException` que lee los datos desde un fichero pasado como argumento. (ver `notas.txt`).

Otros métodos de la clase `Asignatura` son:

```
double getCalificacion(Alumno al) throws AlumnoException
```

que devuelve la calificación del alumno `al` dado si es que existe. Si no existe se lanzará una excepción `AlumnoException`.

También tendrá dos métodos, uno que devuelve la lista de alumnos (`List<Alumno> getAlumnos()`) y otro que devuelve la lista de entradas malas (`List<String> getErrores()`).

El método `Set<String> getNombreAlumnos()` devolverá un conjunto con los nombres de todos los alumnos.

Además, dispondrá de una representación de los objetos de la clase como la que se muestra en el ejemplo del final de este enunciado.

Por último, el siguiente método devuelve la media de las calificaciones de los alumnos de la asignatura. En caso de que no haya alumnos, lanzará una excepción `AlumnoException`.

```
double getMedia(CalculoMedia media)) throws AlumnoException;
```

de tal forma que este método calculará la nota media de los alumnos invocando al método `calcular` proporcionado por la clase recibida como parámetro, que implementa la interfaz `CalculoMedia`. Por lo tanto, será necesario definir la **interfaz** `CalculoMedia` que especifique el siguiente método:

```
double calcular(List<Alumno> alumnos) throws AlumnoException;
```

Además, se deberán definir las clases `MediaAritmetica`, `MediaArmonica` y `MediaSinExtremos` que implementen la interfaz `CalculoMedia`, según las siguientes especificaciones:

[1] NOTA para Scanner. Para poder leer números decimales con el separador punto (ej. 7.1), se debe usar un objeto `Scanner sc` al que se le envía el mensaje `sc.useLocale(Locale.ENGLISH)`.

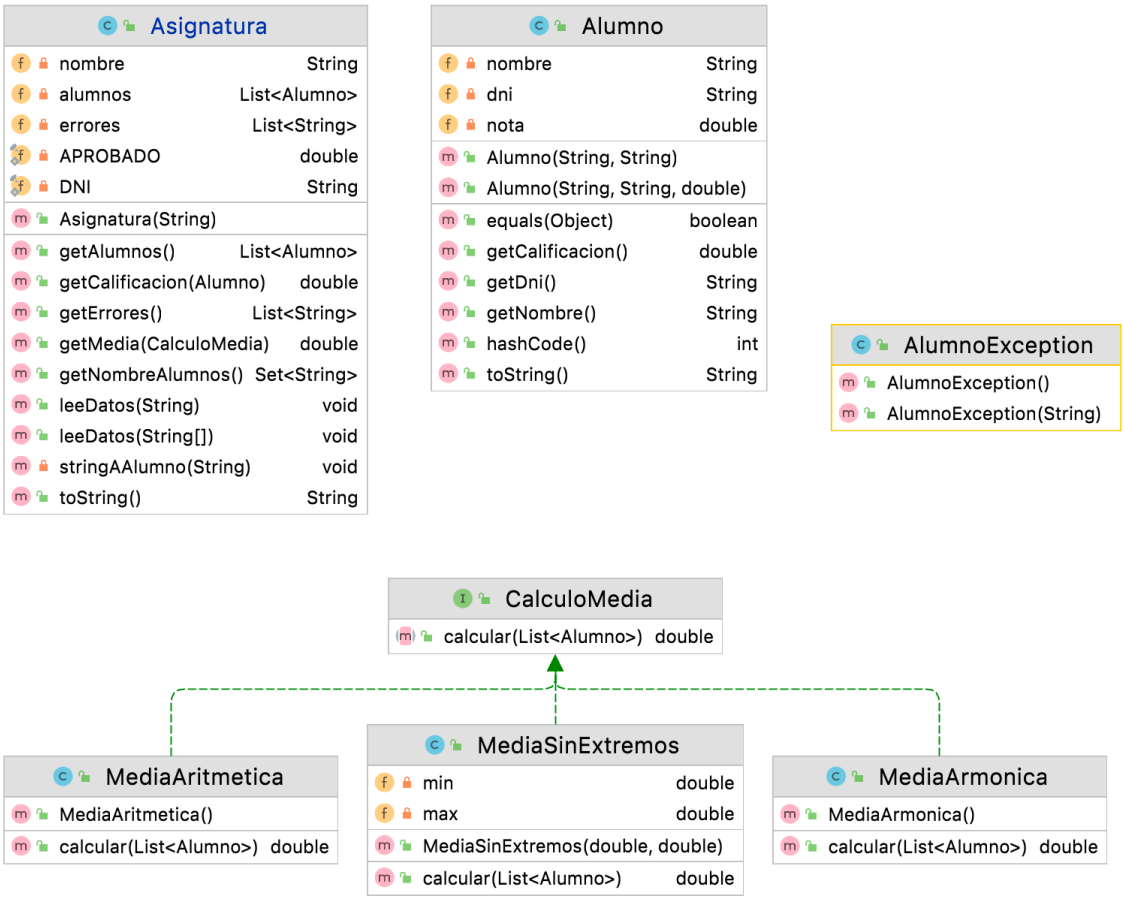
- El método `calcular` proporcionado por la clase `MediaAritmetica` calcula la media aritmética de *n* alumnos según la siguiente ecuación. En caso de que no haya alumnos lanzará una excepción `AlumnoException`:

$$media = \frac{1}{n}\sum_{i=0}^{n-1} calificacion\ Alumno_i$$

- El método `calcular` proporcionado por la clase `MediaArmonica` calcula la media armónica de los *k* alumnos con notas superiores a *0* según la siguiente ecuación. En caso de que no haya alumnos que cumplan el requisito especificado, lanzará una excepción `AlumnoException`:

$$media = \frac{k}{\sum_{j=0}^{k-1} \frac{1}{calificacion\ Alumno_j}}$$

- El método `calcular` proporcionado por la clase `MediaSinExtremos` calcula la media aritmética de aquellos valores comprendidos entre los extremos dados, ellos incluidos. En caso de que no haya alumnos que cumplan el requisito especificado, lanzará una excepción `AlumnoException`. Los valores extremos se pasarán en el constructor de la clase y serán almacenados en sendas variables de instancia, para ser utilizados en el método `calcular`.



- Crea una aplicación (clase distinguida `PruebaAsignatura`) para probar la clase `Asignatura`. En esta clase se crea la asignatura POO con tres alumnos con los siguientes datos:

DNI: 12455666F Nombre: Lopez Perez, Pedro Nota: 6.7
DNI: 33678999D Nombre: Merlo Gomez, Isabel Nota: 5.8
DNI: 23555875G Nombre: Martinez Herrera, Lucia Nota: 9.1

A continuación, muestra la media aritmética de las calificaciones de la asignatura y obtendrá los alumnos de la asignatura, mostrando por pantalla el DNI de cada uno de ellos. Por último, imprime la calificación del alumno Lopez Perez, Pedro. De nuevo ten en cuenta que la excepción `AlumnoException` es de obligado tratamiento a la hora de implementar `PruebaAsignatura`.

Después sustituye los datos del alumno cuya calificación se ha de imprimir por Lopez Lopez, Pedro. Ejecuta de nuevo el programa y observa lo que sucede.

Por último, ejecuta la aplicación Main que se proporciona. El resultado debe ser el siguiente:

Calificacion de Lopez Turo, Manuel 23322443k: 4.3
No existe el alumno Fernandez Vara, Pedro 342424f2J
Media aritmetica 6.199999999999999
Media armonica 5.83482277207447
Media de valores en [5.0,9.0] 6.833333333333333
Alumnos...
Garcia Gomez, Juan 25653443S: 8.1
Lopez Turo, Manuel 23322443K: 4.3
Merlo Martinez, Juana 24433522M: 5.3
Lopez Gama, Luisa 42424312G: 7.1
Malos...
Calificacion negativa: 53553421D;Santana Medina, Petra;-7.1
Faltan datos: 55343442L,Godoy Molina, Marina;6.3
Nota no numérica: 342424f2J;Fernandez Vara, Pedro;2.k
La asignatura completa
Algebra[Garcia Gomez, Juan 25653443S, Lopez Turo, Manuel 23322443K, Merlo Martinez, Juana 24433522M, Lopez Gama, Luisa 42424312G][Calificacion negativa: 53553421D;Santana Medina, Petra;-7.1, Faltan datos: 55343442L,Godoy Molina, Marina;6.3, Nota no numérica: 342424f2J;Fernandez Vara, Pedro;2.k]
Nombre de los alumnos
Lopez Turo, Manuel
Lopez Gama, Luisa
Merlo Martinez, Juana
Garcia Gomez, Juan

Módulo mdAmigoInvisible. (colecciones (list, set), I/O, excepciones, orden)(*advanced*)

Se pretende desarrollar una aplicación en Java que asigne de forma aleatoria "amigos invisibles" entre las personas de un grupo. Cada persona sólo tendrá un amigo invisible (del que recibirá un regalo) y sólo podrá ser el amigo invisible de una persona (a la que hará un regalo). Al final del proceso, todas las personas tendrán asignado un amigo o amiga invisible, y serán a su vez amigo o amiga invisible de alguna otra persona. Obviamente, una persona no puede ser amigo invisible de sí misma. Para ello, se deberá proceder siguiendo las indicaciones y resolviendo los problemas que se plantean en los siguientes apartados.

1. Para tratar situaciones excepcionales propias de la gestión del club (por ejemplo, no hay solución a la asignación de amigos invisibles cuando el número de personas es menor o igual que 2) debe definirse una excepción no comprobada `AmigoException`.

2. Construir la clase `Persona` que represente a una persona determinada por su nombre (una cadena de caracteres) y una referencia a otra persona (`amigo`, que será a su vez una instancia de `Persona`) a la que deberá hacer el regalo. Definir los siguientes constructores y métodos:

a) Constructor con un argumento indicando el nombre de la persona que se quiere crear.
b) Método para asignar una persona (al que habrá que hacer el regalo) a un objeto de la clase (`public void setAmigo(Persona am)`) y métodos para conocer el nombre de la Persona (`public String getNombre()`) y conocer el amigo (`public Persona getAmigo()`).
c) Dos personas son iguales si lo son sus nombres (sin distinguir mayúsculas y minúsculas).
d) La clase Persona debe permitir la comparación entre objetos de esta, de forma natural, atendiendo al orden alfabético del nombre (sin distinguir mayúsculas y minúsculas).
e) Redefinición del método `toString()` para que si una persona llamada `Superman` (el amigo invisible) tiene asignado a `Spiderman` (que recibirá el regalo), su representación sea:

```
Superman --> Spiderman
```

En caso de que alguna persona no tenga aún asignado un amigo al que regalar, su representación será:

```
Superman --> sin amigo
```

3. Para organizar un grupo de amigos se deberá definir una clase `Club`, para almacenar mediante una lista las personas que vayan a participar en el juego del "amigo invisible". Esta clase deberá incluir la siguiente variable de instancia:

`protected List<Persona> socios` que contiene la lista de los socios del club.

a) Constructor que inicializa la estructura.
b) Definir el método

```
public void lee(String  fEntrada, String delim)
                                    throws IOException
```

que lee los datos de los socios del fichero, línea a línea y con cada línea llama al método `private void leeSocios(String linea, String delim)`. Este método leerá los nombres que hay en la línea separados por el delimitador y llamará al método `protected void creaSocioDesdeString(String nombre)` al que se le pasa el nombre de un socio y lo guarda en la lista de socios. El delimitador del fichero será espacio, coma, punto y coma o guion y pueden aparecer una o mas veces (ver `socios.txt`).

c) Definir el método

```
protected void hacerAmigos()
```

que, de forma aleatoria, asigne a cada persona un amigo invisible. Obsérvese que los emparejamientos han de quedar establecidos en los propios objetos de la clase `Persona` en el club. Para ello, generar una lista de enteros (`posAmigos`) con los números de 0 a n-1 donde n es el número de socios. Barajar esta lista (`Collections.shuffle(List<T>)`) hasta que ningún índice quede en su posición (no haya coincidencias). Ahora asignar al socio i-esimo el socio que ocupe la posición `posAmigos.get(i)`.
Por ejemplo, si tenemos 5 personas "Juan", "María", "Pedro", "Luis" y "Ana", construimos la lista con los valores 0,1,2,3,4. Ahora la barajamos hasta que ningún número ocupe su posición. Así, la lista 1,3,2,4,0 no valdría como solución porque hay una coincidencia al estar el 2 en su posición. Supongamos que tras barajar obtenemos la lista 2,4,3,0,1 que no tiene coincidencias y por tanto sí que valdría como solución. Entonces el amigo invisible de "Juan" (posición 0) será "Pedro" (el de la posición 2), el amigo invisible de "María" (1) será "Ana" (el 4), el de "Pedro" (2) será "Luis" (el 3), el de "Luis" (3) será "Juan" (el 0) y el de "Ana" (4) será "María" (el 1).
Para determinar si en una lista de enteros hay coincidencias como las descritas, definir el método `private static boolean hayCoincidencias(List<Integer>)`.

d) Definir los métodos para poder guardar la información de los amigos del club en un fichero y volcarla en un flujo de salida:

```
public void presentaAmigos(String fSalida) throws FileNotFoundException
private void presentaAmigos(PrintWriter pw)
```

La información debe mostrarse como una secuencia de líneas del tipo indicado en el apartado (2.e), y de forma ordenada (`Collections.sort(List<T>)`) atendiendo al orden establecido en la clase `Persona`.

4. Para automatizar la creación del club se va a crear la clase `ClubManager`

a) Definir las siguientes variables de instancia:
`private String fEntrada` nombre del fichero de entrada de datos.
`private String delimitadores` que define los delimitadores del fichero de entrada.
`private String fSalida` nombre del fichero de salida de datos.
`private boolean consola` variable que indica si se presentan los datos por la consola.
`private Club club` que representa al club que se va a manejar.

b) Definir un constructor con un argumento indicando el club que va a manejar
`public ClubManager(Club club)`

c) Definir los siguientes métodos:
`public ClubManager setEntrada (String fEntrada, String delim)` que proporciona nombre del fichero de entrada y el delimitador. Este método deberá devolver el receptor.
`public ClubManager setSalida(String salida)` que proporciona el nombre del fichero de salida. Este método deberá devolver el receptor.
`public ClubManager setConsola(boolean consola)` que indica si se deben mostrar los resultados por la consola. Este método deberá devolver el receptor.

d) Definir el método `private void verify()` que verifica que los datos que contiene el club son correctos, es decir:
Hay un fichero de entrada de datos (no es `null`).
Hay una salida, o bien a fichero o a consola (ambas pueden estar a la vez).
Si alguna de estas condiciones falla, se debe lanzar una `AmigoException` indicando el motivo.

e) Definir el método `public void build()` que realiza las siguientes acciones:
- Verifica que los datos almacenados en las variables de instancia son correctos.

- Le indica al club que lea los datos de la entrada con el delimitador dado.
- Le indica al club que establezca los amigos.
- Presenta los resultados obtenidos en el fichero y/o en la consola.

5. Probar todo con el programa `Main` proporcionado.

Supongamos ahora que se desea contemplar la situación en que en el grupo de amigos existan parejas. En este caso, el problema de asignar amigos invisibles se complica un poco, al no permitirse que un individuo sea el amigo invisible de su pareja.

6. Para prever la situación anterior, se definirá la tupla nombrada `Pareja`, que mantendrá referencias a dos objetos de la clase `Persona`, `uno` y `otro`, de forma que la pareja formada por Romeo y Julieta, por ejemplo, se considerará igual que la compuesta por Julieta y Romeo.

7. Defínase una clase `ClubParejas` con una funcionalidad similar a la de la clase `Club` del ejercicio 3, que permita crear objetos a partir de un fichero de texto, donde, al igual que antes, los individuos se organizan por líneas, y están separados por espacio en blanco, guion, coma o punto y coma. En este caso, sin embargo, se considerará que los individuos separados por un guion constituyen una pareja, y ninguno puede ser amigo invisible del otro. Esta clase deberá tener una variable de instancia privada que contendrá el conjunto de las parejas. La clase deberá redefinir los métodos de crear socio desde `String` y el de hacer amigos. Al método que crea socio desde `String` le puede llegar ahora los nombres de una pareja separadas por guion.

Para comprobar el funcionamiento de la clase `ClubParejas`, utilícese el mismo programa de pruebas cambiando la línea que crea el club manager:

```
ClubManager clubM = new ClubManager(new ClubPareja());
```

y cambiando el delimitador para que no incluya el guion.

Módulo mdUrgencias (Colecciones (set y map), equals, orden, recorridos)

Se trata de diseñar un proyecto Java para simular el proceso del paso de pacientes por un servicio de urgencias en un Centro de Salud. Para ello se pide que implementéis al menos las clases `Ingreso` y `Urgencias`.

1. Defínase una clase `Ingreso` que almacene información sobre los datos relativos a cada paciente tratado en un servicio (por ejemplo, el de urgencias). La información a tener en cuenta es la siguiente: hora de ingreso y hora de alta (ambos de tipo `LocalTime`), identificación de la seguridad social (de tipo `String`), código del médico que trató la urgencia (de tipo `String`), grado de la urgencia (de tipo enumerado `TipoUrgencia` con valores: `LEVE`, `MODERADO` y `GRAVE`. Este tipo enumerado debe crearse como anidada estática pública de `Ingreso`). La clase deberá incluir:
   a) Un constructor que cree ingresos a partir de objetos adecuados:
      `Ingreso(LocalTime,LocalTime,String,String,TipoUrgencia)`
   Métodos de consulta para cada una de las propiedades mencionadas. `getHoraIngreso`, `getHoraAlta`, `getNumSS`, `getCodMedico`, `getTipoUrgencia`.
   b) Un criterio de igualdad que determine que dos objetos de tipo Ingreso son iguales si coinciden en hora de ingreso y número de la seguridad social.
   c) Un orden natural que ordene por hora de ingreso y, en caso de igualdad, por número de la seguridad social.
   d) Una representación textual del tipo:
      hora de ingreso, duración de servicio, número de la SS
      La duración del servicio se calcula como
      `Duration.between(LocalTime, LocalTime)`

2. Defínase una clase `Urgencias` que represente los ingresos en un día determinado de distintas unidades de urgencias de un mismo Centro de Salud. Deberá incluir una colección ordenada de ingresos y además:
   a) Un constructor que cree un objeto de la clase.
   b) Un método `void agregaServicio(String ingreso)` que incorpore los datos del ingreso dado en forma de cadena que se pasan en el argumento. La cadena tiene el formato:

      `9, 15, 9, 30, 123415, MI766, 1`

      que indica que la hora de ingreso es las 9:15, la hora de alta es a las 9:30, el identificador de la Seguridad Social es el 123415, el código del médico que lo atendió es el MI766 y el tipo de urgencia es MODERADO (valor 1). Para crear un objeto `LocalTime` a partir de las horas y los minutos usar el metodo de clase
      `LocalTime LocalTime.of(horas,minutos)`
   c) Un método `void agregaServicios(String [] ingresos)` que incorpore todos los ingresos dados como cadenas en cada línea del array `ingresos`.
   d) Un método `Set<String> medicosDelServicio()` que devuelva el conjunto ordenado de médicos del servicio.

e) Un método `int urgenciasAtendidas(String codMed)` que devuelve el número de servicios (ingresos) atendidos por el médico dado.
f) Un método `void presentaServicio()` que muestre un ingreso por línea en la consola.
g) Un método `Map<String, Set<String>> idSegSocialPorMedico()` que devuelva una aplicación ordenada que asocie a cada código de médico el conjunto ordenado de identificadores de seguridad social de los pacientes que atendió.
h) Un método `Map<String,Integer> numeroDePacientesPorMedico()`, que construye una correspondencia asociando a cada código de médico el número de pacientes que ha atendido.
i) Un método `Set<Ingreso> ingresosPorTiempoDeAtencion()`, que devuelve un conjunto de ingresos ordenado según el tiempo de atención (diferencia entre la hora de alta y la hora de ingreso y en caso de igualdad por el orden natural). NOTA: necesitas crear algún orden alternativo basado en la clase `Duration`. El método de clase `Duration.between(LocalTime, LocalTime)` crea una duración entre dos horas y la clase `Duration` tiene definido un orden natural.
j) Un método `Set<String> medicosConMayorNumeroDePacientes()`que devuelva el conjunto de médicos que atienden al mayor número de pacientes.

La salida del programa `PruebaUrgencias` proporcionado debe ser la siguiente:

```
Ingresos por Tiempo de atención
[09:15,PT-1H-15M,123543, 09:15,PT-20M,124415, 10:15,PT-16M,123455, 10:15,PT-
16M,123465, 09:15,PT-15M,123415, 11:15,PT-15M,123243, 09:17,PT-13M,123724, 09:17,PT-
13M,123734, 09:10,PT-10M,123261, 11:40,PT-10M,223261]
Número de pacientes por médico
{TR454=2, MI766=2, MI765=1, TR325=5}
Médicos con mayor número de pacientes
[TR325]
Presenta el servicio de urgencias
09:10,PT-10M,123261
09:15,PT-15M,123415
09:15,PT-1H-15M,123543
09:15,PT-20M,124415
09:17,PT-13M,123724
09:17,PT-13M,123734
10:15,PT-16M,123455
10:15,PT-16M,123465
11:15,PT-15M,123243
11:40,PT-10M,223261
```

Módulo mdAlturasV2 (mdAlturas, orden)(*mandatory*)

Este ejercicio pretende crear nuevas estructuras en la aplicación principal del módulo `mdAltura` (*Hacer una copia de los ficheros del módulo mdAltura en este módulo*)

Dada una lista de países (la llamamos `listaPaises`) obtenidas a través del método `selecciona` queremos crear diferentes conjuntos ordenados. Añade a `MainMundo` lo necesario para:

a) Crear un conjunto con los países de la `listaPaises` ordenados por altura, de menor a mayor.
b) Crear el conjunto de los países de la `listaPaises` ordenados alfabéticamente.
c) Crear un conjunto de los países de la `listaPaises` ordenados por continente y a igualdad de continente, alfabéticamente.
d) Crear un conjunto de los países de la `listaPaises` ordenados por continente y a igualdad de continente, alfabéticamente en orden inverso.
e) Redefine la clase `Pais` para que tenga un orden natural que ordene los países por altura y a igualdad de altura, los ordena alfabéticamente. Dada la `listaPaises`, ordénarla por el orden natural.
f) Crear un conjunto de los países de la lista ordenados por continente y en caso de igualdad, por el orden natural.

Módulo mdAnagramas (equals, orden, colecciones set y map)

Se desea construir un diccionario de anagramas a partir de una lista de palabras. Se dice que un anagrama de una palabra es otra palabra obtenida mediante una permutación de sus letras. Por ejemplo,

saco es un anagrama de *cosa*
mora es un anagrama de *amor* y de *roma*

Una palabra es un anagrama de otra si tienen la misma signatura, entendiéndose por signatura de una palabra otra palabra resultante de ordenar alfabéticamente las letras de esa palabra. Por ejemplo,

la signatura de *saco* es *acos*
la signatura de *cosa* es *acos*
la signatura de *examen* es *aeemnx*

Un diccionario de anagramas debe hacer corresponder a cada palabra todos sus anagramas.
Crear las clases e interfaces en el paquete `anagramas`.

1) Crear la clase `Signatura` cuyas instancias mantienen información de una palabra y su signatura, para esta clase se deberá:

- poder crear una instancia conocida la palabra;
- disponer de métodos para conocer cada una de las variables de estado que contiene una instancia (`String getSignatura()`, `String getPalabra()`*);*
- definir un método `boolean mismaSignatura(Signatura)` que determine cuándo dos instancias tienen la misma signatura;
- definir todo lo que sea necesario para que los objetos de esta clase puedan ser ordenados por su componente palabra;
- definir el método `String toString()` para que muestre solo la palabra.

2) Crear la clase `DicAnagramas` correspondiente a este diccionario tomando como clave instancias de `Signatura` y como valor conjuntos de instancias `Signatura`, sabiendo que la información que se desea obtener es un listado, ordenado alfabéticamente, con todas las palabras y, por cada una de ellas, todos sus anagramas ordenados.

Por ejemplo, dadas las palabras *cosa*, *lío*, *amor*, *roma*, *olí*, *mora*, *ramo*, *lió* y *saco*, se desea obtener la siguiente información:

```
amor  (mora ramo roma)
cosa  (saco)
lió   ()
lío   (olí)
mora  (amor ramo roma)
olí   (lío)
ramo  (amor mora roma)
roma  (amor mora ramo)
saco  (cosa)
```

La lista de palabras se proporcionará como argumento de la aplicación.

Se deberán proporcionar métodos para:

- Crear la estructura vacía correspondiente al diccionario para almacenar objetos `Signatura`**.**
- Crear el método `void agregaPalabra(String)` que agregue la instancia de signatura correspondiente a esta palabra en el diccionario.
- Crear el métodos `void presentaDiccionario()` para representar la información pedida sobre la consola

Probar con el siguiente programa `TestAnagramas` proporcionado:

3) Además del orden natural definido para las signaturas, constrúyase la clase `SatSignatura` que implemente la interfaz `Comparator<Signatura>` que proporcione un orden alternativo basado en la longitud de las palabras y, en caso de igualdad, en el orden ascendente alfabético, con independencia de su tipografía.

Definir un nuevo constructor de la clase `DicAnagramas` que tome como argumento un `Comparator<Signatura>`.

Probar con el mismo programa quitando ahora los comentarios.

La salida deberá ser:

```
Ordenadas alfabeticamente
amor [mora, ramo, roma]
cosa [saco]
lió  []
lío  [olí]
mora [amor, ramo, roma]
olí  [lío]
ramo [amor, mora, roma]
roma [amor, mora, ramo]
saco [cosa]
Ahora ordenadas por longitud de la palabra
lió  []
lío  [olí]
olí  [lío]
amor [mora, ramo, roma]
cosa [saco]
mora [amor, ramo, roma]
ramo [amor, mora, roma]
roma [amor, mora, ramo]
saco [cosa]
```

Módulo mdIndicePalabrasv1 (Colecciones set y map, herencia, Scanner, orden)(*mandatory*)

Se pretende realizar una aplicación que permita clasificar las palabras significativas (con la intención de descartar artículos, preposiciones, etc. que consideremos no importantes) que aparecen en un texto de manera que podamos conocer para cada palabra la línea o líneas en las que aparece y su posición (o posiciones) dentro de cada línea. De hecho, vamos a construir tres tipos distintos de índices (todas en el paquete `indices`):

- `Indice1aLinea`, que indicará la primera línea en que aparece cada palabra significativa,
- `IndiceLineas`, que indicará todas las líneas en que aparece cada palabra significativa, y
- `IndicePosicionesEnLineas`, que indicará las líneas en que aparece cada palabra significativa y las posiciones dentro de cada línea.

Por ejemplo, para el texto (donde suponemos que no hay retorno de carro entre "ha" y "pegado", ni entre "la" y "porra", es decir, solo hay tres líneas)

Guerra tenía una jarra y Parra tenía una perra, pero la perra de Parra rompió la jarra de Guerra.
Guerra pegó con la porra a la perra de Parra ¡Oiga usted buen hombre de Parra! Por qué ha pegado con la porra a la perra de Parra.
Porque si la perra de Parra no hubiera roto la jarra de Guerra, Guerra no hubiera pegado con la porra a la perra de Parra.

con `Indice1aLinea` obtendríamos la siguiente salida, donde se muestra cada palabra *significativa* seguida de la primera línea en la que aparece:

guerra	**1**
hombre	**2**
jarra	**1**
oiga	**2**
parra	**1**
pegado	**2**
pegó	**2**
perra	**1**
porra	**2**
rompió	**1**
roto	**3**
usted	**2**

Con `IndiceLineas` obtendríamos la siguiente salida, donde se muestra las palabras significativas junto con las líneas donde aparecen:

guerra	**1.2.3.**
hombre	**2.**
jarra	**1.3.**
oiga	**2.**
parra	**1.2.3.**
pegado	**2.3.**
pegó	**2.**
perra	**1.2.3.**
porra	**2.3.**
rompió	**1.**
roto	**3.**
usted	**2.**

Por último, con `IndicePosicionesEnLineas` obtendríamos un índice en el que para cada palabra significativa se muestra las líneas en que aparece y las posiciones de la palabra dentro de la línea:

guerra
1 1.19.
2 1.
3 13.14.
hombre
2 14.
jarra
1 4.17.
3 11.
oiga
2 11.
parra
1 6.14.
2 10.16.28.
3 6.25.
pegado
2 20.
3 17.
pegó
2 2.
perra
1 9.12.
2 8.26.
3 4.23.
porra
2 5.23.
3 20.
rompió
1 15.
roto
3 9.
usted
2 12.

Para poder hacer esto proporcionaremos una cadena de caracteres con los símbolos delimitadores, que separan palabras en una línea, y una lista de palabras no significativas (lista de cadenas de caracteres). En los ejemplos anteriores, la cadena de delimitadores sería **`"[ .,:;-[!][¡][?][¿]]+"`** y la lista de palabras no significativas `["A", "buen", "con", "de", "ha", "hubiera", "la", "NO", "pero", "Por", "porque", "qué", "si", "tenía", "una", "y"]`.

La clase abstracta `Indice`

Dado que, a pesar de sus diferencias, los tres son índices y tienen una funcionalidad similar, podemos definir una clase abstracta `Indice` de la que hereden los tres con las siguientes características:

- Una variable `texto`, de tipo `List<String>`, donde almacenará las líneas del texto en el orden en que se introduzcan.
- Un constructor en el que se inicialice la variable `texto`.
- Un método `void agregarLinea(String texto)` que agrega una línea de texto a las que ya tenga almacenadas. Estas líneas serán las que formarán el texto a analizar. La última línea agregada será la última línea del texto.

- El método `void resolver(String delimitadores, Collection<String> noSignificativas)` que recibe los separadores de las palabras –por ejemplo, la cadena `"[ .,:;-[!]¡?¿]+"` – y una colección de palabras no significativas, y debe construir el índice.
- El método `void presentarIndiceConsola()` permite mostrar el resultado (en el formato indicado arriba para cada caso) en la consola.

Obsérvese que los métodos `resolver` y `presentarIndiceConsola` dependen del tipo de índice, y por tanto deberán ser métodos abstractos en la clase abstracta `Indice`.

La clase `Indice1aLinea`

La clase `Indice1aLinea` hereda de la clase abstracta `Indice`, y tendrá, además de la variable `texto` que hereda de `Indice`, una variable `palabras` donde se almacenará el índice que se construya a partir del texto disponible en un momento dado en la variable de instancia `texto`. El índice a construir es, básicamente, una aplicación en la que a cada palabra significativa del texto se le asocia el número de la primera línea en la que aparece (véase la salida en el caso del ejemplo anterior), es decir, necesitamos una estructura del tipo `Map<String, Integer>`. Obsérvese que la interfaz de la colección utilizada es ordenada, de forma que el índice quedará ordenado de forma automática por palabras.

Además, la clase `Indice1aLinea` proporcionará un constructor e implementará los métodos heredados de `Indice`:

- Un constructor que inicialice adecuadamente las estructuras que sean necesarias para desarrollar la aplicación. Inicialmente no habrá ningún texto sobre el que operar; tanto el texto como los delimitadores y las palabras no significativas se introducirán posteriormente.
- Con respecto al método `void agregarLinea(String texto)` que heredamos de `Indice`, obsérvese que cada vez que se modifica el texto el índice `palabras` deja de ser válido; este método `agregarLinea` debería, por ejemplo, hacer un `clear()` de la estructura tras añadir la nueva línea.
- El método `void resolver(String delimitadores, Collection<String> noSignificativas)` debe construir el índice, calculando las primeras apariciones de cada palabra significativa en el texto y completar la estructura `palabras` para mantener esta información. Obsérvese que:
  - No se debe distinguir entre minúsculas y mayúsculas. Para facilitar el tratamiento, podemos empezar este método `resolver` creando un conjunto de palabras no significativas donde introduzcamos las palabras de `noSignificativas` tras convertirlas, p. ej., a minúsculas, de forma que sea fácil y rápido después comprobar si una palabra está o no en el conjunto de palabras no significativas.
  - Para extraer las palabras una a una de cada línea del texto utilizaremos una instancia de la clase `Scanner`.
  - Varias llamadas a este método con los mismos argumentos deben producir siempre el mismo listado.
- Dada la estructura utilizada para almacenar el índice, para producir la salida en el formato esperado lo único que el método `presentarIndiceConsola()` debe hacer es iterar sobre el conjunto de claves del `Map`, y para cada una de las palabras proporcionamos la primera línea en la que aparece.

Podemos probar el funcionamiento con la clase `EjIndice` proporcionada.

La clase `IndiceLineas`

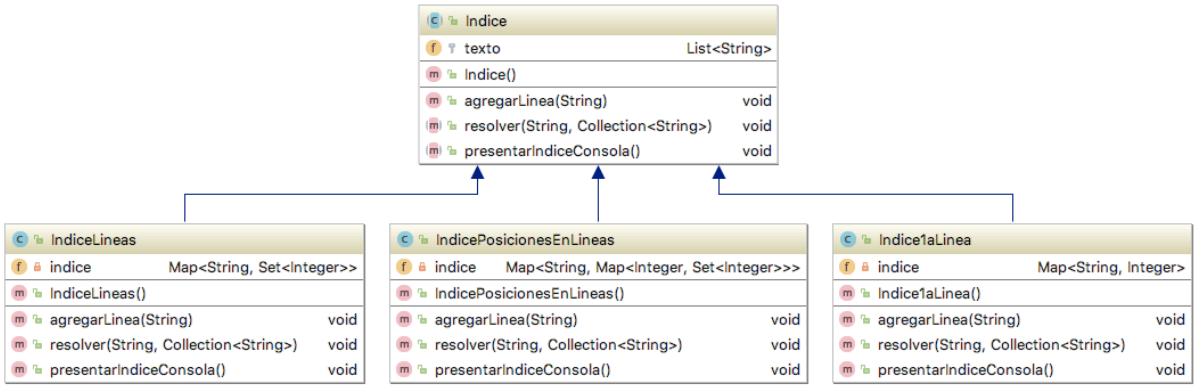
La clase `IndiceLineas` hereda también de `Indice` y sigue un patrón muy similar al de `Indice1aLinea`. Las principales diferencias con esta son:

- La variable `palabras` en este caso almacenará una aplicación en la que a cada palabra significativa del texto se le asocia el conjunto de líneas donde aparece (véase el ejemplo anterior), es decir, necesitamos una estructura del tipo `Map<String, Set<Integer>>`.
- El método `void resolver(String delimitadores, Collection<String> noSignificativas)` debe construir el índice como se indica arriba.
- En este caso, el método `void presentarIndiceConsola()` debe iterar sobre el conjunto de claves del `Map`, y para cada una de las palabras iterar sobre los elementos del conjunto asociado para mostrar las líneas en el formato adecuado.

Podemos probar el funcionamiento de esta clase con una clase similar a `EjIndice1aLinea` donde cambiemos la inicialización de la variable `cp`.

La clase `IndicePosicionesEnLineas`

Por último, la clase `IndicePosicionesEnLineas` hereda también de `Indice` y sigue un patrón muy similar al de `Indice1aLinea` e `IndiceLineas`. En este caso `palabras` almacenará una aplicación en la que a cada palabra significativa se le asocia una segunda aplicación en la que se le asocia el conjunto de posiciones de dicha palabra en cada número de línea en que hay ocurrencias de la misma (véase ejemplo anterior), es decir, necesitamos una estructura del tipo `Map<String, Map<Integer, Set<Integer>>>`. Obsérvese que, como en los casos anteriores, las interfaces de las colecciones utilizadas son todas ordenadas, de forma que el índice quedará de forma automática ordenado por palabras, y para cada una de estas por número de línea, y para cada línea las posiciones de menor a mayor. Dada la estructura utilizada, para mostrar el resultado, iteramos sobre el conjunto de claves del `Map` principal, y para cada una de las palabras obtenemos su aplicación asociada; iteramos nuevamente sobre el conjunto de claves de esta y para cada número de línea mostramos el conjunto de posiciones asociado a ella.



Módulo mdIndicePalabrasv2 (Colecciones (Set y Map), Scanner, io, herencia, orden)

En este ejercicio vamos a modificar las cuatro clases del módulo mdIndicePalabrasv1 para que tomen su información de ficheros y muestren sus resultados en un flujo dado (hacer una copia).

Vamos a realizar las siguientes modificaciones en la clase `Indice`:

- Crea el método **`public void`** `agregarDesdeFichero(String nombreFichero) throws IOException` que lea las líneas de texto desde un fichero de nombre dado. Se proporciona el fichero `frase.txt` que contiene las frases.

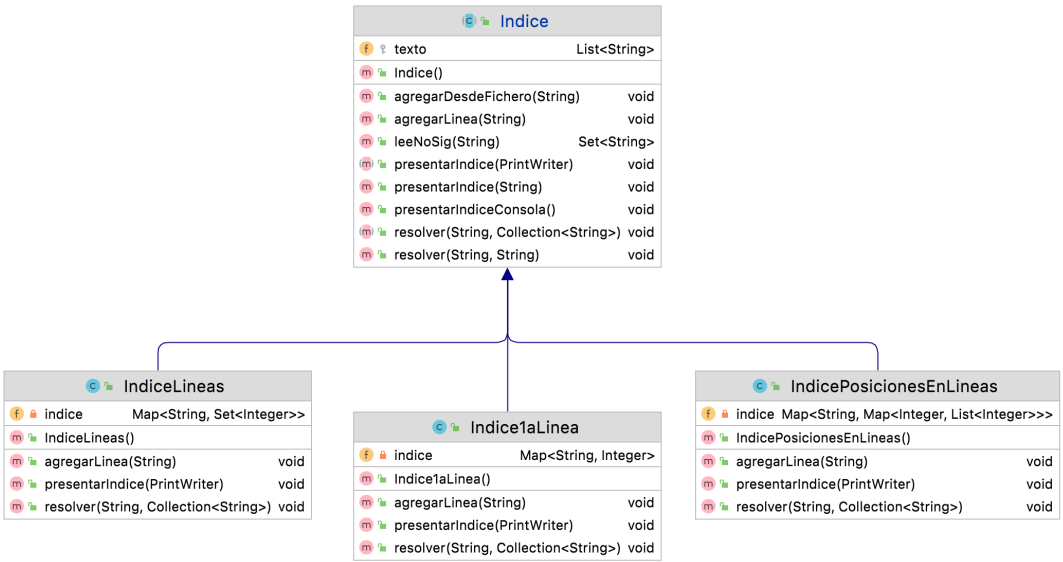
- Crea el método **`public Set<String>`** `leeNoSig(String nombreFichero) throws IOException` que lea las palabras no significativas desde un fichero de nombre dado. Se proporciona el fichero `noSig.txt` que contiene las palabras no significativas.

- Crea el método **`public void`** `resolver(String delim, String nombreFichero) throws IOException` que lea las palabras no significativas desde un fichero de nombre dado y resuelva el índice teniendo en cuenta el delimitador dado.

- Crea el método **`public void`** `presentarIndice(String nombreFichero) throws FileNotFoundException` que presente el índice en el fichero de nombre dado. Para ello crea el método **`abstract public void`** `presentarIndice(PrintWriter pw)` que presentará el índice en un flujo dado. Este método al ser abstracto se redefinirá posteriormente en cada una de las subclases de `Indice`.

- Cambiar el método **`public void`** `presentarIndiceConsola()` para no sea abstracto y se implemente aquí usando estos nuevos métodos. Borrar las implementaciones de este método definidas en las subclases de `Indice` que ya no son necesarios.

Usar la clase de ejemplo `EjIndice` para probar los cambios.



Módulo mdKWIC (equals, orden, colecciones, io)

El objetivo de esta práctica es realizar un glosario de palabras atendiendo a su aparición en un conjunto de frases (*KeyWord In Context*, KWIC), desechando aquéllas que no se consideren significativas. Para ello, contaremos con dos ficheros de entrada. El primero contendrá la relación de palabras no significativas (y que, por lo tanto, no aparecerán en el listado KWIC). El segundo contendrá una relación de títulos, a partir de las cuales deberemos obtener el correspondiente índice. Un ejemplo de fichero con palabras no significativas podría contener las siguientes líneas:

```
el la los las un una unos unas
y o
a ante bajo cabe con contra de desde en entre hacia hasta
para por según sin sobre tras
si no
al del
corre toma llama
```

El siguiente listado de títulos de películas podría servir como ejemplo de contenido de un fichero con los títulos (un título por línea) a partir de las cuales hay que construir un índice KWIC.

```
El color del dinero
Color púrpura
Misión: imposible
La misión
La rosa púrpura del Cairo
El dinero llama al dinero
La rosa del azafrán
El nombre de la rosa
Toma el dinero y corre
```

El índice que se desea generar debe tener el siguiente aspecto:

```
AZAFRÁN
      La rosa del azafrán
CAIRO
      La rosa púrpura del Cairo
COLOR
      Color púrpura
      El color del dinero
DINERO
      El dinero llama al dinero
      Toma el dinero y corre
      El color del dinero
IMPOSIBLE
      Misión: imposible
MISIÓN
      Misión: imposible
      La misión
NOMBRE
      El nombre de la rosa
PÚRPURA
      Color púrpura
      La rosa púrpura del Cairo
```

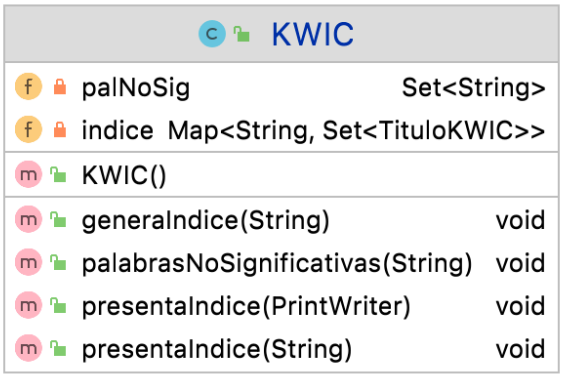
```
ROSA
      La rosa del azafrán
      La rosa púrpura del Cairo
      El nombre de la rosa ROSA
```

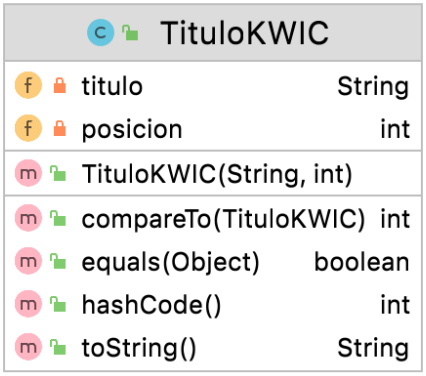
Visto lo anterior, se pide:

a) Definir una clase `TituloKWIC` que mantenga información de un título (de tipo `String`), y una posición (un entero).
   i. Dos títulos kwic son iguales si lo son sus títulos independientemente de la tipografía.
   ii. Un título kwic es menor que otro si lo son sus títulos independientemente de la tipografía.
   iii. Para mostrar un título kwic, se muestra sólo el título (no la posición).

b) Definir una clase `KWIC` que incluya los métodos necesarios para:
   i. Leer (y almacenar) la información de las palabras no significativas,
      `public void palabrasNoSignificativas(String)`
      `public void palabrasNosSignificativas(Scanner)`
      La información se almacenará en un `Set<String>`.
   ii. Generar la estructura del índice a partir de un fichero de texto (p.ej. títulos de películas) teniendo en cuenta las palabras no significativas leídas previamente,
      `public void generarIndice(String)`
      `public void generarIndice(Scanner)`
      La información se almacenará en un `Map<String,Set<TituloKWIC>>`
   iii. Representar el índice sobre un dispositivo.
      `public void presentaIndice(String)`
      `public void presentaIndice(PrintWriter)`

   Probar las clases anteriores con la aplicación `EjemploKWIC`.





Módulo mdCanciones (equals, orden, colecciones, io)

Se desea construir una aplicación que permita emular un el almacenamiento de listas de reproducción de canciones, con información sobre duración, título, intérprete y género. Para ello, se definirán un tipo enumerado Genero con los valores ROCK, POP, HIPHOP, DANCE y las clases que se indicarán a continuación en el paquete `canciones`.

1. Créese la clase `Duracion` para representar la duración en minutos y segundos, con las siguientes operaciones:

1.1. Proporciónese un constructor sin argumentos que cree objetos de duración cero y otro constructor con dos argumentos, que proporcionen los minutos y segundos. En este caso, al menos el valor que indica los segundos debe ser válido (entre 0 y 59); en caso contrario, debe lanzarse la excepción `IllegalArgumentException`, con un mensaje adecuado.

1.2. Proporciónese un método para incrementar la duración con otra que se pasa como argumento:

```
public void incrementa(Duracion tiempo)
```

El efecto debe ser el incremento de la duración que recibe el mensaje con la que se pasa como argumento. Debe tenerse en cuenta que el número de segundos del receptor no debe superar 59. Por ejemplo,

```
Duracion d1 = new Duracion(3,40);

Duracion d2 = new Duracion(2,50);

d1.incrementa(d2);    // d1 será ahora 6 minutos y 30 segundos
```

1.3. Dos objetos de tipo `Duracion` se considerarán iguales cuando sus minutos y segundos coinciden.

1.4. Defínase un orden natural que determine que un objeto `Duracion` es menor que otro cuando representa un periodo temporal menor.

1.5. Impleméntese el método `toString()` de forma que la representación textual de una duración sea "`[mm:ss]`", donde `mm` son los minutos y `ss` los segundos.

2. Constrúyase la clase `Cancion` que mantenga información sobre la duración (de tipo `Duracion`), el título (`String`), el intérprete (`String`) y el género (del tipo enumerado `Genero`), e incluya los siguientes elementos:

2.1. Un constructor para crear instancias de `Cancion` a partir de su título, intérprete, los minutos y segundos de reproducción y el género.

2.2. Operaciones que permitan obtener la información relevante de una canció

```
public Duracion getDuracion()

public String getTitulo()

public String getInterprete()

public Genero getGenero()
```

2.3. Una noción de igualdad que establezca que dos canciones son iguales cuando coinciden el título, el intérprete y la duración.

2.4. Un método `toString()` que permita representar textualmente canciones como:

```
[mm:ss] – TITULO EN MAYÚSCULAS (Intérprete)
```

3. Defínase la clase `Reproductor`, que mantenga información sobre diversas listas de reproducción de canciones, de forma que utilice una correspondencia (`Map`) que asocie a nombres de listas (`String`), listas de canciones (`List<Cancion>`). Para esta clase, debe incluirse:

3.1. Un constructor sin argumentos que inicialice la estructura de forma adecuada.

3.2. Un método para añadir listas de reproducción a partir del nombre de la lista y el nombre de un fichero, que añada a la correspondencia que almacena las listas, una nueva entrada que asocie al nombre de la lista (primer argumento) la información almacenada en fichero indicado como segundo argumento:

```
public void anyadirLista(String nombreLista, String fichero)
```

La información sobre las canciones en el fichero se organiza por líneas, donde cada línea tiene el siguiente formato (véanse los ficheros de prueba proporcionados):

```
Título,Intérprete,minutos,segundos,GENERO
```

3.3. El método anterior deberá llamar al método

```
protected void anyadirLista(String nombreLista, Scanner sc)
```

3.4. Un método para obtener el tiempo de reproducción de una lista de reproducción (indicada en el argumento), consistente en la suma de las duraciones de las canciones asociadas a esa lista, y otro método que devuelva el tiempo total de reproducción de todas las litas:

```
public Duracion tiempoReproduccion(String nombreLista)

public Duracion tiempoTotal()
```

3.5. Un método que vuelque sobre un fichero la información textual de todas las canciones que componen la lista indicada como argumento:

```
public void reproducir(String nombreLista, String ficheroSalida)
```

Proporciónese también un método similar que vuelque esa información sobre un `PrintWriter`:

```
public void reproducir(String nombreLista, PrintWriter salida)
```

4. Indíquese qué habría que hacer para definir una clase `ReproductorDuracion`, que se comporte como la clase `Reproductor`, pero en la que el efecto del método `reproducir` vuelque la información (sobre el fichero o sobre el `PrintWriter`), pero ordenando las canciones de la lista según su duración de menor a mayor.

5. Resolver las siguientes cuestiones:

a) Muestra en consola las duraciones de las canciones de un genero.
b) Obtener un array con los intérpretes de un genero dado (puede haber elementos repetidos).
c) Obtener una lista con los intérpretes de un genero dado (puede haber elementos repetidos).
d) Obtener un conjunto con los intérpretes de un genero dado.
e) Obtener un conjunto ordenado con los intérpretes de un genero dado.
f) Devuelve la suma de las longitudes de los títulos de todas canciones.
g) Crea una correspondencia que asocie a cada intérprete una lista con sus canciones.
h) Crea una correspondencia ordenada con el número de canciones por duración.
i) Crea una correspondencia que indica el número de canciones por género.
j) Devuelve cierto si todas las canciones del género ROCK son de la década de los 80.

Módulo mdPartidos. (colecciones (list, set, map), equals, orden, excepciones, io)(*challenge*)

Se pretende desarrollar una aplicación en Java que calcule los escaños que les corresponden a distintos partidos políticos que participan en unas elecciones según el número de votos obtenidos y los escaños a repartir. Se deben repartir `n` *escaños siguiendo algún criterio de repartición de escaños. El criterio de selección vendrá dado por la interfaz* `CriterioSeleccion` *y las clases que definan esa interfaz proporcionarán un criterio concreto. Así,* `DHontSimple` *proporciona el criterio de la ley D'Hont simple, la clase* `DHont` *implementa la ley D'Hont completa, y la clase* `Proporcional` *implementa un criterio proporcional. Para ello, se deberá proceder siguiendo las indicaciones y resolviendo los problemas que se plantean en los siguientes apartados. Mientras no se especifique lo contrario, las variables de instancia serán privadas y los métodos y constructores públicos y se trabajará en le paquete* `partidos`*.*

1. Constrúyase la excepción no comprobada `EleccionesException` que será lanzada en cualquier situación excepcional.

2. Constrúyase la clase `Partido` que represente a un partido político determinado por su nombre (`String`) y un número de votos (`int`). Defínanse los siguientes constructores y métodos:

a) Constructor con dos argumentos indicando el nombre del partido y el número de votos obtenido.

b) Métodos públicos para conocer el nombre `String getNombre()` y el número de votos `int getVotos()`.

c) Redefinición del método `boolean equals(Object)` para que dos partidos con el mismo nombre (sin distinguir mayúsculas y minúsculas) sean iguales.

d) Redefinición del método `toString()` para que la presentación de un partido sea:

```
nombre : votos
```

3. Constrúyase la interfaz `CriterioSeleccion` que incluye el método

```
Map<Partido, Integer> ejecuta(List<Partido> partidos, int numEsc)
```

que dado una lista de partidos y un número de escaños, reparte los escaños entre los partidos siguiendo el criterio correspondiente. Devuelve una correspondencia en la que a cada partido se le asocia el número de escaños obtenidos.

4. Constrúyase la clase `Elecciones` que contendrá como variable de instancia una lista de partidos.

a) Definir el método `static private Partido stringToPartido(String dato)` que crea un partido con la información que aparece en la cadena dato y lo devuelve. El dato tendrá el formato del ejemplo

```
"PESAO,455342"
```

El separador será la "`,`" y pueden aparecer una o más veces. Cualquier error lanzará una `EleccionesException` indicando el motivo.

b) Defínase el método `public void leeDatos(String [] datos)` que crea la lista de partidos y la rellena con la información que aparece en el array de datos pasado como argumento.

c) Defínase el método `public void leeDatos(String nombreFichero) throws IOException` que crea la lista de partidos con la información proporcionada en el fichero. En cada línea se encuentra la información de un partido.

d) Defínase el método

```
public Map<Partido, Integer>
     generaResultados(CriterioSeleccion cs, int numEsc)
```

que, conociendo el criterio de selección de escaños y el número de escaños a repartir, devuelva una correspondencia que asocie a cada partido el número de escaños que le corresponden.

e) Defínase el método `public void presentaResultados(String nombreFichero, Map<Partidos, Integer> map) throws FileNotFoundException` que genere en el fichero de nombre dado una relación de partidos con el número de escaños que le corresponden. Si un partido no tiene representación aparecerá con la palabra "Sin representación". El formato de salida será parecido al del ejemplo:

```
P.P. : 123655, 19
P.S.O.E. : 57245, 8
IULV-CA : 25354, 3
UPyD : 8099, 1
LOS VERDES : 3197, Sin representación
...
```

Entre los diferentes criterios de selección de escaños (clases que implementan la interfaz `CriterioSeleccion`) vamos a considerar la ley D'Hont simple, la ley D'Hont y un criterio proporcional.

5. Para implementar los criterios usaremos una clase auxiliar que llamaremos `Token` que mantienen como variables a un partido político, `partido` de la clase `Partido` y un `ratio` (de tipo `double`). Para esta clase, defínanse los siguientes constructores y métodos:

a) Constructor con dos argumentos siendo el primero un objeto de la clase `Partido` y el segundo un `double` que representa el ratio.

b) Métodos para conocer el ratio `double getRatio()`y el partido `Partido getPartido()`.

c) Un criterio de ordenación natural que ordene los tokens por ratio de mayor a menor y, en caso de igualdad, por nombre de partido.

d) Un método

```
public static
  Set<Token> seleccioneTokens(Set<Token> tks, int numEsc)
```

que seleccione del conjunto `tks` los primeros `numEsc` tokens (tks vendrá ordenado).

e) Un método

```
public static Map<Partido, Integer>
generaResultados(Set<Token> tks)
```

que genere y devuelva una correspondencia que asigne a cada partido que aparece en el conjunto `tks` un entero que indique cuantas veces aparece el partido en el conjunto.

6. Defínase la clase `DHontSimple` que define como criterio de selección la ley D'Hont simplificada. El método ejecuta define el criterio de selección de escaños por partido. Como argumento tiene la lista de partidos y el número de escaños a repartir. El algoritmo será el siguiente:

a) Para cada partido se crean tantos tokens como escaños hay que repartir. El segundo argumento del constructor serán sucesivamente el número de votos del partido dividido por los valores 1, 2, 3 … hasta el número de escaños a repartir.

b) Se ordenan los tokens según su orden natural.

c) Se seleccionan los primeros tokens, tantos como número de escaños a repartir hay.

d) Se devuelve una correspondencia que asocia a cada partido el número de tokens seleccionados.

Supóngase ahora que se desea contemplar que los partidos políticos que no lleguen a un mínimo porcentaje de votos no se consideren a la hora de repartir los escaños (así lo hace la ley D'Hont). Para ello vamos a crear la clase `DHont` que se comporta como la clase `DHontSimple`, pero que tiene en cuenta esta circunstancia.

7. Defínase la clase `DHont`, que se comportará como el criterio `DHontSimple`, pero además contiene un atributo `double minPor` que representará el mínimo porcentaje de votos admisible para contabilizar a un partido. Defínanse los siguientes constructores y métodos:

a) El constructor `DHont (double mp)` con el mínimo porcentaje admisible. El mínimo porcentaje debe cumplir `0 <= mp < 15`. En caso contrario se deberá lanzar una excepción.

b) Redefínase el criterio de selección de manera que antes de aplicar el criterio de `DHontSimple`, filtre aquellos partidos que no consigan el mínimo porcentaje.

Por último, se va a implementar el criterio de proporcionalidad (clase Proporcional).

8. Defínase la clase `Proporcional`. El método `ejecuta` define el criterio de selección de escaños por partido de forma proporcional. Como argumento tiene la lista de partidos y el número de escaños a repartir. El algoritmo será el siguiente:

a) Se calcula cuantos votos se necesitan para conseguir un escaño. Para ello, se calcula el total de votos emitidos y se divide por el número de escaños a repartir (variable `vpe`).

b) Para cada partido se crean tantos tokens como escaños hay que repartir. El segundo argumento del constructor serán sucesivamente el número de votos del partido menos los valores `0*vpe`, `1*vpe`, `2*vpe`, `3*vpe` … hasta el número de escaños a repartir menos 1 por `vpe`.

c) Se ordenan los tokens según su orden natural.

d) Se seleccionan los primeros tokens, tantos como número de escaños a repartir hay.

e) Se devuelve una correspondencia que asocia a cada partido el número de tokens seleccionados.

9. Para automatizar el proceso de generar unas elecciones, Defínase la clase `EleccionesManager` con las siguientes variables de instancia:

Un array `String [] datos` con los datos de los partidos políticos.
Un entero `numEsc` que indica el número de escaños a repartir.
Un `CriterioSeleccion cs` que indicará el criterio con el que se reparten los escaños.
Un `Elecciones elecciones` que guardará las elecciones que va a manejar.
Un `String fEntrada` que mantiene el nombre del fichero de entrada de datos.
Un `String fSalida` que mantiene el nombre del fichero de salida de datos.
Un `boolean consola` que indica si se deben presentar los resultados por consola

a) Defínase un constructor al que se le pasará como argumento las elecciones que debe manejar.
`public EleccionesManager(Elecciones elecciones)`

b) Defínase los siguientes métodos:
`public EleccionesManager setDatos(String [] datos)` que proporciona el array de datos. Este método deberá devolver el receptor.
`public EleccionesManager setCriterioSeleccion(CriterioSeleccion cs)` que proporciona el criterio de selección de los representantes. Este método deberá devolver el receptor.
`public EleccionesManager setNumEsc(int numEsc)` que proporciona el número de escaños se van a repartir. Este método deberá devolver el receptor.
`public Elecciones setEntrada(String fEntrada)` que proporcione el nombre del fichero de entrada. Este método deberá devolver el receptor
`public Elecciones setSalida(String fSalida)` que proporcione el nombre del fichero de salida. Este método deberá devolver el receptor
`public Elecciones setConsola(boolean consola)` que proporciona si un booleano que indica si se deben mostrar los resultados por la consola. Este método deberá devolver el receptor

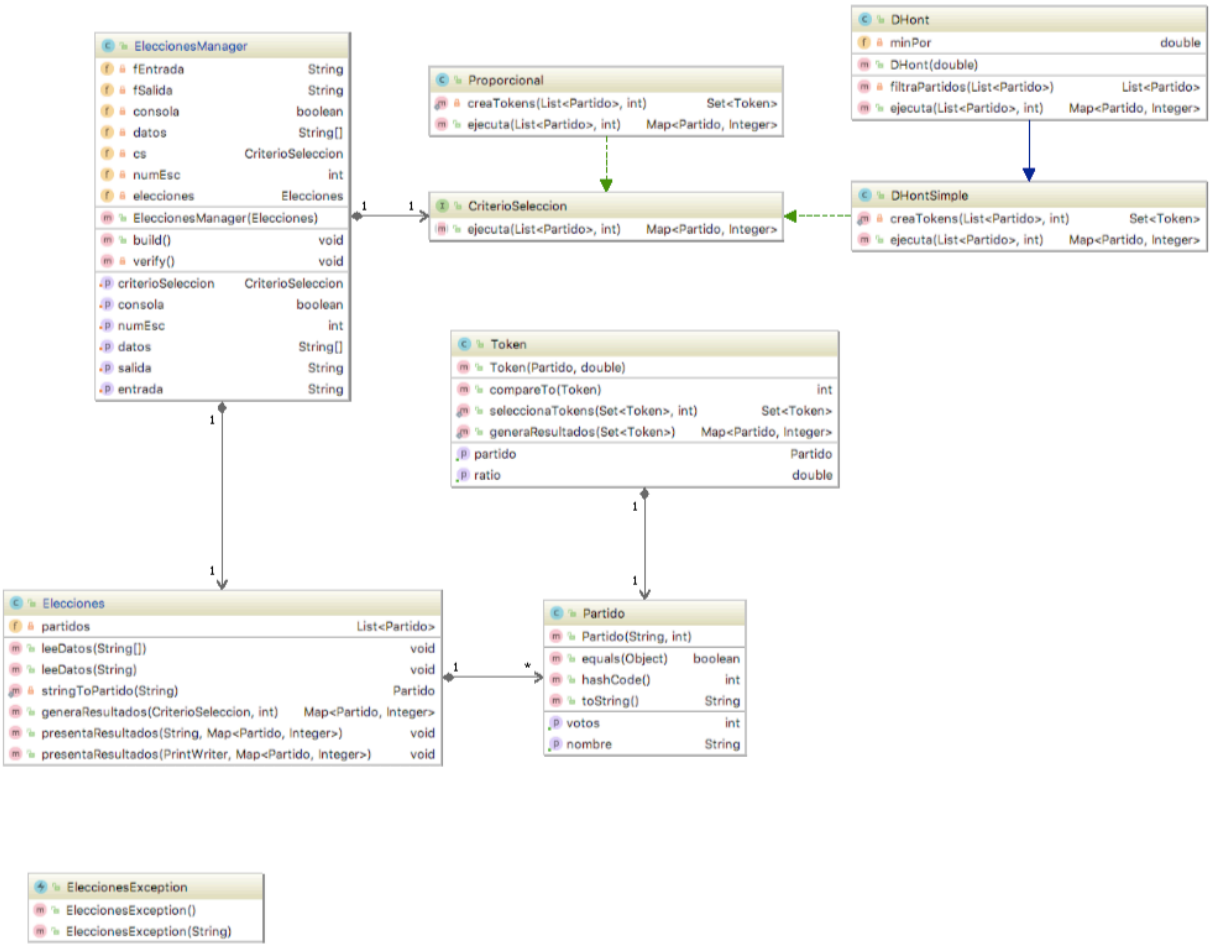
c) Defínase el método `private void verify()` que verifica que los datos que contiene las elecciones son correctos, es decir:

La entrada de datos única. Es decir, o se introducen los datos a través de un array o se introducen a través de un fichero (un debe haber).
Hay criterio de selección (no es `null`).
Hay escaños a repartir (es positivo).
Si hay fichero de salida, es decir fSalida no es null o hay salida por consola (pueden estar los dos pero una debe haber al menos).

Si alguna de estas condiciones falla, se debe lanzar una `ExceptionElecciones` indicando el motivo.

d) Defínase el método `public void build() throws IOException` que realiza las siguientes acciones:
- Verifica que los datos almacenados en las variables de instancia son correctos.
- Pide a `elecciones` que lea los datos de la entrada seleccionada.
- Pide a `elecciones` que genere la correspondencia `Map<Partido, Integer>` a partir del criterio de selección `cs`, la lista de partidos y el número de escaños `numEsc`.
- Pide a `elecciones` que presente en la salida seleccionada los resultados obtenidos.

Utilícese la clase `MainElecciones` proporcionado para hacer pruebas de funcionamiento.



Módulo mdRegata (list, set, map, i/o, orden)

En este ejercicio se va a desarrollar una aplicación que nos permita controlar los barcos que participan en una regata. Para ello, se creará un proyecto `mdRegata` con las clases siguientes en el paquete `regata`, donde la clase `Posicion` se proporciona en el campus virtual y el resto de clases deben crearse:

1) Para la realización del ejercicio, se proporciona la clase `Posicion` que determina una posición conocida su latitud y longitud (en grados centesimales). Esta clase dispone de un constructor donde se proporcionan los valores para la latitud y longitud (`double`). La latitud se normalizará a un valor entre -180 (latitud sur) y 180 (latitud norte) mientras que la longitud se normalizará a un valor entre 0 y 360 (grados a partir del meridiano de Greenwich en sentido horario). Así, una llamada al constructor con los valores (200, -400) quedará registrada como la posición de latitud 20 y longitud 320. Además, esta clase dispone de los siguientes métodos:
   a. `double getLatitud()` y `double getLongitud()` que devuelve la latitud y la longitud.
   b. `double distancia(Posicion p)` que calcula la distancia (en millas) desde el receptor a la posición `p`.
   c. `Posicion posicionTrasRecorrer(int minutos, int rumbo, int velocidad)` que calcula la posición final si partimos de la posición del receptor y viajamos los minutos dados con el rumbo (valor entre 0 y 359 siendo 0 el rumbo norte, 90 el rumbo este, etc.) y velocidad (dada en millas/h= nudos) dados.
   d. `String toString()` devuelve una cadena que representa a la posición. Por ejemplo, la posición de latitud 35 y longitud 156 se representa por `"l = 35 L = 156".`
2) Crear la excepción `RegataException` *no comprobada* para tratar las situaciones excepcionales.
3) La clase `Barco` debe mantener información sobre un barco. En concreto, tendrá una variable de instancia de tipo `String` para el `nombre`, otra de tipo `Posicion` para la `posicion` y dos de tipo `int` para `rumbo` y `velocidad.` El rumbo es un ángulo (0 para rumbo norte, 90 para rumbo este, etc. Así hasta 359.) y la velocidad se mide en km/hora. Todas las variables son *protected*.
   a. Definir un constructor que cree un barco conocida las cuatro variables descritas anteriormente. Comprobar que el rumbo se encuentra entre 0 y 359. Si no es así, lanzar una excepción de tipo `RegataException`.
   b. Definir métodos de acceso a cada variable (`String getNombre()`, etc.).
   c. Dos barcos son iguales si lo son sus nombres, ignorando mayúsculas y minúsculas.
   d. Un barco es menor que otro si su nombre es menor, ignorando mayúsculas y minúsculas
   e. El método `void avanza(int mnt)` cambia la posición del barco a la posición donde estaría una vez que transcurran `mnt` minutos (según su posición, rumbo y velocidad).
   f. El método `String toString()` debe mostrar la información de un barco. Por ejemplo, para un barco de nombre *gamonal*, situado en la posición *(-30, 290)*, con rumbo *0* y velocidad *24* este método mostraría la información de la siguiente manera:

```
gamonal: l= -30 L= 290 R= 0 V= 24
```

4) Crear una aplicación (clase distinguida `PruebaBarco`) que cree cuatro barcos y los introduzca en un array. Luego los ordene con `Arrays.sort(array)` y por último imprima el menor y el mayor.

5) Crear la clase `Velero` que se comporta como un barco, pero cuando su rumbo es menor o igual a 45 o mayor o igual a 315, avanza a una velocidad 3millas/hora inferior a su velocidad. Igualmente, si su rumbo está comprendido entre 145 y 225 avanza a una velocidad 3millas/hora superior a su velocidad. En otro caso se comporta como un barco normal (está simulando que hay viento del norte).

6) Definir un orden alternativo (clase `OrdenDistanciaMalaga`) que ordene los barcos por la distancia que les separa de Málaga (latitud 36.7585406 y longitud -4.3971722). En caso de que dos barcos estén a la misma distancia se ordenarán por el orden natural.

7) La clase `Regata` mantiene información de todos los barcos participantes en una regata. Los barcos se deben guardar en un conjunto ordenado por el orden natural.

   a. El constructor crea las estructuras adecuadamente.

   b. El método `void agrega(Barco)` agrega el barco a los participantes si no estaba ya.

   c. El método `void avanza(int mnt)` hace que todos los participantes se sitúen en la posición que quedarían transcurridos `mnt` minutos.

   d. El método `Set<Barco> participantes()` proporciona una colección con los barcos participantes en la regata.

   e. El método `Set<Barco> ordenadosPorDistanciaAMalaga()` devuelve una colección de participantes ordenados por la distancia a Málaga.

   f. El método `boolean hayBarcoSinArrancada()` devuelve cierto si hay al menos un barco cuya velocidad sea cero.

   g. El método `List<Barco> dentroDelCirculo(Posicion p, int millas)` devuelve los barcos que se encuentra a una distancia menor que `millas` de la posición `p`.

   h. Se llama nivel de un barco al entero que resulta de dividir la velocidad por 10. El método `Map<Integer, Set<Barco>> barcosPorVelocidad()` devuelve los barcos según su nivel. Por ejemplo, si un barco va a 34 millas/h se guardará asociado al entero 3 (34/10 = 3).

8) Añadir a la clase `Regata` los siguientes métodos que facilitan la entrada/salida.

   a. El método `Barco creaBarcoString(String)` que crea un barco y lo devuelve con los datos que aparecen en la cadena que se pasa como argumento. Esa cadena tendrá un formato como el del ejemplo:

   ```
   gamonal: l= -30 L= 290 R= 0 V= 24
   ```

   En este caso, al leer con `next()` de un scanner con delimitadores `"[ :=]+"` podéis encontraros con la `"l"`, `"L"`, `"R"` o la `"V"`. Simplemente, leer esos campos e ignorarlos.

   Si se produce alguna excepción no comprobada, transformarla en una `RegataException`.

   b. El método `void leeFichero(String)` lee los barcos de una regata de un fichero cuyo nombre se pasa como argumento y donde cada línea tiene el formato anterior.

   c. El método `void escribeFichero(String)` escribe los participantes en un fichero con el formato anterior y cuyo nombre se pasa como argumento. Se apoyará en el método `void escribe(PrintWriter)`.

En el campus virtual tenéis varios programas de prueba:

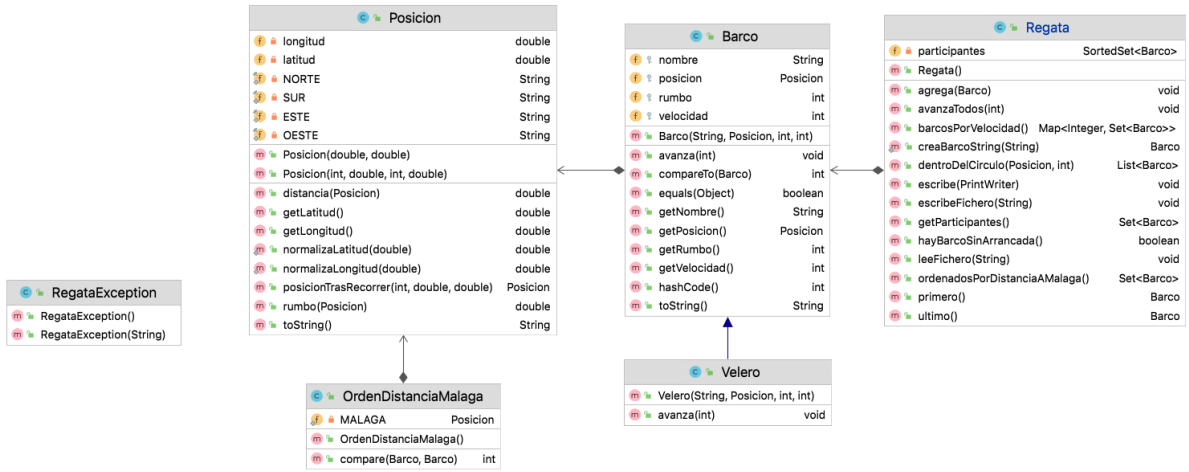
- Main2 (ejecutar al completar el apartado 4). Produce como resultado:

```
[alisa: l=  3,00 L= -1,00 R= 80 V= 20, gamonal: l=  0,00 L= -2,00 R= 0 V= 24, kamira: l= -2,00
L= -6,00 R= 230 V= 33, veraVela: l=  3,00 L=  2,00 R= 20 V= 14]
[alisa: l=  3,01 L= -0,97 R= 80 V= 20, gamonal: l=  0,04 L= -2,00 R= 0 V= 24, kamira: l= -2,04
L= -6,04 R= 230 V= 33, veraVela: l=  3,02 L=  2,01 R= 20 V= 14]
false
[alisa: l=  3,01 L= -0,97 R= 80 V= 20, gamonal: l=  0,04 L= -2,00 R= 0 V= 24]
{1=[veraVela: l=  3,02 L=  2,01 R= 20 V= 14], 2=[alisa: l=  3,01 L= -0,97 R= 80 V= 20, gamonal:
l=  0,04 L= -2,00 R= 0 V= 24], 3=[kamira: l= -2,04 L= -6,04 R= 230 V= 33]}
```

- Main3 (ejecutar al completar el apartado 7). Produce como resultado

```
Boquerón: l= 36,69 L= -4,39 R= 320 V= 14
Chanquete: l= 36,71 L= -4,37 R= 270 V= 33
Concha-Fina: l= 36,70 L= -4,35 R= 259 V= 24
Espeto: l= 36,68 L= -4,40 R= 350 V= 20
Tras avanzar 10 minutos ...
Boquerón: l= 36,71 L= -4,41 R= 320 V= 14
Chanquete: l= 36,71 L= -4,44 R= 270 V= 33
Concha-Fina: l= 36,69 L= -4,40 R= 259 V= 24
Espeto: l= 36,71 L= -4,41 R= 350 V= 20
```

y se debe haber creado el fichero salida.txt con el contenido de las 4 últimas líneas.



Módulo mdZonasMusculacion (equals, orden, colecciones, io)

Se va a crear una aplicación para manejar las zonas de musculación y las máquinas que contienen de la ciudad de Málaga. La información se extraerá de la web de datos abiertos del Ayuntamiento de Málaga. Para ello se crearán las clases `Maquina`, `Zona`, y `Musculacion` en el paquete `musculacion`. Todos los métodos que se piden se considerarán públicos y todas las variables de instancia privadas a menos que se especifique lo contrario.

Para realizar este proyecto se debe leer de un fichero de extensión *csv*. Esto será muy fácil usando la librería `opencsv`. Instalarla con Maven.

Clase `Maquina`

La clase `Maquina` mantiene información de una máquina de musculación. Así una máquina tendrá un identificador único (`int maquinaId`), un nombre (`String nombre`), una URL donde se encuentra una imagen de la máquina (`String urlImagen`), un nivel de dificultad (`int nivel`), una descripción funcional de la máquina (`String funcion`) y una descripción (`String descripcion`).

a) Define un constructor que crea una máquina con la información del identificador y del nombre.
b) Define métodos `set` para `urlImagen`, `nivel`, `funcion` y `descripcion` y métodos `get` para todas las variables de instancia.
c) Una máquina es menor que otra si su identificador es menor.
d) Una máquina con identificador `4` y nombre `Giro de cintura` se representará por:

```
Maquina(4, Giro de cintura)
```

Clase `Zona`

La clase `Zona` mantiene información de una zona de musculación de la ciudad, así como de las máquinas que dispone. Tendrá un identificador único de la zona (`int zonaId`), el nombre de la zona (`String nombre`), la longitud y latitud de la posición de la zona (`double longitud` y `double latitud`), una URL de la imagen de la zona (`String urlImagen`), y el conjunto de máquinas que dispone la zona (`Set<Maquina> maquinas`).

a) Define un constructor que crea una zona con la información del identificador y del nombre. Debe crear el conjunto ordenado de máquinas vacío.
b) Define métodos `set` para `urlImagen`, `latitud` y `longitud`, y métodos `get` para todas las variables.
c) Una zona es menor que otra si lo es su identificador..
d) Define el método `void agrega(Maquina mq)` que agrega una máquina al conjunto de máquinas de esta zona.
e) Una zona de identificador `100` y nombre `Pl. Olletas` se representará por:

```
Zona(100, Pl. Olletas)
```

Clase `Musculacion`

La clase `Musculacion` mantendrá información de todas las zonas que hay en una ciudad. Para ello mantendrá el nombre de la ciudad (`String ciudad`) y una correspondencia que asocia a cada identificador de zona, la zona correspondiente (`Map<Integer, Zona> zonas`).

a) Crea un constructor que proporcione el nombre de la ciudad. Deberá inicializar adecuadamente la estructura `zonas`.
b) Define el método `get` para el `nombre` de la ciudad.

Se van a definir ahora dos métodos para leer los datos de las zonas de musculación. El primero lee los datos de un fichero csv. El segundo de un objeto `CSVReader`. El primero se proporciona y como se ve, llama al segundollamará al segundo

```
public void leeDatos(String ficheroCSV) throws IOException, CsvException {
    try (BufferedReader bin =
                        Files.newBufferedReader(Paths.get(ficheroCSV));
         CSVReader reader = new CSVReader(bin)) {
        leeDatos(reader);
    }
}
```

c) Completa la definición del siguiente método privado:

```
private void leeDatos(CSVReader reader) throws IOException {
    reader.readNext(); // Ignoramos la primera linea
    List<String[]> datos = reader.readAll();// Leemos todas

    for(String[] tokens : datos) {

            // Leemos la maquina
            ...
            // leemos la zona de musculacion
            ...
    }
}
```

La primera línea del fichero de datos contiene las cabeceras por lo que se ignora. Cada una del resto de las líneas contiene información sobre una máquina y la zona en la que se encuentra. Hay también información que no necesitamos (mirar el fichero de ejemplo). Para rellenar la correspondencia, cada línea se convierte a un array de `String` llamado `datos`. De este array solo son significativos algunas posiciones. Por ejemplo, en la posición 18 se encuentra el identificador de la máquina, en la 19 el nombre, etc. Las siguientes constantes definidas en la clase `Musculacion` ayudan a localizar los datos importantes.

```
private final static int MAQUINA_ID = 18;
private final static int MAQUINA_NOMBRE = 19;
private final static int MAQUINA_URL_ICON = 21;
private final static int MAQUINA_NIVEL = 20;
private final static int MAQUINA_FUNCION = 22;
private final static int MAQUINA_DESCRIPCION = 23;
private final static int ZONA_ID = 14;
private final static int ZONA_NOMBRE = 15;
private final static int ZONA_URL_ICON = 17;
private final static int ZONA_UBICACION = 2;
```

El proceso con cada línea será el siguiente:

- Extraemos de `datos` los datos necesarios para crear una máquina y actualizar toda su información (`urlImagen`, `nivel`, `funcion`, etc.).
- Extraemos la información del identificador de zona. Si la zona no estaba creada (no se encuentra en la correspondencia `zonas`), se crea con la información que aparece en esta línea (y se añade a la correspondencia). Si ya existía, la información de la zona de esta línea se ignora.
- Añadimos la máquina creada a la zona correspondiente.

Al final se dispondrá de una correspondencia que a cada identificador de zona asocia la zona. Dentro de cada zona estarán las máquinas que dispone.

d) Define el método `Set<Zona> getZonas()` que devuelve todas las zonas de musculación de la ciudad.

e) Define el método `Set<Maquina> getMaquinasEnZonaId(int zonaId)` que devuelve un conjunto con todas las máquinas que hay en la zona dada. Si la zona no existe devuelve un conjunto vacío.

f) Define el método `Set<Zona> getZonasConMaquinaId(int maquinaId)` que devuelve un conjunto con todas las zonas en las que hay una máquina dada. Si la máquina no existe se devuelve un conjunto vacío.

Probar las clases con el programa Main dado.