Samsung
TECH INSTITUTE

MÓDULO 2. PROGRAMACIÓN ORIENTADA A OBJETOS CON JAVA

Relación de Problemas Nº 5

Proyecto prNotas. (colecciones (list y set), orden, excepciones)

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`.

- La clase `Alumno` mantiene información de un alumno del cual se conocen el nombre (`String`), el dni (`String`) y la calificación obtenida en una asignatura (`double`). La clase tendrá dos constructores, uno en el que se proporcionan el nombre, el dni y la calificación y otro con solo el nombre y el dni siendo en este caso la calificación igual a cero.
  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 getCalificación()`).
  La representación de un alumno debe mostrar el nombre y el dni pero no la calificación.
- Crear la excepción no comprobada `AlumnoException` para manejar situaciones excepcionales que podrán producirse en la siguiente clase `Asignatura`.
- Crear la clase `Asignatura`. Una asignatura se crea a partir del nombre de la misma y de una lista de String con los datos del alumno. Cada línea 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
  ```

  El constructor 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, deberá almacenar esta entrada en otra lista de `String` (llamado `erroneos`) precedido de un comentario que indique cual ha sido el problema por lo que no se ha podido crear el alumno. Por ejemplo, ante la entrada:

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

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

  ```
  Nota no numerica 342424f2J-Fernandez Vara, Pedro-tr
  ```

  La clase `Asignatura` incluirá el método:

  ```
  Optional<Double> getCalificacion(Alumno al)
  ```

  que devuelve un optional con la calificación del alumno `al` dado si es que existe. Si no existe el optional estará vacío.

  También tendrá dos métodos, uno que devuelve una lista con los alumnos (`List<Alumno> getAlumnos()`) y otro que devuelve una lista de entradas erróneas (`List<String> getErroneas()`).

Además incluirá un método que proporcione la media de las calificaciones de los alumnos de la asignatura (`double getMedia()`).

Incluir también el método `List<Alumno> getAlumnosAprobados()` que devuelve una lista con los alumnos cuya calificación es igual o superior a 5.

Aquí se presenta un ejemplo de uso de estas clases y la salida correspondiente.

```
import prNotas.*;
import java.util.*;

public class Main {

    static List<String> als = Arrays.asList(
            "25653443S-Garcia Gomez, Juan-8.1",
            "23322443K-Lopez Turo, Manuel-4.3",
            "24433522U-Merlo Martinez, Juana-5.3",
            "53553421D-Santana Medina, Petra-7.1",
            "55343442L,Godoy Molina, Marina-6.3",
            "34242442J-Fernandez Vara, Pedro-tr",
            "42424312G-Lopez Gama, Luisa-7.1");

    public static void main(String[] args) {
        Asignatura algebra = new Asignatura("Algebra", als);
        Alumno al1 = new Alumno("Lopez Turo, Manuel", "23322443K");
        Alumno al2 = new Alumno("Fernandez Vara, Pedro", "34242442J");

        Optional<Double> od = algebra.getCalificacion(al1);
        if (od.isPresent()) {
            System.out.println("Calificacion de " + al1 + ": "
                    + od.get());
        } else {
            System.err.println("No existe el alumno " + al1);
        }
        od = algebra.getCalificacion(al2);
        if (od.isPresent()) {
            System.err.println("Calificacion de " + al2 + ": "
                    + od.get());
        } else {
            System.out.println("No existe el alumno " + al2);
        }

        System.out.println("Media: " + algebra.getMedia());

        System.out.println("Alumnos...");
        for (Alumno alumno : algebra.getAlumnos()) {
            System.out.println("alumno : " + alumno.getCalificacion());
        }
        System.out.println("Erroneos...");
        for (String malo : algebra.getErroneas()) {
            System.out.println(malo);
        }
        System.out.println("Alumnos aprobados...");
        for (Alumno alumno : algebra.getAlumnosAprobados()) {
            System.out.println(alumno + ": " + alumno.getCalificacion());
        }

        System.out.println("Asignatura...");
        System.out.println(algebra);
```

```
    }
}
```

La salida al ejecutar el programa anterior es (la línea roja puede variar su posición)

```
Calificacion de 23322443K Lopez Turo, Manuel: 4.3
No existe el alumno 34242442J Fernandez Vara, Pedro
Media: 6.65
Alumnos...
alumno : 8.1
alumno : 4.3
alumno : 7.1
alumno : 7.1
Erroneos...
DNI Incorrecto: 24433522U-Merlo Martinez, Juana-5.3
Faltan datos: 55343442L,Godoy Molina, Marina-6.3
Nota no numérica: 34242442J-Fernandez Vara, Pedro-tr
Alumnos aprobados...
25653443S Garcia Gomez, Juan: 8.1
53553421D Santana Medina, Petra: 7.1
42424312G Lopez Gama, Luisa: 7.1
Asignatura...
Algebra : {[25653443S Garcia Gomez, Juan, 23322443K Lopez Turo, Manuel, 53553421D
Santana Medina, Petra, 42424312G Lopez Gama, Luisa], [DNI Incorrecto: 24433522U-Merlo
Martinez, Juana-5.3, Faltan datos: 55343442L,Godoy Molina, Marina-6.3, Nota no
numérica: 34242442J-Fernandez Vara, Pedro-tr]}
```

Orden en la clase Alumno

- Define un orden natural en la clase `Alumno` de forma que se ordenen por nombre y en caso de igualdad por dni sin importar si la letra está en mayúsculas o minúsculas.
    - Añade al `main` lo necesario para mostrar los alumnos aprobados ordenados por el orden natural.
- Crea tres órdenes simples para cada variable de instancia de la clase `Alumno` (tres clases, `OrdenNombre`, `OrdenDNI`, `OrdenNota` que implementen la interfaz `Comparator<Alumno>`).
    - Añade al `main` lo necesario para mostrar los alumnos aprobados ordenados según:
        - Las calificaciones de menor a mayor y en caso de igualdad por nombre.
        - Las calificaciones de mayor a menor y en caso de igualdad por DNI de mayor a menor.
        - Las calificaciones, de mayor a menor y en caso de igualdad por el orden natural.

Proyecto prLlaves (colecciones (list y set), excepción)

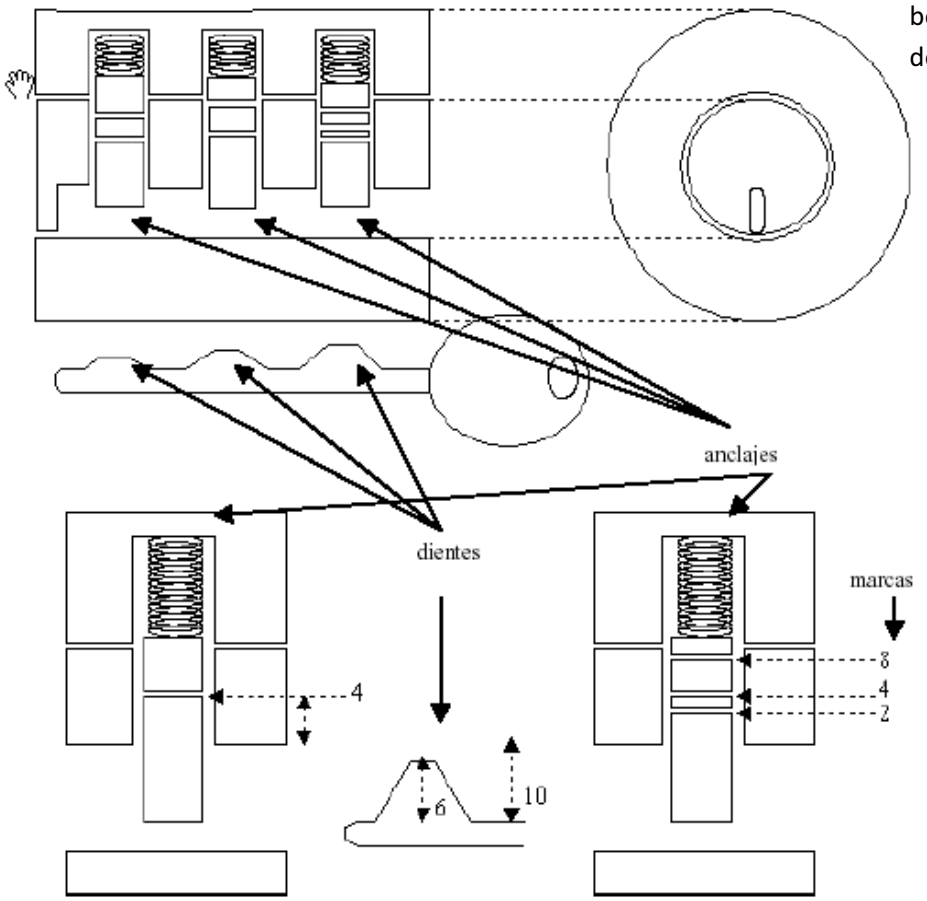
Una llave está formada por un número determinado de dientes, cada uno de una altura (una llave se representará como una lista de enteros). Inicialmente las llaves tienen sus dientes sin limar a una altura de 10 milímetros, obteniéndose el perfil deseado limando cada uno de estos dientes una altura n_i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

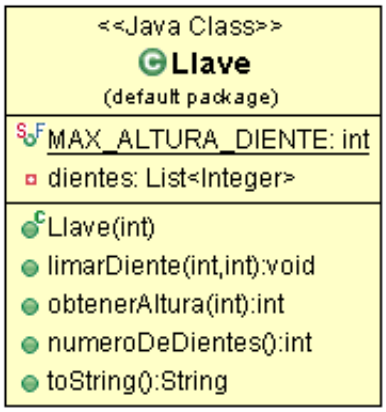
bombillo (visto de perfil)

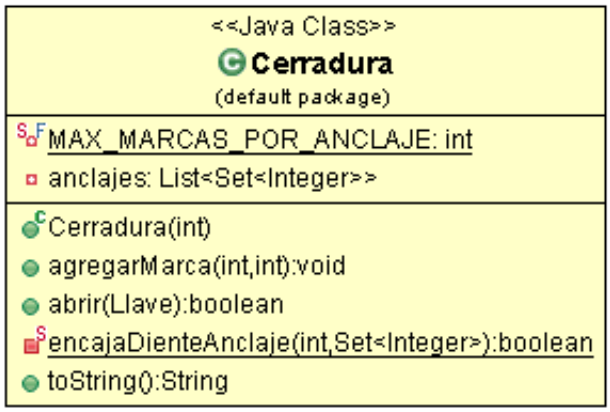


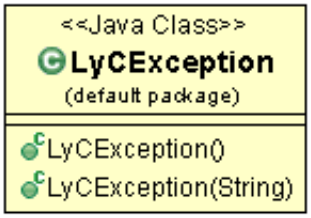
Se pide:

- Implementar las clases `Llave` y `Cerradura` con los métodos especificados en el diagrama de clases mostrado más abajo.
- Implementar una aplicación que pruebe si una cerradura se abre con varias llaves dadas.
- Implementar una segunda aplicación en la que se defina una interfaz gráfica de usuario simple que permita:
    - crear llaves y cerraduras (inicialmente con sus dientes a la altura máxima y sin marcas en sus émbolos) de forma que puedan ser utilizadas más adelante,
    - destruir llaves y cerraduras previamente creadas,
    - limar dientes de llaves previamente creadas,
    - añadir marcas a émbolos de cerraduras previamente creadas,
    - y comprobar si una llave abre una cerradura.

Las operaciones que se realizan sobre una llave o una cerradura se realizarán sobre objetos creados previamente. Es decir, la interfaz dará la posibilidad de, por ejemplo, seleccionar una llave y una cerradura creadas previamente y comprobar si dicha llave abre tal cerradura, o seleccionar una llave creada previamente e indicando un diente y una altura limar dicho diente tal cantidad.







Controlad por medio de la excepción `LyCException` que tanto los dientes como los anclajes usados sean correctos. La excepción debe ser de obligado tratamiento.

Proyecto prPartidos. (colecciones (list, set, map), equals, orden, excepciones)

Se pretende desarrollar una aplicación en Java que calcule los escaños que le corresponden a distintos partidos políticos que participan en unas elecciones según el número de votos obtenidos. 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.*

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) Definir 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) Definir 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.

d) Definir el método `public void presentaResultados(Map<Partidos, Integer> map)` que genere en la consola 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 mantiene 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. Definir 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.

Supongamos 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. Definir 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) Redefinir 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, vamos a implementar el criterio de proporcionalidad (clase Proporcional).

8. Definir 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, definir la clase `EleccionesManager` con las siguientes variables de instancia:

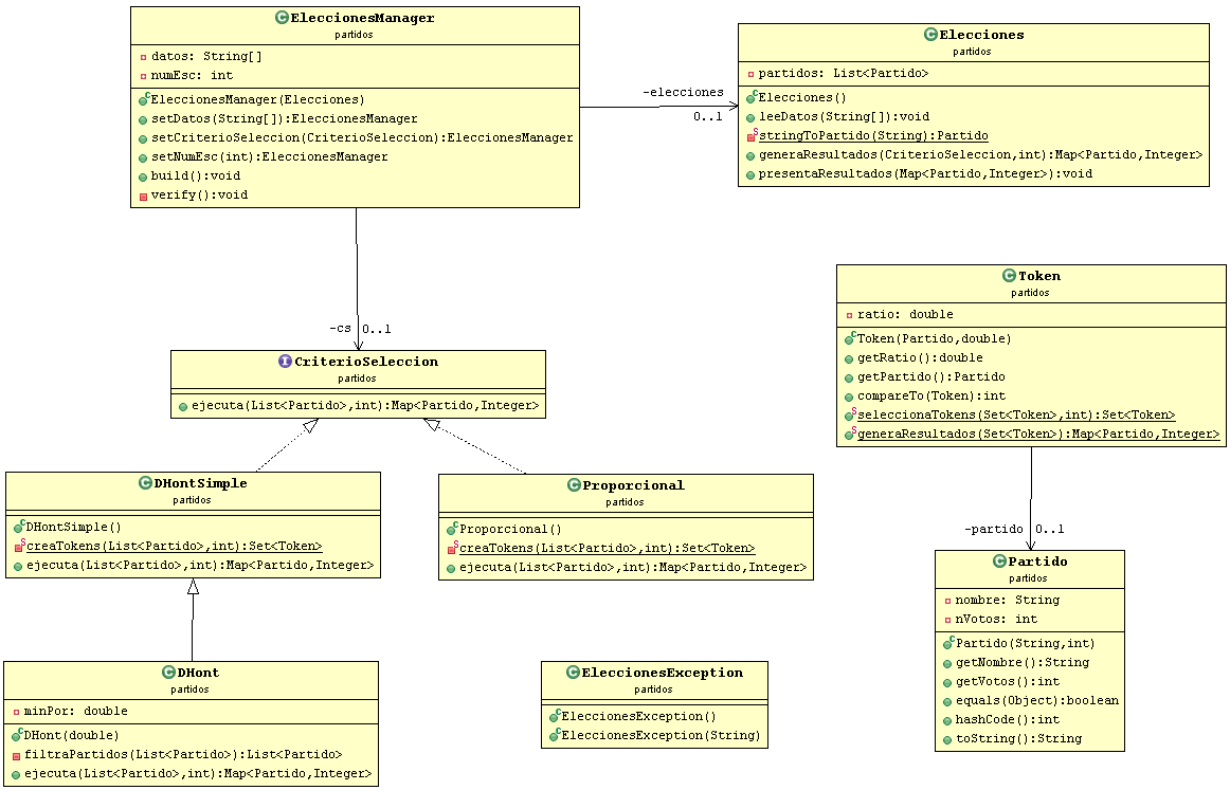
Un array `String [] datos` con los datos de los partidos políticos.
Un entero `numEsc` que indica el número de escaños a repartir.
Un `CriterioSelección` cs que indicará el criterio con el que se reparten los escaños.
Un `Elecciones elecciones` que guardará las elecciones que va a manejar.

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

b) Definir los siguientes métodos:
`public EleecionesManager 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.

c) Definir el método `private void verify()` que verifica que los datos que contiene las elecciones son correctos, es decir:
Hay un array de datos (no es `null`)
Hay criterio de selección (no es `null`)
Hay escaños a repartir (es positivo)
Si alguna de estas condiciones falla, se debe lanzar una `ExceptionElecciones` indicando el motivo.

d) Definir el método `public void build()` que realiza las siguientes acciones:
- Verifica que los datos almacenados en las variables de instancia son correctos.
- Pide a `elecciones` que lea los datos del array `datos`.
- 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 consola los resultados obtenidos.



Proyecto prUrgencias (Colecciones (set y map), equals, compare, 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 han de definir las siguientes clases e interfaces.

1. Defínase una clase `Hora` que permita representar la hora (hora y minutos) en un día. Para ello, la clase incluirá dos propiedades de tipo entero (`int`) para representar hora y minuto, incluyendo:
   a) Un constructor que cree objetos de tipo Hora, a partir de dos argumentos de tipo entero, controlando que el primero represente una hora, entre 0 y 23, y el segundo un minuto, entre 0 y 59. De no ser así, se debe lanzar una excepción de tipo `IllegalArgumentException`.
   b) Métodos de consulta para conocer la hora y el minuto.
   c) Una noción de igualdad por la que dos objetos Hora son iguales cuando coinciden en hora y minuto.
   d) Un criterio de ordenación natural, que compare primero por hora y en caso de coincidencia por minuto.
   e) Un método `int diferenciaMinutos(Hora hora)` que devuelva la diferencia en minutos entre el objeto que recibe el mensaje y la hora que se pasa como argumento. No se tendrá en cuenta si el receptor es menor o mayor que el argumento. Es decir, el entero que se devuelve siempre será positivo.
   f) Una representación textual del tipo: [hora:minuto]

2. 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 Hora), 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`). La clase deberá incluir:
   a) Un constructor que cree ingresos a partir de objetos adecuados:
      `Ingreso(Hora,Hora,String,String,TipoUrgencia)`
   b) Métodos de consulta para cada uno de las propiedades mencionadas.
   c) 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.
   d) Un orden natural que ordene por hora de ingreso y, en caso de igualdad, por número de la seguridad social.
   e) Una representación textual del tipo:

      {hora de ingreso, tiempo de consulta – número de la SS – código del médico}

3. Defínase una clase Urgencias que represente los ingresos en un día determinado de distintas unidades de urgencias de un mismo Centro de Salud. Esta case, debe incluir propiedades que permitan representar el día del mes (de tipo entero), el mes (de tipo entero) y una colección ordenada de ingresos. La clase `Urgencias` debe incluir:
   a) Un constructor que cree un objeto de la clase a partir del día del mes y el mes, que se pasan como argumentos, con la colección de ingresos inicialmente vacía. Debe controlarse que la fecha sea consistente (sin tener en cuenta si el año es bisiesto o no). En caso de que no sea así, debe lanzarse la excepción `IllegalArgumentException`.

b) Un método `void agregaServicio(String ingreso)` que lea los datos del `String ingreso`, genero un ingreso y lo incorpore a la colección de ingresos. El formato del `String ingreso` será el siguiente

`hora:minuto, hora:minuto, númeroSS, códigoMédico, gravedad`

donde hora y minuto son enteros que representan la hora de ingreso (la primera) y hora de salida o alta (la segunda), númeroSS representa el identificador de la SS, códigoMédico es una cadena de caracteres que representa el código del médico que atiende la urgencia y gravedad es un entero indicando la gravedad: 0-LEVE, 1-MODERADO y 2-GRAVE.

c) Un método `int urgenciasAtendidas()` que devuelve el número de servicios realizados.

d) Un método `Map<String,Integer> pacientesPorMedico()`, que construye una correspondencia asociando a cada código de médico el número de pacientes que ha atendido.

e) Un método `Set<Ingreso> ingresosPorTiempoDeAtencion()`, que devuelve un conjunto de ingresos ordenado según el tiempo de atención (diferencia entre la hora de ingreso y la hora de salida o alta).

f) Un método `void presentaIngresos()` que vuelca sobre la consola la siguiente información:

dia/mes/2014
colección de ingresos: uno por línea

Un programa de prueba

```
import prUrgencias.Urgencias;

public class PruebaUrgencias {
	public static void main(String[] args) {
		String [] ingresos = {"9, 15, 9, 30, 123415,MI766, 1",
                                      "9, 15, 10, 30, 123543, MI765, 0",
                                      "9, 17, 9, 30, 123734, TR454, 2",
                                      "10, 15, 10, 31, 123465, TR325, 0",
                                      "9, 10, 9, 20, 123261, TR325, 0",
                                      "9, 15, 9, 35, 124415,MI766, 2",
                                      "11, 15, 11, 30, 123243, TR325, 0",
                                      "9, 17, 9, 30, 123724, TR454, 2",
                                      "10, 15, 10, 31, 123455, TR325, 0",
                                      "11, 40, 11, 50, 223261, TR325, 1"};

		Urgencias urg = new Urgencias(12, 7);
		for (String ingreso: ingresos) {
			urg.agregaServicio(ingreso);
		}

		System.out.println(urg.ingresosPorTiempoDeAtencion());
		System.out.println(urg.pacientesPorMedico());

		urg.presentaServicio();

	}
}
```

Proyecto prAnagramas (equals, compare, 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.

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:

```
import prAnagramas.DicAnagramas;
import prAnagramas.SatSignatura;

public class TestAnagramas{
	public static void main (String[] args) {

		System.out.println("Ordenadas alfabeticamente");
		DicAnagramas dic = new DicAnagramas();
		for (String palabra : args) {
			dic.agregarPalabra(palabra);
		}
		dic.presentaDiccionario();
	}
}
```

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 siguiente programa:

```java
import prAnagramas.DicAnagramas;
import prAnagramas.SatSignatura;

public class TestAnagramas{
	public static void main (String[] args) {

		System.out.println("Ordenadas alfabeticamente");
		DicAnagramas dic = new DicAnagramas();
		for (String palabra : args) {
			dic.agregarPalabra(palabra);
		}
		dic.presentaDiccionario();

		System.out.println("Ahora ordenadas por longitud de la palabra");
		dic = new DicAnagramas(new SatSignatura());
		for (String palabra : args) {
			dic.agregarPalabra(palabra);
		}
		dic.presentaDiccionario();
	}
}
```

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]
```

Proyecto prIndicePalabrasv1 (Colecciones set y map, herencia, Scanner)

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 índice:

- `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")

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` creada con la línea de texto a "romper" y los delimitadores a utilizar (dados como argumento de `resolver` y ajustados para usar con `Scanner`). Esta clase nos permite de una forma muy simple acceder a cada uno de los *tokens* como si de un iterador se tratara. La forma de utilizarlo es básicamente la siguiente:

```
try (Scanner sc = new Scanner(linea)) {
      sc.useDelimiter(delim);
      while (sc.hasNext()) {
            String palabra = sc.next().toLowerCase();
            ...
      }
}
```

- 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 damos la primera línea en la que aparece.

Podemos probar el funcionamiento de esta clase con la siguiente aplicación:

```
import java.util.Arrays;
import java.util.List;
import prIndicePalabras1.*;

public class EjIndice {
    public static void main(String args[]) {
       String delimitadores = "[ .,:;-[¡][!][¿][?]]+";
       List<String> noSignificativas =
             Arrays.asList("A", "buen", "con", "de", "ha", "hubiera",
                           "la", "NO", "pero", "Por", "porque", "qué",
                           "si", "tenía", "una", "y");
       Indice cp = new Indice1aLinea();
       // Indice cp = new IndiceLineas();
       // Indice cp = new IndicePosicionesEnLineas();
       cp.agregarLinea("Guerra tenía una jarra y Parra tenía una perra, "
                     + "pero la perra de Parra rompió la jarra de Guerra.");
       cp.agregarLinea("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.");
       cp.agregarLinea("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.");
       cp.resolver(delimitadores, noSignificativas);
       cp.presentarIndiceConsola();
    }
}
```

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.