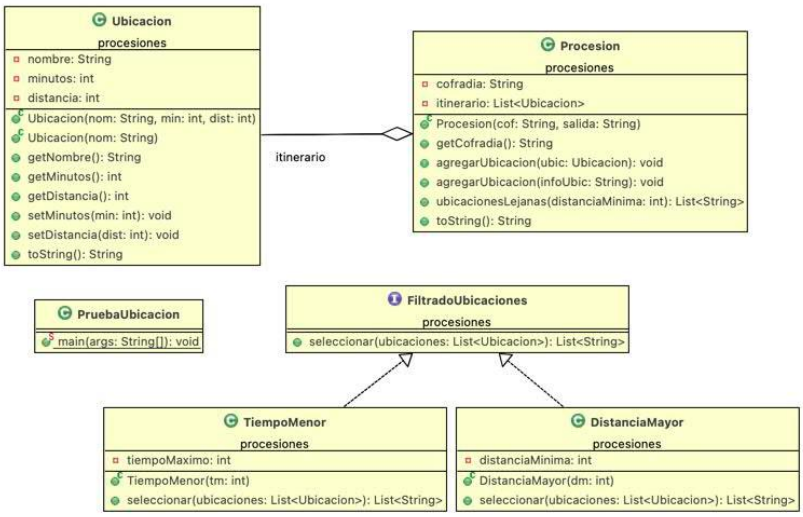
DEPARTAMENTO LENGUAJES Y CIENCIAS DE LA COMPUTACIÓN	**PROGRAMACIÓN AVANZADA I** (Prueba del 13 de mayo de 2024)	
APELLIDOS, Nombre	TITULACIÓN Y GRUPO	
	MÁQUINA	

NOTAS PARA LA REALIZACIÓN DEL EJERCICIO:

- Al **inicio del contenido de cada fichero** realizado deberá aparecer un comentario con tus **apellidos y nombre, titulación y grupo.**
- Los diferentes apartados tienen una determinada puntuación. Si un apartado no lo sabes hacer, *no debes pararte en él indefinidamente*. Puedes abordar otros.
- **Está permitido:**
  - Consultar los apuntes (CV), y la guía rápida de la API (CV).
  - Añadir métodos privados a las clases.
- **No está permitido**:
  - Intercambiar documentación con otros compañeros.
  - Recibir ayuda de otras personas. Se debe realizar personal e individualmente la solución del ejercicio propuesto.
  - Añadir métodos no privados a las clases.
  - Añadir variables o constantes a las clases.
  - Modificar la visibilidad de las variables, constantes y métodos que se indican.
  - Modificar el código suministrado.
- Una vez terminado el ejercicio, debéis subir (a la tarea creada en el campus virtual para ello) un fichero comprimido de la carpeta **src** que hayáis realizado y usáis vuestros apellidos y nombre para su denominación **(Apellido1Apellido2Nombre.rar** o **.zip)**.
- La evaluación tendrá en cuenta la claridad de los algoritmos, del código y la correcta elección de las estructuras de datos, así como los criterios de diseño que favorezcan la reutilización.
- Para la corrección del ejercicio se utilizarán **programas de detección de copias/plagios**.
- Con posterioridad a la realización del ejercicio, el profesor podrá convocar a determinado/as alumno/as para realizar **entrevistas personales** con objeto de comprobar la autoría de las soluciones entregadas.

Proyecto prProcesiones

Se va a crear una aplicación para representar procesiones de Semana Santa. Para ello se crearán en el paquete `procesiones` las clases que siguen: `ProcesionesException`, `Ubicacion` y `Procesion`; además de la interfaz `FiltradoUbicaciones` y las clases `TiempoMenor` y `DistanciaMayor`. También hay que implementar la clase `PruebaUbicacion` en el paquete por defecto. Se proporciona una versión simplificada de la clase `Ubicacion`, correspondiente a una solución de la anterior prueba de seguimiento.



1) **(0,25 puntos)** Defínase la clase `ProcesionesException` que represente excepciones **no comprobadas** para tratar las situaciones excepcionales en el proyecto.

2) **(1,25 puntos)** Se debe modificar la clase `Ubicacion` para incluir un atributo que almacene la distancia en metros a la salida (de tipo `int`) y la siguiente funcionalidad:

   a) La distancia debe ser siempre mayor o igual a 0. Modifíquese adecuadamente el constructor de la clase para garantizarlo y que, en caso contrario, se lance la correspondiente excepción del tipo `ProcesionesException`, con un mensaje adecuado.

   b) Un método para obtener la distancia de la ubicación a la salida:

   ```
   int getDistancia();
   ```

   y otro para cambiar dicha distancia, de forma que se lance una excepción del tipo `ProcesionesException` en caso de que se intente cambiar por una distancia negativa:

   ```
   void setDistancia(int nuevaDistancia);
   ```

   c) Dos ubicaciones son iguales cuando sus nombres coinciden, independientemente de mayúsculas o minúsculas.

   d) La representación textual de una ubicación vendrá dada por el nombre y, entre paréntesis y separados por una coma: el tiempo seguido de una comilla simple y la distancia seguido por "m". Por ejemplo:

   ```
   Larios(210', 1000m)
   ```

3) **(0,75 puntos)** Debe definirse una aplicación (clase distinguida) `PruebaUbicacion`, en el paquete por defecto, que pruebe la clase anterior, creando dos objetos de la clase `Ubicacion`, con los datos siguientes (nombre, minutos, metros):

   "Larios", 210, 1000

   "Molina Larios", 270, 1900

   Posteriormente, se mostrarán en consola ambas ubicaciones y un mensaje indicando si las ubicaciones son iguales o no.

   A continuación, debe crearse un tercer objeto de la clase `Ubicacion` con la denominación, tiempo y distancia proporcionados, en ese orden, en el argumento del `main`.

   Como es posible que los datos que se proporcionen en los argumentos no sean numéricos, sean insuficientes o sean negativos, deben capturarse las correspondientes excepciones, imprimiendo un mensaje sobre la variable `System.err`.

   Por ejemplo, si los argumentos que se pasan son:

   ```
   Alameda 140 -1500
   ```

   la salida debe ser algo como:

   ```
   Larios(210', 1000m)
   Molina Larios(270', 1900m)
   Las ubicaciones no son iguales
   La distancia desde la salida no puede ser negativa
   ```

4) **(5 puntos)** La clase `Procesion` pretende representar procesiones de una cofradía por la ciudad. Deberá incluir información privada sobre el nombre de la cofradía (de tipo `String`), y una lista representando el itinerario o recorrido con las ubicaciones que constituyen la procesión. En este caso, la clase incluirá:

a) Un constructor con dos argumentos: el nombre de la cofradía y el nombre de la ubicación de salida (en ese orden). Inicialmente, la lista de ubicaciones solo incluirá una ubicación correspondiente a la salida, con tiempo de recorrido 0 y distancia desde la salida de 0.

b) Un método para obtener el nombre de la cofradía que realiza la procesión:

```
String getCofradia()
```

c) Un método para añadir una ubicación a la procesión:

```
void agregarUbicacion(Ubicacion ubic)
```

En caso de que ya exista una ubicación en el itinerario con el mismo nombre, se sustituirá por la ubicación que se pasa como argumento. En otro caso, la ubicación se añadirá al final del itinerario.

d) Un método para añadir ubicaciones a una procesión:

```
void agregarUbicacion(String infoUbicacion)
```

cuyos datos se proporcionan en una cadena de caracteres, donde cada cadena contiene los datos de una ubicación con el siguiente formato:

```
Nombre Ubicación#Tiempo#Distancia
```

En caso de que el formato de los datos sea incorrecto, el tiempo o distancia no sean numéricos o sean negativos, esa cadena no se agregará como ubicación y se lanzará una excepción de tipo `ProcesionesException` con un mensaje distinto en cada caso, incluyendo los datos que han proporcionado el error. Por ejemplo, dependiendo del caso:

```
Formato incorrecto (faltan datos): Alameda#140
Formato incorrecto (dato no numérico): Alameda#I40#1500
Formato incorrecto (número negativo): Alameda#140#-1500
```

e) Un método para obtener una lista con los nombres de las ubicaciones que se encuentran a más de una distancia proporcionada en el argumento:

```
List<String> ubicacionesLejanas(int distancia)
```

Se devolverá una lista con los nombres de las ubicaciones en mayúsculas.

f) La representación textual de una procesión vendrá dada por el nombre de la cofradía, seguido del símbolo @ y seguido de la secuencia de ubicaciones separadas por " -> " y encerradas entre llaves. Por ejemplo:

```
Estudiantes @ { Casa Hermandad C/ Alcazabilla(0', 0m) -> Tribuna(250', 900m) -> Larios(270', 1000m) -> Alameda(300', 1500m) }
```

Utilícese `StringJoiner` o `StringBuilder`, según se considere.

5) **(2,75 puntos)** Con objeto de no tener que definir en la clase `Procesión` distintos métodos para seleccionar los nombres de las ubicaciones que cumplan determinado criterio (por ejemplo, el que se utiliza en el apartado (e) del ejercicio anterior), se deben definir (en el caso de los apartados (b), y (c) solo es necesario responder a uno de ellos):

a) La interfaz `FiltradoUbicaciones` incluirá un solo método:

```
List<String> seleccionar (List<Ubicacion> ubicaciones)
```

con la intención de devolver una lista con los nombres (en mayúscula) de las ubicaciones de la lista que se pasa como argumento que cumplan un determinado criterio. Ese criterio será definido por clases (como las de los apartados siguientes) que implementen la interfaz.

b) La clase `DistanciaMayor` implementará la interfaz anterior, de forma que incluirá:

i. Una variable de instancia (`int`) que almacenará una distancia de referencia.
ii. Un constructor que permita inicializar esa variable de instancia.
iii. La implementación del método `seleccionar`, que devuelva una lista con los nombres (en mayúscula) de las ubicaciones de la lista que se pasa como argumento y cuya distancia a la salida es mayor que la distancia de referencia.

c) La clase `TiempoMenor` implementará también la interfaz del apartado (a) de forma que incluirá:

   i. Una variable de instancia (`int`) que almacenará un tiempo de referencia.
   ii. Un constructor que permita inicializar ese tiempo de referencia.
   iii. La implementación del método `seleccionar`, que devuelva una lista con los nombres (en mayúscula) de las ubicaciones de la lista que se pasa como argumento y que tengan un tiempo de recorrido menor que el establecido referencia.

d) Incluir en la clase `Procesion` el método:

```
List<String> seleccionarUbicaciones(FiltradoUbicaciones filtro)
```

que devuelva la lista con los nombres (en mayúscula) de ubicaciones que cumplan el criterio de selección que se pasa como argumento.

Se proporciona una clase de pruebas para incluir en el paquete por defecto, `PruebaProcesion`, para ilustrar el uso de las clases propuestas, y cuya salida debería ser:

```
Formato incorrecto (número negativo): Mármoles#60#-100
Formato incorrecto (faltan datos): Catedral#390
Estudiantes @ { Casa Hermandad C/ Alcazabilla(0', 0 m) -> Tribuna(250', 900 m) ->
Larios(270', 1000 m) -> Alameda(300', 1500 m) -> Molina Larios(330', 1900 m) ->
Torre Sur(360', 2100 m) }
Las ubicaciones con tiempo menor de 300 minutos son:
[CASA HERMANDAD C/ ALCAZABILLA, TRIBUNA, LARIOS, ALAMEDA]
Las ubicaciones con distancia mayor de 1000 metros son:
[LARIOS, ALAMEDA, MOLINA LARIOS, TORRE SUR]
```