DEPARTAMENTO DE LENGUAJES Y CIENCIAS DE LA COMPUTACIÓN	**PROGRAMACIÓN ORIENTADA A OBJETOS** **Septiembre 2017**	
APELLIDOS, Nombre	TITULACIÓN Y GRUPO	
	MÁQUINA	

Notas para la realización del examen:

- El examen se almacenará en el directorio `C:\POO\SEP17`. En caso de que no exista deberá crearse, y si ya existiese, deberá borrarse todo su contenido antes de comenzar.
- Al inicio del contenido de cada fichero deberá indicarse el **nombre del alumno, titulación, grupo y código del equipo** que está utilizando.
- 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.
- El código del alumno debe compilar correctamente. Para ello, el alumno puede poner entre comentarios aquellas zonas del código que no consigan compilar correctamente.
- Los diferentes apartados tienen una determinada puntuación. Si un apartado no se sabe hacer, el alumno *no debe pararse en él indefinidamente*. Puede abordar otros.
- **Está permitido**: Consultar la API y la Guía rápida de la API.
- **No está permitido**: Utilizar otra documentación electrónica o impresa. Intercambiar documentación con otros compañeros. Utilizar soportes de almacenamiento.
- Una vez terminado el examen, se debe subir un fichero comprimido de la carpeta `src` del proyecto a la tarea del campus virtual adecuada para ello.

En este examen se va a desarrollar un conjunto de clases (ver diagrama en el *Campus Virtual*) que nos permitirán controlar los mensajes de una red social. Para ello, se creará un proyecto `prSept17` con las clases siguientes en el paquete `prSept17`.

La excepción `AppException` (0.25 pts.)

Defina la clase `AppException` (del paquete `prSept17`) como una **excepción no comprobada** para notificar las situaciones excepcionales que se produzcan.

La clase `Mensaje` (1.00 pts.)

La clase `Mensaje` (del paquete `prSept17`) contiene información sobre un determinado mensaje, tal como el orden de secuencia, el nombre del emisor, el nombre del receptor, y el texto del mensaje (todas las variables son privadas).

- `Mensaje(e:String, r:String, txt:String)`
  El constructor recibe, en el siguiente orden, el nombre del emisor, el nombre del receptor, y el texto del mensaje. Si los parámetros son erróneos (algún nombre o texto del mensaje *nulo* o de *longitud cero*), entonces lanza la excepción `AppException`. En otro caso almacena los valores recibidos y registra el orden de secuencia del mensaje. Para ello asigna a la variable de instancia `secuencia` el valor actual de la variable de clase `cntSecuencia`, y posteriormente incrementa en `1` el valor de la variable de clase `cntSecuencia`. Nótese que el valor inicial de la variable de clase `cntSecuencia` es el valor `1`.
- `getEmisor(): String`
  Devuelve el nombre del emisor del mensaje almacenado en el objeto.
- `getReceptor(): String`
  Devuelve el nombre del receptor del mensaje almacenado en el objeto.
- `getTexto(): String`
  Devuelve el contenido del texto del mensaje almacenado en el objeto.
- `toString(): String`
  Devuelve la representación textual del objeto, en el siguiente formato (incluyendo los paréntesis):
  `(emisor; receptor; texto del mensaje)`
- Se considera que dos mensajes son *iguales* si el orden de secuencia de ambos mensajes es igual, y el nombre del emisor y el nombre del receptor son iguales (ignorando mayúsculas y minúsculas) en ambos mensajes.

- Los objetos de la clase `Mensaje` proporcionan el **orden natural** comparando el orden de la variable secuencia, en caso de igualdad de secuencia, se compara el nombre del emisor, y en caso de igualdad de emisor, se compara el nombre del receptor (ignorando mayúsculas y minúsculas tanto en la comparación del emisor como del receptor) de ambos objetos.

La interfaz `Filtro` (0.25 pts.)

La interfaz `Filtro` (del paquete `prSept17`) define un método que permite seleccionar un determinado mensaje dependiendo de diversas circunstancias.

- `select(m:Mensaje): boolean`
  Devuelve `true` si el mensaje recibido como parámetro debe ser seleccionado por este filtro. En otro caso devuelve `false`.

La clase `FiltroReceptor` (0.25 pts.)

La clase `FiltroReceptor` (del paquete `prSept17`) implementa la interfaz `Filtro`, y contiene información sobre el nombre del receptor de los mensajes que serán seleccionados. Permite seleccionar un determinado mensaje si el receptor del mensaje es un determinado usuario registrado, especificado durante la creación del objeto `FiltroReceptor` (todas las variables son privadas).

- `FiltroReceptor(r:String)`
  Construye un nuevo objeto y almacena el nombre del receptor del mensaje a seleccionar, recibido como parámetro.
- `select(m:Mensaje): boolean [@Redefinición]`
  Devuelve `true` si el receptor del mensaje recibido como parámetro es igual (ignorado mayúsculas y minúsculas) al receptor almacenado en este objeto durante su construcción. En otro caso devuelve `false`.

La clase `FiltroTexto` (1.00 pts.)

La clase `FiltroTexto` (del paquete `prSept17`) implementa la interfaz `Filtro`, y contiene información sobre el conjunto de claves de los mensajes que serán seleccionados. Permite seleccionar un determinado mensaje si en el texto del mensaje aparece alguna de las claves registradas, especificadas durante la creación del objeto `FiltroTexto` (todas las variables son privadas).

- `FiltroTexto(c:Set<String>)`
  Construye un nuevo objeto y almacena (en mayúsculas) las claves recibidas como parámetro.
- `select(m:Mensaje): boolean [@Redefinición]`
  Devuelve `true` si el texto del mensaje recibido como parámetro contiene alguna de las claves (ignorando mayúsculas y minúsculas) almacenadas en este objeto durante su construcción. En otro caso devuelve `false`. Nótese que las claves a buscar pueden aparecer en el texto mezcladas con cualquier otro texto sin delimitar.

La clase `Cuenta` (2.25 pts.)

La clase `Cuenta` (del paquete `prSept17`) contiene información sobre una determinada cuenta de usuario, tal como el nombre del usuario y un conjunto ordenado (según el orden natural) de los mensajes enviados por el mismo usuario de la cuenta (todas las variables son privadas).

- `Cuenta(usr:String)`
  Si los parámetros son erróneos (usuario nulo o de longitud cero), entonces lanza la excepción `AppException`. En otro caso, construye el objeto con el nombre del usuario recibido como parámetro, y el conjunto de mensajes vacío.
- `getUsuario(): String`
  Devuelve el nombre del usuario del objeto actual.
- `addMsj(receptor:String, txt:String): void`
  Crea un nuevo objeto `Mensaje` donde el emisor es el usuario de esta cuenta, y el receptor y el texto del mensaje provienen de los parámetros recibidos. Finalmente añade el mensaje creado al conjunto de mensajes de esta cuenta.
- `getMsjs(flt:Filtro): SortedSet<Mensaje>`
  Si el filtro recibido como parámetro es nulo, entonces devuelve un **nuevo** conjunto ordenado (según el orden natural) con todos los mensajes almacenados en esta cuenta. En otro caso, devuelve un **nuevo** conjunto ordenado (según el orden natural) con todos los mensajes de esta cuenta seleccionados al aplicarles el objeto `Filtro` recibido como parámetro, considerando que un mensaje será seleccionado si el método `select` del filtro devuelve `true` al ser aplicado sobre el mensaje.

- `toString(): String [@Redefinición]`
  Devuelve la representación textual del objeto, según el formato del siguiente ejemplo (sin considerar los saltos de línea):

```
[(Emisor1; Receptor3; este es el mensaje 1),
 (Emisor1; Receptor2; este es el mensaje 2),
 (Emisor1; Receptor1; este es el mensaje 4)]
```

La clase `CuentaModerada` (1.00 pts.)

La clase `CuentaModerada` (del paquete `prSept17`) es una **subclase** de la clase `Cuenta`, y redefine su comportamiento para bloquear mensajes que contengan insultos y palabras malsonantes. Para ello contiene un conjunto de palabras clave a bloquear, y redefine el método `addMsj` de tal forma que un determinado mensaje será añadido si su texto **no** contiene ninguna de las palabras registradas (todas las variables son privadas).

- `CuentaModerada(usr:String, c:Set<String>)`
  Si los parámetros son erróneos (usuario nulo o de longitud cero o conjunto nulo), entonces lanza la excepción `AppException`. En otro caso, construye el objeto con el nombre del usuario recibido como parámetro, el conjunto de mensajes vacío, y almacena (en mayúsculas) las palabras a bloquear recibidas como parámetros.
- `addMsj(receptor:String, txt:String): void [@Redefinición]`
  Si el texto del mensaje contiene alguna de las palabras del conjunto a bloquear, entonces lanza la excepción `AppException`. En otro caso, crea y añade un nuevo mensaje según lo especificado en el mismo método de la super clase.

La clase `RedSocial` (3.00 pts.)

La clase `RedSocial` (del paquete `prSept17`) permite gestionar múltiples cuentas de usuario, así como el usuario actualmente activo. Para ello, contiene una correspondencia ordenada que asocia nombres de usuario (almacenados en mayúsculas) con la cuenta de dicho usuario. Además, también contiene el nombre del usuario actualmente activo en el sistema (todas las variables son privadas).

- `RedSocial()`
  Construye el objeto con la correspondencia de cuentas vacía, y el nombre del usuario activo nulo. Además, crea una nueva cuenta con el usuario `"ADMIN"` y la añade a la correspondencia.
- `login(usr:String): void`
  Si no existe ninguna cuenta correspondiente al parámetro `usr` (en mayúsculas), entonces lanza la excepción `AppException`. En otro caso, almacena el nombre (en mayúsculas) recibido como parámetro en la variable de instancia correspondiente al usuario activo.
- `logout(): void`
  Pone a nulo la variable de instancia correspondiente al usuario activo.
- `crearCuenta(usr:String): void`
  Si el usuario activo no es `"ADMIN"`, o si ya existe alguna cuenta correspondiente al parámetro `usr` (en mayúsculas), entonces lanza la excepción `AppException`. En otro caso, crea una nueva cuenta con el usuario recibido como parámetro (en mayúsculas) y la añade a la correspondencia de cuentas.
- `addMsj(receptor:String, txt:String): void`
  Crea y añade un nuevo mensaje en la cuenta del usuario activo con los parámetros recibidos (invoca a `addMsj` del objeto `Cuenta` correspondiente).
- `getMsjsCon(usuario:String): SortedSet<Mensaje>`
  Si no existe ninguna cuenta correspondiente al parámetro `usuario` (en mayúsculas), entonces lanza la excepción `AppException`. En otro caso, devuelve un **nuevo** conjunto ordenado (según el orden natural) con todos los mensajes de la cuenta correspondiente al *usuario activo*, cuyo receptor es igual (ignorando mayúsculas y minúsculas) al parámetro `usuario`, junto a todos los mensajes de la cuenta correspondiente al parámetro `usuario`, cuyo receptor es igual (ignorando mayúsculas y minúsculas) al *usuario activo*. Para ello, debe utilizar objetos que implementen la interfaz `Filtro`.
- `getMsjsClaves(c:Set<String>): SortedSet<Mensaje>`
  Devuelve un **nuevo** conjunto ordenado (según el orden natural) con todos los mensajes de la cuenta correspondiente al usuario activo que contengan en su texto alguna de las palabras especificadas en el conjunto recibido como parámetro. Para ello, debe utilizar un objeto que implemente la interfaz `Filtro`.
- `toString(): String [@Redefinición]`
  Devuelve la representación textual del objeto, utilizando la clase `StringBuilder`, según el formato del siguiente ejemplo (sin considerar los saltos de línea):

```
{ EMISOR1:
[(Emisor1; Receptor3; este es el mensaje 1),
 (Emisor1; Receptor1; este es el mensaje 2)]
EMISOR2:
[(Emisor2; Receptor3; este es el mensaje 1),
 (Emisor2; Receptor1; este es el mensaje 2)]}
```

- `crearCuentaModerada(usr:String, c:Set<String>): void`
  Si el usuario activo no es `"ADMIN"`, o si ya existe alguna cuenta correspondiente al parámetro `usr` (en mayúsculas), entonces lanza la excepción `AppException`. En otro caso, crea una nueva cuenta moderada (`CuentaModerada`) con el usuario y conjunto de palabras a bloquear recibidos como parámetros y la añade a la correspondencia de cuentas.
- `cargarDeFichero(n:String): void`
  Si se produce alguna excepción de entrada/salida, será notificada como una `IOException`. Además, cualquier otro error que se produzca será notificado como una `AppException`. Lee los mensajes del fichero cuyo nombre se recibe como parámetro, y creará las cuentas necesarias y mensajes para almacenar la infomación adecuadamente. En el fichero, cada línea contiene la información de un mensaje, donde cada mensaje contiene el emisor, el receptor y el texto del mensaje separados por los delimitadores: `" *[;] *"` (punto-y-coma, precedido y seguido por secuencias de espacios en blanco opcionales):

  ```
  Usuario1 ; Usuario3 ; este es el mensaje 1
  Usuario2 ; Usuario1 ; este es el mensaje 2
  ```
- `guardarEnFichero(n:String): void`
  Si se produce alguna excepción de entrada/salida, será notificada como una `IOException`. Además, cualquier otro error que se produzca será notificado como una `AppException`. Escribe en el fichero cuyo nombre se recibe como parámetro todos los mensajes de todas las cuentas, **entremezclados y ordenados** según su orden natural, según el formato del método `cargarDeFichero`.

La clase `Controlador` (1.00 pts.)

La clase `Controlador` (del paquete `prSept17`) interactúa con la `RedSocial` como modelo y con un objeto que implementa la interfaz `Vista` proporcionada, y tiene el siguiente comportamiento:

- El constructor recibe tanto a la vista como al modelo (en el orden especificado) y los almacena en sendas variables de instancia. Además, desactiva la interacción en la vista.
- Cuando recibe los siguientes eventos, muestra el comportamiento especificado a continuación, considerando que si se produce cualquier error, será notificado mediante el mensaje `"Operacion Erronea"` (sin tildes), y si la operación es correcta, será notificado mediante un mensaje `"Operacion Correcta"` (sin tildes), ambos mensajes en la *línea de estado* de la vista:
  - `Vista.LOGIN`: accede en la vista al nombre del usuario que se quiere conectar (`getLogin`), e intenta la conexión (`login`) de ese usuario en la red social. Si la conexión es correcta, entonces muestra en la vista el nombre del usuario conectado (`setLogin`) y activa la interacción en la vista.
  - `Vista.LOGOUT`: realiza la desconexión del usuario en el modelo (`logout`), limpia de la vista el nombre del usurio conectado (`setLogin`), limpia los mensajes (`cleanMensajes`) y desactiva la interacción en la vista.
  - `Vista.CREARCNT`: accede en la vista tanto al nombre del nuevo usuario como al conjunto de claves de bloqueo. Si el conjunto de claves de bloqueo **no** está vacío, entonces crea una *cuenta moderada* en el modelo con el nombre del nuevo usuario y el conjunto de claves de bloqueo. En otro caso, crea una *cuenta normal* en el modelo con el nombre del nuevo usuario.
  - `Vista.ENVMSJ`: accede en la vista tanto al nombre del receptor como al texto del mensaje y envía un *mensaje* en el modelo con los datos recolectados de la vista.
  - `Vista.LEERMSJ`: accede en la vista tanto al nombre del usuario como al conjunto de claves de filtro de mensajes. Si el nombre del usuario **no** está vacío (longitud mayor que cero), entonces accede en el modelo a los mensajes intercambiados con dicho usuario. En otro caso, si el conjunto de claves de filtro **no** está vacío, entonces accede en el modelo a los mensajes del usuario activo con el conjunto de claves de filtro. En ambos casos limpia y muestra los mensajes accedidos en la vista. En otro caso, la operación es errónea.
  - `Vista.CARGARFICH`: accede en la vista al nombre del fichero, y carga del fichero del nombre especificado en el modelo.
  - `Vista.GUARDARFICH`: accede en la vista al nombre del fichero, y guarda en el fichero del nombre especificado en el modelo.
