Práctica 4: programación funcional en Scala; objetos

Nuevas tecnologías de la programación

Contenido:

1 Objetivos

En esta práctica se trata de trabajar con la clase clase **ConjuntoTweet** y sus clases derivadas. Todas ellas tiene por objeto almacenar colecciones de objetos de la clase **Tweet**. Esta última clase ya está implementada y contiene información sobre mensajes de este tipo: usuario, texto y número de retweets.

2 Clases de interés

La clase **ConjuntoTweet** es abstracta y cuenta con dos subclases: **ConjuntoTweetVacio** y **ConjuntoTweetNoVacio**. Esta última permite almacenar los mensajes en forma de árbol binario, estando los mensajes ordenados lexicográficamente. Es decir, el mensaje almacenado en el objeto de la raíz es mayor (lexicográficamente hablando) que todos los almacenados a la izquierda y menor que todos los almacenados a la derecha. Se trata, en cualquier caso, de clases inmutables. Es decir, las operaciones sobre ellas deben generar nuevos objetos.

Entre todas ellas se debe aportar la funcionalidad que se indica a continuación.

2.1 Filtrado

Esta operación permite obtener los mensajes (objetos de la clase **Tweet**) del conjunto que cumplan una determinada condición expresada en forma de predicado. El resultado de la operación será un nuevo conjunto de mensajes que contiene únicamente a los mensajes que cumplen la condición expresada en el predicado. Por ejemplo:

```
1 conjunto.filtrar(mensaje => mensaje.retweets > 10)
```

generaría un nuevo conjunto con todos los mensajes almacenados en el objeto **conjunto** sobre el que se hace la llamada que cumplan que el número de retweets sea mayor de 10.

Para implementar esta función se debe implementar un método auxiliar, llamado **filtrar0**, que usa como argumento un conjunto auxiliar en que se va acumulando el resultado de la operación. La declaración de ambos métodos es la siguiente: *Estas son de conjunto tweet*

IMPLEMENTAR

```
1 def filtrar(predicado: Tweet => Boolean): ConjuntoTweet
2 def filtrar0(predicado: Tweet => Boolean, conjunto: ConjuntoTweet): ConjuntoTweet
```

Se le pasa conjunto para poder hacerlo recursivo

Usando el método auxiliar, la implementación de **filtrar** debería ser inmediata. Por su parte, **filtrar0** debería quedar abstracto en la clase base e implementarse en las clases derivadas.

Tenemos que decidir si estos métodos se quedan en la clase abstracta o no. Hay que ver si bajamos una a clases inferiores

2.2 Union

La unión permite agregar conjuntos de mensajes. La declaración del método es:

```
1 def union(otro: ConjuntoTweet): ConjuntoTweet
```

La unión hay que decidir si la dejamos como abstracta en ConjuntoTweet o la metemos dentro de las específicas

Recibe como argumento otro conjunto (**otro**) y genera un nuevo conjunto con todos los mensajes contenidos en el objeto en que se hace la llamada y en el objeto pasado como argumento. Aquí debes decidir dónde se ubicará la declaración de esta operación (¿debería ser abstracto en la clase **ConjuntoTweet** o puede implementarse en ella?).

Aquí pueden ser de interés algunos métodos de utilidad ya implementados en las clases, como **head** y **tail**, que se comportan de forma análoga a como lo harían sobre listas (devolviendo respectivamente el primer mensaje de un conjunto y todos menos el primero). Se ha usado aquí el nombre inglés para estos métodos para remarcar la coincidencia con los métodos ofertados por la clase **List**.

2.3 Intersección

La intersección permite obtener conjuntos de mensajes comunes a dos conjuntos: el usado para hacer la llamada al método y el pasado como argumento. La declaración del método será:

```
1 def interseccion(otro : ConjuntoTweet) : ConjuntoTweet
```

Aquí también hay que decidir cuál es el lugar adecuado para incluir su implementación: ¿se deja abstracto en la clase base o se implementa en ella?

2.4 Ordenación de mensajes por influencia

Se considera que un mensaje es más influyente que otro si cuenta con mayor número de retweets. El objetivo de esta operación es implementar el método

- Buscar mensaje con mínimo # RTs (buscarMinimo)
- Agregarlo a Tendencia (Ini. TendenciaVacía)
- Borrar sms mínimo del conjunto (Con Eliminar)
- Repetir el proceso hasta procesar todos los sms

```
1 def ordenacionAscendentePorRetweet: Tendencia
```

que produce una secuencia lineal de mensajes (como una instancia de la clase **Tendencia**), ordenados por su influencia, de menor a mayor. Esta clase (y sus derivadas **TendenciaVacia** y **TendenciaNoVacia**) ya se ofrecen implementadas de forma completa y sólo hay que investigar sobre ellas para ver cómo pueden usarse.

Este método presenta un patrón común en operaciones de transformación de estructuras de datos. Mientras se recorre una colección (en este caso un objeto de la clase **ConjuntoTweet**) se va creando una segunda estructura (un objeto de la clase **Tendencia**). Se debería comenzar con un objeto de la clase **TendenciaVacia** y:

- encontrar el mensaje con el menor número de retweets en el conjunto de mensajes.
- borrado del mensaje , usando el método **eliminar**, ya implementado en las clases **ConjuntoTweet** y derivadas. Observad que este método respeta el principio de inmutabilidad de las clases.
- agregar el mensaje al objeto que representa la tendencia (que, al final de la agregación, ya no será vacía).
- continuar repetiendo este proceso hasta tratar todos los mensajes.

También está implementada la funcionalidad para encontrar el mensaje con el menor número de retweets de un conjunto (método **buscarMinimo** y método auxiliar **buscarMinimo0**). Observad que los métodos auxiliares emplean siempre la estrategia del uso del acumulador para facilitar la construcción recursiva de colecciones.

2.5 Número de mensajes en la colección (ConjuntoTweet)

Este método permite conocer cuántos mensajes forman parte de un conjunto. La declaración del método será:

```
1 def numeroMensajes : Integer
```

Su implementación se incluirá en la clase (o clases) que consideréis oportuna (oportunas).

2.6 Número de mensajes en la tendencia (Tendencia)

Se trata de dar aquí la implementación del método **length** (para mantener la coherencia con los nombres en inglés **head**, **tail**, etc...). La declaración es:

IMPLEMENTAR
```
1 def length : Integer (Decidir si lo metemos en la clase abstracta o derivadas)
```

Como ocurre en otros métodos queda a vuestra elección el lugar de implementación del método (en **Tendencia** o en las clases derivadas).

2.7 Funcionalidad ya implementada

Observad que algunas clases ya ofrecen funcionalidad implementada. Está debidamente marcada en los archivos de código fuente y, aunque no debe modificarse, sí que conviene estudiar la forma de implementación para obtener ideas sobre la forma de implementar los métodos pedidos.

3 Uniendo todo

El último paso de la práctica consiste en detectar los mensajes más influyentes de un conjunto de mensajes. El objeto **DatosMensajes** contiene varios cientos de mensajes que usaremos para probar. El objeto **LectorTweets** contiene métodos de utilidad para leerlos y procesarlos. En concreto, el dato miembro **mensajes** de este objeto almacena todos los mensajes disponibles.

Por su parte, en el archivo **Main.scala** se incluyen dos objetos: el primero se denomina **TerminosGoogleApple** y ofrece una lista de términos de interés asociados a **google** y **apple**. Estas listas deben usarse para recuperar de la colección completa de mensajes aquellos que contienen alguno de sus términos. Así, debemos obtener

```
1 mensajesGoogle : ConjuntoTweet
2 mensajesApple : ConjuntoTweet
```

a partir del conjunto completo de mensajes. Para ello debemos implementar un método auxiliar en el objeto **LectorTweets**, con la siguiente declaración:

```
1 def obtenerConjuntoConTerminos(terminos : List[String]) : ConjuntoTweet
```

Observad que para implementar este método podríamos usar también el patrón de uso de método auxiliar para ir acumulando la colección creada a medida que se van considerando los términos de la lista.

A partir de estos dos conjuntos, tras unirlos y ordenarlos por influencia, se obtendrá un objeto de la clase **tendencia**:

```
1 val tendencia: Tendencia
```

Por su parte, en el objeto **Main** se incluirán las sentencias necesarias para informar al usuario de:

- número de mensajes en cada conjunto (**mensajesGoogle** y **mensajesApple**)
- número de mensajes en el objeto que representa la tendencia
- número de mensajes comunes entre ambos conjuntos
- orden de influencia de los mensajes comunes
- número máximo y mínimo de retweets de los mensajes comunes

4 Material a entregar

Al final de la realización de la práctica se entregará un archivo comprimido con el contenido completo de la práctica, tal y como se integra en el proyecto con el entorno de desarrollo que hayáis usado. Se incluirá también un pequeño documento indicando el entorno de desarrollo y una breve valoración de la práctica (si los conceptos vistos son novedosos, si os ha parecido de interés, problemas encontrados, etc) en tres o cuatro líneas.

Pueden realizarse los casos de prueba que se consideren necesarios para ir comprobando que el código funciona de forma correcta.

La fecha de entrega se fijará más adelante. La entrega se hará mediante la plataforma **PRADO**.

- Hay un Object (DatosMensajes): Aquí está la colección de Tweets.
- LectorTweets: parsea los mensajes de arriba y va creando las clases.
- Dentro de esta el que se encarga de parsear es AnalizadorTweets.
- Hay distintos métodos auxiliares para leer los tweets.
- IMPLEMENTAR obtenerConjuntoConTerminos(terminos:List[String]): ConjuntoTweet
- La val mensajes contiene todos los mensajes del archivo

En la clase Main:
val google = List("android", "Android", "Galaxy"....), esto se le pasa a obtenerConjuntoConTerminos
val apple = List("otras cosas de apple")

Dentro del main tenemos que hacer:
- mensajesGoogle (Con los mensajes que hablan de Google)
- lo mismo para Apple (mensajesApple), ambos de tipo ConjuntoTweet
- val Tendencia(mensajesGoogle, mensajesApple), les hacemos la unión y a partir de ellos creamos la tendencia.
menApple
- Número de mensajes de la tendencia.
- Número de mensajes comunes.
- Orden de influencia de los mensajes comunes (Crear una Tendencia para ver cual es dicha Tendencia)
- Máx y Min # RTs de mensajes
- Máx y Min # RTs de toda la colección.

Para el método de filtrar:
Se basa en el uso de filtrar0:

```
def filtrar(predicado:Tweet => Boolean):ConjuntoTweet =
    filtrar0(predicado, new ConjuntoTweetVacio)

def filtrar0(predicado: Tweet => Boolean, conjuntoActual:ConjuntoTweet) =  (Sobre conjunto Base no vacío)
    // Hay que ver si el predicado se cumple sobre la raiz, en ese caso:
    val conjuntoNuevo =
        if (predicado(raiz) && !conjuntoActual.contiene(raiz))
            conjuntoActual.incluir(raiz)
        else
            // Si no tenemos que incluir la raiz devolvemos el conjunto actual
            conjuntoActual
    // Ahora tenemos que seguir haciendo un filtrado, ya sea con la raíz o sin ella.
    derecha.filtrar0( predicado , izquierda.filtrar0(predicado, conjuntoNuevo))
```