Tecnologías y Paradigmas de la Programación. Examen Junio 2019

Ejercicio 1 (2 puntos):

Se quiere modelar un software que gestione **información de autenticación** de usuarios con los siguientes requisitos:

- La información de autenticación está compuesta por un *nombre de usuario* y una *clave*. Si bien el nombre es un dato que puede consultarse y modificarse por parte de usuarios externos, la clave **no será visible** una vez construido un objeto que contenga información de autenticación.
- La clase o clases que guarden la información de autenticación soportarán las siguientes funcionalidades:
  - Un método `Validate` que admita un nombre de usuario y una clave. Devolverá `true` si el nombre de usuario y la clave pasada son los que contienen la información de autenticación. No obstante, por motivos de seguridad las claves nunca se guardan en texto plano, sino **hasheadas** con un algoritmo. Por tanto, este método debe **hashear** la clave pasada antes de hacer la comparación. Se debe desarrollar funcionalidad para soportar **dos métodos de hashing**: MD5 y SHA1, cuyas funciones de hasheo **se entregan** en el fichero `Utils.cs`.
  - Un método `Validate` que admita un nombre de usuario solamente. Devolverá `true` si el nombre de usuario es el que contiene la información de autenticación.
  - Un método `ToString` que solo muestre el nombre de usuario y **NO** la clave.
  - A la hora de modificar el nombre de usuario, **no se admiten** nombres de usuarios nulos ni inferiores a 3 caracteres ni superiores a 15.

Usando todas las técnicas de POO que considere más adecuadas, cree las clases que soporten estos requisitos.

Ejercicio 2 (2 puntos):

- Crear una función de orden superior que reciba como primer parámetro otra función (que no recibe parámetros y devuelve un `string`) y un tamaño como segundo parámetro. Esta función de orden superior deberá devolver un enumerable del tamaño indicado como segundo parámetro, conteniendo `string` generados por la función pasada como primero. Probar la función con la función de generación de `string` aleatorios suministrada en `Utils.cs` que considere más adecuada *(1 punto)*
- Hacer **una versión lazy** de la función anterior que devuelva un `IEnumerable` potencialmente infinito. El tamaño pasado como segundo parámetro indicará en este caso **el tamaño máximo de los `string` que contiene** dicho `IEnumerable` devuelto. Probar la función con la función de generación de `string` aleatorios suministrada en `Utils.cs` que considere más adecuada *(1 punto)*.

Este ejercicio debe entregarse con las dos funciones, es decir, una no sustituye a la otra.

Ejercicio 3 (3 puntos):

- Mediante `Linq`, generar un array de 100 `string` aleatorios de tamaño máximo 50 *(0,5 puntos)*. Probarlo con la función de generación de `string` aleatorios suministrada en `Utils.cs` que considere más adecuada.
- A partir del array de `string` creado en el apartado anterior o bien de uno creado manualmente, seleccionar una posición al azar de este. Crear un objeto de la clase del ejercicio 1 capaz de guardar **claves hasheadas en `MD5`** cuyo nombre de usuario sea "tpp" y la clave sea el `string` en esa posición. Hecho esto, y mediante `Linq`, contar cuantas veces aparece la clave del usuario en el array de `string` usado *(1 punto)*.
- Mediante `Linq` y un enumerable de 200 claves de hasta 500 caracteres de tamaño generado a partir de las funciones suministradas en `Util.cs`, obtener todas las claves menores de 150 caracteres, convertirlas a mayúsculas y, finalmente, concatenarlas todas separándolas por `';'`. Las 3 operaciones pueden hacerse concatenadas o por separado, a *0,5 puntos cada una*.

Ejercicio 4 (2 puntos):

Crear un array de 1000 `string` aleatorios y otro array de 10 objetos de la clase creada en el ejercicio 1 con la capacidad de guardar **claves hasheadas en SHA1**, cuyas claves estén inicializadas con posiciones aleatorias de este array de `string` (los nombres de usuario se pueden suministrar manualmente o aleatorios con las funciones de `Util.cs` que considere oportunas).

Mediante **un esquema *Master-Worker***, hacer lo siguiente:

- Pasarle el array de usuarios generado
- Pasarle el array de `string` aleatorios usado para dar valor a sus claves
- Hacer que cada *Worker* busque en qué posición se encuentra la clave de un usuario concreto en el trozo de array de claves que se le ha asignado. Por tanto, cada *Worker* tendrá asignado un usuario solamente y se deben crear tantos *Worker* como usuarios introducidos (en este ejemplo son 10)
- Si la encuentra, cada *Worker* debe introducir el usuario en un diccionario que guarde {posición: objeto con los datos del usuario}. Este diccionario **está compartido** por todos los *Worker*.
- Si un *Worker* encuentra la posición de la clave del usuario que tiene asignado, **termina su procesamiento**.
- El *Master* devuelve el diccionario al final.

NOTA: Como ayuda, **se entrega un esquema Máster-Worker incompleto**, idéntico al que se vio en el módulo de prácticas correspondiente.

Conteste en un comentario del programa a la siguiente pregunta: ¿Cree que un esquema Master-Worker como éste es adecuado para encontrar las posiciones de todas las claves de todos los usuarios? ¿Por qué?

Ejercicio 5 (1 punto):

Hacer el ejercicio anterior **usando *TPL***, pero asegurándose de que se encuentran las posiciones de todas las claves.