Programación orientada a objetos. Primer ejercicio.

25 de marzo de 2008.

Mira el siguiente programa compuesto por los archivos Program.cs y Downloader.cs. El programa muestra en la consola el contenido de un recurso -un archivo- de tu disco local pero puedes modificarlo fácilmente para mostrar el contenido de cualquier recurso local o remoto.

¡Prueba armar un proyecto con esos archivos en tu editor favorito y descarga páginas de Internet! Busca en el código de la clase Program las pistas para hacerlo.

Program.cs:

```
using System;
using System.IO;
using System.Reflection;

namespace ExerciseOne
{
   /// <summary>
   /// Pequeño programa para probar el funcionamiento de la clase Downloader.
   /// </summary>
   public class Program
   {
      /// <summary>
      /// Punto de entrada
      /// </summary>
      public static void Main()
      {
         // Para uniformizar la entrega del obligatorio les pedimos que usen un archivo
         // llamado "archivo.txt" que debe estar en el mismo directorio que el programa.
         // Para facilitar el desarrollo desde el entorno integrado de Visual Studio 2005
         // Express Edition, pueden agregar un nuevo elemento del tipo archivo de texto y
         // llamarlo "archivo.txt" a su solución; luego cambien la propiedad
         // "CopyToOutputDirectory de ese nuevo elemento a "Copy always": con esto podrán
         // modificar el archivo desde el entorno integrado y asegurarse que al depurar la
         // versión más reciente del archivo se copia al directorio desde donde se ejecuta el
         // programa.

         const String fileName = "archivo.txt";

         String path = Path.Combine(
           Path.GetDirectoryName(Assembly.GetExecutingAssembly().Location), fileName);
         UriBuilder builder = new UriBuilder("file", "", 0, path);
         String uri = builder.Uri.ToString();

         // Creamos un nuevo descargador pasándole una ubicación.
         Downloader downloader = new Downloader(uri);

         // Pedimos al descargador que descargue el contenido
```

```
            string content;
            content = downloader.Download();

            // Imprimimos el contenido en la consola y esperamos una tecla para terminar
            Console.WriteLine(content);
            Console.ReadKey();
        }
    }
}
```

Downloader.cs:

```
using System;
using System.IO;
using System.Net;
namespace ExerciseOne {
    /// <summary>
    /// Descarga archivos de una ubicación de la forma "http://server/directory/file" o
    /// "file:///drive:/directory/file"
    /// </summary>
    public class Downloader {
        private string url;
        /// <summary>
        /// La ubicación de la cual descargar
        /// </summary>
        public string Url { get { return url; } set { url = value;  } }

        /// <summary>
        /// Crea una nueva instancia asignando la ubicación de la cual descargar
        /// </summary>
        /// <param name="url"></param>
        public Downloader(string url) {
            this.url = url;
        }

        /// <summary>
        /// Descarga contenido de la ubicación de la cual descargar
        /// </summary>
        /// <returns>Retorna el contenido descargado</returns>
        public string Download() {
            // Creamos una nueva solicitud para el recurso especificado por la URL recibida
            WebRequest request = WebRequest.Create(url);

            // Asignamos las credenciales predeterminadas por si el servidor las pide
            request.Credentials = CredentialCache.DefaultCredentials;

            // Obtenemos la respuesta
            WebResponse response = request.GetResponse();
```

```
            // Obtenemos la stream con el contenido retornado por el servidor
            Stream stream = response.GetResponseStream();

            // Abrimos la stream con un lector para accederla más fácilmente
            StreamReader reader = new StreamReader(stream);

            // Leemos el contenido
            string result = reader.ReadToEnd();

            // Limpiamos cerrando lo que abrimos
            reader.Close();
            stream.Close();
            response.Close();
            return result;
        }
    }
}
```

Parte 1

Si has visto alguna vez el código HTML de una página web, habrás visto que se organiza mediante *tags*[1]. A grandes rasgos un *tag* tiene la siguiente forma:

```
<nombreTag ([claveAtributo="valorAtributo"] …)>[Contenido]</nombreTag>
```

- `nombreTag` es el **nombre del tag**
- los **atributos** son una lista de tuplas **clave**/**valor**, de forma que solo pude haber un valor asociado a una clave para un *tag* determinado. Un tag puede tener de `0` a `n` atributos. Los valores del atributo van siempre entre comillas dobles.
- El **contenido** puede ser cualquier texto.
- Luego del contenido, un tag siempre se cierra mediante una entrada de la forma `</nombreTag>`
- Opcionalmente, si no hay contenido, un tag puede estar cerrado "**en el lugar**" (in place) mediante la siguiente sintaxis:

```
<nombreTag ([claveAtributo="valorAtributo"] …) />
```

- Se permiten cualquier cantidad de espacios antes del nombre del *tag*, entre el nombre del tag y el final del tag o primer atributo, entre un atributo y el símbolo de =, entre el símbolo de = y el valor del atributo y entre éste y el final del tag o el próximo atributo.
- Se dice que el tag esta **mal formado** cuando no respeta las reglas de sintaxis, por ejemplo:,no se cierra, tiene múltiples atributos con la misma clave, etc.

Ejemplos

`<b>Negrita</b>`

Un tag de nombre `b`, sin atributos, cuyo contenido es `Negrita`.

1Marcas

`<br/>`

Un tag de nombre `br`, sin atributos, sin contenido.

`<font color="red">Rojo</font>`

Un tag de nombre `font`, con un atributo con clave `color` y valor `red`, con contenido `Rojo`

`<input type="text" name="nombre"/>`

Un tag de nombre input, con un atributo con clave `type` y valor `text` y un atributo `name` con valor `nombre`.

Los siguientes son tags mal formados:

`<input type="text" type="text"/>`

Mal formado, clave de atributo repetido.

`<input type="text" name="nombre">`

Mal formado, el tag no se cierra.

`</input>`

Mal formado, el tag no se abre.

`<type="text" name="nombre"/>`

Mal formado, no tiene nombre.

Tu primera tarea es construir un programa que lea un archivo de texto e imprima el nombre de cada *tag* y sus atributos. Los *tags* deben ser impresos en el orden en el que aparecen en el archivo. El orden en que se impriman los atributos no es importante. Tampoco es necesario, para esta primera instancia, que evalúes si el *tag* se cierra correctamente o no.

Por ejemplo, si el archivo contiene:

```
<html><body><font color=”blue” size="3">Ingrese su nombre </font><input
type=”text” name=”nombre” maxlength=”8”/><br/><font size="2”>Máximo 8
caracteres</font></html></body>
```

(que, por cierto, es contenido HTML “válido” que puedes abrir con tu navegador)

El programa debe imprimir por consola:

```
html
body
font
color=blue
size=3
input
type=text
name=nombre
maxlength=8
br
font
```

```
size=2
```

Tu programa puede imprimir lo mismo incluso si el archivo contiene *tags* mal formados porque no se cierran, como por ejemplo:

```
<font color=”blue” size=”3”>Ingrese su nombre <input type=”text” name=”nombre” maxlength=”8”><br><font size="2”>Máximo 8 caracteres
```

Ésto solo para simplificar el problema. Por supuesto que si deseas chequear que todos los *tags* deben estar bien formados, adelante.

Para resolver el problema, pueden serte útiles las clases del `namespace` System.Collections, en especial Hashtable y ArrayList. Usa la documentación de .NET Framework SDK para saber como utilizarlas y que colaboración puedes pedir a instancias de éstas clases.

Parte 2

En tu programa anterior, si has programado pensando en objetos, deberías estar usando múltiples clases y deberían existir varios objetos que se envían mensajes entre sí.

Enumera las clases que se utilizan en el programa y las responsabilidades de hacer y conocer de cada una de ellas. Considera sólo las que son usadas en el programa.

Entrega este trabajo como parte de la documentación de la clase Program.

Importante

Casos de prueba `nunit` y documentación ndoc

Recuerda que un programa *debe* incluir casos de prueba y documentación ndoc. Los casos de prueba deben asegurarse de que cada método de cada clase que programaste funcione como es debido. Intenta utilizar ésto como una herramienta para trabajar menos y más seguro, y no como un requisito “burocrático”.

Reglas de colaboración

Los ejercicios son individuales. Puedes diseñar una solución en conjunto con otros compañeros y aprovechar comentarios o correcciones de los profesores, pero debes entregar un código que tú comprendiste, escribiste, compilaste, ejecutaste y probaste. Si entregas una clase que a nuestro criterio es idéntica a la de un compañero, puedes estar en problemas.

Entregas tardías

No se aceptarán entregas fuera de fecha. Los trabajos se entregan para que sigas constantemente el curso y no con objetivo de evaluarte, pese a que influyen en tu evaluación final en caso de que estés en riesgo de perder el curso. Un trabajo a las apuradas no tiene valor en este contexto.

Entregas por correo electrónico

No se aceptarán entregas por correo electrónico excepto que Web Asignatura no esté disponible seis horas antes a la fecha final de entrega. La entrega por correo eletctrónico debe enviarse a todos los profesores y pedirles confirmación de entrega. Es tu responsabilidad asegurarte de que el trabajo haya sido recibido.