Universidad Abierta Interamericana

Campus Virtual UAIOnline

PROGRAMACIÓN I

LECTURA

UNIDAD 4

ARCHIVOS

Autor de contenidos:
Nicolás Battaglia

OBJETIVOS

En esta clase veremos cómo se genera, manipula, trabaja y procesa un archivo de texto secuencial.

Desde el punto de vista lógico, explicaremos la existencia de archivos con organización secuencial e indexada. Desde el punto de vista de .Net, analizaremos solamente el manejo de los archivos secuenciales dado que el lenguaje no soporta indexados.

Esperamos que usted, a través del estudio de esta unidad, adquiera capacidad para:

- Generar un archivo de texto.
- Dar de Alta, Baja y Modificar sus datos.
- Realizar procesos de corte de control y apareos de archivos.

ENUNCIADO

Para el estudio de estos contenidos usted deberá consultar la bibliografía que aquí se menciona:

BIBLIOGRAFÍA OBLIGATORIA

Brizuela, R. **Operaciones básicas con archivos.** UAI. Buenos Aires; 2008.

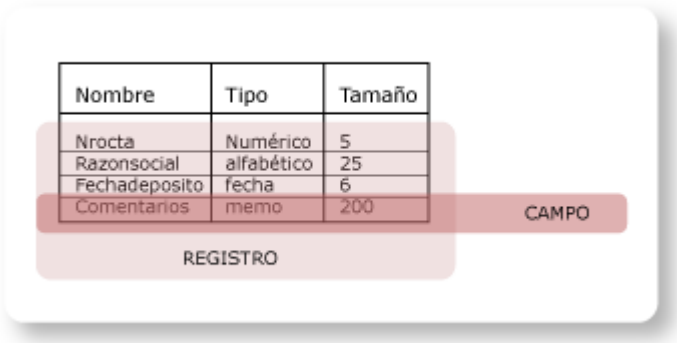
Martinez, J. Actualización al Lenguaje C#, UAI, Buenos Aires; 2019.

1. Archivos

Un **archivo** se encuentra compuesto por un conjunto de **registros** y estos, por un conjunto de **campos** donde se guardan los datos.

Veremos a continuación un detalle de los campos de un registro.

1.1 Campos de un registro



Los campos de un registro son las unidades lógicas donde se almacenan los datos de un registro. Sus características son el nombre, tipo y tamaño.

Los tipos de datos y su tamaño están restringidos por el lenguaje que se utilice. No todos los lenguajes soportan los mismos tipos de datos y los rangos de los mismos pueden variar.

Un conjunto de **campos** componen un **registro**, como lo ejemplifica la tabla anterior donde los cuatro datos tomados como una unidad, componen el registro. Un conjunto de **registros** componen un **archivo.**

Existen básicamente dos tipos de archivos:

- Secuenciales
- Indexados

Explicaremos a continuación cada uno de ellos.

Archivos Secuenciales

Tienen como característica principal que **su organización es secuencial** y su forma de acceso a los distintos registros es únicamente secuencial. Es decir, para acceder al registro 50 debo haber pasado por los 49 anteriores.

Estos archivos pueden o no estar ordenados y pueden o no tener **registros** repetidos, o sea, varios registros para una misma cuenta. Veamos los ejemplos donde cada fila representa un registro del archivo y, las columnas, los campos de los mismos:

Los registros están desordenados y hay uno solo

Artículo	Denominación	Cantidad vendida
10	Libro	5
5	Sacapuntas	15
1	Lápiz	20
3	Regla	30
2	Goma	10

Lo mismo que el caso anterior pero ordenado.

Artículo	Denominación	Cantidad vendida
1	Lápiz	20
2	Goma	10
3	Regla	30
5	Sacapuntas	15
10	Libro	5

Este archivo está ordenado y con registros repetidos para cada artículo.

Artículo	Denominación	Cantidad vendida
1	Lápiz	30
1	Lápiz	10
2	Goma	15
2	Goma	5

Para definir un registro, tomaremos como notación la siguiente y colocaremos debajo de cada campo el tipo de dato y el tamaño del campo:

S/R sin repetidos

C/R con repetidos

Artículo	Descripción	Cantidad
C/R N (5)	A (25)	F (6)

El ordenamiento del archivo puede ser realizado por más de un campo.

Archivos Indexados

Los archivos indexados tienen como particularidad el **acceso restringido**, por lo que es necesario tener clave para acceder a los mismos. Están ordenados por claves. Si bien la organización física de los registros es de acuerdo con el orden de grabación, su organización lógica está dada por la clave.

La organización de un archivo indexado es indexada y, según el lenguaje, las formas de acceso pueden ser secuencial, random o al azar y dinámica.

- ***Secuencial*** es la lectura de un registro a posteriori del anterior con la única seguridad de que el archivo se encuentra ordenado por la clave de acceso definida.
- El acceso ***random*** es el que se realiza en forma directa y única a un registro en particular debiéndose conocer el valor completo de la clave de acceso. Si el registro está compuesto por más de un campo se deberán conocer todos los valores de esos campos que componen la clave.
- ***Dinámica*** es la forma de acceso a un registro en particular. Una vez allí, permite leer el archivo en forma secuencial hasta una condición dada. En este caso no es necesario conocer la clave completa. Si ésta se encuentra compuesta por más de un campo y existe peso de orden entre ellos será necesario que conozcamos aquellos campos de mayor peso. No podemos conocer los campos de menor peso y desconocer los de mayor peso cuando las claves están ordenadas

  Veamos un ejemplo de una situación en la que la clave está compuesta por más de un campo. Suponga el archivo notas de un alumno. Este estará ordenado por legajo y dentro de los legajos ordenados por materias, entonces yo no puedo querer ingresar a una materia si antes no sé de qué alumno se trata.

Este ejemplo es una clave compuesta por dos campos que explica en parte lo anteriormente explicado:

En este caso de un archivo de infracciones, la clave está compuesta por la patente del automóvil y, como ésta puede estar repetida por varias multas, hay que agregarle el campo **nroinfraccion** para lograr que ese valor sea único.

Este archivo estará lógicamente ordenado por patente y dentro de ella por número de infracción.

Un **archivo indexado** puede tener varias claves de acceso definidas a la vez. Eso dependerá del lenguaje a utilizar.

Podemos decir que un archivo indexado está compuesto por la **tabla base donde** se encuentran los registros propiamente dichos y distintas **tablas índices** asociadas para su más rápido y fácil acceso.

Los archivos indexados no existen en .Net. Sí en otros lenguajes de programación.

A continuación, le presentamos otras operaciones que puede realizar con los archivos.

Lectura requerida

Brizuela, R. **Operaciones básicas con archivos.** UAI. Buenos Aires; 2008.

Martinez, J. Actualización al Lenguaje C#, UAI, Buenos Aires; 2019.

Le proponemos a continuación una serie de trabajos prácticos que le permitirán ejercitar las operaciones aprendidas. Tenga en cuenta que este trabajo tiene fecha de entrega. Administre su tiempo y cumpla entregando en tiempo y forma.

Consulte el Cronograma de Actividades de la asignatura.

2. Introducción al sistema de objetos de archivo

Todo lo referido a Entrada / Salida (E/S) se maneja desde la clase **System.IO** del marco .NET y está basado en el concepto de **streams.**

Un **streams** es un conjunto de bytes en el cual podemos escribir, leer, situarnos dentro de una posición determinada y mucho más, siempre dependiendo de la capacidad del dispositivo en el que estemos trabajando.

Los archivos pertenecen a la clase **FileStream** y podemos sintetizar sus operaciones en las siguientes:

- Crear un archivo o abrir uno ya existente
- Leer o escribir datos en un archivo
- Cerrar el archivo

Lo primero que debemos hacer para generar un archivo es crear un objeto de tipo **FileStream** asociado a un archivo en particular. El código podría ser:

Le presentamos algunos ejemplos en función de la operación a realizar:

El archivo ya existe y desea leer o escribir.

```
FileStream miArchivo = new FileStream("a1.txt", FileMode.Open);
```

Queremos abrir un archivo. No existe, lo crea.

```
FileStream miArchivo = new FileStream("a1.txt", FileMode.OpenOrCreate);
```

Abrir un archivo para leer (el archivo existe)

```
FileStream miArchivo = new FileStream("a1.txt", FileMode.Open, FileAccess.Read);
```

Crear un archivo y escribirlo (tenga en cuenta que si ya existía lo pisa).

```
FileStream miArchivo = new FileStream("a1.txt",filemode.create,fileaccess.write”)
```

Permitir que otro proceso escriba y lea el archivo.

```
FileStream miArchivo = new FileStream("a1.txt", FileMode.create, FileAccess.write,
FileShare.ReadWrite)
```

La siguiente tabla le muestra un resumen de las operaciones básicas de manejo de archivos:

Modos de apertura de un archivo **FileMode**	
Append	Si no existe lo crea. Si existe lo abre y se sitúa al final del mismo
Create	Crea un archivo nuevo .si existía lo pisa
createNew	Crea un archivo. Si existe uno con igual nombre envía una Excepción
open	Abre un archivo existente. Si no existe envía una excepción
openOrCreate	Abre el archivo y si no existe lo crea
Truncate	Abre un archivo existente y lo deja en cero bytes

Modos de acceso a un archivo FileAccess	
Read	**Permite leer datos de un archivo**
Write	**Permite escribir datos en un archivo**
ReadWrite	**Permite leer y escribir datos en un archivo**

2.1. Tipos de archivos

Podemos tener básicamente tres tipos de archivos de texto:

- Archivos con texto, visible desde un editor de Windows.
- Archivos con datos binarios.
- Archivos con el estado de un objeto guardado.

Explicaremos los dos primeros y esperamos que usted complete el estudio leyendo la bibliografía correspondiente.

Archivos de texto

Para trabajar con archivo de texto puro utilizaremos **StreamReader** y **StreamWriter.**

Veamos primero el **SteamReader.**

```
string fileName = "temp.txt";
```

Entonces, cuando queremos leer un archivo debemos codificar lo siguiente:

Creamos el **FileStream:**

```
FileStream stream = new FileStream(fileName, FileMode.Open, FileAccess.Read);
```

Creamos el **StreamReader** para leerlo:

```
StreamReader reader = new StreamReader(stream);
```

Veamos en esta tabla los métodos principales del objeto **StreamReader**.

Read	**Lee el próximo caracter disponible a partir de la posición actual y avanza.**
ReadBlock	**Lee un bloque de caracteres y lo almacena en un vector de chars.**
Readline	**Lee una línea del archivo.**
ReadToEnd	**Lee el contenido de todo el archivo y lo guarda en un string.**
Peek	**Obtiene el valor del próximo caracter disponible pero no avanza.**
Close	**Cierra el stream liberando recursos y referencias al archivo.**

Le presentamos algunos ejemplos:

- Lectura por línea de un archivo

```
FileStream miArchivo = new FileStream("a1.txt", FileMode.OpenOrCreate);
StreamReader lector = new StreamReader(miArchivo);
string linea;
linea = lector.ReadLine();
this.label1.Text = lector.ReadToEnd();
this.listBox1.Items.Clear();

while (linea != null)
{
   this.listBox1.Items.Add(linea);
   linea = lector.ReadLine();
}
lector.Close();
```

```
miArchivo.Close();
```

- Podemos escribir lo mismo utilizando en el **while**, **Peek.**

```
string fileName = "temp.txt";
FileStream stream = new FileStream(fileName, FileMode.Open, FileAccess.Read);
StreamReader reader = new StreamReader(stream);

while (reader.Peek() > -1)
 {
Console.WriteLine(reader.ReadLine());
  }
reader.Close();Archivo.close( )
```

- Para leer todo el archivo de una sola vez:

```
string fileName = "temp.txt";
FileStream stream = new FileStream(fileName, FileMode.Open, FileAccess.Read);
StreamReader reader = new StreamReader(stream);
string contenido = reader.ReadToEnd;
reader.Close();Archivo.close( );
```

- Para **escribir** un archivo de texto deberíamos hacer lo siguiente siguiendo el mismo razonamiento del **StreamReader**.

```
FileStream miArchivo = new FileStream("a1.txt", FileMode.Append);
StreamWriter escritor = new StreamWriter(miArchivo);
String registro = txtUni.Text + "|" + txtFacu.Text + "|" + txtCarrera.Text+ "|" + txtLegajo.Text;
```

```
escritor.WriteLine(registro);

escritor.Close();

miArchivo.Close();
```

Recuerde que si usted nombra a este nuevo archivo **a1.dat** y resulta que éste ya existía, perderá el contenido del archivo existente.

Le presentamos algunos métodos del **StreamWriter**

Write	Escribe una cadena de texto en el archivo.
Writeline	Escribe una cadena y le agrega un salto de fin de línea.
Flush	Devuelve el contenido del buffer del **stream** al archivo.
Close	Realiza cualquier escritura pendiente sobre el archivo (**flush**) y lo cierra.

Veamos un ejemplo

- Para escribir 100 números en un archivo

```
StreamWriter escritor = new StreamWriter(("a1.txt",);

int i;

for( i = 1; i<=100;i++)

{

        Archivo.writeLine ( i.tostring());

}

escritor.Close();
```

Recuerde que hay que pasar de **integer** a **string**

Archivos binarios

Para los tipos de archivos que no guardan texto plano legible sino código binario debemos utilizar otras clases. Estas clases serán **BinaryReader** y **BinaryWriter.**

Comencemos por la clase **BinaryWriter**

Para escribir archivos binarios es obligatorio crear previamente un **FileStream.**

```
string fileName = "temporal.txt";
int[] data = {0, 1, 22, 31, 4, 5,45};
FileStream stream = new FileStream(fileName, FileMode.Open, FileAccess.Write);
BinaryWriter writer = new BinaryWriter(stream);

for(int i=0; i<data.Length; i++)
    {
       writer.Write(data[i]);
    }

 writer.Close();
 stream.Close();
```

Los métodos más comunes de esta clase son:

Write	Guarda una variable en el archivo.
Flush	Vuelca en el archivo cualquier escritura pendiente.
Seek	Se sitúa en una posición particular dentro del archivo.
Close	Cierra el **binarywriter**, el **filestream** y libera recursos.

Veamos ahora qué pasa con el **BinaryReader**

Para leer un archivo binario es similar al de texto y tiene las mismas restricciones que ellos para leer.

Para abrir el archivo y leerlo, codificamos lo siguiente:

```
string fileName = "temp.txt";
int letter = 0;
FileStream stream = new FileStream(fileName, FileMode.Open, FileAccess.Read);
BinaryReader reader = new BinaryReader(stream);


while (letter != -1)
  {
    letter = reader.Read();
    if (letter != -1)
{
Console.Write((char)letter);
}
  }
```

```
reader.Close();

stream.Close();
```

Los métodos más comunes de esta clase son los que volcamos en la tabla.

Peekchar	Consulta el próximo carácter disponible pero no avanza el cursor.
Read	Lee un bloque de bytes del archivo.
ReadBoolean	Lee una variable booleana.
ReadByte,ReadBytes	Lee un byte o un bloque de bytes.
ReadChar,ReadChars	Lee un carácter o un bloque de carácter.
ReadSingle	Lee una variable single.
ReadDecimal	Lee una variable decimal.
Close	Cierra el **binarywritery** el **Filestream**.

Las propuestas de trabajo en el Laboratorio recuperan y amplían los contenidos trabajados hasta este momento.

Recuerde que si durante la realización del trabajo surgen dudas o inquietudes sería muy provechoso que las compartiera con sus pares. En la Web, los Foros creados por los programadores son herramientas fundamentales para el aprendizaje compartido y la profesionalización. Iníciese, si aún no lo ha hecho, en esta práctica de trabajo colaborativo.

A continuación, le dejamos un detalle del uso de Streams, a partir de una explicación y los conceptos asociados

Clase Stream

La clase abstracta **Stream** representa una secuencia de bytes que va o que viene de un medio de almacenamiento (por ejemplo: un archivo) o de un dispositivo virtual o físico (por ejemplo: un puerto paralelo, una tubería de comunicación entre procesos o un socket TCP/IP).

Las secuencias (**stream**) les permitirá leer o escribir de/en un almacén que puede corresponderse con uno de entre varios medios de almacenamiento.

Operaciones con secuencia

Las operaciones fundamentales que pueden ejecutar sobre las secuencias son: lectura, escritura y búsqueda. No todos los tipos de secuencia permiten todas estas operaciones.

La mayoría de los objetos de secuencia almacenan los datos en búferes de una forma transparente. No es necesario decir que el almacenamiento en búfer mejora notablemente el rendimiento.

Las secuencias de archivos se almacenan en búferes, mientras que las secuencias de memoria no porque no existe un motivo para almacenar en un búfer una secuencia asignada de memoria.

Podría utilizar el objeto **BufferedStream** para agregar capacidad de almacenamiento en búfer a un objeto **Stream** que no cuente con ella de forma nativa.

Lectura y escritura de secuencia

Cuando el objeto **stream** genérico solo puede leer y escribir bytes individuales o grupos de bytes, la mayor parte de las veces deberá utilizar objetos auxiliares "lectores de secuencia y escritores de secuencia" que le permitirán trabajar con datos de forma mas estructurada.

.Net Framework dispone de varios lectores y escritores de secuencia, como: Las clases **BinaryReader**, **BinaryWriter**, Las Clases **stremReader**, **streamWriter**, entre otras.

Lectura y Escritura de Archivos de Texto

Normalmente, utilizará el objeto **StreamReader** para leer desde un archivo de texto.

Después de que tenga una referencia a un objeto **StreamReader**, podrá utilizar uno de sus numerosos métodos para leer uno o más caracteres e, incluso, líneas de texto completas.

Lectura y escritura de archivos binarios

Las clases **BinaryReader** y **BinaryWriter** resultan apropiadas para trabajar con secuencia binarias; una de estas secuencias se puede asociar con un archivo que contenga datos en formato nativo. En este contexto formato nativo significa los bits reales utilizados para almacenar el valor en memoria.

Trabajar con el objeto **BinaryWriter** resulta especialmente sencillo porque su método **Write** ha sido sobrecargado para que acepte todos los tipos de .NET Framework.

Lectura y escritura de secuencia de memoria

Los lectores y escritores de secuencia no sirven exclusivamente para los archivos. Por ejemplo, podrá utilizarlos en unión del objeto **MemoryStream** para trabajar con la memoria como si fuera un archivo temporal (proporciona un mejor rendimiento que utilizar un archivo real).

Lectura y escritura de cadenas en memoria

Si los datos que desea leer se encuentran ya contenidos en una variable de cadena, tal vez desee utilizar un objeto **StringReader** para recuperarlo.

Lectores y escritores de secuencias personalizados

Podrá crear con facilidad lectores y escritores de secuencias personalizados que trabajen con objetos personalizados, gracias a la herencia.