

تمنحنا لغة سي الشارب الصنف `FileStream` الذي يصور المعلومات المخزنة على انها سلسلة البايتات .من اجل معالجة كل الأمور التي تتعلق بالملفات (قراءة – كتابة –إنشاء- تحديد حجم الملف الخ)يوجد لدى الصنف `FileStream` عدد من الحقول أهمها:

الحقل	الوصف
<code>CanRead</code>	يحدد إمكانية القراءة من الملف الحالي ويأخذ القيمة <code>true</code> إذا كانت القراءة ممكنة و <code>false</code> إذا كان الأمر عكس ذلك . يحدث خطأ إذا كانت هناك محاولة قراءة لملف لا يسمح بقراءته . يستخدم عادة قبل أي عملية قراءة.
<code>CanSeek</code>	يحدد إمكانية التنقل الحر لمؤشر الملف داخل الملف فإذا كان هذا الأمر متاحاً سنحصل على القيمة <code>true</code> وإلا فسنحصل على <code>false</code> . يحدث خطأ إذا كانت هناك محاولة تنقل حر مع وجود قيمة <code>false</code> للحقل
<code>CanWrite</code>	يستخدم عادة قبل أي عملية كتابة لفحص إمكانية الكتابة . فإذا كان الملف يدعم عملية كتابة فسنحصل على القيمة <code>true</code> فيما عدا ذلك سنحصل على القيمة <code>false</code> . ويحدث خطأ إذا كانت هناك محاولة الكتابة داخل ملف لا يدعم الكتابة.
<code>Position</code>	نستطيع بواسطة هذا الحقل أن نعين موقع جديد لمؤشر ملف أو نعرف موقعه ولكن قبل تغيير قيمة هذا الحقل يجب أن نتأكد من قيمة الحقل <code>CanSeek</code> أي من وجود القيمة <code>true</code> داخل هذا الحقل وهو من النوع <code>long</code>
<code>Length</code>	هذا الحقل للقراءة فقط ويعطينا حجم الملف بالبايتات وهو من النوع <code>long</code>

تفاصيل المنهج الباني من الصنف `FileStream`

من اجل التعامل مع الملفات داخل الشفرة في `C#` نحتاج الى تضمين الترويسة

```
using System.IO;
```

بعد ذلك سنستخدم الصنف `FileStream` لإنشاء ملف كما مبين في السطر التالي:

```
FileStream MyFile = new FileStream("C:\\MyFolder\\SyncDemo.txt",
    FileMode.OpenOrCreate);
```

في الشفرة السابقة قمنا بتعريف كائن من الصنف `FileStream` وارسلنا له بارامترين (وسيطين) **يدل الأول** على المسار الى الملف **يدل الثاني** على نمط الانشاء تحمل الشفرة السابقة المعنى التالي:

سينشأ ملف في المجلد `C:\\MyFolder` باسم `MyStream.txt`

ياخذ البارامتر الدال على نمط الانشاء القيم التالية :

القيمة	الوصف
<code>Append</code>	يستخدم لاضافة بيانات في آخر الملف (يجعل مؤشر الملف في الملف ولذا لا يمكن القراءة مباشرة)
<code>Create</code>	يستخدم لإنشاء ملف وفي حالة وجود ملف بهذا الاسم يمسح الملف
<code>CreateNew</code>	يستخدم لإنشاء ملف وفي حالة وجود ملف بهذا الاسم نحصل على رسالة خطأ
<code>Open</code>	نفتح ملف وفي حالة غياب ملف بهذا الاسم نحصل على رسالة خطأ
<code>OpenOrCreate</code>	يفتح الملف ان كان موجوداً أو ينشئه عند عدم وجود الملف
<code>Truncate</code>	يفتح الملف ويمسح محتوياته ويجعل مؤشر الملف في البداية

هناك أيضا بعض الامور الاخرى التي نستطيع التحكم بها مثلا يمكن ان ننشئ ملفا ونجعله صالحا للقراءة فقط أو للكتابة فقط أو للقراءة و الكتابة وذلك بارسالة بارامتر (وسيط) ثالث يدل على خيارات الوصول الى الملف كما في المثال التالي:

```
FileStream StrFl = new FileStream("MyStream.txt ", FileMode.OpenOrCreate,
    FileAccess.Write, FileShare.Read);
```

وهذه هي القيم التي يأخذها البارامتر الثالث:

القيمة	الوصف
Read	الملف للقراءة فقط
Write	الملف للكتابة فقط
ReadWrite	الملف للقراءة والكتابة

كما يمكن التحكم بمشاركة الملف بين العمليات وذلك ببارامتر (وسيط) رابع يساعدنا على التحكم بمدى المشاركة بين العمليات على الملف – انظر المثال:

```
// Using file path, file mode, file access and sharing permission
// Open file for writing, other processes will get read-only access
FileStream StrFl = new FileStream("C:\\MyFolder\\MyStream.txt ",
    FileMode.Open, FileAccess.Write, FileShare.Read);
```

القيمة	الوصف
Inheritable	تستطيع اي عملية ابن أن ترث من عملية اب مقيس الملف
None	لا توجد أي عملية تستطيع الوصول الى الملف الحالي بما في ذلك العملية الحالية
Read	يسمح للعمليات قراءة الملف فقط
Write	يسمح للعمليات بما في ذلك العملية الحالية بالكتابة فقط
ReadWrite	يسمح للعمليات بالقراءة والكتابة الى هذا الملف

سنجرب كيف يمكن أن ننشئ ملفا وكيف نكتب داخله :

```
using System;
```

تفاصيل مهمة لعمل الصنف FileStream

يوجد في الصنف FileStream عدد من المناهج التي تسهل معالجة البيانات المخزنة داخل الملفات

المنهج Read : يأخذ كمداخلات مصفوفة من البايتات و الموقع (الموقع من داخل المصفوفة ويبدأ من صفر) الذي سيبدأ منه القراءة وعدد البايتات المطلوب قراءته ويعيد كمخرجات عدد البايتات التي قام بقراءته فعليا. يقوم هذا المنهج بزيادة موقع مؤشر الملف بعدد البايتات المقروءة . يكافئ موقع مؤشر الملف ما يقوم به العنوان أو index في المصفوفات مع فارق بسيط وهو أن موقع مؤشر الملف يتزايد دائما تزايد تلقائيا بعد كل عملية قراءة وكتابة حتى يصل إلى نهاية الملف.

إذا كان عدد البايتات الممكن قراءته اكبر اقل من المطلوب قراءته (مثلا حجم الملف 50 بايت وطلبنا 100 بايت أو مثلا حجم الملف 1000 بايت وطلبنا 200 بايت ابتداء من الموقع 950) سيعيد المنهج ما قرأه فعليا أو يعيد 0 إذا لم يستطع قراءة شيء (موقع مؤشر الملف يقع في نهاية الملف).

مثال على استخدامه

```
byte[] AllBytes=new byte[500];

int count = 0;

count =MyFile.Read(AllBytes, 0, AllBytes.Length);
```

الآن أصبح **count** يحتوي على عدد البايتات المقروء من الملف.

يقوم المنهج ReadByte بقراءة بايت (يقوم بزيادة موقع مؤشر الملف **Position**) كما أن المنهج **ReadByte** لا يأخذ أي مدخلات و يعيد قيمة من النوع **int** تحتوى بداخلها على البايت المقروء (معلوم ان البايت يمكن أن يكتب على هيئة عدد صحيح)

```
//*****
byte[] b=new byte[10];
// .....
FileStream MyFile=new FileStream("C:\\MyFolder\\MyStream.txt",
                                FileMode.OpenOrCreate);
char c = Convert.ToChar(MyFile.ReadByte());
```

إذا تحدثنا عن القراءة من ملف فلا عجب أن يدور الحديث عن الكتابة في ملف , ولفعل ذلك لدينا منهجان:

المنهج Write : ويأخذ كمدخل مصفوفة من البايتات والموقع (الموقع من داخل المصفوفة ويبدأ من صفر)الذي سيبدأ منه الكتابة وعدد البايتات المراد كتابته. كما يقوم هذا المنهج بزيادة موقع مؤشر الملف **Position** بمقدار يساوي عدد البايتات المكتوبة

```
byte[] b=new byte[10];
Random R = new Random(); R.NextBytes(b);
FileStream MyFile=new FileStream("C:\\MyFolder\\MyStream.txt",
                                FileMode.OpenOrCreate);
MyFile.Write(b,0,b.Length);
```

المنهج WriteByte يأخذ كمدخلات بايتا واحد سيقوم بكتابته داخل الملف ويزيد مؤشر الكتابة بمقدار واحدة

```
byte[] b=new byte[10];
// .....
Random R = new Random(); R.NextBytes(b);
FileStream MyFile=new FileStream("C:\\MyFolder\\MyStream.txt",FileMode.OpenOrCreate);
```

```
MyFile.Write(b,0,b.Length);
```

المنهج Seek يؤمن هذا المنهج التنقل الحر في الاماكن المختلفة من الملف

```
MyFile.Seek(0, SeekOrigin.Begin);
```

تفاصيل المنهج الباني من الصنف FileStream

من اجل التعامل مع الملفات داخل الشفرة في C# نحتاج إلى تضمين الترويسة

```
using System.IO;
```

بعد ذلك سنستخدم الصنف **FileStream** لإنشاء ملف كما مبين في السطر التالي:

```
FileStream MyFile = new FileStream("C:\\MyFolder\\SyncDemo.txt",  
FileStreamMode.OpenOrCreate);
```

في الشفرة السابقة قمنا بتعريف كائن من الصنف **FileStream** وارسلنا له بارامترين (وسيطين) **يدل الأول** على المسار الى الملف **يدل الثاني** على نمط الإنشاء تحمل الشفرة السابقة المعنى التالي:

سينشأ ملف في المجلد **C:\\MyFolder** باسم **MyStream.txt** يأخذ البارامتر الدال على نمط الإنشاء القيم التالية :

الوصف	القيمة
يستخدم لإضافة بيانات في آخر الملف (يجعل مؤشر الملف في الملف ولذا لا يمكن القراءة مباشرة)	Append
يستخدم لإنشاء ملف وفي حالة وجود ملف بهذا الاسم يسمح الملف	Create
يستخدم لإنشاء ملف وفي حالة وجود ملف بهذا الاسم نحصل على رسالة خطأ	CreateNew
نفتح ملف وفي حالة غياب ملف بهذا الاسم نحصل على رسالة خطأ	Open
يفتح الملف ان كان موجودا أو ينشئه عند عدم وجود الملف	OpenOrCreate
يفتح الملف ويمسح محتوياته ويجعل مؤشر الملف في البداية	Truncate

هناك ايضا بعض الامور الاخرى التي نستطيع التحكم بها مثلا يمكن ان ننشئ ملفا ونجعله صالحا للقراءة فقط أو للكتابة فقط أو للقراءة و الكتابة وذلك بارسالة بارامتر (وسيط) ثالث **يدل** على خيارات الوصول الى الملف كما في المثال التالي:

```
FileStream StrFl = new FileStream("MyStream.txt", FileMode.OpenOrCreate,  
FileAccess.Write, FileShare.Read);
```

وهذه هي القيم التي يأخذها البارامتر الثالث:

الوصف	القيمة
الملف للقراءة فقط	Read
الملف للكتابة فقط	Write
الملف للقراءة والكتابة	ReadWrite

كما يمكن التحكم بمشاركة الملف بين العمليات وذلك ببارامتر (وسيط) رابع يساعدنا على التحكم بمدى المشاركة بين العمليات على الملف – انظر المثال:

```
// Using file path, file mode, file access and sharing permission
// Open file for writing, other processes will get read-only access
FileStream StrFl = new FileStream("C:\\MyFolder\\MyStream.txt",
    FileMode.Open, FileAccess.Write, FileShare.Read);
```

الوصف	القيمة
تستطيع اي عملية ابن أن ترث من عملية اب مقبس الملف	Inheritable
لا توجد أي عملية تستطيع الوصول الى الملف الحالي بما في ذلك العملية الحالية	None
يسمح للعمليات قراءة الملف فقط	Read
يسمح للعمليات بما في ذلك العملية الحالية بالكتابة فقط	Write
يسمح للعمليات بالقراءة والكتابة الى هذا الملف	ReadWrite

سنجرب كيف يمكن أن ننشئ ملفا وكيف نكتب داخله :

```
using System;
```

هناك منهجان من مكتبة .NET. سوف نستخدمها لانجاز بعض المهام وهما:

1. المنهج `Encoding.ASCII.GetBytes` يأخذ كمدخلات سلسلة من الرموز string ويعيد كمخرجات مصفوفة من البايتات، يقابل كل بايت منها رمزا من رموز السلسلة المدخلة.
2. المنهج `Encoding.ASCII.GetString` يأخذ كمدخلات مصفوفة من البايتات ويعيد كخرج سلسلة من الرموز string .

مثال

```
using System.IO;
using System.Text;
class SyncIO
{
    public static void Main(string[] args)
    {
        // Create FileStream instance
        FileStream MyFile = new FileStream("SyncDemo.txt", FileMode.OpenOrCreate);
```

لاحظ أننا قمنا بإنشائه كائن `MyFile` مرتبط مع الملف `SyncDemo.txt` الموجود داخل نفس مجلد الملف التنفيذي إذا لم يكن الملف موجودا سينشأ ملف بهذا الاسم لأن نمط الإنشاء هو `OpenOrCreate`.

```
MyFile.WriteByte(Convert.ToByte('A'));
```

يقوم المنهج `Convert.ToByte` بتحويل الرمز 'A' إلى بايت بعدها سيقوم المنهج `WriteByte` من الكائن `MyFile` بكتابته داخل الملف

```

Console.WriteLine("Using Write Method");
byte[] writeBytes = Encoding.ASCII.GetBytes(" Very Beautiful Language");
MyFile.Write(writeBytes, 0, writeBytes.Length);

```

لا يخفى على القارئ الكريم أن السطر الأول هو لإخراج سلسلة على الشاشة بينما يقوم السطر الثاني بتحويل سلسلة رموز إلى مصفوفة من البايتات . يقوم السطر الثالث بكتابة مصفوفة البايتات الناتجة من السطر الثاني داخل الملف ابتداء من العنوان 0 داخل المصفوفة إلى نهاية المصفوفة (أي بعدد عناصر المصفوفة).

قلنا سابقا أنك إذا أجريت عملية قراءة أو كتابة فإن موقع مؤشر الكتابة يزداد بمقدار البايتات المقروءة أو المكتوبة ، سنواصل مثالنا ونعيد مؤشر الملف إلى بداية الملف باستخدام الشفرة

```
MyFile.Seek(0, SeekOrigin.Begin);
```

إن الشفرة السابقة تقول لنا ابتداء من بداية الملف يجب أن يتحرك مؤشر الملف بمقدار 0 من البايتات ولو كانت الشفرة هكذا

```
MyFile.Seek(2, SeekOrigin.Begin);
```

لقلنا الشفرة لنا : ابتداء من بداية الملف تحرك بمقدار 2 بايت . انظر إلى الشفرة التي تليها

```
char c;
```

```
MyFile.Seek(3, SeekOrigin.Current);
```

```
c = Convert.ToChar(MyFile.ReadByte());
```

إنها تقول : ابتداء من موقعك الحالي تحرك 3 بايتات ثم اقرأ بايتاً واحداً وحوله إلى رمز باستخدام الأمر `Convert.ToChar`

أما الشفرة التالية فتقول لنا من نهاية الملف تراجع خطوة واحدة وقم بالقراءة وأطبع ما قمت بقراءته (من نهاية الملف لا يمكن القراءة دون التراجع إلى الخلف كما لا يمكن التقدم)

```
MyFile.Seek(-1, SeekOrigin.End);
```

```
c = Convert.ToChar(MyFile.ReadByte());
```

```
Console.WriteLine(c);
```

بعد التفاصيل السابقة سنواصل كتابة برنامجنا ونعود إلى بداية الملف

```
// Set pointer at origin
```

```
MyFile.Seek(0, SeekOrigin.Begin);
```

بعد أن أعدنا مؤشر الملف إلى بداية الملف نكتب الشفرة التالية:

```
Console.WriteLine("Using Read Method ");
```

```
// Read byte and display
```

```
char c= Convert.ToChar(MyFile.ReadByte());
```

```
Console.WriteLine("First character is ->" +c.ToString());
```

من نافذة القول أننا هنا عرفنا متغيراً من النوع `char` (رمز) ثم حولنا البايت الذي قرأناه من الملف إلى رمز `Convert` باستخدام وقمنا بتخزين القيمة في المتغير `c` , بعد ذلك طبعنا ما قرأناه على الشاشة.

بدلاً من قراءة الرموز من داخل الملف رمزاً تلو الآخر ، نستطيع أن نقرأ كلها دفعة واحدة لفعل ذلك اكتب الشفرة التالية:

```
// Use of Read method
```

```
Console.WriteLine("Using Read Method ");
```

```
// Allocate buffer
```

```
byte[] readBuf = new byte[MyFile.Length-1];
```

يظهر السطر الأول سلسلة على الشاشة بعدها عرفنا مصفوفة من البايتات طولها يساوي طول الملف - 1 (لماذا ؟ لان الرمز الأول قمنا بقراءته وطباعته على الشاشة). والآن اكتب الشفرة التالية :

```
// Read file
int count=Convert.ToInt32(MyFile.Length)-1;
MyFile.Read(readBuf,0,count);
}}
```

من المعروف أن الحقل **Length** من النوع **long** وان البارامتر الثالث من المنهج **Read** (الذي يعطي عدد البايتات المطلوب قراءته) هو من النوع **int** لذلك قمنا بتحويل طول الملف إلى قيمة من النوع **int** بواسطة المنهج **Int32** من الصنف **Convert** وقمنا بتخزين القيمة المحولة في المتغير **count** كما هو واضح من الشفرة وارسلناها إلى المنهج **Read** من الكائن **MyFile** و هنا ستجد شفرة المثال كاملة:

```
using System.Text;
using System.IO;
using System;

namespace StreamProcessing
{
    class Program
    {
        static void Main(string[] args)
        {
            FileStream MyFile = new FileStream("SyncDemo.txt",
                FileMode.OpenOrCreate);
            MyFile.WriteByte(Convert.ToByte('A'));
            Console.WriteLine("Using Write Method ");

            byte[] writeBytes = Encoding.ASCII.GetBytes(" is a first
character.");
            MyFile.Write(writeBytes, 0, writeBytes.Length);
            // Set pointer at origin
            MyFile.Seek(0, SeekOrigin.Begin);
            // Use of Read method
            Console.WriteLine("Using Read Method ");
            char c= Convert.ToChar(MyFile.ReadByte());
            Console.WriteLine("First character is ->" +c.ToString());
            // Allocate buffer
            byte[] readBuf = new byte[MyFile.Length-1];
            // Read file
            int count=Convert.ToInt32(MyFile.Length)-1;
            MyFile.Read(readBuf,0,count);
            // Display contents
            string s = Encoding.ASCII.GetString(readBuf);
            Console.WriteLine("The rest of the file is : " +s);

        }
    }
}
```

أنواع الملفات

من الأمور المتفق عليها ان هناك نوعين من الملفات : ملفات مخصصة لتخزين البيانات النصية **Text File** وملفات ثنائية **Binary File** مخصصة لبقية أنواع البيانات (اعداد صحيحة – اعداد حقيقية – بايتات ... الخ)

الملفات الثنائية

تقدم لنا مكتبات ال **Net** الاصناف **BinaryReader** و **BinaryWriter** لقراءة و كتابة أنواع بيانات ثنائية داخل الملفات، تجدر الإشارة إلى أن كائنات الصنفين المذكورين يستخدمان دائما مع كائن من الصنف **FileStream** وهذه قائمة باهم مناهج الصنف

BinaryReader

المنهج	عدد البايتات المقروءة من الStraem
<code>bool ReadBoolean()</code>	1
<code>byte ReadByte()</code>	1
<code>byte[] ReadBytes(int count)</code>	count
<code>char ReadChar()</code>	حسب نظام الترميز المستخدم
<code>decimal ReadDecimal()</code>	16
<code>double ReadDouble()</code>	8
<code>short ReadInt16()</code>	2
<code>int ReadInt32()</code>	4
<code>long ReadInt64()</code>	4
<code>float ReadSingle()</code>	4
<code>string ReadString()</code>	حسب طول السلسلة

واضح من اسماء المناهج أن الصنف **BinaryReader** مخصص لقراءة انواع البيانات البسيطة (**primitive data**) , والسؤال الذي يطرح نفسه ما الفائدة من الصنف **BinaryReader** طالما نستطيع أن نقرأ مباشرة من كائن من الصنف **FileStream** بواسطة المنهج **Read** أو المنهج **ReadByte** ؟ الجواب : عند القراءة من كائن من الصنف **FileStream** مباشرة نحتاج إلى تحديد عدد البايتات المطلوب عند كل قراءة أو يمكن نقرأ بايتا واحدا فقط بواسطة **ReadByte** وهو شيء لا نحتاجه عند استخدام مناهج الصنف **BinaryReader** نحتاج فقط تحديد نوع البيانات المطلوب قراءته كما هو واضح من اسم مناهج الصنف ، وسيتولى المنهج المطلوب مسألة تحديد الحجم (انظر الجدول السابق). يعطينا الصنف **BinaryReader** منهجا آخر رائعا وهو المنهج **PeekChar** حيث يقرأ المنهج المذكور الرمز الذي يقف عنده مؤشر الملف، دون يزيد موقع مؤشر الملف. يعطينا هذا المنهج القيمة 1- إذا كان المؤشر يقع عند آخر الملف أو يعطينا الرمز المقروء ممثلا داخل عدد صحيح من النوع **int** وهذا المثال عمل هذا المنهج:

```
static void Main(string[] args)
{
    char c;
    FileStream MyFile = new FileStream("SyncDemo.txt", FileMode.OpenOrCreate);
    MyFile.WriteByte(Convert.ToByte('A'));
    MyFile.Seek(0, SeekOrigin.Begin);
    Console.WriteLine("Position Before Calling PeekChar=" +
    MyFile.Position.ToString());
    BinaryReader br = new BinaryReader(MyFile);
```

```

if (br.PeekChar() > -1)
{
    c= Convert.ToChar(br.PeekChar());
    Console.WriteLine(c.ToString() + " is First character");
    Console.WriteLine("Position =" + MyFile.Position.ToString() );
}
}

```

تبدو بداية هذا البرنامج مألوقة لنا ، حيث قمنا بتعريف الكائن **MyFile** من الصنف وكتبنا رمزا داخله باستخدام المنهج **Write** وعدنا إلى بداية الملف باستخدام المنهج **Seek** ولكن انظر إلى تهيئة الكائن **br** حيث ربطناه مباشرة بكائن من النوع **FileStream** وهو الكائن **MyFile** لاحظ أنني قمت بطباعة موقع مؤشر الملف (**Position**) قبل اختبار شرط الوصول إلى اخر الملف وستجد أنني أحاول طباعة موقع المؤشر حتى بعد أن قمت باستدعاء المنهج **PeekChar** مرتين ولكنني سأحصل على القيمة 0، على ماذا يدل الحصول على هذه القيمة ؟ يدل على أن المنهج **PeekChar** يقرأ الرمز من الموقع الذي يقف عنده المؤشر دون أن يغير موقع مؤشر الملف أي أن قيمة **Position** لا تتغير . لمزيد من المتعة استبدل الشفرة التالية بالجزء الأخير من الشفرة السابقة :

```

if (br.PeekChar() > -1)
{
    c = br.ReadChar();
    Console.WriteLine(c.ToString() + " is First character");
    Console.WriteLine("Position =" + MyFile.Position.ToString() );
}

```

لاحظ أنك ستحصل على القيمة صفر وبعدها على القيمة واحد .

بعد أن تحدثنا بشكل خاطف عن قراءة الملفات يمكن أن نتحدث عن الكائن **BinaryWriter** الذي يختص بكتابة البيانات البسيطة داخل ملف بواسطة المنهج **write**. في الحقيقة يمكن بواسطة المنهج **write** أن تكتب أنواع متعددة من البيانات البسيطة والأمثلة القادمة توضح هذا الأمر . انظر الم

```

static void Main(string[] args)
{
    FileStream MyFile = new FileStream("BinaryDemo.txt",
    FileMode.OpenOrCreate);
    int i = 0,a;
    //*****
    Random R = new Random();
    BinaryWriter bw = new BinaryWriter(MyFile);
    for (i = 0; i < 10; i++)
    {
        a = R.Next(10, 100); Console.WriteLine(a.ToString());
        bw.Write(a);
    }
    Console.WriteLine("*****");
    bw.Close();
    BinaryReader br = new BinaryReader(MyFile);
    MyFile.Seek(0, SeekOrigin.Begin);
    while (br.PeekChar() > -1)
    {
        a = br.ReadInt32();
        Console.WriteLine(a.ToString());
    } br.Close(); MyFile.Close();
}

```

تقوم الشفرة السابقة بما يلي:

1. قمنا بتعريف كائن من **FileStream** وسميناه بالاسم **MyFile** وربطناه بملف موجود في المسار المذكور
 2. ربطنا كائنا من الصنف **BinaryWriter** بالكائن **MyFile** وسميناه بالاسم **bw**
 3. عرفنا كائنا مخصصا لتوليد الأعداد العشوائية من الصنف **Random** وسميناه **R**
 4. نكرر العمل التالي 10 مرات :
 - a. نولد عددا عشوائيا جديد ونخزن القيمة داخل متغير من نوع **int** يحمل الاسم **a** (لاحظ أن تعريف المتغيرين **a,i** تم سلفا)
 - b. نطبع العدد العشوائي الذي ولدناه على الشاشة
 - c. نكتب العدد العشوائي المولد في الملف
 5. نغلق الكائن **bw** المرتبط بالكائن **MyFile** (معلوم ان **MyFile** يشكل واجهة التعامل مع الملف)
 6. نطبع فاصلا من النجوم
 7. ربطنا كائنا من النوع **BinaryReader** بالكائن **MyFile** وسميناه بالاسم **br**
 8. أعدنا مؤشر الملف إلى بداية الملف **MyFile**
 9. طالما أننا لم نصل إلى نهاية الملف اعمل الآتي:
 - a. اقرأ عدد صحيحا من داخل الملف وأخزن القيمة المقروءة داخل المتغير **a**
 - b. اطبع المتغير **a**
 10. تقوم باغلاق الملف والكائن **br** المرتبط به
- أهم اللحظات أثناء تتبع الشفرة السابقة هناك لحظتان الأولى: الكتابة حيث استخدمنا المنهج **Write** ولم نقم بتحديد أي عدد من البايتات و لا حتى نوع المتغير المكتوب اللحظة الثانية هي لحظة القراءة حيث لم نعين عدد البايتات المطلوب قراءته و إنما حددنا نوع البيانات المقروءة بواسطة المنهج **ReadInt32** مثل هذه الأمور هي التي تجعلنا نلجأ إلى استخدام كائنات الصنفين **BinaryReader and BinaryWriter** (لاحظ أننا طبعنا العدد العشوائي المولد قبل كتابته في الملف، حتى نضمن التوافق بين ما كتب في الملف و ما سنقرؤه من الملف) . اللحظة الثالثة هي لحظة الدخول إلى الحلقة **while** ،حيث يمكننا أن نقرأ جميع البيانات الموجودة داخل الملف دون نعرف طول الملف، وما أن نصل إلى نهاية الملف حتى نحصل على القيمة 1- ولا نستطيع الدخول إلى الحلقة وهذا بفضل المنهج **PeekChar** . إغلاق الكائنات أمر ضروري توفيراً للمصادر وثانياً من أجل سلامة الملفات في حالة توقف البرنامج بشكل مفاجئ. لحسن الحظ تستطيع الملفات الثنائية تخزين أنواع غير متجانسة من البيانات (اعداد صحيحة- اعداد حقيقية - رموز) بشكل متتابع دون أي مشاكل .
- انظر المثال التالي

```
static void Main(string[] args)
{
    FileStream MyFile = new FileStream("IntDoubleDemo.txt",
    FileMode.OpenOrCreate);
    int i = 0,a=0,b=0;
    double d;
    char c;
    //*****
    Random R = new Random();
    BinaryWriter bw = new BinaryWriter(MyFile);
```

```

for (i = 0; i < 10; i++)
{
    a = R.Next(10, 100);
    b = R.Next(10, 100);
    d = Math.Sqrt(a*b);
    c = (char)R.Next(65, 90);
    bw.Write(a);
    bw.Write(b);
    bw.Write(d);
    bw.Write(c);
    Console.WriteLine("{0} {1} {2} {3}",a,b,d,c);
}

Console.WriteLine("*****");
BinaryReader br = new BinaryReader(MyFile);
MyFile.Seek(0, SeekOrigin.Begin);
while (br.PeekChar() > -1)
{
    a = br.ReadInt32();
    b = br.ReadInt32();
    d = br.ReadDouble();
    c = br.ReadChar();
    Console.WriteLine("{0} {1} {2} {3}", a, b, d, c);
}
//*****
}

```

لاحظ أننا استخدمنا نفس المنهج **Write** لكتابة الرمز والعدد الحقيقي وكذلك الأعداد الصحيحة. يجدر بنا أن نشير إلى أن المنهج **Write** يملك العديد من الأشكال التي يمكن استدعاءها.

يوضح الشكل إلى اليسار أهم أنواع البيانات التي تستطيع كائنات الصنف **BinaryWriter** كتابتها داخل الملفات والقائمة تطول.

يحتوي الصنف على مناهج أخرى مثل **Seek**

Write(Boolean)

Write(Byte)

Write(Byte[])

Write(Char)

Write(Char[])

Write(Int32)

Write(Int64)

Write(Double)

Write(String)

Write(Char[], Int32, Int32)

الملفات النصية Text Files

من أجل القراءة والكتابة إلى الملفات النصية نستخدم الكائنين **StreamReader** و **StreamWriter**. يستخدم الأول منهما للقراءة من ملف نصي بينما يستخدم الآخر للكتابة داخل ملف نصي. نحتاج إلى ربط الملف النصي مع كائن من **FileStream** ومن ثم نستخدمه لتهيئة كائنات من الصنف **StreamReader** أو **StreamWriter** حسب الحاجة.

سنتناول أولاً الصنف **StreamReader** وهذه قائمة بأهم المناهج التي يقدمها هذا الصنف

Method	Description
Peek()	يعيد الرمز الذي يقف عنده مؤشر الملف دون أن يغير موقعه ويعيد 1 إذا كان المؤشر يقف عند آخر الملف
Read()	يستخدم لقراءة سلسلة أو رمز (حسب الصيغة المستدعاة)
ReadBlock()	يقرأ من ملف إلى مصفوفة من الرموز
ReadLine()	يقرأ سطر من ملف ويعيد الرموز المقروءة على هيئة سلسلة
ReadToEnd()	يقرأ كل الرموز المكتوبة ويعيدها على شكل سلسلة
Close()	إغلاق القارئ

لكي نرى قدرات الصنف **StreamReader** وبعدها قم بإنشاء ملف نصي و أعطه الاسم **FirstDayWithTextFiles.txt** واكتب داخله النص التالي:

It's so interesting to Work Files in C#

and I'm going to do it every day.

C# gives us very powerful tools

try to do it and you will not stop

إحفظ التعديلات في الملف ثم أنشئ مشروعاً جديداً واكتب داخله الشفرة التالية (لا تنسى أن تضع الملف داخل المجلد **Debug** الخاص بالملف التنفيذي بالبرنامج):

```
static void Main(string[] args)
{
    FileStream MyTextFile = new FileStream("FirstDayWithTextFiles.txt",
        FileMode.OpenOrCreate);
    StreamReader Sr = new StreamReader(MyTextFile);
    string myLine;
    int Lines = 0;
    myLine = Sr.ReadLine();
    while (myLine != null)
    {
        Console.WriteLine(myLine);
        Lines++;
        myLine = Sr.ReadLine();
    }
    Console.WriteLine(Lines.ToString());
    Sr.Close();
    MyTextFile.Close();
}
```

يقرأ البرنامج السابق النص الموجود داخل ملف ويطبعه على الشاشة ويعد الأسطر أثناء ذلك، ليطلع عددها بعد الانتهاء من طباعتها

المثال التالي يأخذ محتويات ملف ويحوّله إلى كلمات متفرقة

```
static void Main(string[] args)
{
    FileStream MyTextFile = new
FileStream("FirstDayWithTextFiles.txt", FileMode.OpenOrCreate);
    StreamReader Sr = new StreamReader(MyTextFile);
    string MyLine="";
    int Lines = 0;
    char c;
    //*****
    while (Sr.Peek()>-1)
    {
        c = Convert.ToChar(Sr.Read());
        if (!char.IsWhiteSpace(c))
            MyLine = MyLine + c;
        else
        {
            if (MyLine != "")
            {
                Console.WriteLine(MyLine);
                MyLine = "";
            }
        }
        Sr.Close();
        MyTextFile.Close();
    }
}
```

بعد أن قمنا بالقراءة من داخل الملف `FirstDayWithTextFiles` سنرى كيف يقوم الصنف `StreamWriter` بالكتابة داخل ملف ولفعل ذلك انظر إلى المثال التالي:

```
static void Main(string[] args)
{
    FileStream MyTextFile = new FileStream("FirstDayWithTextFiles.txt",
FileMode.OpenOrCreate);
    StreamWriter Sw = new StreamWriter(MyTextFile);
    // Display the encoding type
    Console.WriteLine("Encoding type : " + Sw.Encoding.ToString());
    //Display Format Provider
    Console.WriteLine("Format Provider : " +
Sw.FormatProvider.ToString());
    Sw.WriteLine("Today is {0}.", DateTime.Today.DayOfWeek);
    Sw.WriteLine("Today we will mostly be using StreamWriter.");
    for (int i = 0; i < 5; i++)
        Sw.WriteLine("Value {0}, its square is {1}", i, i * i);
    Sw.Write("Arrays can be written : ");
    char[] myarray = new char[] { 'a', 'r', 'r', 'a', 'y' };
    Sw.Write(myarray);
    Sw.WriteLine("\r\nAnd parts of arrays can be written");
    Sw.Write(myarray,0,3);
    Sw.Close();
    MyTextFile.Close();
}
```

تقوم الشفرة في المثال الأخير بالأمور التالية:

1. يرتبط الكائن **MyTextFile** بالملف **FirstDayWithTextFiles** والذي قمنا بقراءته في المثال السابق لهذا المثال
2. يرتبط كائن **StreamWriter** بكائن الملف ولاحظ أن الربط هنا لا يختلف عن ربط كائنات الصنف **StreamReader**
3. يقوم كائن الكتابة بتحديد نوعية الترميز ومزود التنسيق (قد يكون هناك أكثر من تنسيق حتى من أجل اللغة الواحدة)
4. نكتب بعض الجمل داخل الملف
5. نكتب بعض الأعداد داخل الملف (لاحظ أنه يتعامل مع الملف وكأنه يتعامل مع الشاشة السوداء أي ينسق السلسلة كما يريد قبل كتابتها).
6. نكتب مصفوفة من الرموز داخل الملف, (كما نستطيع أن نكتب جزءا منها، حسب موقع بداية الكتابة وعدد الرموز المطلوب كتابتها).
7. إغلاق الكائنات.

ملاحظة هامة :

تحتوي كائنات الصنف **StreamReader** على ذاكرة مؤقتة **Buffer** تقوم بتخزين أول كيلو بايت من الملف المقروء ثم قراءة الرموز من داخل الذاكرة المؤقتة. الفائدة من هذا العمل هو تأمين السرعة بدلا من الرجوع المتكرر إلى الملف بالمقابل سيغير هذا الأمر موقع مؤشر الملف مرة واحدة كل 1024 قراءة رمز وسيكون التغيير بمقدار 1024 . للتأكد من هذا الأمر نفذ البرنامج التالي:

```
static void Main(string[] args)
{
    FileStream MyTextFile = new FileStream("44.txt", FileMode.OpenOrCreate);
    StreamReader Sr = new StreamReader(MyTextFile);
    StreamWriter Sw = new StreamWriter(MyTextFile);
    Random R = new Random();
    //*****
    int i = 0;
    Console.WriteLine(MyTextFile.Position.ToString());
    for (i = 0; i < 5000; i++)
        Sw.Write((char)R.Next(65,90));
    MyTextFile.Position = 0;
    Console.WriteLine(MyTextFile.Position.ToString());
    Sr.Read();
    Console.WriteLine(MyTextFile.Position.ToString());
}
```

تقوم الشفرة السابقة بما يلي :

1. ربطنا كائنا من النوع **FileStream** مع ملف باسم **44.txt** وربطنا مع هذا كائنين للقراءة والكتابة وهما **Sr** و **Sw** على الترتيب.
2. عرفنا متغير لتوليد أعداد عشوائية سنستخدمه لتوليد رموز عشوائية.
3. طبعنا موقع مؤشر الملف قبل الكتابة فنحصل على 0
4. كتبنا رموز عشوائية 5000 مرة
5. أعدنا مؤشر الملف إلى بداية الملف وللتأكد من ذلك قمنا بطباعة موقع مؤشر الملف
6. قمنا بعملية قراءة واحدة
7. طبعنا موقع مؤشر الملف قبل الكتابة فنحصل على 1024 !!!

على ماذا يدل هذا ؟ على أن عملية المنهج **Read** قام بالقراءة 1024 مرة ولهذا توقف مؤشر الملف عند هذه النقطة ولكن ماذا عن الرموز المقروءة؟ إنها موجودة داخل ذاكرة مؤقتة داخل كائنات الصنف **StreamReader** للتأكد من ذلك. ضع الشفرة التالية بدل السطرين الأخيرين من الشفرة السابقة:

```
char c = (char)Sr.Read();
Console.WriteLine("First char is {0} and Position={1}",
c,MyTextFile.Position);
c = (char)Sr.Read();
Console.WriteLine("Second char is {0} and Position={1}",c,
MyTextFile.Position);
```

1. قمنا بقراءة أول رمز وطباعته وطباعة موقع مؤشر الملف

2. قمنا بقراءة ثاني رمز وطباعته وطباعة موقع مؤشر الملف

قرأنا من الملف مرتين ولكن موقع مؤشر الملف لم يتغير...!!؟ عدم تغير موقع مؤشر الملف يدل على احد امرين:

a. أن القراءة لم تحدث مرتين ابدا بدليل أن موقع مؤشر الملف لم يتغير. ولكن لماذا تغيرت قيمة الرمز المقروء c من اين حصل على القيمة الثانية المطبوعة ؟

b. أن القراءة تمت بدليل أن قيمة الرمز المقروء تغيرت ولكن لماذا لم يتغير موقع مؤشر الملف ؟

الجواب هو وجود **Buffer** داخل كائن الصنف **StreamReader** طوله 1024 بايت بحيث يقرأ هذه الكتلة مرة واحدة من الملف إلى داخل الذاكرة لذلك تجد أن مؤشر الملف يقف عند 1024 (إذا كان الملف يحتوي على عدد رموز اقل سيقرا جميع الرموز) ويستمر في تزويدنا بالرموز من داخل الذاكرة عند كل استدعاء للمنهج **Read** فإذا قمنا بالقراءة 1024 مرة سيقوم بتزويدنا ب 1024 رمزا الذين يأتون بعد المجموعة الاولى وسترى أن مؤشر الملف يقف عند 2048 بينما يستمر الإمداد بالرموز من ال **Buffer** الممتلئ بالمجموعة الجديدة من الرموز.

```
static void Main(string[] args)
{
    FileStream MyTextFile = new FileStream("44.txt",
    FileMode.OpenOrCreate);
    StreamReader Sr = new StreamReader(MyTextFile);
    StreamWriter Sw = new StreamWriter(MyTextFile);
    Random R = new Random();

    //*****
    int i = 0;
    Console.WriteLine(MyTextFile.Position.ToString());
    for (i = 0; i < 5000; i++)
        Sw.Write((char)R.Next(65,90));
    MyTextFile.Position = 0;
    Console.WriteLine(MyTextFile.Position.ToString());
    char c = (char)Sr.Read();
    Console.WriteLine("First char is {0} and Position={1}",
c,MyTextFile.Position);
    for (i = 0; i < 1024; i++)
        Sr.Read();
    Console.WriteLine("And now Position= " +
MyTextFile.Position.ToString());
}
```

ستلاحظ أننا قمنا بعملية القراءة 1024 وبعدها أصبح موقع مؤشر الملف 2048 أي أن الكائن `sr` قام بقراءة الكيلوبايت الثاني ولذلك يقف مؤشر موقع الملف عند 2048 عند 2048 .

ماذا عن الكائن `StreamWriter`

هل هناك أمور أخرى يجب أن نأخذها بعين الاعتبار عند استخدام كائن الصنف `StreamWriter`؟ جرب المثال التالي:

```
static void Main(string[] args)
{
    FileStream MyTextFile = new FileStream("55.txt",
    FileMode.OpenOrCreate);
    StreamWriter Sw = new StreamWriter(MyTextFile);
    Random R = new Random();
    //*****
    int i = 0;
    Console.WriteLine(MyTextFile.Position.ToString());
    for (i = 0; i < 5000; i++)
        Sw.Write((char)R.Next(65,90));
    Console.WriteLine(MyTextFile.Position.ToString());
}
```

1. في البداية قمنا بطباعة موقع مؤشر الملف. حصلنا على 0 يؤكد وقوعنا عند بداية الملف
 2. قمنا بكتابة رموز عشوائية 5000 مرة
 3. قمنا بطباعة موقع مؤشر الملف. ويا للمفاجأة لم نحصل على 5000 بل حصلنا على 4096 !! أين بقية الرموز؟ الآن اجعل مكان العدد 5000 العدد 500 ستلاحظ أننا حصلنا على 0 و الآن اجعل مكان العدد 5000 العدد 4000 ماذا وجدت؟ ستجد 3072 . . ؟ إلا تلاحظ أن هذه كلها مضاعفات 1024؟
- في الحقيقة تقوم كائنات الصنف `StreamWriter` بتخزين الرموز المرسله داخل `buffer` حتى يصبح عدد الرموز 1024 ثم يكتب الرموز في الملف ثم يخزن المجموعة التالية وهكذا لذا تجد أن موقع مؤشر الملف عبارة مضاعفات ال 1024 . `true` . ذا كان عدد الرموز اقل من 1024 فإن الكائن ينتظر حتى نستدعي `Close` أو بواسطة المنهج `Flush` الذي يجعل الكائن يقوم بإرسال الرموز المخزنة عنده إلى الملف مباشرة أو بجعل الخاصية `AutoFlush` تأخذ القيمة `true` عندها سيقوم المنهج `Write` بإرسال أي رمز مقروء إلى الملف مباشرة.

مثال :

اكتب برنامج يستبدل بكل حرف `s` موجود داخل الملف الحرف `t`. الحل:

أنشئ مشروعا جديدا واكتب داخله الشفرة التالية :

```
static void Main(string[] args)
{
    FileStream MyTextFile = new FileStream("66.txt",
    FileMode.OpenOrCreate);
    StreamWriter Sw = new StreamWriter(MyTextFile);
    StreamReader Sr = new StreamReader(MyTextFile);
    //*****
    Sw.AutoFlush = true;
    MyTextFile.Seek(0, SeekOrigin.Begin);
    int index = 0; char c;
    while (Sr.Peek() != -1)
    {
```

```

c = (char)Sr.Read(); ;
if (c == 's')
{
    Sw.BaseStream.Seek(index, SeekOrigin.Begin);
    Sw.Write('t');
    Sw.BaseStream.Seek(index, SeekOrigin.Begin);
}
index++;
}
Sw.Close();

```

```

//*****
}

```

1. لاحظ انني جعلت الخاصية `AutoFlush = true` حتى تتم الكتابة مباشرة من نفس موقع المؤشر وليس بعد الانتهاء من القراءة.
2. لاحظ أن المتغير `index` يحفظ أماكن الرموز المقروءة فإذا كان هذا الرمز مساوياً لـ `s` استبدلناه بالرمز `t`.
3. لاحظ إنني استخدمت الحقل `BaseStream` وهو يحفظ كائن الملف الذي ترتبط به كائنات الـ `StreamReader` و لذلك يمكن الاستغناء عن `MyTextFile.Position` واستخدام `Sw.BaseStream.Position` مثلاً وكل المناهج والحقول التي يقدمها لنا الكائن `MyTextFile`.

مثال : من الملف `77.txt` ضع الرمز 'a' بدلا من 's' في جميع اسطر الملف

```

static void Main(string[] args)
{
    FileStream MyTextFile = new FileStream("77.txt",
    FileMode.OpenOrCreate);
    StreamReader Sr = new StreamReader(MyTextFile);
    StreamWriter Sw = new StreamWriter(MyTextFile);
    Sw.AutoFlush = true;
    //*****
    string AllChar = Sr.ReadToEnd();
    AllChar = AllChar.Replace('s', 'a');
    Sw.BaseStream.Seek(0, SeekOrigin.Begin);
    Sw.Write(AllChar);
    //*****
    Sw.Close();
    Sr.Close();
    MyTextFile.Close();
}

```

مثال : اقرأ جميع اسطر الملف `88.txt` واضف الحرف n إلى نهاية كل سطر

```

static void Main(string[] args)
{
    FileStream MyTextFile = new FileStream("88.txt",
    FileMode.OpenOrCreate);
    StreamReader Sr = new StreamReader(MyTextFile);
    StreamWriter Sw = new StreamWriter(MyTextFile);
    Sw.AutoFlush = true;
    string Line = "";
    int index = 0;
    //*****
    Sr.BaseStream.Seek(0, SeekOrigin.Begin);

```

```

while (Sr.Peek() >= 0)
{
    Line = Sr.ReadLine();
    Console.WriteLine(Line);
}

Console.WriteLine("*****");
Line = "";
Sw.BaseStream.Seek(0, SeekOrigin.Begin);
Sr.DiscardBufferedData();
//*****
while (Sr.Peek() >= 0)
{
    Line = Sr.ReadLine();
    Sw.BaseStream.Seek(index, SeekOrigin.Begin);
    Sw.WriteLine(Line + "\n");
    index = (int) Sw.BaseStream.Position;
}
//*****
Sw.BaseStream.Position = 0;
Sr.DiscardBufferedData();
Line = Sr.ReadToEnd();
Console.WriteLine(Line);
//*****
Sw.Close();
Sr.Close();
MyTextFile.Close();
//*****
}

```

1. تقوم حلقة **While** الأولى بقراءة الاسطر من داخل الملف وطباعتها على الشاشة
2. لاحظ أننا استخدمنا المنهج **DiscardBufferedData** من الكائن **Sr**. يقوم هذا المنهج بمسح الذاكرة المؤقتة الخاصة بالكائن **Sr** ويعيد منطقة القراءة إلى حيث يقف مؤشر الملف (في حالتنا هذه 0). لا يكفي جعل مؤشر الملف للكائن **MyTextFile** مساوياً للصفر حتى يبدأ الكائن **Sr** بالقراءة من بداية الملف (لماذا) ؟ لأن الكائن **Sr** لا يقوم بالقراءة من الملف وإنما ينسخ الجزء الضروري داخل ذاكرته المؤقتة، ويعطينا ما يلزم من داخل هذه الذاكرة حتى يصل إلى آخرها. ثم ينسخ كتلة أخرى وهكذا.

مثال : في الملف الثنائي 11.dat اكتب 20 عدد صحيح عشوائي ثم استبدل كل عدد زوجي بالعدد صفر

```

static void Main(string[] args)
{
    FileStream MyBinaryFile = new FileStream("11.dat", FileMode.OpenOrCreate);
    BinaryReader Reader = new BinaryReader(MyBinaryFile);
    BinaryWriter Writer = new BinaryWriter(MyBinaryFile);
    int i = 0; Random R = new Random();
    //*****
    for (i = 0; i < 20; i++)
        Writer.Write(R.Next(10, 100));
    //*****
    Reader.BaseStream.Seek(0, SeekOrigin.Begin);
    while (Reader.PeekChar() != -1)
        Console.Write(Reader.ReadInt32().ToString() + " ");
    //*****
}

```

```

Reader.BaseStream.Seek(0, SeekOrigin.Begin);
int flag = 0;
while (Reader.PeekChar() != -1)
{
    flag = Reader.ReadInt32();
    if (flag % 2 == 0)
    {
        Writer.BaseStream.Position -= 4;
        Writer.Write((int)0);
    }
}
//*****
Console.WriteLine();
Reader.BaseStream.Seek(0, SeekOrigin.Begin);
while (Reader.PeekChar() != -1)
{
    Console.Write(Reader.ReadInt32().ToString() + " ");
}
}

```

1. تقوم حلقة For بكتابة 20 عشوائي داخل الملف
2. بعد إعادة مؤشر الملف إلى بدايته تقوم حلقة while الأولى بقراءة الأعداد الصحيحة وطباعتها على الشاشة
3. تقوم حلقة while الثانية بقراءة عدد صحيح من داخل الملف، فإذا تأكدت أن العدد المقروء عدد زوجي جعلت مؤشر الملف يتراجع إلى الخلف بمقدار اربع بايتات (حجم العدد الصحيح من النوع int. راجع الجدول) لان القراءة تجعل مؤشر الملف يتراجع فوراً ومن ثم تكتب صفراً في الموضع الذي قرأت منه العدد الزوجي.

الصف File

بعد أن تناولنا مسألة وضع البيانات داخل الملفات سنرى كيف يمكن أن نحصل على المزيد من إمكانيات التعامل مع الملفات. تقدم لنا مكتبات .Net الصف File الذي يمدنا بمناهج مختلفة لمعالجة الملفات دون فتحها مثل نسخ ملفات، حذف ملفات، مواصفات الملفات، نقل ملفات الخ.

في الجدول التالي قائمة بأهم المناهج التي يقدمها لنا هذا الصف

الوصف	المنهج
يضيف نصاً إلى ملف أو تنشئ الملف في حال عدم وجوده و تعيد كائننا من النوع StreamWriter	AppendText
يأخذ مسارين الملفين وينشئ نسخة من ملف المسار الأول في المسار الثاني	Copy
ينشئ ملفاً	Create
ينشئ كائننا من الصف StreamWriter يستخدم للكتابة داخل ملف نصي	CreateText
يحذف ملف المعطى بالمسار	Delete
يحدد فيما إذا كان الملف المحدد بالمسار موجوداً أم لا	Exist
يعيد مواصفات ملف (ملف نظام، مخفي، للقراءة فقط .. الخ)	GetAttributes
يعيد تاريخ و وقت إنشاء الملف	GetCreationTime
يعيد تاريخ و وقت آخر وصول للملف	GetLastAccessTime

المنهج	الوصف
GetLastWriteTime	يعيد تاريخ و وقت آخر عملية كتابة للملف
Move	ينقل ملف إلى موقع جديد
Open	ينشئ كائن FileStream من اجل الملف المحدد بالمسار
OpenRead	يفتح ملف للقراءة وتعيد كائن من FileStream
OpenText	ينشئ كائن من الصنف StreamReader
OpenWrite	يفتح ملف للقراءة والكتابة وتعيد من اجله كائن من الصنف FileStream
SetAttribute	يحدد مواصفات ملف
SetCreationTime	يحدد تاريخ و وقت الإنشاء
SetLastAccessTime	يضببط تاريخ و وقت آخر وصول
SetLastWriteTime	يضببط تاريخ و وقت آخر عملية كتابة إلى ملف

واضح من الجدول السابق أن الصنف File غني بالمناهج المفيدة والتي تحتاجها أثناء التعامل مع الملفات فيما يلي امثلة على استخدامها.

مثال:

اكتب برنامجا ينشئ ملف بالاسم **MyFile.dat** فإذا كان الملف موجودا اخبر المستخدم عن ذلك وقم بإنهاء عمل البرنامج فيما عدا ذلك

اكتب 20 عدد عشوائيا داخل الملف

```
static void Main(string[] args)
{
    if (File.Exists("MyFile2.dat"))
        Console.WriteLine("File with this name exists, Try an other
name");
    else
    {
        FileStream F = File.Open("MyFile2.dat",
FileMode.OpenOrCreate);
        BinaryWriter Writer = new BinaryWriter(F);
        BinaryReader Reader = new BinaryReader(F);
        Random R = new Random();
        //*****
        Writer.BaseStream.Seek(0, SeekOrigin.Begin);
        for (int i = 0; i < 20; i++)
            Writer.Write(R.Next(10, 100));
        //*****
        Writer.BaseStream.Seek(0, SeekOrigin.Begin);
        while (Reader.PeekChar() > -1)
            Console.Write(Reader.ReadInt32().ToString() + " ");
        Console.WriteLine();
    }
}
```

تقوم حلقة **While** الأخيرة بقراءة القيم من الملف وطباعتها على الشاشة. لاحظ أن تشغيل البرنامج أول مرة سيظهر أرقام على الشاشة (إذا لم يوجد ملف بنفس الاسم في نفس المسار)، بينما سيظهر رسالة عن وجود الملف في المرة الثانية لأن المنهج **Open** قام بإنشاء الملف. كذلك لاحظ المنهج **Open** اخذ نفس القيم التي أن يأخذها باني **Constructor** الصنف **FileStream**.

مثال: قم بكتابة برنامج يقوم بنقل الملف 22.dat إلى المسار C:\Users\Adel\Documents\22.dat

الحل: سنستخدم المنهج **Move** الذي يأخذ بارامترين الأول مسار الملف المصدر والثاني المسار الذي سيكون فيه الملف بعد النقل

```
static void Main(string[] args)
{
    File.Move("MyFile1.dat", @"C:\Users\Adel\Documents\MyFile1.dat");
}
```

طبعا هناك شرطان يجب توفرهما كي تعمل الشفرة السابقة بشكل صحيح

1. أن يكون الملف المصدر موجودا في المسار المذكور

2. أن لا يكون في المسار المستقبل قبل النقل ملف بنفس الاسم

إذا لم يتوفر احد الشرطين فستحصل على رسالة خطأ ولتجنب ذلك يجب فحص المسارين أولاً، انظر الرسالة التالية:

```
static void Main(string[] args)
{
    if (File.Exists("MyFile1.dat") &&
        !File.Exists(@"C:\Users\Adel\Documents\MyFile3.dat"))
    {
        File.Move("MyFile1.dat", @"C:\Users\Adel\Documents\MyFile1.dat");
    }
}
```

ملاحظة:

في الشفرة السابقة البارامتر الثاني ليس من الضروري أن يكون اسم الملف في مسار النقل هو **MyFile1.dat** أي نفس اسم الملف المصدر، المهم أن الاسم الذي تختاره لن يكون موجودا في المجلد الذي ستنتقل إليه.

مثال:

انقل ملف من مسار إلى آخر بحيث يستبدل الملف الجديد بالملف الموجود (أي نحذف الملف من المسار المستقبل في حالة وجود ملف بنفس الاسم الذي اخترناه كملف مستقبل).

الحل:

```
static void Main(string[] args)
{
    string Path1="MyFile1.dat";
    string Path2=@"C:\Users\Adel\Documents\MyFile1.dat";
    //*****
    if (File.Exists("MyFile1.dat"))
    {
        if (File.Exists(Path2))
            File.Delete(Path2);
        File.Move(Path1, Path2);
    }
    else
    {
        Console.WriteLine("Make sure That File With This name is Exist");
    }
}
```

لاحظ أننا تأكدنا من وجود الملف في المسار **Path2** قبل أن نقوم بأي عملية حذف، فإذا لم يكن الملف موجودا في المسار **Path2** لم نحتاج إلى الحذف وننقل الملف الموجود في المسار **Path1** إلى المسار **Path2** مباشرة . أما إذا كان موجودا فإننا نحذفه ثم وننقل الملف الموجود في المسار **Path1** إلى المسار **Path2** بعد ذلك.

قد تحتاج إلى نسخ ملف من مكان إلى آخر ولفعل هذا نستخدم المنهج **Copy** من الصنف **File** كما هو موضح بالمثال:

خلافًا للمنهج **Move** لا يقوم المنهج **Copy** بحذف الملف الموجود في المسار المصدر ، انظر :

```
static void Main(string[] args)
{
    string SourcePath="20.dat";
    string DestPath=@"C:\Users\Adel\Documents\40.dat";
    int i = 0;
    File.Copy(SourcePath, DestPath);
    //File.Copy(SourcePath, DestPath,true);
}
```

هناك صيغتان للمنهج **Copy** الاولى لا تسمح بإعادة الكتابة في الملف المستقبل في حال وجود ملف بنفس الاسم في المسار المستقبل في حين تسمح الاخرى بذلك (انظر السطر الأخير من الشفرة السابقة). لو قمت بتنفيذ هذه الشفرة فستحصل في المرة الثانية على خطأ لأن الملف قد تم نسخه في مسار الملف المستقبل ولكن بتعديل البرنامج هكذا:

```
static void Main(string[] args)
{
    string SourcePath="20.dat";
    string DestPath=@"C:\Users\Adel\Documents\40.dat";
    int i = 0;
    // File.Copy(SourcePath, DestPath);
    File.Copy(SourcePath, DestPath,true);
}
```

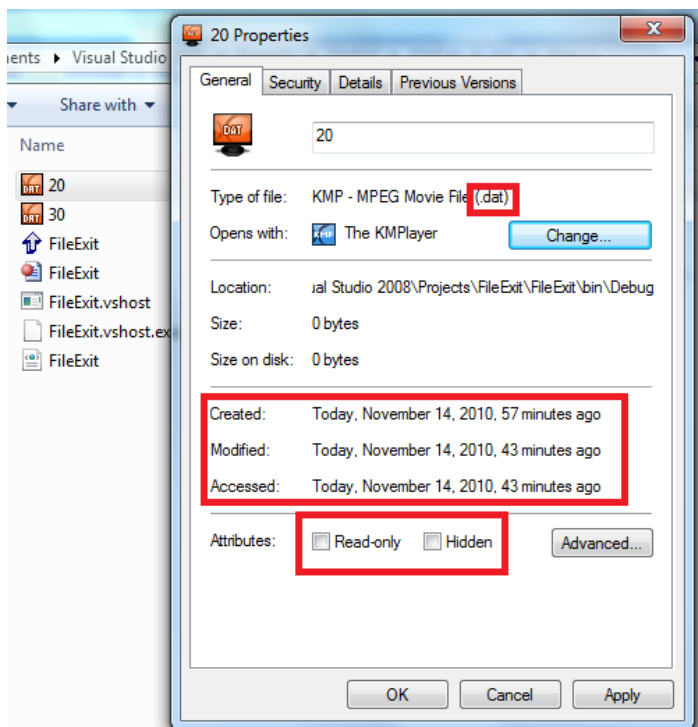
تسمح هذه الشفرة بالكتابة فوق الملف في حال تطابق اسم الملف المستقبل المحدد في البارامتر الثاني من المنهج مع ملف موجود سلفا في

نفس المسار.

سنرى الآن كيف يمكن أن نعيد ضبط مواصفات ولكن قبل ذلك قم بما يلي:

1. أنشئ ملفا في مسار معين أو اختر احد الملفات التي انشأناها في أمثلة سابقة (أنا سأختار الملف **20.dat**).

2. اذهب إلى مسار الملف وانقر على أيقونة الملف باليمين واختر البند خصائص من القائمة ستحصل على نافذة تشبه النافذة التالية:



لاحظ المستطيلات الحمراء سترى في المستطيل الصغير (**.dat**). تنسيق الملف , ثم في الكبير سترى تاريخ الإنشاء والتعديل و آخر مرة فتحنا فيها الملف وكذلك إلى الوقت ثم في المستطيل الذي يليه نرى إذا كان الملف للقراءة فقط ام لا , وكذلك حالته من حيث الظهور والإخفاء. قبل أن نقوم بتغيير أي لابد أن نعرف أن مواصفات الملف **FileAttributes** عبارة سلسلة من البتات (**FlagsAttribute**) يمثل كل بت منها دليل أو علم (**Flag**) على شيء معين ولتغيير احد بت معين دون التأثير على بقية البتات نحتاج إلى استخدام **Operations Bitwise** على الأعداد. مثال على سلاسل البتات يشكل منها علما لون معين

```
[FlagsAttribute]
enum MultiHue
{
    Black = 0,
    Red = 1,
    Green = 2,
    Blue = 4
}
```

لنرى بمثال كيف نقوم بجعل ملف معين ملفا للقراءة فقط:

```
static void Main(string[] args)
{
    string SourcePath="20.dat";
    if (File.Exists(SourcePath))
    {
        FileAttributes Atr = File.GetAttributes(SourcePath);
        File.SetAttributes(SourcePath, Atr | FileAttributes.ReadOnly);
    }
}
```

للتأكد من نجاح العملية اذهب إلى مسار الملف وانقر على ايقونة الملف باليمين ثم اختر البند خصائص وراقب التغييرات. تأمل الشفرة الماضية ستلاحظ أنني اختبرت وجود الملف كما فعلنا ذلك مرارا ثم حصلت على مواصفات الملف بواسطة المنهج **GetAttributes** و وضعتها داخل المتغير **Atr** . و لكي نضمن جعل الملف ملفا للقراءة فقط اضفنا العلم **FileAttributes.ReadOnly** بواسطة العملية | والتي تعني **OR** أي نقوم بعمل جمع منطقي لكل بت مع نظيره. ماذا لو أردت أن اعكس العملية السابقة؟ أي أن اجعل ملفا معيناً , ليس للقراءة فقط ؟

```
static void Main(string[] args)
{
    string SourcePath="20.dat";
    if (File.Exists(SourcePath))
    {
        FileAttributes Atr = File.GetAttributes(SourcePath);
        File.SetAttributes(SourcePath, Atr & ~FileAttributes.ReadOnly);
    }
}
```

تأكد بنفس الطريقة الماضية؟ ماذا تلاحظ؟ ستلاحظ أن الخاصية **ReadOnly** لم تعد فعالة.. هل رأيت كم هو سهل ضبط خواص ملف؟ اعمل نفس الشيء مع الخاصية **Hidden** استبدل السطر التالي بالسطر الأخير من الشفرة السابقة :

```
File.SetAttributes(SourcePath, Atr | FileAttributes.Hidden);
```

ستلاحظ أن الخاصية **Hidden** أصبحت فعالة والان اعكس الحال أي اجعلها غير فعالة باستخدام السطر

```
File.SetAttributes(SourcePath, Atr & ~FileAttributes.Hidden);
```

تأكد بعد كل مرة تنفذ فيها شفرة. وستلاحظ أن بإمكانك ضبط أي خاصية. هناك سؤال يطرح نفسه ماذا لو كنت أريد أن أعرف إذا كانت الخاصية فعالة أم غير فعالة ولا أحتاج إلى ضبطها كما كنت أفعل قبل قليل ؟
مثال : تأكد إذا كانت الخاصية **Hidden** فعالة أم غير فعالة وهات رسالة تفيد ذلك. الحل:

```
static void Main(string[] args)
{
    string SourcePath="20.dat";
    if (File.Exists(SourcePath))
    {
        FileAttributes Atr = File.GetAttributes(SourcePath);
        short Active = (short) (Atr & FileAttributes.Hidden);
        if (Active!=0)
            Console.WriteLine("Yes Active");
    }
}
```

سنرى كيف سنحصل على معلومات تخص الوقت والتاريخ (فقد نحتاج إلى ترتيب الملفات حسب التاريخ ذات يوم) من أجل هذا النوع من المعلومات لدينا العديد من المناهج نبدأ بالمنهج **GetLastWriteTime** الذي يعطينا تاريخ آخر تعديل. انظر إلى الشفرة التالية:

```
static void Main(string[] args)
{
    //You can write path to any file here
    string SourcePath = "30.dat";
    if (File.Exists(SourcePath))
    {
        DateTime dt = File.GetLastWriteTime(SourcePath);
        Console.WriteLine("The last write time for this file was {0}.", dt);
    }
}
```

قمنا بتخزين بيانات التاريخ والوقت الخاصة بآخر مرة كتبنا فيها داخل الملف في المتغير **dt** من الصنف **DateTime**. يعطينا الصنف **DateTime** إمكانات جبارة في معالجة الوقت حيث نستطيع على الحصول الساعة والدقيقة والثانية بشكل منفصل كل حدة و على شكل سلسلة إضافة إلى التاريخ (يوم شهر سنة) لكن دراسة هذا الصنف ليست موضوعنا الآن. في المثال التالي سنرى كيف نستطيع تعديل تاريخ آخر تعديل بواسطة المنهج **SetLastWriteTime**

```
static void Main(string[] args)
{
    //You can write path to any file here
    string SourcePath = "30.dat";
    if (File.Exists(SourcePath))
    {
        DateTime dt = File.GetLastWriteTime(SourcePath);
        Console.WriteLine("Last time I wrote time for this file was {0}.", dt);

        File.SetLastWriteTime(SourcePath, new DateTime(1990, 12, 2));
        dt = File.GetLastWriteTime(SourcePath);
        Console.WriteLine("My New Last time {0}.", dt);
    }
}
```

في المثال القادم سنرى كيف سنحصل على وقت و تاريخ انشاء ملف ثم تعديله :

```
static void Main(string[] args)
{
    //You can write path to any file here
    string SourcePath = "30.dat";
    if (File.Exists(SourcePath))
    {
        DateTime dt = File.GetCreationTime(SourcePath);
        Console.WriteLine("Last time I wrote time for this file was {0}.", dt);
        File.SetCreationTime(SourcePath, DateTime.Now);
        dt = File.GetCreationTime(SourcePath);
        Console.WriteLine("Last time I wrote time for this file was {0}.", dt);
    }
}
```

يعيد المنهج **GetCreationTime** تاريخ و وقت الإنشاء وبعد ذلك يمكن تعديلها بواسطة المنهج **SetCreationTime** . ولا يختلف الأمر كثير عما هو عليه في المناهج الخاصة بآخر تعديل **GetLastWriteTime** و **SetLastWriteTime**. انظر المثال التالي الذي يعالج اخر مرة حصل فيها الوصول إلى ملف .

```
public static void Main()
{
    //*****
    string SourcePath = "30.dat";
    if (File.Exists(SourcePath))
    {
        DateTime dt = File.GetLastAccessTime(SourcePath);
        Console.WriteLine("The last write time for this file was {0}.", dt);

        File.SetLastAccessTime(SourcePath, new DateTime(1982, 2, 12));

        dt = File.GetLastAccessTime(SourcePath);
        Console.WriteLine("The last write time for this file was {0}.", dt);
    }
}
```

ملاحظة: بعد أن تقوم بتنفيذ الشفرات السابقة تأكد من تواريخ الإنشاء التعديل بالذهاب إلى مسار الملف ثم من خصائص الملف كما بينا ذلك في وقت سابق.

المناهج CreateText,AppendText,OpenText

يقوم المنهج **CreateText** بإنشاء ملف نصي باسم يساوي البارامتر المسند إلى المنهج. إذا كان هناك ملف يملك نفس الاسم المسند إلى المنهج فإن المنهج **يمسح** ما داخله ويسمح لنا بالكتابة فوق الملف ويعيد كائن من النوع **StreamWriter**.
 يقوم المنهج **AppendText** بإنشاء ملف نصي باسم يساوي البارامتر المسند إلى المنهج. إذا كان هناك ملف يملك نفس الاسم المسند إلى المنهج فإن المنهج **لا يمسح** ما داخل الملف وإنما يذهب إلى آخره ليضيف النص الجديد و يعيد كائن من النوع **StreamWriter**.
 يقوم المنهج **OpenText** بفتح ملف موجود ويعيد كائن من النوع **StreamReader**.
 سنرى في المثال التالي كيف يمكن الاستفادة من المناهج المذكورة

```

public static void Main()
{
    string path = @"MyTestFile.txt";
    StreamWriter FirstWriter;
    StreamWriter SecondWriter;
    StreamReader Reader;
    //*****
    // This text is added only once to the file.
    if (!File.Exists(path))
    {
        // Create a file to write to.
        FirstWriter = File.CreateText(path);
        FirstWriter.WriteLine("Hello,");
        FirstWriter.WriteLine("How");
        FirstWriter.WriteLine("Are You");
        FirstWriter.Close();
    }
    // This text is always added, making the file longer over time
    // if it is not deleted.
    SecondWriter = File.AppendText(path);
    SecondWriter.WriteLine("This");
    SecondWriter.WriteLine("is new");
    SecondWriter.WriteLine("Line");
    SecondWriter.Close();
    //*****
    // Open the file to read from.
    Reader = File.OpenText(path);
    string str = "";
    while ((str = Reader.ReadLine()) != null)
    {
        Console.WriteLine(str);
    }
}

```

نقوم باختبار وجود الملف الممثل في المسار **path** فإذا لم يكن موجودا نقوم بإنشائه بواسطة المنهج **CreateText** ونضيف اليه 3 اسطر (Hello,- How - Are You) ،بعد ذلك نقوم المجموعة الثانية بإضافة 3 اسطر اخرى هي (This - is new - Line) بعد فتحه بالمنهج **AppendText**. في حالة تنفيذ البرنامج اكثر من لاحظ ان المجموعة الاولى ستضيف اسطرها مرة واحدة فقط لانه الشرط **if** يمنع الوصول إلى المجموعة الاولى من الشفرة بسبب كون الملف موجود في تستمر المجموعة الثانية باضافة الاسطر الخاصة بها. وتقوم المجموعة الأخيرة بقراءة الملف وطباعة شفرته.

هناك مناهج اخرى مثل **WriteAllLines** و **AppendAllText** و **AppendAllText** و **ReadAllBytes** و **WriteAllBytes** و **WriteAllText** و **ReadAllText** و **ReadAllLines** .. الخ وسأترك أمر البحث عن عملها كتمرين للقاريء.

الصف Directory

إذا كان الصف File يقوم بالعمل مع الملفات فإن الصف Directory هو نظيره الذي يعمل مع المجلدات. يقدم لنا الصف Directory قائمة بالمناهج التي نحتاجها لمعالجة المجلدات اليك في الجدول التالي قائمة باهم المناهج:

المنهج	الوصف
CreateDirectory	ينشئ مجلد
Delete	يحذف مجلد مع كل محتوياته
Exist	يعيد True إذا كان المجلد موجودا
GetCreationTime	يعيد تاريخ و وقت إنشاء مجلد
GetCurrentDirectory	يعيد المجلد الحالي (المجلد الذي يحتوي على ملف)
GetDirectories	يعيد المجلد الفرعية لمجلد معين
GetDirectoryRoot	يعيد الدليل الجذر لمجلد معين (C:\,D:\ ...)
GetFiles	يعيد الملفات التي في مجلد معين
GetFileSystemEntries	يعيد ادخالات نظام الملفات لمجلد معين
GetLastAccessTime	يعيد تاريخ و وقت آخر وصول للمجلد
GetLastWriteTime	يعيد تاريخ و وقت آخر عملية تعديل للمجلد
GetLogicalDrives	يعيد اسماء محركات الاقراص المنطقية للحاسب
GetParent	يعيد المجلد الاب لمجلد معين
Move	نقل مجلد (مع مجلداته الفرعية)
SetCreationTime	ضبط وقت إنشاء المجلد
SetCurrentDirectory	تحديد المجلد الحالي (المجلد الافتراضي)
SetLastAccessTime	ضبط تاريخ و وقت آخر وصول
SetLastWriteTime	ضبط تاريخ و وقت آخر عملية كتابة إلى ملف

يوضح المثال التالي كيف يقوم المنهج CreateDirectory بإنشاء مجلد جديد:

```
public static void Main()
{
    string path = @"C:\Users\Adel\Documents\MyNewFolder";
    Directory.CreateDirectory(path);
}
```

إذا لم يكن كان مجلد بنفس الاسم لن يقوم المنهج بعمل أي شيء فيما عدا ذلك سينشئ المنهج مجلدا بالاسم المطلوب.

يوضح المثال التالي كيفية حذف مجلد بمحتوياته بواسطة المنهج **Delete**

```
public static void Main()
{
    string path = @"C:\Users\Adel\Documents\MyNewFolder";
    Directory.Delete(path, true);
}
```

إذا كان البارامتر الثاني يساوي **False** فإن المنهج لن يحذف إلا مجلدات فارغة وسيحدث خطأ في حالة محاولتك لحذف مجلد غير فارغ مع القيمة **False** البارامتر الثاني. طبعاً غير الأخطاء التي ممكن أن تحدث بسبب عدم وجود المجلد...الخ
يمكن التأكد من وجود المجلد من عدمه بواسطة المنهج **Exist** والشفرة التالية توضيح كيفية القيام بذلك:

```
public static void Main()
{
    string path = @"C:\Users\Adel\Documents\Armik Music";
    if (Directory.Exists(path))
        Directory.Delete(path, true);
}
```

يمكن الحصول على أسماء الملفات الموجودة داخل المجلد مباشرة (مباشرة يعني ليست موجودة داخل مجلدات فرعية) باستخدام المنهج **GetFiles** كما هو موضح في المثال التالي:

```
public static void Main()
{
    string path = @"C:\Users\Adel\Documents\Armik Music";
    if (Directory.Exists(path))
    {
        string[] MyFiles = Directory.GetFiles(path);
        foreach (string str in MyFiles)
            Console.WriteLine(str);
    }
}
```

كما يمكن أن بنفس الطريقة تحصل على كل المجلدات التي تقع داخل مجلد معين باستخدام المنهج **GetDirectories**

```
public static void Main()
{
    string path = @"C:\Users\Adel\Documents\Armik Music";
    if (Directory.Exists(path))
    {
        string[] MyFolder = Directory.GetDirectories(path);
        foreach (string str in MyFolder)
            Console.WriteLine(str);
    }
}
```

يمكن استخدام الشفرة التالية للحصول على كافة المجلدات مع المجلدات الفرعية الموجودة في مجلد معين

```
public static ArrayList List = new ArrayList();
public static void Main()
{
    string path = @"C:\Users\Adel\Documents\Armik Music";
    int i = 0, j=0;
    if (Directory.Exists(path))
    {
        List.Add(path);
        while (i < List.Count)
        {
```

```

        path = (string)List[i];
        MyFolder = Directory.GetDirectories(path);
        for (j = 0; j < MyFolder.Length; j++)
            List.Add(MyFolder[j]);
        Console.WriteLine(path);
        i++;
    }
    // Here You Can Add Code To Get All Files From Subdirectories
    //
    // ضع الشفرة التالية هنا للحصول كافة الملفات
    //*****
}
}

```

لاحظ اننا استخدمنا مصفوفة ديناميكية **ArrayList** لاننا لا نعرف عدد المجلدات الفرعية وابناءها. لا تنسى إضافة الترويسة **using System.Collections;** إلى بداية الملف. لاحظ أننا أخذنا جميع المجلدات التي تقع مباشرة داخل المجلد الموصوف بالمسار **path** بواسطة المنهج **GetDirectories** و وضعناها داخل المصفوفة **MyFolder** مؤقتاً، ثم نقلنا عناصر هذه المصفوفة إلى المصفوفة الديناميكية **List** التي تخزن مسارات المجلدات ، ثم نختار المجلد التالي من مصفوفة المسارات . يمكن إضافة القليل من الأسطر إلى الشفرة السابقة بعد نهاية الحلقة **While** مباشرة للحصول على كل الملفات الموجودة في كل المجلدات الفرعية . وهذه هي الشفرة المطلوبة

```

Console.WriteLine("*****");
i = 0;
while (i < List.Count)
{
    path = (string)List[i];
    MyFiles = Directory.GetFiles(path);
    for (j = 0; j < MyFiles.Length; j++)
        Console.WriteLine(MyFiles[j]);
    i++;
}

```

سنرى في المثال القادم كيف يمكن أن نحصل على الملف الذي يعمل فيه البرنامج وكذلك القرص المنطقي الذي يحتوي على البرنامج وقائمة بالأقراص الموجودة. لدينا الشفرة التالية ما يمكن عمله بواسطة المناهج **GetCurrentDirectory-** **(GetDirectoryRoot- GetLogicalDrives**

```

public static void Main()
{
    string CurDir=Directory.GetCurrentDirectory();
    Console.WriteLine(CurDir);
    Console.WriteLine(Directory.GetDirectoryRoot(CurDir));
    Console.WriteLine("*****");
    string[] Drivers = Directory.GetLogicalDrives();
    foreach (string str in Drivers)
        Console.WriteLine(str);
}
}

```

قد يحدث أنك تريد محتويات مجلد مباشرة دون التطرق إلى محتويات المجلد الفرعية ، كما أنك لا تريد التفريق بين مجلد وملف؛ أي أنك تريد عمل المنهجين **GetFiles** , **GetDirectories** في آن واحد لفعل ذلك لدينا المنهج **GetFileSystemEntries**

```
public static void Main()
{
    string[] Folders =
    Directory.GetFileSystemEntries(@"C:\Users\Adel\Documents\Armik Music");
    foreach (string str in Folders)
        Console.WriteLine(str);
}
```

و إذا أردنا أن نعرف خصائص مجلد أب لمجلد ما (المجلد الحاوي لمجلد ما)، نستخدم المنهج GetParent كما موضح بالمثل التالي:

```
public static void Main()
{
    DirectoryInfo FolderParent =
    Directory.GetParent(@"C:\Users\Adel\Documents\Armik Music");
    Console.WriteLine(FolderParent.FullName);
    Console.WriteLine(FolderParent.Name);
}
```

لاحظ أن المسار الكامل للمجلد الحاوي لمجلد معين موجود داخل الحقل FullName من كائنات الصنف DirectoryInfo. انظر كيف يمكن نقل مجلد من مكان إلى آخر باستعمال المنهج

```
public static void Main()
{
    Directory.Move(@"mm", @"C:\Users\Adel\Documents\Armik Music1");
}
```

تنطبق عليها شروط Move من أجل الملفات وهي للتذكير :

1. أن يكون المجلد المصدر موجودا في المسار المذكور
 2. أن لا يكون في المسار المستقبل قبل النقل مجلد بنفس الاسم
- لذلك لا بد من استخدام المنهج Exist لفحص وجود مجلد من عدمه قبل القيام بعملية نقل.

```
public static void Main()
{
    string source = @"mm";
    string destination = @"C:\Users\Adel\Documents\MyNewMM";
    if (Directory.Exists(source) && Directory.Exists(destination))
        Directory.Move(source, destination);
    else
        Console.WriteLine("No Way To Remove");
    //*****
}
```

لا تختلف دوال معالجة وقت الانشاء و التعديل وآخر وصول عما هو عليه في الملفات لذلك سأتركها كتمرين للقاريء