Лабораторная работа №4

ХАРЬКОВСКИЙ НАЦИОНАЛЬНЫЙ АВТОМОБИЛЬНО-ДОРОЖНЫЙ УНИВЕРСИТЕТ

ФАКУЛЬТЕТ МЕХАТРОНИКИ ТРАНСПОРТНЫХ СРЕДСТВ

Кафедра информатики

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

по проведению лабораторных работ по дисциплине «Программирование» для студентов специальности 6.050201 “Системная инженерия”

Разработчик - доцент кафедры информатики
кандидат технических наук,
старший научный сотрудник
Тимонин Владимир Алексеевич

Харків 2011

Лабораторная работа №4
Исследование возможностей интегрированной среды разработки Visual C# для создания приложений, работающих с файловой системой (диски, папки и файлы).

Цель работы – исследовать возможности интегрированной среды разработки Visual Studio 2010 и получить практические навыки по созданию приложений, работающих с файловой системой (диски, папки и файлы).

1. Теоретические сведения

Классы для работы с файловой системой, которые позволяют манипулировать индивидуальными файлами, а также взаимодействовать со структурой каталогов, находятся в пространстве System. IO. Основными классами являются Directory, DirectoryInfo, File, FileInfo и Path.

Классы Directory и DirectoryInfo используются для манипуляций структурой каталогов машины. Классы File и FileInfo служат для манипуляций множеством файлов данной машины.

Классы Directory и File предлагают операции создания, удаления, копирования и перемещения с использованием различных статических членов. Родственные им классы FileInfo и DirectoryInfo предлагают подобную функциональность в виде методов уровня экземпляра (и потому должны размещаться в памяти с помощью ключевого слова new).

Класс Path выполняет операции для экземпляров класса String, содержащих сведения пути к файлу или каталогу.

1.1. Класс DirectoryInfo

Класс DirectoryInfo унаследовал значительную часть своего поведения от абстрактного базового класса FileSystemInfo, члены которого используются для получения общих характеристик(таких как время создания, различные атрибуты и т.д.) определенного файла или каталога. Основными свойствами класса FileSystemInfo являются:

- **Attributes** - получает или устанавливает ассоциированные с текущим файлом атрибуты, которые представлены перечислением FileAttributes (доступный только для чтения, зашифрованный, скрытый или сжатый);
- **CreationTime** - получает или устанавливает время создания текущего файла или каталога;
- **Exists** - использоваться для определения, существует ли данный файл или каталог;
- **Extension** - извлекает расширение файла;
- **FullName** - получает полный путь к файлу или каталогу;
- **LastAccessTime** - получает или устанавливает время последнего доступа к текущему файлу или каталогу;
- **LastWriteTime** - получает или устанавливает время последней записи в текущий файл или каталог;
- **Name** - получает имя текущего файла или каталога.

Класс DirectoryInfo содержит набор членов, используемых для создания, перемещения, удаления и перечисления каталогов и подкаталогов.

Основными членами класса являются:

- **Create() (CreateSubdirectory())** - создает каталог (или набор подкаталогов) по заданному путевому имени;
- **Delete()** - удаляет каталог и все его содержимое;
- **GetDirectories()** - возвращает массив объектов DirectoryInfo, представляющих все подкаталоги в текущем каталоге;

- **GetFiles()** - извлекает массив объектов FileInfo, представляющий множество файлов в заданном каталоге;
- **MoveTo()** - перемещает каталог со всем содержимым по новому пути;
- **Parent** - извлекает родительский каталог данного каталога;
- **Root** - получает корневую часть пути.

Работа с типом DirectoryInfo начинается с указания определенного пути в качестве параметра конструктора. Если требуется получить доступ к текущему рабочему каталогу (т.е. каталогу выполняющегося приложения), применяется нотация ".". Например, оператор

DirectoryInfo dirl = new DirectoryInfo(".");

позволяет привязаться к текущему рабочему каталогу, а оператор

DirectoryInfo dir2 = new DirectoryInfo(@"C:\Windows");

позволяет привязаться к C:\Windows, используя литеральную строку.

Пример 1. Нижеприведенный пример демонстрирует возможность получения информации о каталоге C:\WINDOWS.

```
private void button1_Click(object sender, EventArgs e)
    {
     DirectoryInfo dir = new DirectoryInfo(@"C:\Windows");
     listBox1.Items.Add("*** Информация о каталоге ***");
     listBox1.Items.Add("Полное имя: "+dir.FullName);
     listBox1.Items.Add("Имя каталога: "+dir.Name);
     listBox1.Items.Add("Родительский каталог: "+dir.Parent);
     listBox1.Items.Add("Время создания: "+dir.CreationTime);
     listBox1.Items.Add("Атрибуты: "+dir.Attributes);
     listBox1.Items.Add("Корневой каталог: "+dir.Root);
    }
```

Пример 2. Нижеприведенный пример демонстрирует возможность получения информации с помощью метода GetFiles() о всех файлах, например, имеющих расширение jpg, расположенных в каталоге C:\Windows\Web\Wallpaper.

```
private void button1_Click(object sender, EventArgs e)
    {
       DirectoryInfo dir1 = new DirectoryInfo(@"C:\Windows\Web\Wallpaper");
       // Получить все файлы с расширением *.jpg.
       FileInfo[] imageFiles = dir1.GetFiles("*.jpg", SearchOption.AllDirectories);
       listBox2.Items.Add("Найдено " + imageFiles.Length + " *.jpg файлов");
       foreach (FileInfo f in imageFiles)
       {
          listBox2.Items.Add("Имя файла: " + f.Name);
          listBox2.Items.Add("Размер файла: " + f.Length);
          listBox2.Items.Add("Время создания: " + f.CreationTime);
          listBox2.Items.Add("Атрибуты: " + f.Attributes);
       }
    }
```

1.2. Класс Directory

Класс Directory позволяет выполнять такие стандартные операции, как создание, копирование, перемещение, переименование и удаление папок. Класс Directory также позволяет получать и задавать сведения о типе DateTime, относящиеся к созданию, доступу и записи каталога.

Все методы Directory статические, поэтому если необходимо выполнить только одно действие,

более эффективным может оказаться использование метода Directory, а не соответствующего экземпляра метода DirectoryInfo. Большинство методов класса Directory требуют указывать путь к каталогу, с которым работает пользователь.
Статические члены класса Directory повторяют функциональность, предоставленную членами уровня экземпляра, которые определены в Directorylnfo, и обычно возвращают строковые данные вместо строго типизированных объектов Directorylnfo.

Все статические методы класса Directory выполняют проверку безопасности для всех методов. Если необходимо использовать объект неоднократно, рекомендуется использовать соответствующий метод экземпляра DirectoryInfo , поскольку в этом случае проверка безопасности будет требоваться не всегда.

В операторах путь должен указывать на файл или каталог. Заданный путь может также указывать на относительный путь или путь к серверу с именем общего ресурса, например: c:\\MyDir, MyDir\\MySubdir, \\\\MyServer\\MyShare.

По умолчанию всем пользователям предоставляется неограниченный доступ к новым каталогам с правом чтения и записи. Если запрашивается разрешение для каталога, путь к которому завершается символом-разделителем каталогов, то запрос разрешения относится ко всем вложенным каталогам, например "D:\Temp\". Если разрешение требуется только для одного конкретного каталога, то в конце строки должен быть символ ".",например, "D:\Temp\.".

Основными методами класса Directory являются:

- **CreateDirectory()** - создает все каталоги по заданному пути;
- **Delete()** - удаляет заданный каталог;
- **Exists()** - определяет, указывает ли заданный путь на существующий каталог на диске;
- **GetAccessControl()** – возвращает права доступа для указанной папки;
- **GetCreationTime()** - получает дату и время создания каталога;
- **GetCurrentDirectory()** – возвращает текущую рабочую папку приложения, в которую будут сохраняться файлы, для которых не указан конкретный путь;
- **GetDirectories()** - получает имена подкаталогов в указанном каталоге;
- **GetDirectoryRoot()** - возвращает для заданного пути сведения о томе и корневом каталоге по отдельности или сразу;
- **GetFiles()** - возвращает имена файлов в заданном каталоге;
- **GetFileSystemEntries()** - возвращает имена всех файлов и подкаталогов из указанного каталога;
- **GetLastAccessTime()** - возвращает время и дату последнего обращения к указанному файлу или каталогу;
- **GetLastWriteTime()** - возвращает время и дату последней операции записи в указанный файл или каталог;
- **GetLogicalDrives()** - извлекает имена логических устройств данного компьютера в формате "<имя устройства>:\".;
- **GetParent()** - извлекает родительский каталог, на который указывает абсолютный или относительный путь;
- **Move()** - перемещает файл или каталог со всем его содержимым в новое местоположение. Имеет синтаксис:

public static void Move(string sourceDirName, string destDirName),

где **sourceDirName** - путь к файлу или каталогу, который необходимо переместить, **destDirName** - путь к новому местоположению sourceDirName. Если sourceDirName является файлом, то параметр destDirName также должен быть именем файла.

- **SetAccessControl()** - назначает права доступа к заданному каталогу;
- **SetCreationTime()** - устанавливает дату и время создания заданного файла или каталога;
- **SetCurrentDirectory()** - устанавливает заданный каталог в качестве текущего рабочего каталога приложения;
- **SetLastAccessTime()** - устанавливает время и дату последнего обращения к заданному файлу или каталогу;

- **SetLastWriteTime()** - устанавливает дату и время последней записи в файл или каталог.


Пример 3. В нижеприведенном примере кода определяется, существует ли определенный каталог, если каталог существует, он удаляется, в противном случае — создается. Затем каталог перемещается, в нем создается файл и производится подсчет файлов в каталоге.

```
private void button1_Click(object sender, EventArgs e)
    {
     string path = @"d:\MyDir";
     string target = @"d:\TestDir";
     if (!Directory.Exists(path)) // Определяем, существует ли каталог
       {
        // Создаем каталог, если он не существует
        Directory.CreateDirectory(path);
       }
     if (Directory.Exists(target))
       {
        // Удаляем каталог TestDir, если он там есть
        Directory.Delete(target, true);
       }
     // Заменяем каталог MyDir на каталог TestDir
     Directory.Move(path, target);
     // Создаем файл myfile.txt в каталоге TestDir
     File.CreateText(target + @"\myfile.txt");
     // Определяем количество файлов в каталоге TestDir
     textBox1.Text = "Количество файлов в каталоге " +  target + " - " +
                     Directory.GetFiles(target).Length;
    }
```


Пример 4. Нижеприведенный пример демонстрирует возможность отобразить имена всех устройств, представленных на текущем компьютере:

```
private void button1_Click(object sender, EventArgs e)
    {
       string[] drives = Directory.GetLogicalDrives();
       textBox1.Text = "На ПК имеются логические диски:";
       foreach (string s in drives)
          textBox1.Text +=" " + s + "; ";
    }
```

1.3. Класс FileInfo

Класс FileInfo позволяет получать подробную информацию относительно существующих файлов на жестком диске (т.е. время создания, размер и атрибуты) и предназначен для создания, копирования, перемещения и удаления файлов. Вдобавок к набору функциональности, унаследованной от FileSystemInfo, есть некоторые члены, уникальные для класса FileInfo:

- **AppendText()** - создает объект StreamWriter и добавляет текст в файл;
- **CopyTo()** - копирует существующий файл в новый файл;
- **Create()** - создает новый файл и возвращает объект FileStream для взаимодействия с вновь созданным файлом;
- **CreateText()** - создает объект StreamWriter, записывающий новый текстовый файл;
- **Delete()** - удаляет файл, к которому привязан экземпляр FileInfo;
- **Directory** - получает экземпляр родительского каталога;

- **DirectoryName** - получает полный путь к родительскому каталогу;
- **Length** - получает размер текущего файла или каталога;
- **MoveTo()** - перемещает указанный файл в новое местоположение, предоставляя возможность указать новое имя файла;
- **Name** - получает имя файла;
- **Open()** - открывает файл с различными привилегиями чтения/записи и совместного доступа;
- **OpenRead()** - создает доступный только для чтения объект FileStream;
- **OpenText()** - создает объект StreamReader (описанный ниже) и читает из существующего текстового файла;
- **OpenWrite()** - создает доступный только для записи объект FileStream.

Большинство методов класса FileInfo возвращают специфический объект ввода-вывода (т.е. FileStream и StreamWriter), который позволяет начать чтение и запись данных в ассоциированный файл в разнообразных форматах.

1.3.1. Способы получения дескриптора файла с использованием класса FileInfo.

Метод Create(). Один из способов создания дескриптора файла предусматривает использование метода **FileInfo.Create()**:

```
// Создать новый файл на диске C
FileInfo f = new FileInfo(@"D:\Test.dat");
FileStream fs = f.Create();
// Использовать объект FileStream
// Закрыть файловый поток.
fs.Close();
```

Метод FileInfo.Create() возвращает тип FileStream, который предоставляет синхронную и асинхронную операции записи/чтения лежащего в его основе файла. Необходимо помнить, что объект FileStream, возвращенный FileInfo.Create(), открывает полный доступ по чтению и записи всем пользователям. Кроме того после окончания работы с текущим объектом FileStream следует закрыть его дескриптор, чтобы освободить лежащие в основе потока неуправляемые ресурсы. Учитывая , что FileStream реализует интерфейс IDisposable, можно применить оператор using и позволить компилятору сгенерировать логику завершения:

```
// Определение контекста using для файлового ввода-вывода.
FileInfo f = new FileInfo(@"D:\Test.dat") ;
using (FileStream fs = f.Create())
{
// Использовать объект FileStream
}
```

Метод Open(). С помощью метода FileInfo.Open() можно открывать существующие файлы, а также создавать новые файлы с гораздо более высокой точностью, чем FileInfo.Create(), учитывая, что Open() обычно принимает несколько параметров для описания общей структуры файла, с которым будет производиться работа. В результате вызова Open() получается возвращенный им объект FileStream:

```
// Создать новый файл через FileInfo.Open()
FileInfo f2 = new FileInfo(@"D:\Test2.dat") ;
using (FileStream fs2 = f2.Open(FileMode.OpenOrCreate, FileAccess.ReadWnte,
                                FileShare .None))
    {
      // Использовать объект FileStream
    }
```

Метод Open() требует три параметра. Первый параметр указывает общий тип запроса ввода-вывода (т.е. создать новый файл, открыть существующий файл и дописать в файл), указываемый в виде перечисления FileMode:

- **CreateNew** - информирует операционную систему о создании нового файла. Если файл уже существует, генерируется исключение IOException;

- **Create** - информирует операционную систему о создании нового файла. Если файл уже существует, он будет перезаписан;

- **Open** - открывает существующий файл. Если файл не существует, генерируется исключение FileNotFoundException;

- **OpenOrCreate** - открывает файл, если он существует; в противном случае создает новый;

- **Truncate** - открывает файл и усекает его до нулевой длины;

- **Append** - открывает файл, переходит в его конец и начинает операции чтения(этот флаг может быть использован только с потоками, доступными лишь для чтения). Если файл не существует, то создается новый

Второй параметр метода Open() — значение перечисления FileAccess —используется для определения поведения чтения/записи лежащего в основе потока:

public enum FileAccess {Read, Write, ReadWrite}.

Третий параметр метода Open() — значение перечисления FileShare — указывает, как файл может быть разделен с другими файловыми дескрипторами:

public enum FileShare {Delete, Inheritable, None, Read, ReadWrite, Write},

где **Delete** - разрешает последующее удаление файла**; Inheritable -** разрешает наследование дескриптора файла дочерними процессами; **None -** отклоняет совместное использование текущего файла. Любой запрос на открытие файла (данным процессом или другим процессом) не выполняется до тех пор, пока файл не будет закрыт; **Read** - разрешает последующее открытие файла для чтения. Если этот флаг не задан, любой запрос на открытие файла для чтения (данным процессом или другим процессом) не выполняется до тех пор, пока файл не будет закрыт. Однако, даже если этот флаг задан, для доступа к данному файлу могут потребоваться дополнительные разрешения; **ReadWrite** - разрешает последующее открытие файла для чтения или записи. Если этот флаг не задан, любой запрос на открытие файла для записи или чтения (данным процессом или другим процессом) не выполняется до тех пор, пока файл не будет закрыт. Однако, даже если этот флаг задан, для доступа к данному файлу могут потребоваться дополнительные разрешения; **Write** - разрешает последующее открытие файла для записи. Если этот флаг не задан, любой запрос на открытие файла для записи (данным процессом или другим процессом) не выполняется до тех пор, пока файл не будет закрыт. Однако, даже если этот флаг задан, для доступа к данному файлу могут потребоваться дополнительные разрешения.

Методы OpenRead() и OpenWrite(). Хотя метод Open() позволяет получить дескриптор файла довольно гибким способом, в классе FileInfo также предусмотрены для этого члены OpenRead() и OpenWrite(). Эти методы возвращают объект FileStream, соответствующим образом сконфигурированный только для чтения или только для записи, без необходимости применять различные значения перечислений. Методы OpenRead() и OpenWrite() возвращают объект FileStream, например:

```
// Получить объект FileStream с правами только для чтения
FileInfo f3 = new FileInfo (@"D:\Test3.dat");
using (FileStream readOnlyStream = f3.OpenRead())
    {
      // Использовать объект FileStream
    }
// Получить объект FileStream с правами только для записи
FileInfo f4 = new FileInfo(@"D:\Test4.dat");
using (FileStream writeOnlyStream = f4.OpenWrite ())
    {
      // Использовать объект FileStream
    }
```

Метод OpenText(). В отличие от Create(), Open(), OpenRead() и OpenWrite(), метод OpenText() возвращает экземпляр типа StreamReader, а не FileStream:

```
// Получить объект StreamReader.
FileInfo f5 = new FileInfo(@"D:\boot.ini");
using (StreamReader sreader = f5.OpenText ())
    {
      // Использовать объект StreamReader
    }
```

Тип StreamReader предоставляет способ чтения символьных данных из лежащего в основе файла.

Методы FileInfo.CreateText() и FileInfo.AppendText(). Оба возвращают объект StreamWriter:

```
FileInfo f6 = new FileInfo(@"D:\Test6.txt");
using (StreamWriter swriter = f6.CreateText())
    {
      // Использовать объект StreamWriter
    }
FileInfo f7 = new FileInfo(@"D:\FinalTest.txt");
using (StreamWriter swriterAppend = f7.AppendText())
    {
      // Использовать объект StreamWriter
    }
```

1.4. Класс File

Класс File предоставляет статические методы для создания, копирования, удаления, перемещения и открытия файлов, а также помогает при создании объектов FileStream. Используется класс File для обычных операций, таких как копирование, перемещение, переименование, создание, открытие, удаление файла, а также добавление данных в файлы. Класс File можно также использовать для получения и задания атрибутов или сведений DateTime, связанных с созданием файла, доступом к нему и записью в файл.

Все методы File статические, поэтому, если необходимо выполнить только одно действие, более эффективным может оказаться использование метода File, а не соответствующего экземпляра метода FileInfo.

Для всех методов File требуется путь к файлу, с которым проводится операция. Этот путь должен указывать на файл или каталог. Заданный путь может также указывать на относительный путь или путь к серверу, например: d:\\MyDir\\MyFile.txt , d:\\MyDir, MyDir\\MySubdir, \\\\MyServer\\MyShare, причем требуют, чтобы путь был правильным, в противном случае будет вызвано исключение. Например, если полный путь начинается с пробела, он не будет обрезаться при использовании в методах данного класса. Таким образом, путь будет неверным и вызовет исключение.

Все статические методы класса File выполняют проверку безопасности для всех методов. Если необходимо использовать объект неоднократно, рекомендуется использовать соответствующий метод экземпляра FileInfo, поскольку в этом случае проверка безопасности будет требоваться не всегда.

По умолчанию всем пользователям предоставляется полный доступ к новым файлам с правом на чтение и запись.

В следующей таблице описаны перечисления, используемые для настройки поведения различных методов File используются перечисления:

- **FileAccess** - устанавливает доступ к файлу с правом чтения и записи;
- **FileShare** - устанавливает уровень разрешенного доступа для уже используемого файла;
- **FileMode** - позволяет установить, будет ли содержание существующего файла сохранено или перезаписано, а также будет ли вызываться исключение в случае запроса на создание существующего файла.

Основными методами класса File являются:

- **AppendAllText(String, String)** - открывает файл, добавляет в него указанную строку и затем закрывает файл. Если файл не существует, этот метод создает файл, записывает в него указанную строку и затем закрывает файл;

- **AppendText()** - создает StreamWriter, добавляющий в существующий файл текст в кодировке UTF-8. Кодирование - это процесс преобразования набора символов Юникода в последовательность байтов. Декодирование представляет собой процесс преобразования последовательности закодированных байтов в набор символов Юникода. Кодировка UTF-8, в которой каждая кодовая точка представляется в виде последовательности от одного до четырех байтов.

- **Copy(String, String)** - копирует существующий файл в новый файл. Перезапись файла с тем же именем не разрешена;

- **Copy(String, String, Boolean)** - копирует существующий файл в новый файл. Перезапись файла с тем же именем разрешена;

- **Create()** - создает или перезаписывает файл в указанном пути;

- **CreateText()** - создается или открывается файл для записи текста в кодировке UTF-8;

- **Decrypt()** - расшифровывает файл, зашифрованный текущей учетной записью с помощью метода Encrypt();

- **Delete()** - удаляет указанный файл;

- **Encrypt()** - шифрует файл таким образом, чтобы его можно было расшифровать только с помощью учетной записи, которая использовалась для шифрования;

- **Exists()** - определяет, существует ли заданный файл;

- **GetAccessControl()** - возвращает права доступа к файлу;

- **GetAttributes()** - возвращает атрибуты файла;

- **GetCreationTime()** - возвращает дату и время создания заданного файла или каталога;

- **GetLastAccessTime()** - возвращает время и дату последнего обращения к указанному файлу или каталогу;

- **GetLastWriteTime()** - возвращает время и дату последней операции записи в указанный файл или каталог;

- **Move()** - перемещает заданный файл в новое местоположение и разрешает переименование файла;

- **Open()** - открывает объект FileStream по указанному пути с доступом для чтения и записи;

- **OpenRead()** - открывает для чтения существующий файл;

- **OpenText()** - открывает для чтения существующий файл, содержащий текст в кодировке UTF-8;

- **OpenWrite()** - открывает существующий файл или создает новый файл для записи;

- **ReadAllBytes()** - открывает двоичный файл, считывает содержимое файла в массив байтов и затем закрывает файл;

- **ReadAllLines()** - открывает текстовый файл, считывает все строки файла и затем закрывает файл;

- **ReadAllText()** - открывает текстовый файл, считывает все строки файла и затем закрывает файл;

- **ReadLines()** - считывает строки файла;

- **Replace()** - заменяет содержимое заданного файла на содержимое другого файла, удаляя исходный файл и создавая резервную копию замененного файла;

- **SetAccessControl()** - назначает права доступа;

- **SetAttributes()** - устанавливает заданные атрибуты FileAttributes файла по заданному пути;

- **SetCreationTime()** - устанавливает дату и время создания файла;

- **SetLastAccessTime()** - устанавливаются дата и время последнего доступа к заданному файлу;

- **SetLastWriteTime()** - устанавливаются дата и время последней операции записи в заданный файл;

- **WriteAllBytes()** - создает новый файл, записывает в него указанный массив байтов и затем закрывает файл. Если целевой файл уже существует, он будет переопределен;

- **WriteAllLines(String, String[])** - создает новый файл, записывает в него указанный массив

строк и затем закрывает файл;

- **WriteAllText(String, String)** - Создает новый файл, записывает в него указанную строку и затем закрывает файл. Если целевой файл уже существует, он будет переопределен.

Фактически во многих случаях типы File и FileInfo могут использоваться взаимозаменяемым образом, например:

```
// Получить объект FileStream через File.Create()
using (FileStream fs = File.Create(@"D:\Test.dat"))
      {}
// Получить объект FileStream через File.Open()
using (FileStream fs2 = File.Open(@"D:\Test2.dat",
        FileMode.OpenOrCreate, FileAccess .ReadWrite, FileShare .None))
      {}
// Получить объект FileStream с правами только для чтения.
using (FileStream readOnlyStream = File.OpenRead(@"Test3.dat"))
      {}
// Получить объект FileStream с правами только для записи.
using (FileStream writeOnlyStream = File.OpenWrite(@"Test4.dat"))
      {}
// Получить объект StreamReader.
using (StreamPeacler sreader = File .OpenText (@"D:\boot.ini" ))
      {}
// Получить несколько объектов StreamWriter.
using (StreamWriter swriter = File.CreateText(@"D:\Test6.txt"))
     {}
using (StreamWriter swriterAppend = File.AppendText(@"D:\FinalTest.txt"))
    {}
```

Пример 5. Нижеприведенный пример демонстрирует использование методов класса **File**. Осуществляется проверка на наличие файла и в зависимости от результата, либо создаётся новый файл и записывается в него, или открывается существующий файл и происходит чтение из него.

```
private void button1_Click(object sender, EventArgs e)
      {
       string path = @"d:\temp\MyTest.txt";
       if (!File.Exists(path))
        {
         // Создание файла и запись в него
         using (StreamWriter sw = File.CreateText(path))
           {
            sw.WriteLine("Создали");
            sw.WriteLine(" файл");
            sw.WriteLine(" для чтения");
           }
        }
       // Открытие файла и чтение из него
       using (StreamReader sr = File.OpenText(path))
        {
         string s = "";
         while ((s = sr.ReadLine()) != null)
               {
                textBox1.Text += s;
               }
```

```
    }
}
```

Пример 6. Нижеприведенный пример демонстрирует использование метода **AppendText()** класса **File**, то есть производится добавление текста в файл.

```
private void button1_Click(object sender, EventArgs e)
    {
    string path = @"d:\temp\MyTest.txt";
    if (!File.Exists(path))
      {
       // Создание файла и запись в него
       using (StreamWriter sw = File.CreateText(path))
         {
          sw.WriteLine("Создали");
          sw.WriteLine(" файл");
          sw.WriteLine(" для чтения");
         }
      }
    using (StreamWriter sw = File.AppendText(path))
      {
       sw.WriteLine(" Добавили");
       sw.WriteLine(" к содержимому");
       sw.WriteLine(" новый текст.");
      }
    // Открытие файла и чтение из него
    using (StreamReader sr = File.OpenText(path))
      {
       string s = "";
       while ((s = sr.ReadLine()) != null)
            {
             textBox1.Text += s;
            }
      }
}
```

1.5. Класс Path

Выполняет операции для экземпляров класса String, содержащих сведения пути к файлу или каталогу.

Путь — это строка, предоставляющая размещение файла или каталога. Путь не обязательно указывает место на диске. Например, путь может отображаться в памяти или в устройстве. Точный формат пути определяется текущей платформой. Например, в некоторых системах путь может начинаться с буквы диска или тома, тогда как в других системах этот элемент может отсутствовать. В некоторых системах пути к файлам могут содержать расширения, показывающие тип данных, хранящихся в файле. Формат расширения имени файла зависит от платформы. Например, некоторые системы ограничивают расширения тремя символами. Текущая платформа также определяет набор символов, используемых для разделения элементов пути, и набор символов, которые при задании пути использовать нельзя. В связи с этими различиями поля класса Path, так же как и точное поведение некоторых элементов класса Path, зависят от платформы.

Путь может содержать абсолютные или относительные сведения о размещении. В абсолютном пути размещение указано полностью (файл или каталог может быть однозначно определен независимо от текущего размещения). В относительном пути указано неполное размещение (при поиске файла по

указанному относительному пути в качестве начальной точки используется текущее размещение). Чтобы определить текущий каталог, необходимо вызвать метод Directory.GetCurrentDirectory().

Большинство членов класса Path не взаимодействует с файловой системой и не проверяет существование файла, указанного строкой пути. Члены класса Path, которые изменяют строку пути, например ChangeExtension(), не оказывают влияния на имена файлов в файловой системе. Члены Path, однако, проверяют содержимое указанной строки пути и вызывают исключение ArgumentException если строка содержит символы, недопустимые в пути, как определено в строках символов, возвращаемых из метода GetInvalidPathChars().

Все элементы класса Path являются статическими и могут быть вызваны без наличия экземпляра пути.

Основными методами класс Path являются:

- **ChangeExtension()** - изменяет расширение строки пути;
- **Combine(String[])** - объединяет массив строк в путь;
- **Combine(String, String)** - объединяет две строки в путь;
- **Combine(String, String, String)** - объединяет три строки в путь;
- **Combine(String, String, String, String)** - объединяет четыре строки в путь;
- **GetDirectoryName()** - возвращает для указанной строки пути сведения о каталоге;
- **GetExtension()** - возвращает расширение указанной строки пути;
- **GetFileName()** - возвращает имя файла и расширение указанной строки пути;
- **GetFileNameWithoutExtension()** - возвращает имя файла указанной строки пути без расширения;
- **GetFullPath()** - возвращает для указанной строки пути абсолютный путь;
- **GetPathRoot()** - получает сведения о корневом каталоге для указанного пути;
- **GetRandomFileName()** - возвращает произвольное имя каталога или файла;
- **GetTempFileName()** - создает на диске временный пустой файл с уникальным именем и возвращает полный путь этого файла;
- **GetTempPath()** - возвращает путь к временной папке текущего пользователя;
- **IsPathRooted()** - получает значение, показывающее, содержит ли указанная строка пути корень.

2. Рабочее задание

Задание 1. Разработать приложение «Файловая структура диска», с помощью которого можно отобразить файловую структуру диска.

При первоначальном запуске отображается содержимое корневого каталога диска **D** (рис.1а), после выбора любого каталога из списка каталогов по нажатию кнопки «Отобразить» отображается содержимое выбранного каталога (рис.1б). При нажатии на кнопку «Назад» отображается содержимое предыдущего каталога.

При разработке интерфейса приложения использовать компоненты **Label, Button, ListBox** (один из вариантов интерфейса представлен на рис. 1).

Рис. 1а. Внешний вид приложения «Файловая структура диска» при первоначальном запуске приложения

Рис. 1б. Внешний вид приложения «Файловая структура диска» после выбора соответствующего каталога

Задание 2. Разработать приложение «Поиск каталогов и файлов», с помощью которого можно осуществить поиск папок и файлов, размещенных на дисках ПК.

При выполнении приложения пользователь должен ввести имя каталога или файла, а также указать место (каталог), где искать требуемые каталог или файлы.

При разработке интерфейса приложения использовать компоненты **Label, Button, TextBox, ListBox** (один из вариантов интерфейса представлен на рис. 2).

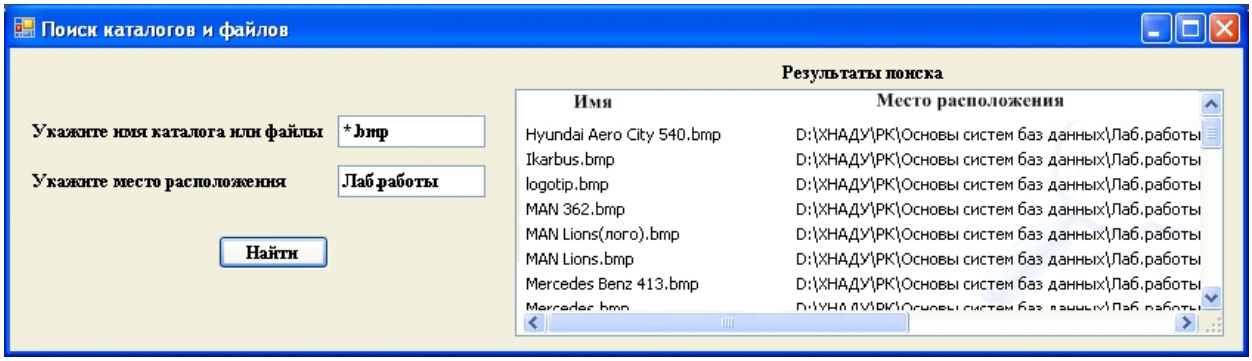

Рис. 2. Внешний вид приложения «Поиск каталогов и файлов»

3. Контрольные вопросы

Литература

1. Голощапов А.Л. Microsoft Visual Studio 2010. – СПб.:БХВ-Петербург, 2011. – 544 с.: ил.
2. Культин Н.Б. Microsoft Visual C# в задачах и примерах. – СПб.: БХВ-Петербург, 2009. – 320 с.: ил.
3. Лабор В.В. Си Шарп: Создание приложений для Windows. – Мн.: Харвест, 2003. – 384 с.
4. Петцольд Ч. Программирование для Microsoft Windows на C#. В 2-х томах. Том 1. Пер. с англ. -

М.: «Русская Редакция», 2002.- 576 с.: ил.
5. Петцольд Ч. Программирование для Microsoft Windows на C#. В 2-х томах. Том 2. Пер. с англ. - М.: «Русская Редакция», 2002.- 624 с.: ил.
6. Троелсен Э. Язык программирования C# 2010 и платформа .NET 4.0. Пер. с англ. - М.: Издательский дом "Вильямс", 2011. — 1392 с.: ил.
7. Фаронов В.В. Программирование на языке C#. – СПб.: Питер, 2007. – 240 с.: ил.
8. Фленов М.Е. Библия C#. - СПб.: БХВ-Петербург, 2011.– 560с.: ил.