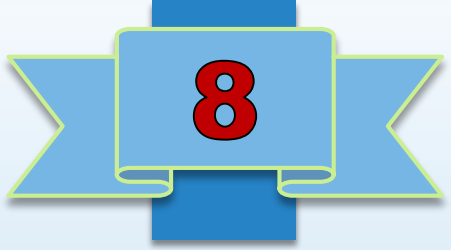
ЛЕКЦИЯ 8



СЪДЪРЖАНИЕ

I. Потоците в .NET Framework.

II. Буферирани потоци

III. Файлови потоци

IV. Операции с файлове

V. Работа с директории

VI. Особености и предназначение на Атрибутите в C#

VII. Делегати и събития

I. ПОТОЦИТЕ В .NET FRAMEWORK

I.1. Какво представляват потоците?

Потокът е подредена серия от байтове, която служи като абстрактен канал за данни. Този виртуален канал свързва програмата с устройство за съхранение или пренос на данни (напр. файл върху хард диск), като достъпът до канала е последователен. Потоците предоставят средства за четене и запис на поредици от байтове от и към устройството. Това е стандартният механизъм за извършване на входно-изходни операции в .NET Framework.

Потоците в .NET Framework се делят на две групи – **базови и преходни**. И двете групи потоци наследяват абстрактния **клас System.IO.Stream**, базов за всички потоци.

```
1   using System;
2   using System.Collections.Generic;
3   using System.Linq;
4   using System.Text;
5   using System.Threading.Tasks;
6   using System.IO;
```

Пример за използване на потоци:

програмата реализира копиране на файлове чрез потоци. Тя създава копие на обикновен текстов файл (test.txt)

Клас за стандартен поток за работа с файлове

Междинен масив от байтове

```
class Program
{
    public const string INPUT_FILE = @"D:\test.txt";
    public const string OUTPUT_FILE = @"D:\test1.txt";
    0 references
    static void Main(string[] args)
    {
        using (
        FileStream inFile = new FileStream(INPUT_FILE,
        FileMode.Open),
        outFile = new FileStream(OUTPUT_FILE, FileMode.Create))
                    {
                        byte[] buf = new byte[1024];
                        while (true)
                        {
                            int bytesRead = inFile.Read(buf, 0, buf.Length);
                            if (bytesRead == 0)
                                break;
                            outFile.Write(buf, 0, bytesRead);
                        }
                    }
    }
}
```

Обяснение на примерната програма

Със създаването на **inFile** и **outFile** от клас **FileStream**, създаваме два двоични потока, като ги свързваме с два файла (test.txt и test1.txt), намиращи се в съответните директории (D:\test.txt и D:\test1.txt). Другите параметри в конструктора (**FileMode.Open** и **FileMode.Create**) показват, че първият файл се отваря като вече съществуващ, докато вторият се създава при изпълнението на програмата.

Използваната **using** клауза гарантира затварянето на използваните в нея потоци след приключване на работа с тях. Следва цикъл, който чете байтове от файла test.txt (като използва метода **Read()** на inFile обекта), записва ги в междинния масив от байтове **buf**, след което записва съдържанието на buf във файла test1.txt. Read() връща действително прочетените байтове – те може да бъдат и по-малко от заявените. Действително прочетените байтове се запиват в изходния файл. Когато inFile.Read() върне 0, входният файл вече е прочетен и копирането приключва.

I.2. Основни групи потоци в .NET

Базови потоци (base streams)

Базовите потоци пишат и четат директно от някакъв външен механизъм за съхранение, като файловата система (например класът **FileStream**), паметта (**MemoryStream**) или данни, достъпни по мрежата (**NetworkStream**).

Преходни потоци (pass-through streams)

Преходните потоци пишат и четат от други потоци (най-често в базови потоци), като при това посредничество добавят допълнителна функционалност, например буфериране (**BufferedStream**) или кодиране (**CryptoStream**).

I.3. Основни операции с потоци

Когато работим с потоци, върху тях можем да извършваме следните основни операции:

- **Конструиране (създаване)** – свързваме потока с механизма за пренос (съхранение) на данните **(в случай на базов поток)** или с друг поток **(в случай на преходен поток)**. При това конструиране подаваме необходимата информация. Например, в случай на файлов поток подаваме име на файл и режим, в който го отваряме. В примера за копиране на файл вече показахме конструиране на файлов поток.
- **Четене** – извличане на данни от потока по специфичен за него начин. Извличането се извършва последователно, започвайки от текущата позиция.
- **Запис** – изпращат се данни в потока по специфичен за него начин. Записът става от текущата позиция.

- **Позициониране** – премества текущата позиция на потока (ако потокът поддържа позициониране). Можем да позиционираме спрямо текуща позиция, спрямо начало на потока или спрямо края на потока. Някои потоци не поддържат позициониране (напр. NetworkStream).
- **Затваряне** – приключваме работата с потока и освобождаваме ресурсите, свързани с него. Например, ако потокът е файлов, записваме на диска данните от вътрешните буфери, които не са още записани и затваряме файла.
- **Други операции** – изпразване на вътрешните буфери (**flush**), асинхронно четене/запис (при "Многонишково програмиране и синхронизация") и други.

I.3.1. Четене от поток

За четене на данни от поток се използва методът

int Read(byte[] buffer, int offset, int count).

Той чете най-много ***count*** на брой байта от текущата позиция на входния поток, увеличава позицията и връща броя прочетени байтове или 0 при достигне края на потока.

Четенето може да блокира за неопределено време. Например, ако при четене от мрежа извикаме метода ***NetworkStream.Read(…)***, а не са налични данни за четене, операцията блокира до тяхното получаване. В такива случаи е уместно да се използва свойството **NetworkStream.DataAvailable**, което показва дали в потока има пристигнали данни, които още не са прочетени, т. е. дали последваща операция Read() ще блокира или ще върне резултат веднага.

I.3.2. Писане в поток

Методът

***Write*(byte[] buffer, int offset, int count)**

записва в изходния поток *count* на брой байта, като започва от зададеното отместване в байтовия масив. И тази операция е блокираща, т.е. може да предизвика забавяне за неопределено време. Не е гарантирано, че байтовете, записани в потока с Write(…), са достигнали до местоназначението си след успешното изпълнение на метода. Възможно е потокът да буферира данните и да не ги изпраща веднага.

I.3.3. Изчистване на работните буфери

Методът ***Flush()*** изчиства вътрешните буфери, като изпраща съдържащите се в тях данни към механизма за съхранение. След успешното приключване на изпълнението на ***Flush()*** е гарантирано, че всички данни, записани в потока, са изпратени към местоназначението си, но няма гаранция, че ще пристигнат успешно до него.

I.3.4. Затваряне на поток

Методът ***Close()*** извиква ***Flush()***, затваря връзката към механизма за съхранение (пренос) на данни и освобождава използваните ресурси.

Алтернатива на ***Close()*** е **using** конструкцията. Когато използваме using, дефинираме програмния блок, в който е видим създавания обект. При достигане края на блока, гарантирано се извиква методът Dispose() на посочения в клаузата обект, а той вътрешно извиква Close().

Винаги затваряйте потоците, които използвате, за да не предизвиквате загуба на ресурси!

Правилно затваряне на поток

Често срещан подход за начинаещия програмист е да напише следния код за работа с поток:

```
StreamClass streamObj = ...; // отваряне на нов поток
// Извършвани действия с потока
streamObj.Close();
```

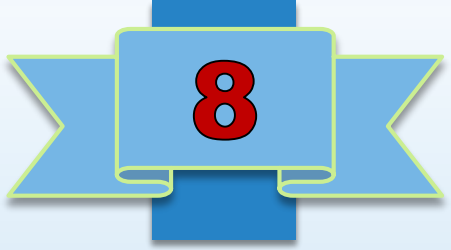
Проблемът на този код е, че ако по време на работата с отворения поток възникне изключение, операцията Close() няма да се изпълни и потокът ще остане отворен.

Това е сериозен проблем, защото води до потенциална загуба на ресурси, а ресурсите са ограничени и не трябва да се пропиляват. При изчерпване на ресурсите приложението започва да става нестабилно и да предизвиква неочаквани грешки и сривове.

Правилната работа с потоци изисква затварянето им да бъде гарантирано след приключване на работата с тях или чрез **using** конструкцията в C# или чрез употребата на **try-finally** блок.

Ето как можем да използваме конструкцията using за правилно освобож-даване на поток:

```
StreamClass streamObj = ...; // Отваряне на нов поток
using (streamObj)
{
    // Извършване на действия с потока
}
// Тук потокът вече гарантирано ще бъде затворен
```



Ето и алтернативният вариант в **try-finally** конструкция:

```
StreamClass streamObj = ...; // Отваряне на нов поток
try {
    // Извършване на действия с потока
}

finally {
// ръчно затваряне на потока след приключване на работата с него
streamObj.Close();
}
```

И в двата варианта е предвиден случаят, в който по време на работа възниква изключение. В този случай потокът ще бъде затворен преди да бъде обработено изключението.

I.3.5. Промяна на текущата позиция в поток

Методът

Seek(int offset, SeekOrigin origin)

премества текущата позиция на потока с ***offset*** на брой байта спрямо зададена отправна точка (начало, край или текуща позиция на потока). Методът е приложим за потоците, за които CanSeek връща true, за останалите хвърля изключение NotSupportedException.

Методът ***SetLength(long length)*** променя дължината на потока (ако това се поддържа). Промяната на дължината на поток е рядко използвана операция и се поддържа само от някои потоци, например MemoryStream.

II. БУФЕРИРАНИ ПОТОЦИ

Буферираните потоци използват вътрешен буфер за четене и запис на данни, с което значително подобряват производителността.

Когато четем данни от някакво устройство, при заявка дори само за един байт, в буфера попадат и следващите го байтове до неговото запълване.

При следващо четене, данните се взимат директно от буфера, което е много по-бързо.

По този начин се извършва кеширане на данните, след което се четат кеширани данни.

При запис, всички данни попадат първоначално в буфера. Когато буферът се препълни или когато програмистът извика Flush(), те се записват върху механизма за съхранение (пренос) на данни.

Класът, който реализира буфериран поток в .NET Framework, е System.IO.BufferedStream. Този клас или негов наследник трябва да бъде използван, когато трябва да се подобри производителността на входно-изходните операции в приложението.

III. ФАЙЛОВИ ПОТОЦИ

Файловите потоци в .NET Framework са реализирани в **класа FileStream**, който вече беше използван в примера за потоци. Като наследник на Stream, той поддържа всичките му методи и свойства (четене, писане, позициониране) и добавя някои допълнителни.

III.1. Създаване на файлове поток

В .NET Framework файлов поток се създава по следния начин:

```
FileStream fs = new FileStream(string fileName,
   FileMode [, FileAccess [, FileShare]]);
```

При конструирането, посочваме името на файла, с който свързваме потока (**fileName**), начина на отваряне на файла (**FileMode**), правата, с които го отваряме (**FileAccess**) и правата, които притежават другите потребители, докато ние държим файла отворен (**FileShare**).

III. ФАЙЛОВИ ПОТОЦИ

FileMode може да има една от следните стойности:

- **Open -** отваря съществуващ файл.
- **Append -** отваря съществуващ файл и придвижва позицията веднага след края му.
- **Create** – създава нов файл. Ако файлът вече съществува, той се презаписва и старото му съдържание с губи.
- **CreateNew** – аналогично на Create, но ако файлът съществува, се хвърля изключение.
- **OpenOrCreate** – отваря файла, ако съществува, в противен случай го създава.
- **Truncate** – отваря съществуващ файл и изчиства съдържанието му, като прави дължината му 0 байта.

FileAccess и **FileShare** могат да приемат стойности **Read**, **Write** и **ReadWrite**. FileShare може да бъде и None.

III.2. Четене и писане във файлов поток

Четенето и писането във файлови потоци, както и другите по-рядко използвани операции, се извършват както при всички наследници на класа Stream – с методите **Read**(), **Write**() и т. н.

Файловите потоци поддържат **пряк достъп до определена позиция** от файла чрез метода **Seek**(…).

III. ФАЙЛОВИ ПОТОЦИ


```
const int BUFFSIZE = 16384;
const byte SPACE_SYMBOL_CODE = 32;
0 references
static void Main(string[] args)
{
    FileStream fs = new FileStream("test.txt",
    FileMode.Open, FileAccess.ReadWrite, FileShare.None);
    using (fs)
    {
        byte[] buf = new byte[BUFFSIZE];
        while (true)
        {
            int bytesRead = fs.Read(buf, 0, buf.Length);
            if (bytesRead == 0)
                break;
            for (int i = 0; i < bytesRead; i++)
            {
                if (buf[i] == SPACE_SYMBOL_CODE)
                    buf[i] = (byte)'-';
            }
            fs.Seek(-bytesRead, SeekOrigin.Current);
            fs.Write(buf, 0, bytesRead);
        }
    }
}
```

Как работи примерът?

В примера “test.txt” е текстов файл от директорията bin\Debug на проекта. Преди изпълнението на програмата, в него можем да сложим произволно текстово съдържание. Примерът търси всички срещания на стойността 32 (ASCII кода на символа интервал) и я заменя с 45 (ASCII кода на символа тире). След като масивът от байтове buf се запълни с байтове от файла, правим замяната, след което връщаме позицията на потока и записваме коригираната информация върху старата. Получаваме файл, в който интервалите са заменени с тирета. Тъй като в случая “test.txt” съдържа текстова информация, той може да бъде отворен и разгледан с Notepad, както преди замяната, така и след нея.

III.3. Четци и писачи

Четците и писачите (**readers** и **writers**) в .NET Framework са класове, които улесняват работата с потоците. При работа например само с файлов поток, програмистът може да чете и записва единствено байтове. Когато този поток се обвие в четец или писач, вече са позволени четенето и записа на различни структури от данни, например примитивни типове, текстова информация и други типове. Четците и писачите биват двоични и текстови.

Двоични четци и писачи

Двоичните четци и писачи осигуряват четене и запис на примитивни типове данни в двоичен вид – **ReadChar(), ReadChars(), ReadInt32(), ReadDouble**() и др. за четене и съответно **Write(char), Write(char[]), Write(Int32), Write(double)** – за запис. Може да се чете и записва и string, като той се представя във вид на масив от символи и префиксно се записва дължината му – ReadString(), респективно Write(string).

III. ФАЙЛОВИ ПОТОЦИ

Текстови четци и писачи

Текстовите четци и писачи осигуряват четене и запис на текстова информация, представена във вид на низове, разделени с нов ред. Базови текстови четци и писачи са абстрактните класове **TextReader** и **TextWriter**. Основните методи за четене и запис са следните:

- **ReadLine**() – прочита един ред текст.
- **ReadToEnd**() – прочита всичко от текущата позиция до края на потока.
- **Write**(…) – вмъква данни в потока на текущата позиция.
- **WriteLine**(…) – вмъква данни в потока на текущата позиция и добавя символ за нов ред.

Текстовите четци прочитат само текст, а писачите записват различни типове данни в текстов формат.

Всяка от изброените операции е блокираща операция. Това означава, че извикването на някои от описаните методи може да се забави известно време, например докато пристигнат или бъдат изпратени данните към техния източник.

III. ФАЙЛОВИ ПОТОЦИ

Тъй като класовете TextReader и TextWriter **са абстрактни**, за конкретни входно-изходни операции с текстови данни се използват техни наследници, например класовете:

- **StreamReader** – чете текстови данни от поток или файл.
- **StreamWriter** – записва текстови данни в поток или файл.
- **StringReader** – чете текстови данни от символен низ.
- **StringWriter** – записва текстови данни в символен низ.

III. ФАЙЛОВИ ПОТОЦИ

Номериране на редове в текстов файл – пример:

```
static void Main(string[] args)
{
    StreamReader reader = new StreamReader("in.txt");
    using (reader)
    {
        StreamWriter writer = new StreamWriter("out.txt");
        using (writer)
        {
            int lineNumber = 0;
            string line = reader.ReadLine();
            while (line != null)
            {
                lineNumber++;
            }
            writer.WriteLine("{0,5} {1}", lineNumber, line);
            line = reader.ReadLine();
        }
    }
}
```

След като свържем файловете in.txt и out.txt съответно с текстов четец и текстов писач, започва цикъл по редовете на входния файл, който копира всеки ред от in.txt в стринга line (line = reader.ReadLine();), след което записва line във файла out.txt (writer.WriteLine(…)). Записът на line се предшества от номера на реда, изведен в поле с ширина 5 символа. При достигане края на файла, ReadLine() връща null.

IV. ОПЕРАЦИИ С ФАЙЛОВЕ. КЛАСОВЕ FILE И FILEINFO

Класовете **File** и **FileInfo** са помощни класове за работа с файлове. Те **дават възможност за стандартни операции върху файлове като създаване, изтриване, копиране** и др. В тях са дефинирани следните методи:

- Create(), CreateText() – създаване на файл.
- Open(), OpenRead(), OpenWrite(), AppendText() – отваряне на файл.
- CopyTo(…) – копиране на файл.
- MoveTo(…) – местене (преименуване) на файл.
- Delete() – изтриване на файл.
- Exists(…) – проверка за съществуване.
- LastAccessTime и LastWriteTime – момент на последен достъп и последен запис във файла.

В класа File, изброените методи са статични, а в класа FileInfo – достъпни чрез инстанция. Ако извършваме дадено действие еднократно (например създаваме един файл, след това го отваряме), класът File е за предпочитане. Работата с FileInfo и създаването на обект биха имали смисъл при многократното му използване. В примера за търсене на низ в текстов файл класът File вече бе използван.

File и FileInfo – пример

```
static void Main(string[] args)
{
    StreamWriter writer = File.CreateText("test1.txt");
    using (writer)
    {
        writer.WriteLine("Налей ми бира!");
    }
    FileInfo fileInfo = new FileInfo("test1.txt");
    fileInfo.CopyTo("test2.txt", true);
    fileInfo.CopyTo("test3.txt", true);
    if (File.Exists("test4.txt"))
    {
        File.Delete("test4.txt");
    }
    File.Move("test3.txt", "test4.txt");
}
```

В примера, извикването на CreateText(…) създава файла test1.txt и отваря текстов писач върху него. С този писач можем да запишем някакъв произволен текст във файла. След създаването на FileInfo обект, копираме създадения файл в два други. Параметърът true означава, че при вече съществуващ файл test2.txt, респективно test3.txt, новият файл ще бъде записан върху стария. След това се проверява дали съществува файл test4.txt и се изтрива, след което test3.txt се преименува като test4.txt.

V. РАБОТА С ДИРЕКТОРИИ.

КЛАСОВЕ DIRECTORY И DIRECTORYINFO

Класовете **Directory** и **DirectoryInfo** са помощни класове за работа с директории. Ще изброим основните им методи, като отбележим, че за Directory те са статични, а за DirectoryInfo – достъпни чрез инстанция.

- Create(), CreateSubdirectory() – създава директория или поддиректория.
- GetFiles(…) – връща всички файлове в директорията.
- GetDirectories(…) – връща всички поддиректории на директорията.
- MoveTo(…) – премества (преименува) директория.
- Delete() – изтрива директория.
- Exists() – проверява директория дали съществува.
- Parent – връща горната директория.
- FullName – пълно име на директорията.

V. РАБОТА С ДИРЕКТОРИИ.

КЛАСОВЕ DIRECTORY И DIRECTORYINFO

V.1. Създаване на директория в C#

```
string root = @"C:\Temp";
string subdir = @"C:\Temp\Mahesh";
// If directory does not exist, create it.
if (!Directory.Exists(root))
{
    Directory.CreateDirectory(root);
}
```

V.2. Създаване на директория в C#

```
// Create sub directory
if (!Directory.Exists(subdir))
{
    Directory.CreateDirectory(subdir);
}
```

V. РАБОТА С ДИРЕКТОРИИ.

КЛАСОВЕ DIRECTORY И DIRECTORYINFO

V.3. Изтриване на директория в C# - Методът Directory.Delete изтрива празна директория от посочения път за постоянно. Ако дадена папка има подпапки и файлове, трябва да ги изтриете, преди да можете да изтриете папка. Ще получите съобщение за грешка, ако се опитате да изтриете празен файл.

```
string root = @"C:\Temp";
// If directory does not exist, don't even try
if (Directory.Exists(root))
{
    Directory.Delete(root);
}
```

V.4. Преместване на папка в C#

```
string sourceDirName = @"C:\Temp";
string destDirName = @"C:\NewTemp";
try
{
    Directory.Move(sourceDirName, destDirName);
}
catch (IOException exp)
{
    Console.WriteLine(exp.Message);
}
```

V. РАБОТА С ДИРЕКТОРИИ.

КЛАСОВЕ DIRECTORY И DIRECTORYINFO

V.5. Получаване и/или задаване дата и час за създаване на файл в директория - Методите SetCreationTime и GetCreationTime се използват за задаване и получаване на датата и часа на създаване на посочения файл. Следният кодов фрагмент задава и получава времето за създаване на файл.

```
// Get and set file creation time
string fileName = @"c:\temp\Mahesh.txt";
File.SetCreationTime(fileName, DateTime.Now);
DateTime dt = File.GetCreationTime(fileName);
Console.WriteLine("File created time: {0}",dt.ToString());
```

V.6. Получаване и/или задаване времето на последен достъп до файл

```
// Get and set file last access time
string fileName = @"c:\temp\Mahesh.txt";
File.SetLastAccessTime(fileName,DateTime.Now);
DateTime dt = File.GetLastAccessTime(fileName);
Console.WriteLine("File last access time: {0}", dt.ToString());
```

V. РАБОТА С ДИРЕКТОРИИ.

КЛАСОВЕ DIRECTORY И DIRECTORYINFO

V.7. Получаване и/или задаване времето на последен запис във файл

```
// Get and set file last write time
string fileName = @"c:\temp\Mahesh.txt";
File.SetLastWriteTime(fileName,DateTime.Now);
DateTime dt = File.GetLastWriteTime(fileName);
Console.WriteLine("File last write time: {0}", dt.ToString());
```

V.8. Получаване и/или задаване дата и час на създаване на файл

```
string root = @"C:\Temp";
// Get and Set Creation time
Directory.SetCreationTime(root, DateTime.Now);
DateTime creationTime = Directory.GetCreationTime(root);
Console.WriteLine(creationTime);
```

V. РАБОТА С ДИРЕКТОРИИ.

КЛАСОВЕ DIRECTORY И DIRECTORYINFO

V.9. Изброяване на директории - Методът Directory.EnumerateDirectories връща изброима колекция от имена на директории в указаната директория.

```
string root = @"C:\Temp";
// Get a list of all subdirectories
var dirs = from dir in
    Directory.EnumerateDirectories(root)
              select dir;
Console.WriteLine("Subdirectories: {0}", dirs.Count<string>().ToString());
Console.WriteLine("List of Subdirectories");
foreach (var dir in dirs)
{
    Console.WriteLine("{0}", dir.Substring(dir.LastIndexOf("\\") + 1));
}

// Get a list of all subdirectories starting with 'Ma'
var MaDirs = from dir in
    Directory.EnumerateDirectories(root, "Ma*")
              select dir;
Console.WriteLine("Subdirectories: {0}", MaDirs.Count<string>().ToString());
Console.WriteLine("List of Subdirectories");
foreach (var dir in MaDirs)
{
    Console.WriteLine("{0}", dir.Substring(dir.LastIndexOf("\\") + 1));
}
```

V. РАБОТА С ДИРЕКТОРИИ.

КЛАСОВЕ DIRECTORY И DIRECTORYINFO

V.10. Изброяване на файлове

Методът EnumerateFiles връща изброима колекция от имена на файлове в указаната директория.

```
string root = @"C:\Temp";
// Get a list of all subdirectories
var files = from file in
Directory.EnumerateFiles(root)
            select file;
Console.WriteLine("Files: {0}", files.Count<string>().ToString());
Console.WriteLine("List of Files");
foreach (var file in files)
{
    Console.WriteLine("{0}", file);
}
```

V. РАБОТА С ДИРЕКТОРИИ.

КЛАСОВЕ DIRECTORY И DIRECTORYINFO

V.11. Взимане името на текущата директория

```
string root = @"C:\Temp";
Directory.SetCurrentDirectory(root);
Console.WriteLine(Directory.GetCurrentDirectory());
```

V.12. Взимане името на основната (root) директория

```
string root = @"C:\Temp";
Console.WriteLine(Directory.GetDirectoryRoot(root));
```

V.13. Взимане на всички файлове намиращи се в директория

```
string root = @"C:\Temp";
string[] fileEntries = Directory.GetFiles(root);
foreach (string fileName in fileEntries);
Console.WriteLine(fileName);
```

V. РАБОТА С ДИРЕКТОРИИ.

КЛАСОВЕ DIRECTORY И DIRECTORYINFO

V.14. Откриване на всички поддиректории

```
public void GetSubDirectories()
{
    string root = @"C:\Temp";
    // Get all subdirectories
    string[] subdirectoryEntries = Directory.GetDirectories(root);
    // Loop through them to see if they have any other subdirectories
    foreach (string subdirectory in subdirectoryEntries)
        LoadSubDirs(subdirectory);
}
private void LoadSubDirs(string dir)
{
    Console.WriteLine(dir);
    string[] subdirectoryEntries = Directory.GetDirectories(dir);
    foreach (string subdirectory in subdirectoryEntries)
    {
        LoadSubDirs(subdirectory);
    }
}
```

VI. ОСОБЕНОСТИ И ПРЕДНАЗНАЧЕНИЕ НА АТРИБУТИТЕ В C#

1. Какво представляват атрибутите в .NET?

Атрибутите представляват описателни декларации към типове, полета, методи, свойства и други елементи на кода, подобни на ключовите думи от езиците за програмиране.

Атрибутите позволяват да се добавят собствени описателни елементи (анотации) към кода, написан на C# или на някой от другите езици от .NET платформата, без да се налага промяна в компилатора. По време на компилация те се записват в метаданните на асемблито и при изпълнение на кода могат да бъдат извличани и да влияят на поведението му.

Атрибутите са описателни тагове, които могат да се прилагат към раз-лични елементи от кода, наричани цели. Целите могат да бъдат най-разнообразни: асемблита, типове, свойства, полета, методи, параметри и други елементи от кода.

Атрибутите се делят на две групи – вградени в .NET Framework (които са част от CLR) и дефинирани от програмистите за целите на отделните приложения. Последните се наричат собствени (потребителски) атрибути.

2. Прилагане на атрибути

По-долу са разгледаме начините, по които можем да приложим атрибут към дадена цел. За да се приложи атрибут, името му се огражда в квадратни скоби и се поставя непосредствено преди декларацията, за която се отнася.

Пример:

```
[Flags]
public enum FileAccess
{
   Read = 1,
   Write = 2,
   ReadWrite = Read | Write
}
```

Прилагане на атрибута System.FlagsAttribute енумерацията FileAccess

В посочения пример системният атрибут **Flags** (реално това е типът System.FlagsAttribute) е приложен към дефиницията на изброения тип FileAccess и указва, че този изброен тип може да се третира като битово поле, т.е. като множество от битови флагове.

VI. ОСОБЕНОСТИ И ПРЕДНАЗНАЧЕНИЕ НА АТРИБУТИТЕ В C#

За да бъде приложен атрибут към дадена дефиниция в кода, трябва да се изпълнят следните стъпки:

1. Да се дефинира нов атрибут или да се използва съществуващ, като неговото пространство от имена (namespace) се импортира в началото на текущия файл от сорс кода;
2. Да се изпише името на атрибута в квадратни скоби точно преди целта, към която се прилага. По желание могат да му бъдат предадени някакви параметри (инициализиращи данни).

Атрибутите са обекти - Атрибутите в .NET Framework реално представляват .NET обекти (инстан-ции на клас, наследник на системния клас System.Attribute). Като такива те могат да имат един или няколко конструктора (вкл. конструктор по подразбиране), публични и частни полета, свойства и др. членове. Най-често атрибутите дефинират конструктори, публични полета и свойства, които използват за съхраняване на данните, подавани им като параметри по време на инициализация.

3. Параметри на атрибутите

Някои атрибути могат да приемат параметри. Параметрите биват два вида: позиционни и именувани. Позиционните параметри се подават с оп-ределена последователност и се инициализират от конструктора на атри-бута, докато именуваните се подават в произволен ред и задават стойност на свойство или публично поле. Ето един пример:

```
[DllImport("user32.dll", EntryPoint="MessageBox")]
public static extern int ShowMessageBox(int hWnd, string text,
  string caption, int type);
...
ShowMessageBox(0, "Some text", "Some caption", 0);
```

В примера е използвана комбинация между позиционни и непозиционни параметри.

4. Задаване на цел към атрибут

Атрибутите в .NET Framework могат да се прилагат към различни цели, например асемблы, клас, интерфейс, член-променлива на тип и др. Възможните цели на атрибутите се дефинират от изброения тип AttributeTargets както следва:

Име на целта	Употреба (прилага се към)
Assembly	самото асемблы
Module	текущия модул
Class	клас
Struct	структура
Interface	интерфейс
Enum	изброен тип
Delegate	делегат
Constructor	конструктор
Method	метод
Parameter	параметър на метод

ReturnValue	връщаната стойност от метод
Property	свойство
Field	поле (член-променлива)
Event	събитие
All	всички възможни цели

VI. ОСОБЕНОСТИ И ПРЕДНАЗНАЧЕНИЕ НА АТРИБУТИТЕ В C#

При прилагане на атрибут целта обикновено се подразбира. Например, ако поставим атрибут преди декларацията на даден метод, той ще се отнася за съответния метод.

Понякога не може да се използва целта по подразбиране, например ако искаме да приложим атрибут към асемблито. В такива случаи целта може да се зададе преди името на атрибута, отделена от него с двоеточие:

```
// The following attributes are applied to the target "assembly"
[assembly: AssemblyTitle("Attributes Demo")]
[assembly: AssemblyCompany("DemoSoft")]
[assembly: AssemblyProduct("Entreprise Demo Suite")]
[assembly: AssemblyCopyright("(c) 1963-1964 DemoSoft")]
[assembly: AssemblyVersion("2.0.1.37")]

[Serializable] // The compiler assumes [type: Serializable]
class TestClass
{
   [NonSerialized] // The compiler assumes [field: NonSerialized]
   private int mStatus;
   ...
}
```

5. Дефиниране на собствени атрибути

За да бъде създаден потребителски атрибут, той задължително трябва да наследява класа System.Attribute и на компилатора трябва да се укаже към какъв вид елементи от кода може да се прилага атрибутът, т.е. какви са неговите цели. Това става с помощта на мета-атрибута AttributeUsage.

Да разгледаме следния пример: в даден проект има изискване всеки клас да съдържа в себе си информация за своя автор. Една възможност да се реализира това е във всеки един от класовете да се сложи коментар, подобен на този:

```
// This class is written by Person X.
```

За четящия кода ще бъде ясно кой е авторът, но няма да е възможно тази информация да се извлича по време на изпълнение на програмата, след като сорс кодът е бил вече компилиран. За да решим този проблем, можем вместо горния коментар да ползваме специален атрибут:

```
[Author("Person X")]
```

Пример за дефиниране на собствен атрибут

Нека сега дефинираме нашия атрибут за автор. Както вече отбелязахме, всички атрибути са инстанции на класове, а всеки клас, дефиниращ собствен атрибут, наследява класа System.Attribute. В нашия случай, когато създаваме атрибут, съдържащ името на автора на класа, можем да използваме следната дефиниция:

```
public class AuthorAttribute: System.Attribute { … }
```

Следващото нещо, което е необходимо, за да стане нашият клас AuthorAttribute потребителски атрибут, е да му приложим атрибута AttributeUsage. Чрез него указваме кои са целите, към които може да се прилага, и дали се допуска многократно прилагане към една и съща цел.

За да е възможно подаването на параметър при създаването на атрибута, за него трябва да се дефинира и подходящ конструктор.

Следва примерна реализация на атрибута AuthorAttribute:

VI. ОСОБЕНОСТИ И ПРЕДНАЗНАЧЕНИЕ НА АТРИБУТИТЕ В C#


```
using System;

[AttributeUsage(AttributeTargets.Struct |
    AttributeTargets.Class | AttributeTargets.Interface)]

public class AuthorAttribute: System.Attribute
{
   private string mName;

   public AuthorAttribute(string aName)
   {
      mName = aName;
   }

   public string Name
   {
      get
      {
         return mName;
      }
   }
}
```

VI. ОСОБЕНОСТИ И ПРЕДНАЗНАЧЕНИЕ НА АТРИБУТИТЕ В C#

Както се вижда от декларацията, нашият атрибут може да се прилага само към структури, класове и интерфейси, като към дадена цел не може да се прилага повече от веднъж (това се подразбира ако не е указано друго).

Понеже нашият атрибут е клас, който има само един конструктор, един-ственият начин да го инстанцираме е, като извикаме този конструктор. Следователно при използване на нашия атрибут винаги трябва да подаваме позиционния параметър за име на автор.

И така, веднъж деклариран, нашият атрибут вече може да бъде прилаган като всички останали атрибути:

```
[Author(„Иван Иванов")]
class CustomAttributesDemo
{
…
}
```

VII. ДЕЛЕГАТИ И СЪБИТИЯ

1. Какво представляват делегатите?

Делегатите са референтни типове, които описват сигнатурата на даден метод (броя, типа и последователността на параметрите му) и връщания от него тип. Могат да се разглеждат като "обвивки" на методи - те представляват структури от данни, които приемат като стойност методи, отговарящи на описаната от делегата сигнатура и връщан тип.

Пример:

```
// Declaration of a delegate
public delegate void SimpleDelegate(string aParam);

class TestDelegate
{
  public static void TestFunction(string aParam)
  {
    Console.WriteLine("I was called by a delegate.");
    Console.WriteLine("I got parameter {0}.", aParam);
  }

  public static void Main()
  {
    // Instantiation of a delegate
    SimpleDelegate simpleDelegate =
      new SimpleDelegate(TestFunction);
    // Invocation of the method, pointed by a delegate
    simpleDelegate("test");
  }
}
```

```
I was called by a delegate.
I got parameter test.
Press any key to continue_
```

Описание на примера:

В първия ред от кода се декларира делегат. За целта се използва ключо-вата дума delegate. След това в класа се дефинира функция, която има сигнатура и връщан тип като тези, декларирани от делегата. В главния метод на класа се инстанцира делегата, като дефинираният метод се подава като параметър и след това той се извиква чрез делегата.

2. Видове делегати

Делегатите в .NET Framework са специални класове, които наследяват System.Delegate или System.MulticastDelegate. От тези класове обаче явно могат да наследяват само CLR и компилаторът. Всъщност, те не са от тип делегат – тези класове се използват, за да се наследяват от тях типове делегат. Всеки делегат има "списък на извикване" (invocation list), който представлява наредено множество делегати, като всеки елемент от него съдържа конкретен метод, реферирaн от делегата. Делегатите могат да бъдат единични и множествени.

- **Единични (singlecast) делегати** - Единичните делегати наследяват класа System.Delegate. Тези делегати извикват точно един метод. В списъка си на извикване имат единствен елемент, съдържащ референция към метод.
- **Множествени (multicast) делегати** - Множествените делегати наследяват класа System.MulticastDelegate, който от своя страна е наследник на класа на System.Delegate. Те могат да викат един или повече метода. Техните списъци на извикване съдър-жат множество елементи, всеки реферирaщ метод. В тях може един и същ метод да се среща повече от веднъж.

Извикване на multicast делегати

При извикване на multicast делегат се изпълняват всички методи от него-вия списък на извикване. Методите се викат в реда, в който се намират в списъка, като дублиращите се методи (ако има такива) се викат толкова пъти, колкото се срещат в списъка.

Делегати и връщани стойности

Ако сигнатурата на методите, викани от делегата включва връщана стойност, връща се стойността, получена при изпълнението на последния елемент от списъка на делегата. Когато сигнатурата включва out или ref параметър, то всички извикани методи променят неговата стойност после-дователно в реда си на извикване и крайният резултат е резултата от последния извикан метод.

Делегати и изключения

Възможно е при извикване на multicast делегат някой от методите от списъка му на извикване да хвърли изключение. В този случай методът спира изпълнение и управлението се връща в кода, извикал делегата. Останалите методи от списъка не се извикват. Дори извикващият метода да хване изключението, останалите методи от списъка не се изпълняват.

System.MulticastDelegate

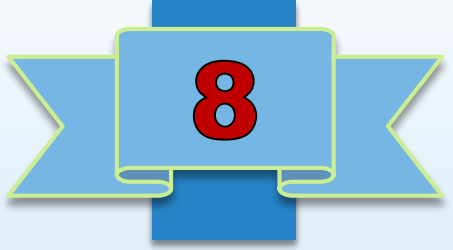
Класът System.MulticastDelegat е наследник на System.Delegate. Той е базов клас за всички делегати в C#, но самият той не е тип делегат – при срещане на ключовата дума delegate компилаторът създава клас, наследник на System.MulticastDelegat. Всички делегати наследяват от него няколко важни метода, които сега ще разгледаме.

Комбиниране на делегати

Multicast делегатите могат да участват в комбиниращи операции. Това се реализира с метода Combine() на класа. Той слива списъците от методи на няколко делегата от еднакъв тип. Този метод е предефиниран и може да приема като параметри както два multicast делегата от еднакъв тип, така и масив от multicast делегати от еднакъв тип. В резултат връща нов multicast делегат, чийто списък от методи съдържа списъците на подадените като параметри делегати.

Метод Remove()

Освен, че списъците от методи на няколко делегата могат да бъдат обединявани в един, възможно е също от списъка на един делегат да се извади списъкът на друг. Това се извършва чрез метода Remove(). Той приема като параметри два делегата и в резултат връща нов делегат, чийто списък е получен като от списъка на първия аргумент е премахнато последното срещане на списъка на втория аргумент. Ако двата списъка са еднакви, или ако списъкът на втория аргумент не се среща в списъка на първия, резултатът е null. В езика C# е предефиниран операторът -= за изваждане на списъци на делегати.



3. Събития (Events)

Събитията могат да се разглеждат като съобщения за настъпване на някакво действие. В компонентно-ориентираното програмиране компонен-тите изпращат събития (events) към своя притежател за да го уведомят за настъпването на интересна за него ситуация. Този модел е много характе-рен например за графичните потребителски интерфейси, където контро-лите уведомяват чрез събития други класове от програмата за действия от страна на потребителя. Например, когато потребителят натисне бутон, бутонът предизвиква събитие, с което известява, че е бил натиснат. Раз-бира се, събития могат да се предизвикват не само при реализиране на потребителски интерфейси. Нека вземем за пример програма, в която като част от функционалността влиза трансфер на файлове. Приключването на трансфера на файл може да се съобщава чрез събитие.

Шаблонът "Наблюдател"

Механизмът на събитията реализира шаблона "Наблюдател" (Observer) или, както още се нарича, Публикуващ/Абонати (Publisher/Subscriber), при който един клас публикува събитие, а произволен брой други класове могат да се абонират за това събитие. По този начин се реализира връзка на зависимост от тип един към много, при която когато един обект проме-ни състоянието си, зависещите от него обекти биват информирани за промяната и автоматично се обновяват.

Изпращачи и получатели

Обектът, който предизвиква дадено събитие се нарича изпращач на събитието (event sender). Обектът, който получава дадено събитие се нарича получател на събитието (event receiver). За да могат да получават дадено събитие, получателите му трябва преди това да се абонират за него (subscribe for event).

За едно събитие могат да се абонират произволен брой получатели. Изпращачът на събитието не знае кои ще са получателите на събитието, което той предизвиква. Затова чрез механизма на събитията се постига по-ниска степен на свързаност (coupling) между отделните компоненти на програмата.

VII. ДЕЛЕГАТИ И СЪБИТИЯ

3.1. Събитията в .NET Framework

Събитията могат да се разглеждат като съобщения за настъпване на някакво действие. В компонентно-ориентираното програмиране компонен-тите изпращат събития (events) към своя притежател за да го уведомят за настъпването на интересна за него ситуация. Този модел е много характе-рен например за графичните потребителски интерфейси, където контро-лите уведомяват чрез събития други класове от програмата за действия от страна на потребителя. Например, когато потребителят натисне бутон, бутонът предизвиква събитие, с което известява, че е бил натиснат. Раз-бира се, събития могат да се предизвикват не само при реализиране на потребителски интерфейси. Нека вземем за пример програма, в която като част от функционалността влиза трансфер на файлове. Приключването на трансфера на файл може да се съобщава чрез събитие.

Деклариране на събития - В C# събитията представляват специални инстанции на делегати. Те се декларират с ключовата дума event, която може да се предшества от модификатори, като например модификатори за достъп. Обикновено съби-тията са с модификатор public. След ключовата дума event се записва името на делегата, с който се свързва съответното събитие. За тази цел делегатът трябва да бъде дефиниран предварително. Той може да бъде дефиниран от потребителя, но може да се използва и вграден делегат. Тези делегати трябва да бъдат от тип void.

3.2. Разлика между събитие и делегат

Събитията и делегатите са много тясно свързани. Въпреки това член-променлива от тип делегат не е еквивалентна на събитие, декларирано с ключовата дума event, т.е. public MyDelegate m не е същото като public event MyDelegate m. Първото е декларация на променлива m, която е от тип MyDelegate, докато второто декларира събитие, което ще се обра-ботва от делегат от тип MyDelegate.

Между делегатите и събитията има и други разлики, освен в деклари-рането. Например, делегатите не могат да бъдат членове на интерфейси, докато събитията могат. В такъв случай класът, който имплементира интерфейса, трябва да предостави подходящо събитие.

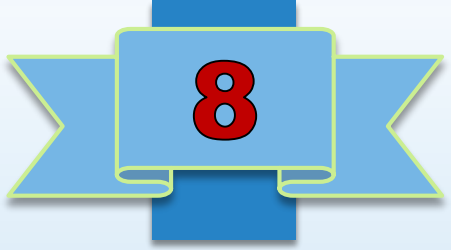
Извикване на събитие - Извикването на събитие може да стане само в класа, в който то е дефинирано. Това означава, че само класът, в който се дефинира събитие, може да предизвика това събитие. Това е наложително, за да се спази шаблонът на Публикуващ/Абонати – абонираните класове се информират при промяна на състоянието на публикуващия и именно публикуващият е отговорен за разпращане на съобщенията за промяната, настъпила у него.

Предизвикване на събитие

За предизвикване на събитие се създава protected void метод. Прието е името му да започва с On, следвано от името на събитието (например OnEventName). Този метод предизвиква събитието като извиква делегата.

Методът трябва да е protected, защото това позволява при наследяване на класа, в който е декларирано събитието, наследниците да могат да предизвикват събитието. Ако методът не е protected, наследниците няма да могат да предизвикат събитието, защото не могат да се обърнат директно към него, тъй като то е достъпно единствено в класа, в който е декларирано. За още по-голяма гъвкавост е възможно освен protected, методът да бъде обявен virtual, което би позволило на наследниците да го предефинират. Така те биха могли да прихващат извикването на събитията от базовия клас и евентуално да извършват собствена обработка. Следващият пример показва как се декларира метод за предизвикване на събитие:

```
protected void OnItemChanged() { … }
```



Конвенция за обработчиците - Обикновено името на метода-получател (обработчикът) на събитието има вида Обект_Събитие, както се илюстрира в следния пример:

```
private void OrderList_ItemChanged () { … }
```