

Multithreading

Multithreading in C# is a process in which multiple threads work simultaneously. It is a process to achieve multitasking. It saves time because multiple tasks are being executed at a time. To create multithreaded application in C# we need to use **System.Threading** namespace.

System.Threading Namespace

The System.Threading namespace contains classes and interfaces to provide the facility of multithreaded programming. It also provides classes to synchronize the thread resource. A list of commonly used classes are given below:

- Thread
- Mutex
- Timer
- Monitor
- Semaphore
- ThreadLocal
- ThreadPool
- Volatile etc.

Process and Thread

A process represents an application whereas a thread represents a module of an application. Process is a heavyweight component whereas the thread is lightweight. A thread can be termed as a lightweight subprocess because it is executed inside a process.

Whenever you create a process, a separate memory area is occupied. But threads share a common memory area.

Thread Life Cycle

Each thread has a lifecycle, the life cycle of a thread is started when an instance of System.Threading.Thread class is created. When the task execution of the thread is completed, its lifecycle is ended.

There are following states in the lifecycle of a thread in c#

- Unstarted
 - When the instance of a thread class is created, it is in an unstarted state by default.
- Runnable(Ready to run)
 - When the start() method on the thread is called, it is in runnable or ready to run state.
- Running
 - Only one thread within a process can be executed at a time. At time of execution, the thread is in running state.
- Not Runnable
 - The thread is in not runnable state, if sleep() or wait() method is called on the thread, or input/output operation is blocked.
- Dead(Terminated)
 - After completing the task, thread enters into dead or terminated state.

Thread Class

Thread class provides properties and methods to create and control threads. It is found in the System.Threading namespace.

C# Thread Properties

A list of important properties of thread class are listed below:

Property	Description
CurrentThread	Returns the instance of currently running thread
isAlive	Checks whether the current thread is alive or not. It is used to find the execution status of the thread.
isBackground	Is used to get or set value whether the current thread is in background or not.
ManagedThreadId	Is used to get a unique id for the current managed thread.

Name	Is used to get or set the name of the current thread.
Priority	Is used to get or set the priority of the current thread.
ThreadState	Is used to return a value representing the thread state.

C# Thread Methods

A list of important methods of threads in C# are listed below:

Method	Description
Abort()	Is used to terminate the thread. It raises ThreadAbortException.
Interrupt()	Is used to interrupt a thread which is in WaitSleepJoin state.
Join()	Is used to block all the calling threads until this thread terminates.
ResetAbort()	Is used to cancel the abort request for the current thread.
Resume()	Is used to resume the current thread. It is obsolete.
Sleep(Int 32)	Is used to suspend the current thread for the specified milliseconds.
Start()	Changes the current state of the thread to Runnable.
Suspend()	Suspends the current thread if it is not suspended. It is obsolete.
Yield()	Is used to yield the execution of the current thread to another thread.

Main Thread Example

The first thread which is created inside the process is called the main thread. It starts first and ends at last.

Example of main thread in C#

```
using System;
using System.Threading;
public class ThreadExample
{
    public static void Main(string[] args)
    {
        Thread t = Thread.CurrentThread;
        t.Name = "MainThread";
        Console.WriteLine(t.Name);
    }
}
```

Output:

MainThread

Threading Example: static method

We can call static and non-static methods on the execution of the thread. To call the static and non-static methods, you need to pass a method name in the constructor of the ThreadStart class. For static methods, we don't need to create the instance of the class. You can refer to it by the name of the class.

```
using System;
using System.Threading;
public class MyThread
{
    public static void Thread1()
    {
        for (int i = 0; i < 10; i++)
        {
            Console.WriteLine(i);
        }
    }
}
public class ThreadExample
{
    public static void Main()
    {
        Thread t1 = new Thread(new ThreadStart(MyThread.Thread1));
        Thread t2 = new Thread(new ThreadStart(MyThread.Thread1));
        t1.Start();
        t2.Start();
    }
}
```

Output:

0
1
2
3
4
5
0
1
2
3
4
5
6
7
8
9

6
7
8
9

Threading Example: non-static method

For non-static methods, you need to create an instance of the class so that you can refer to it in the constructor of the ThreadStart class.

```
using System;
using System.Threading;
public class MyThread
{
    public void Thread1()
    {
        for (int i = 0; i < 10; i++)
        {
            Console.WriteLine(i);
        }
    }
}
public class ThreadExample
{
    public static void Main()
    {
        MyThread mt = new MyThread();
        Thread t1 = new Thread(new ThreadStart(mt.Thread1));
        Thread t2 = new Thread(new ThreadStart(mt.Thread1));
        t1.Start();
        t2.Start();
    }
}
```

Output:

Like above program output, the output of this program can be anything because there is context switching between the threads.

Threading Example: performing different tasks on each thread

Let's see an example where we are executing different methods on each thread.

```
using System;
using System.Threading;

public class MyThread
{
    public static void Thread1()
    {
        Console.WriteLine("task one");
    }
    public static void Thread2()
    {
        Console.WriteLine("task two");
    }
}

public class ThreadExample
{
    public static void Main()
    {
        Thread t1 = new Thread(new ThreadStart(MyThread.Thread1));
        Thread t2 = new Thread(new ThreadStart(MyThread.Thread2));
        t1.Start();
        t2.Start();
    }
}
```

Output:

task one

task two

Thread Sleep

The Sleep() method suspends the current thread for the specified milliseconds. So, other threads get the chance to start execution.

```
using System;
using System.Threading;
public class MyThread
{
    public void Thread1()
    {
        for (int i = 0; i < 10; i++)
        {
            Console.WriteLine(i);
            Thread.Sleep(200);
        }
    }
}
public class ThreadExample
{
    public static void Main()
    {
        MyThread mt = new MyThread();
        Thread t1 = new Thread(new ThreadStart(mt.Thread1));
        Thread t2 = new Thread(new ThreadStart(mt.Thread1));
        t1.Start();
        t2.Start();
    }
}
```

Output:

```
0
0
1
1
2
2
3
3
4
4
5
5
6
6
7
7
8
8
9
9
```


Threading Example: Abort() Method

The Abort() method is used to terminate the thread. It raises ThreadAbortException if the Abort operation is not done.

```
using System;
using System.Threading;
public class MyThread
{
    public void Thread1()
    {
        for (int i = 0; i < 10; i++)
        {
            Console.WriteLine(i);
            Thread.Sleep(200);
        }
    }
}
public class ThreadExample
{
    public static void Main()
    {
        Console.WriteLine("Start of Main");
        MyThread mt = new MyThread();
        Thread t1 = new Thread(new ThreadStart(mt.Thread1));
        Thread t2 = new Thread(new ThreadStart(mt.Thread1));

        t1.Start();
        t2.Start();
```

```
    try
    {
        t1.Abort();
        t2.Abort();
    }
    catch (ThreadAbortException tae)
    {
        Console.WriteLine(tae.ToString());
    }
    Console.WriteLine("End of Main");
}
```

Output:

Output is unpredictable because thread may be in running state.

```
Start of Main  
0  
End of Main
```

Threading Example: Join() Method

It causes all the calling threads to wait until the current thread (joined thread) is terminated or completes its task.

```
using System;  
using System.Threading;  
public class MyThread  
{  
    public void Thread1()  
    {  
        for (int i = 0; i < 5; i++)  
        {  
            Console.WriteLine(i);  
            Thread.Sleep(200);  
        }  
    }  
}  
public class ThreadExample  
{  
    public static void Main()  
    {  
        MyThread mt = new MyThread();  
        Thread t1 = new Thread(new ThreadStart(mt.Thread1));  
        Thread t2 = new Thread(new ThreadStart(mt.Thread1));  
        Thread t3 = new Thread(new ThreadStart(mt.Thread1));  
        t1.Start();  
        t1.Join();  
        t2.Start();  
        t3.Start();  
    }  
}
```

Output:

```
0
1
2
3
4
0
0
1
1
2
2
3
3
4
4
```

Threading Example: Naming Thread

You can change or get the name of the thread by using the Name property of Thread class.

Example:

```
using System;
using System.Threading;

public class MyThread
{
    public void Thread1()
    {
        Thread t = Thread.CurrentThread;
        Console.WriteLine(t.Name+" is running");
    }
}

public class ThreadExample
{
    public static void Main()
    {
        MyThread mt = new MyThread();
        Thread t1 = new Thread(new ThreadStart(mt.Thread1));
        Thread t2 = new Thread(new ThreadStart(mt.Thread1));
        Thread t3 = new Thread(new ThreadStart(mt.Thread1));
        t1.Name = "Player1";
        t2.Name = "Player2";
        t3.Name = "Player3";
        t1.Start();
        t2.Start();
        t3.Start();
    }
}
```

Output:

```
Player1 is running
Player2 is running
Player3 is running
```

Threading Example: ThreadPriority

Let's see an example where we are changing the priority of the thread. The high priority thread can be executed first. But it is not guaranteed because thread is highly system dependent. It increases the chance of the high priority thread to execute before the low priority thread.

```
using System;
using System.Threading;
public class MyThread
{
    public void Thread1()
    {
        Thread t = Thread.CurrentThread;
        Console.WriteLine(t.Name+" is running");
    }
}
public class ThreadExample
{
    public static void Main()
    {
        MyThread mt = new MyThread();
        Thread t1 = new Thread(new ThreadStart(mt.Thread1));
        Thread t2 = new Thread(new ThreadStart(mt.Thread1));
        Thread t3 = new Thread(new ThreadStart(mt.Thread1));
        t1.Name = "Player1";
        t2.Name = "Player2";
        t3.Name = "Player3";
        t3.Priority = ThreadPriority.Highest;
        t2.Priority = ThreadPriority.Normal;
        t1.Priority = ThreadPriority.Lowest;

        t1.Start();
        t2.Start();
        t3.Start();
    }
}
```

Output:

The output is unpredictable because threads are highly system dependent. It may follow any algorithm preemptive or non-preemptive.

```
Player1 is running  
Player3 is running  
Player2 is running
```

FILES

A file is a collection of data stored in a disk with a specific name and a directory path. When a file is opened for reading or writing it becomes a stream.

The stream is basically the sequence of bytes passing through the communication path. There are two main streams:

- The Input Stream
 - Used for reading data from file(read operation).
- The Output Stream
 - Used for writing into the file(write operation).

FileStream

C# FileStream class provides a stream for file operation. It can be used to perform synchronous and asynchronous read and write operations. By the help of FileStream class, we can easily read and write data into a file.

FileStream Example: Writing single byte into file

Here, we are using OpenOrCreate file mode which can be used for read and write operations.

```

using System;
using System.IO;
public class FileStreamExample
{
    public static void Main(string[] args)
    {
        FileStream f = new FileStream("e:\\b.txt", FileMode.OpenOrCreate);//creating file stream
        f.WriteByte(65);//writing byte into stream
        f.Close();//closing stream
    }
}

```

Output:

A

FileStream Example: Writing multiple bytes into file

Let's see another example to write multiple bytes of data into a file using a loop.

```

using System;
using System.IO;
public class FileStreamExample
{
    public static void Main(string[] args)
    {
        FileStream f = new FileStream("e:\\b.txt", FileMode.OpenOrCreate);
        for (int i = 65; i <= 90; i++)
        {
            f.WriteByte((byte)i);
        }
        f.Close();
    }
}

```

Output:

ABCDEFGHIJKLMNOPQRSTUVWXYZ

FileStream Example: Reading all bytes from file

Here, the ReadByte() method of FileStream class returns a single byte. To read all the bytes, you need to use a loop.

```
using System;
using System.IO;
public class FileStreamExample
{
    public static void Main(string[] args)
    {
        FileStream f = new FileStream("e:\\b.txt", FileMode.OpenOrCreate);
        int i = 0;
        while ((i = f.ReadByte()) != -1)
        {
            Console.Write((char)i);
        }
        f.Close();
    }
}
```

Output:

ABCDEFGHIJKLMNOPQRSTUVWXYZ

StreamWriter

StreamWriter class is used to write characters to a stream in specific encoding. It inherits the TextWriter class. It provides overloaded write() and writeln() methods to write data into a file.

StreamWriter Example:

A simple example of StreamWriter class which writes a single line of data into the file.


```
using System;
using System.IO;
public class StreamWriterExample
{
    public static void Main(string[] args)
    {
        FileStream f = new FileStream("e:\\output.txt", FileMode.Create);
        StreamWriter s = new StreamWriter(f);

        s.WriteLine("hello c#");
        s.Close();
        f.Close();
        Console.WriteLine("File created successfully...");
    }
}
```

Output:

File Created Successfully...

Now open the file, you will see the text "hello c#" in the output.txt file.

Output.txt:

hello c#

StreamReader

The StreamReader class is used to read strings from the stream. It inherits the TextReader class. It provides Read() and ReadLine() methods to read data from the stream.

StreamReader Example

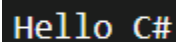
Example of StreamReader class that reads a single line of data from the file.

```
using System;
using System.IO;
public class StreamReaderExample
{
    public static void Main(string[] args)
    {
        FileStream f = new FileStream("e:\\output.txt", FileMode.OpenOrCreate);
        StreamReader s = new StreamReader(f);

        string line=s.ReadLine();
        Console.WriteLine(line);

        s.Close();
        f.Close();
    }
}
```

Output:



Hello C#

StreamReader Example to read all lines

```
using System;
using System.IO;
public class StreamReaderExample
{
    public static void Main(string[] args)
    {
        FileStream f = new FileStream("e:\\a.txt", FileMode.OpenOrCreate);
        StreamReader s = new StreamReader(f);

        string line = "";
        while ((line = s.ReadLine()) != null)
        {
            Console.WriteLine(line);
        }
        s.Close();
        f.Close();
    }
}
```

Output:

```
Hello C#  
this is file handling
```

TextWriter

The `TextWriter` class is an abstract class. It is used to write text or a sequential series of characters into a file. It is found in the `System.IO` namespace.

TextWriter Example

```
using System;  
using System.IO;  
namespace TextWriterExample  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            using (TextWriter writer = File.CreateText("e:\\f.txt"))  
            {  
                writer.WriteLine("Hello C#");  
                writer.WriteLine("C# File Handling by JavaTpoint");  
            }  
            Console.WriteLine("Data written successfully...");  
        }  
    }  
}
```

Output:

```
Data written successfully...
```

f.txt

```
Hello C#  
C# File Handling by JavaTpoint
```

TextReader

The TextReader class is found in the System.IO namespace. It represents a reader that can be used to read text or sequential series of characters.

TextReader Example

Example of TextReader class that reads data till the end of file.

```
using System;
using System.IO;
namespace TextReaderExample
{
    class Program
    {
        static void Main(string[] args)
        {
            using (TextReader tr = File.OpenText("e:\\f.txt"))
            {
                Console.WriteLine(tr.ReadToEnd());
            }
        }
    }
}
```

Output:

```
Hello C#
C# File Handling by JavaTpoint
```

TextReader Example: Read one line

```
using System;
using System.IO;
namespace TextReaderExample
{
    class Program
    {
        static void Main(string[] args)
        {
            using (TextReader tr = File.OpenText("e:\\f.txt"))
            {
                Console.WriteLine(tr.ReadLine());
            }
        }
    }
}
```

Output:

```
Hello C#
```

BinaryWriter

BinaryWriter class is used to write binary information into stream. It is found in the System.IO namespace. It also supports writing strings in specific encoding.

BinaryWriter Example

Example of BinaryWriter class which writes data into dat file.

```

using System;
using System.IO;
namespace BinaryWriterExample
{
    class Program
    {
        static void Main(string[] args)
        {
            string fileName = "e:\\binaryfile.dat";
            using (BinaryWriter writer = new BinaryWriter(File.Open(fileName, FileMode.Create)))
            {
                writer.Write(2.5);
                writer.Write("this is string data");
                writer.Write(true);
            }
            Console.WriteLine("Data written successfully...");
        }
    }
}

```

Output:

```
Data written successfully...
```

BinaryReader

The BinaryReader class is used to read binary information from stream. It is found in the System.IO namespace. It also supports reading strings in specific encoding.

BinaryReader Example

```

static void ReadBinaryFile()
{
    using (BinaryReader reader = new BinaryReader(File.Open("e:\\binaryfile.dat", FileMode.Open)))
    {
        Console.WriteLine("Double Value : " + reader.ReadDouble());
        Console.WriteLine("String Value : " + reader.ReadString());
        Console.WriteLine("Boolean Value : " + reader.ReadBoolean());
    }
}

```

Output:

```
Double Value : 12.5
String Value : this is string data
Boolean Value : true
```

StringWriter Class

This class is used to write and deal with string data rather than files. It is derived from the `TextWriter` class. The string data written by the `StringWriter` class is stored into `StringBuilder`.

The purpose of this class is to manipulate string and save results into the `StringBuilder`.

StringWriter class signature

```
[SerializableAttribute]
[ComVisibleAttribute(true)]
public class StringWriter : TextWriter
```

StringWriter Constructors

Constructors	Description
<code>StringWriter()</code>	It is used to initialize a new instance of the <code>StringWriter</code> class.
<code>StringWriter(IFormatProvider)</code>	It is used to initialize a new instance of the <code>StringWriter</code> class with the specified format control.
<code>StringWriter(StringBuilder)</code>	It is used to initialize a new instance of the <code>StringWriter</code> class that writes to the specified <code>StringBuilder</code> .
<code>StringWriter(StringBuilder,? IFormatProvider)</code>	It is used to initialize a new instance of the <code>StringWriter</code> class that writes to the specified <code>StringBuilder</code> and has the specified format provider.

StringWriter Properties

Property	Description
Encoding	It is used to get the Encoding in which the output is written.
FormatProvider	It is used to get an object that controls formatting.
NewLine	It is used to get or set the line terminator string used by the current TextWriter .

StringWriter Methods

Methods	Description
Close()	It is used to close the current StringWriter and the underlying stream.
Dispose()	It is used to release all resources used by the TextWriter object.
Equals(Object)	It is used to determine whether the specified object is equal to the current object or not.
Finalize()	It allows an object to try to free resources and perform other cleanup operations.
GetHashCode()	It is used to serve as the default hash function.
GetStringBuilder()	It returns the underlying StringBuilder.
ToString()	It returns a string containing the characters written to the current StringWriter.
WriteAsync(String)	It is used to write a string to the current string asynchronously.
Write(Boolean)	It is used to write the text representation of a Boolean value to the string.
Write(String)	It is used to write a string to the current string.
WriteLine(String)	It is used to write a string followed by a line terminator to the string or stream.
WriteLineAsync(String)	Writes a string followed by a line terminator asynchronously to the current string.(Overrides TextWriter.WriteLineAsync(String).)

StringWriter Example

We are using the StringWriter class to write string information to the StringBuilder class. The StringReader class is used to read written information to the StringBuilder.


```

using System;
using System.IO;
using System.Text;
namespace CSharpProgram
{
    class Program
    {
        static void Main(string[] args)
        {
            string text = "Hello, Welcome to the javatpoint \n" +
                "It is nice site. \n" +
                "It provides technical tutorials";
            // Creating StringBuilder instance
            StringBuilder sb = new StringBuilder();
            // Passing StringBuilder instance into StringWriter
            StringWriter writer = new StringWriter(sb);
            // Writing data using StringWriter
            writer.WriteLine(text);
            writer.Flush();
            // Closing writer connection
            writer.Close();
            // Creating StringReader instance and passing StringBuilder
            StringReader reader = new StringReader(sb.ToString());
            // Reading data

            while (reader.Peek() > -1)
            {
                Console.WriteLine(reader.ReadLine());
            }
        }
    }
}

```

Output:

```

Hello, Welcome to the javatpoint
It is nice site.
It provides technical tutorials

```

StringReader

StringReader class is used to read data written by the StringWriter class. It is a subclass of the TextReader class. It enables us to read a string synchronously or asynchronously. It provides constructors and methods to perform read operations.

StringReader Signature

```
[SerializableAttribute]  
[ComVisibleAttribute(true)]  
public class StringReader : TextReader
```

StringReader Constructors

Constructors	Description
StringReader(String)	Initializes a new instance of the StringReader class that reads from the specified string.

StringReader Methods

Method	Description
Close()	It is used to close the StringReader.
Dispose()	It is used to release all resources used by the TextReader object.
Equals(Object)	It determines whether the specified object is equal to the current object or not.
Finalize()	It allows an object to try to free resources and perform other cleanup operations.
GetHashCode()	It serves as the default hash function.
GetType()	It is used to get the type of the current instance.
Peek()	It is used to return the next available character but does not consume it.
Read()	It is used to read the next character from the input string.
ReadLine()	It is used to read a line of characters from the current string.
ReadLineAsync()	It is used to read a line of characters asynchronously from the current string.
ReadToEnd()	It is used to read all the characters from the current position to the end of the string.
ReadToEndAsync()	It is used to read all the characters from the current position to the end of the string asynchronously.
ToString()	It is used to return a string that represents the current object.

StringReader Example

The StringWriter class is used to write the string information and the StringReader class is used to read the string, written by the StringWriter class.

```
using System;
using System.IO;
namespace CSharpProgram
{
    class Program
    {
        static void Main(string[] args)
        {
            StringWriter str = new StringWriter();
            str.WriteLine("Hello, this message is read by StringReader class");
            str.Close();
            // Creating StringReader instance and passing StringWriter
            StringReader reader = new StringReader(str.ToString());
            // Reading data
            while (reader.Peek() > -1)
            {
                Console.WriteLine(reader.ReadLine());
            }
        }
    }
}
```

Output:

```
Hello, this message is read by StringReader class
```

FileInfo

The FileInfo class is used to deal with file and its operations in C#. It provides properties and methods that are used to create, delete and read files. It uses the StreamWriter class to write data to the file. It is a part of the System.IO namespace.

FileInfo Constructors

Constructor	Description
FileInfo(String)	It is used to initialize a new instance of the FileInfo class which acts as a wrapper for a file path.

FileInfo Properties

Properties	Description
Attributes	It is used to get or set the attributes for the current file or directory.
CreationTime	It is used to get or set the creation time of the current file or directory.
Directory	It is used to get an instance of the parent directory.
DirectoryName	It is used to get a string representing the directory's full path.
Exists	It is used to get a value indicating whether a file exists.
FullName	It is used to get the full path of the directory or file.
IsReadOnly	It is used to get or set a value that determines if the current file is read only.
LastAccessTime	It is used to get or set the time from current file or directory was last accessed.
Length	It is used to get the size in bytes of the current file.
Name	It is used to get the name of the file.

FileInfo Methods

Method	Description
AppendText()	It is used to create a StreamWriter that appends text to the file represented by this instance of the FileInfo.
CopyTo(String)	It is used to copy an existing file to a new file.
Create()	It is used to create a file.
CreateText()	It is used to create a StreamWriter that writes a new text file.
Decrypt()	It is used to decrypt a file that was encrypted by the current account using the Encrypt method.
Delete()	It is used to permanently delete a file.
Encrypt()	It is used to encrypt a file so that only the account used to encrypt the file can decrypt it.
GetAccessControl()	It is used to get a FileSecurity object that encapsulates the access control list (ACL) entries.
MoveTo(String)	It is used to move a specified file to a new specified location.
Open(FileMode)	It is used to open a file in the specified mode.
OpenRead()	It is used to create a read-only FileStream.
OpenText()	It is used to create a StreamReader with UTF8 encoding that reads from an existing text file.
OpenWrite()	It is used to create a write-only FileStream.
Refresh()	It is used to refresh the state of the object.
Replace(String,String)	It is used to replace the contents of a specified file with the file described by the current FileInfo object.
ToString()	It is used to return the path as a string.

FileInfo Example: Creating a file

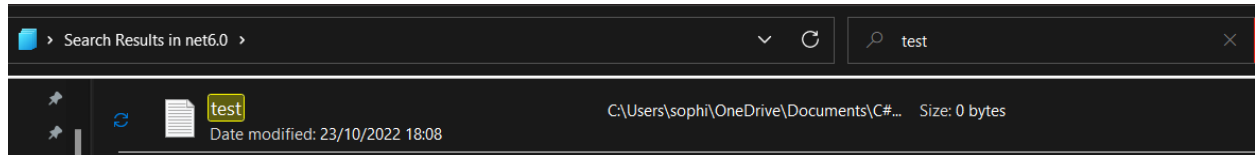
```
using System;
using System.IO;
namespace CSharpProgram
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                // Specifying file location
                string loc = "F:\\abc.txt";
                // Creating FileInfo instance
                FileInfo file = new FileInfo(loc);
                // Creating an empty file
                file.Create();
                Console.WriteLine("File is created Successfully");
            } catch (IOException e)
            {
                Console.WriteLine("Something went wrong: "+e);
            }
        }
    }
}
```

```
1 reference
public class FileInfoLesson
{
    1 reference
    public static void fInfo()
    {
        try
        {
            FileInfo f = new FileInfo("test.txt");
            f.Create();
            Console.WriteLine("File created successfully");
        }
        catch (IOException e)
        {
            Console.WriteLine("Something Went Wrong: {0}", e);
        }
    }
}
```

Output:

File is created Successfully

We can see the file **test.txt** is created. A screenshot is given below.

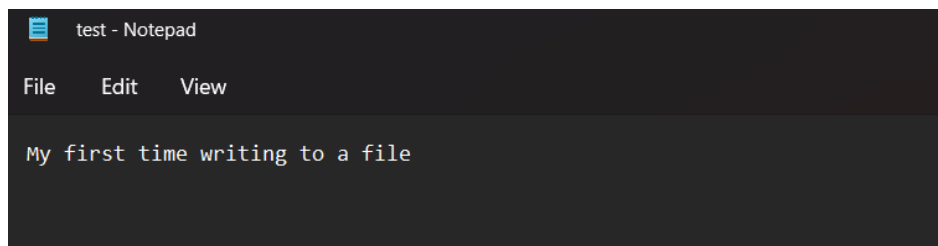


FileInfo Example: writing to a file

```
public static void fInfo()
{
    try
    {
        FileInfo f = new FileInfo("test.txt");
        Console.WriteLine("File created successfully");

        StreamWriter sw = f.CreateText();
        sw.WriteLine("My first time writing to a file");
        sw.Close();
    }
    catch(IOException e)
    {
        Console.WriteLine("Something Went Wrong: {0}", e);
    }
}
```

Output:



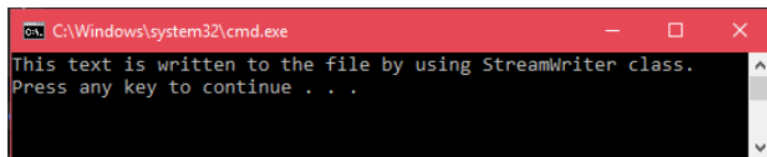
FileInfo Example: Reading text from file

```

namespace CSharpProgram
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                // Specifying file to read
                string loc = "F:\\abc.txt";
                // Creating FileInfo instance
                FileInfo file = new FileInfo(loc);
                // Opening file to read
                StreamReader sr = file.OpenText();
                string data = "";
                while ((data = sr.ReadLine()) != null)
                {
                    Console.WriteLine(data);
                }
            }
            catch (IOException e)
            {
                Console.WriteLine("Something went wrong: " + e);
            }
        }
    }
}

```

Output:



A screenshot of a Windows command prompt window. The title bar shows the path "C:\Windows\system32\cmd.exe". The window contains the following text: "This text is written to the file by using StreamWriter class." followed by "Press any key to continue . . .". The text is displayed in a monospaced font on a black background.

Directory Info

The DirectoryInfo class is a part of the System.IO namespace. It is used to create, delete and move directory. It provides methods to perform operations related to directory and subdirectory. It is a sealed class so we cannot inherit it.

DirectoryInfo Example:

```

using System;
using System.IO;
namespace CSharpProgram
{
    class Program
    {
        static void Main(string[] args)
        {
            // Provide directory name with complete location.
            DirectoryInfo directory = new DirectoryInfo(@"F:\javatpoint");
            try
            {
                // Check, directory exist or not.
                if (directory.Exists)
                {
                    Console.WriteLine("Directory already exist.");
                    return;
                }
                // Creating a new directory.
                directory.Create();
                Console.WriteLine("The directory is created successfully.");
            }
            catch (Exception e)
            {
                Console.WriteLine("Directory not created: {0}", e.ToString());
            }
        }
    }
}

```

Output:

```
The directory is created successfully.
```

In the screenshot below, we can see that a directory is created.

The **DirectoryInfo** class also provides a delete method to delete created directory. In the following program, we are deleting a directory that we created in the previous program.

DirectoryInfo Example: Deleting Directory


```

using System;
using System.IO;
namespace CSharpProgram
{
    class Program
    {
        static void Main(string[] args)
        {
            // Providing directory name with complete location.
            DirectoryInfo directory = new DirectoryInfo(@"F:\javatpoint");
            try
            {
                // Deleting directory
                directory.Delete();
                Console.WriteLine("The directory is deleted successfully.");
            }
            catch (Exception e)
            {
                Console.WriteLine("Something went wrong: {0}", e.ToString());
            }
        }
    }
}

```

Output:

```
The directory is deleted successfully.
```

DATETIME

We used the DateTime when there is a need to work with the dates and times in C#.

We can format the date and time in different formats by the properties and methods of the DateTime.

The value of the DateTime is between 12:00:00 midnight, January 1 0001 and 11:59:59 PM, December 31, 9999 A.D.

Here we will explain how to create the DateTime in C#.

We have different ways to create the DateTime object. A DateTime object has Time, Culture, Date, Localization, Milliseconds.

Here we have a code which shows the various constructor used by the DateTime structure to create the DateTime objects.

```
// From DateTime create the Date and Time
DateTime DOB= new DateTime(19, 56, 8, 12, 8, 12, 23);
// From String creation of DateTime
string DateString= "8/12/1956 7:10:24 AM";
DateTime dateFromString =
    DateTime.Parse(DateString, System.Globalization.CultureInfo.InvariantCulture);
Console.WriteLine(dateFromString.ToString());
// Empty DateTime
DateTime EmpDateTime= new DateTime();
// Just date
DateTime OnlyDate= new DateTime(2002, 10, 18);
// DateTime from Ticks
DateTime OnlyTime= new DateTime(10000000);
// Localization with DateTime
DateTime DateTimeWithKind = new DateTime(1976, 7, 10, 7, 10, 24, DateTimeKind.Local);
// DateTime with date, time and milliseconds
DateTime WithMilliseconds= new DateTime(2010, 12, 15, 5, 30, 45, 100);
```

Properties of DateTime in c#

The DateTime has the Date and Time property. From DateTime, we can find the date and time. DateTime contains other properties as well, like Hour, Minute, Second, Millisecond, Year, Month, and Day.

The other properties of DateTime are:

1. We can get the name of the day from the week with the help of the DayOfWeek property.
2. To get the day of the year, we will use the DayOfYear property.
3. To get time in a DateTime, we use TimeOfDay property.
4. Today property will return the object of the DateTime, which is having today's value. The value of the time is 12:00:00
5. The Now property will return the DateTime object, which is having the current date and time.
6. The Utc property of DateTime will return the Coordinated Universal Time (UTC).

7. The one tick represents the One hundred nanoseconds in DateTime. Ticks property of the DateTime returns the number of ticks in a DateTime.
8. The Kind property returns value where the representation of time is done by the instance, which is based on the local time, Coordinated Universal Time (UTC). It also shows the unspecified default value.

Example:

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp8
{
    class Program
    {
        static void Main(string[] args)
        {

            DateTime DateTimeProperty = new DateTime(1974, 7, 10, 7, 10, 24);
            Console.WriteLine("Day:{0}", DateTimeProperty.Day);
            Console.WriteLine("Month:{0}", DateTimeProperty.Month);
            Console.WriteLine("Year:{0}", DateTimeProperty.Year);
            Console.WriteLine("Hour:{0}", DateTimeProperty.Hour);
            Console.WriteLine("Minute:{0}", DateTimeProperty.Minute);
            Console.WriteLine("Second:{0}", DateTimeProperty.Second);
            Console.WriteLine("Millisecond:{0}", DateTimeProperty.Millisecond);
        }
    }
}
```

```
Console.WriteLine("Day of Week:{0}", DateTimeProperty.DayOfWeek);
Console.WriteLine("Day of Year: {0}", DateTimeProperty.DayOfYear);
Console.WriteLine("Time of Day:{0}", DateTimeProperty.TimeOfDay);
Console.WriteLine("Tick:{0}", DateTimeProperty.Ticks);
Console.WriteLine("Kind:{0}", DateTimeProperty.Kind);
    }
}
}
```

Output:

```
Day:10
Month:7
Year:1974
Hour:7
Minute:10
Second:24
Millisecond:0
Day of Week:Wednesday
Day of Year: 191
Time of Day:07:10:24
Tick:622782690240000000
Kind:Unspecified
Press any key to continue . . .
```

Addition and subtraction of DateTime

The DateTime structure provides the methods to add and subtract the date and time to and from the DateTime object. We can add and subtract the date in the DateTime structure to and from the DateTime object. For the Addition and Subtraction in the DateTime, we use the TimeSpan structure.

For Addition and Subtraction, we can use the Add and Subtract method from the DateTime object. Firstly, we create the TimeSpan with the values of the date and time where we use the Add and Subtract methods.

Example:

```

using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp8
{
    class Program
    {
        static void Main(string[] args)
        {

            DateTime Day = DateTime.Now;
            TimeSpan Month = new System.TimeSpan(30, 0, 0, 0);
            DateTime aDayAfterAMonth = Day.Add(Month);
            DateTime aDayBeforeAMonth = Day.Subtract(Month);
            Console.WriteLine("{0:ddd}", aDayAfterAMonth);
            Console.WriteLine("{0:ddd}", aDayBeforeAMonth);
        }
    }
}

```

DateTime structure contains the methods to add years, days, hours, minutes, seconds.

To add the different components to the DateTime object, the Add method is used.

```

// To Add the Years and Days
day.AddYears(2);
day.AddDays(12);
// Add Hours, Minutes, Seconds, Milliseconds, and Ticks
Day.AddHours(4.25);
day.AddMinutes(15);
day.AddSeconds(45);
day.AddMilliseconds(200);
day.AddTicks(5000);

```

The DateTime does not contain the subtract method. To subtract the component of the DateTime, we will use only the subtract method. For example: if we need to subtract the 12 days

from the DateTime, we can create another object of the DateTime or TimeSpan object with 12 days. Now we will subtract this object from the DateTime. In addition to this, we can also use the minus operator to subtract the DateTime or TimeSpan from the DateTime.

Example:

```
DateTime DOB = new DateTime(2000, 10, 20, 12, 15, 45);
DateTime SubtractDate = new DateTime(2000, 2, 6, 13, 5, 15);

// Use the TimeSpan with 10 days, 2 hrs, 30 mins, 45 seconds, and 100 milliseconds
TimeSpan ts = new TimeSpan(10, 2, 30, 45, 100);

// Subtract the DateTime
TimeSpan Different = DOB.Subtract(SubtractDate);
Console.WriteLine(Different.ToString());

// Subtract the TimeSpan
DateTime Different2 = DOB.Subtract(ts);
Console.WriteLine(Different2.ToString());

// Subtract 10 Days by creating the object SubtractedDays
DateTime SubtractedDays = new DateTime(DOB.Year, DOB.Month, DOB.Day - 10);
Console.WriteLine(SubtractedDays.ToString());

// Subtract hours, minutes, and seconds with creating the object HoursMinutesSeconds
DateTime HoursMinutesSeconds = new DateTime(DOB.Year, DOB.Month, DOB.Day, DOB.Hour - 1, DOB.Minute - 15, DOB.Second - 45);
Console.WriteLine(HoursMinutesSeconds.ToString());
```

Searching of the days in the months

To find the number of days in the month, we used the static **DaysInMonth** method. This searching method [] takes the parameter in numbers from 1 to 12.

```
int NumberOfDays = DateTime.DaysInMonth(2004, 2);
Console.WriteLine(NumberOfDays);
```

With the same technique, we can find out the total number of days in a year. For that, we will use the method DaysInYear.

```

private int DaysInYear(int year)
{
    int DaysIN= 0;
    for (int j = 1; j <= 12; j++)
    {
        DaysIN += DateTime.DaysInMonth(year, j);
    }
    return DaysIN;
}

```

Comparison of two DateTime

The comparer static method is used to compare the object of the two datetime. If the objects of both DateTime are the same, then the result will be 0. If the first DateTime is earlier, then the result will be 0 else the first DateTime would be later.

```

using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp8
{
    class Program
    {
        static void Main(string[] args)
        {

            DateTime DateOfFirst = new DateTime(2002, 10, 22);
            DateTime DateOfSecond = new DateTime(2009, 8, 11);
            int result1 = DateTime.Compare(DateOfFirst, DateOfSecond);

            if (result1 < 0)
                Console.WriteLine("Date of First is earlier");
            else if (result1 == 0)
                Console.WriteLine("Both dates are same");
            else
                Console.WriteLine("Date of First is later");

        }
    }
}

```

Output:

```
Date of First is earlier  
Press any key to continue . . .
```

CompareTo Method

The CompareTo method is used to compare the two dates. We will assign the DateTime or object in this method.

```
using System;  
using System.Collections;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
using System.Threading.Tasks;  
  
namespace ConsoleApp8  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
  
            DateTime DateOfFirst = new DateTime(2001, 10, 20);  
            DateTime DateOfSecond = new DateTime(2009, 8, 11);  
            int ResultOfComparison = DateOfFirst.CompareTo(DateOfSecond);  
            if (ResultOfComparison < 0)  
                Console.WriteLine("Date Of First is Earlier");  
            else if (ResultOfComparison == 0)  
                Console.WriteLine("Date of Both are same");  
            else  
                Console.WriteLine("Date of First is Later");  
  
        }  
    }  
}
```

Output:

```
Date of First is earlier  
Press any key to continue . . .
```


Formatting of the DateTime

In C#, we can format the DateTime to any type of string format as we want.

Here we have a C# code that returns the array of the strings of all the possible standard formats.

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp8
{
    class Program
    {
        static void Main(string[] args)
        {

            DateTime DateOfMonth = new DateTime(2020, 02, 25);
            string[] FormatsOfDate = DateOfMonth.GetDateTimeFormats();
            foreach (string format in FormatsOfDate)
                Console.WriteLine(format);

        }
    }
}
```

Output:

```

25-02-2020
25-02-20
25-2-20
25.2.20
2020-02-25
25 February 2020
25 February 2020
Tuesday, 25 February, 2020
25 February 2020 00:00
25 February 2020 0:00
25 February 2020 12:00 AM
25 February 2020 00:00
25 February 2020 0:00
25 February 2020 12:00 AM
Tuesday, 25 February, 2020 00:00
Tuesday, 25 February, 2020 0:00
Tuesday, 25 February, 2020 12:00 AM
25 February 2020 00:00:00
25 February 2020 0:00:00
25 February 2020 12.00.00 AM
25 February 2020 12:00:00 AM
25 February 2020 00:00:00
25 February 2020 0:00:00
25 February 2020 12.00.00 AM
25 February 2020 12:00:00 AM
Tuesday, 25 February, 2020 00:00:00
Tuesday, 25 February, 2020 0:00:00
Tuesday, 25 February, 2020 12.00.00 AM
Tuesday, 25 February, 2020 12:00:00 AM
25-02-2020 00:00
25-02-2020 0:00
25-02-2020 12:00 AM
25-02-20 00:00

```

```

2020-02-25T00:00:00.0000000
2020-02-25T00:00:00.0000000
Tue, 25 Feb 2020 00:00:00 GMT
Tue, 25 Feb 2020 00:00:00 GMT
2020-02-25T00:00:00
00:00
0:00
12:00 AM
00:00:00
0:00:00
12.00.00 AM
12:00:00 AM
2020-02-25 00:00:00Z
24 February 2020 18:30:00
24 February 2020 18:30:00
24 February 2020 6.30.00 PM
24 February 2020 06:30:00 PM
24 February 2020 18:30:00
24 February 2020 18:30:00
24 February 2020 6.30.00 PM
24 February 2020 06:30:00 PM
Monday, 24 February, 2020 18:30:00
Monday, 24 February, 2020 18:30:00
Monday, 24 February, 2020 6.30.00 PM
Monday, 24 February, 2020 06:30:00 PM
February, 2020
February, 2020
Press any key to continue . . .

```

We can overload the **GetDateTimeFormats** method, which takes the format specifier as a good parameter and converts the **DateTime** to that format. To get the desired format, we need to understand the format of the **DateTime** specifiers.

We will specify the format of the DateTime in the below C# Code.

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp8
{
    class Program
    {
        static void Main(string[] args)
        {

            DateTime FormatOfDate = new DateTime(2020, 02, 25);
            // DateTime Formats: d, D, f, F, g, G, m, o, r, s, t, T, u, U,
            Console.WriteLine("-----");
            Console.WriteLine("d Formats");
            Console.WriteLine("-----");
            string[] DateFormat = FormatOfDate.GetDateTimeFormats('d');
            foreach (string format in DateFormat)
            {
                Console.WriteLine(format);
            }
            Console.WriteLine("-----");
            Console.WriteLine("D Formats");
            Console.WriteLine("-----");
            DateFormat = FormatOfDate.GetDateTimeFormats('D');
            foreach (string format in DateFormat)
            {
                Console.WriteLine(format);
            }
            Console.WriteLine("-----");
            Console.WriteLine("f Formats");
            Console.WriteLine("-----");
            DateFormat = FormatOfDate.GetDateTimeFormats('f');
            foreach (string format in DateFormat)
            {
                Console.WriteLine(format);
            }
            Console.WriteLine("-----");
            Console.WriteLine("F Formats");
            Console.WriteLine("-----");
            DateFormat = FormatOfDate.GetDateTimeFormats('F');
            foreach (string format in DateFormat)
            {
                Console.WriteLine(format);
            }
        }
    }
}
```

Output:

```
-----  
d Formats  
-----  
25-02-2020  
25-02-20  
25-2-20  
25.2.20  
2020-02-25  
-----  
D Formats  
-----  
25 February 2020  
25 February 2020  
Tuesday, 25 February, 2020  
-----  
f Formats  
-----  
25 February 2020 00:00  
25 February 2020 0:00  
25 February 2020 12:00 AM  
25 February 2020 00:00  
25 February 2020 0:00  
25 February 2020 12:00 AM  
Tuesday, 25 February, 2020 00:00  
Tuesday, 25 February, 2020 0:00  
Tuesday, 25 February, 2020 12:00 AM
```

```
Tuesday, 25 February, 2020 0:00  
Tuesday, 25 February, 2020 12:00 AM  
-----  
F Formats  
-----  
25 February 2020 00:00:00  
25 February 2020 0:00:00  
25 February 2020 12.00.00 AM  
25 February 2020 12:00:00 AM  
25 February 2020 00:00:00  
25 February 2020 0:00:00  
25 February 2020 12.00.00 AM  
25 February 2020 12:00:00 AM  
Tuesday, 25 February, 2020 00:00:00  
Tuesday, 25 February, 2020 0:00:00  
Tuesday, 25 February, 2020 12.00.00 AM  
Tuesday, 25 February, 2020 12:00:00 AM  
Press any key to continue . . .
```

We can also do the formatting of the DateTime by passing the format specifier in the ToString() method of DateTime. Now we will write the C# code for the formatting of the DateTime using the ToString() method.

```
Console.WriteLine(DateOfFormat.ToString("r"));
```

Get the Leap year and Daylight-Saving time

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp8
{
    class Program
    {
        static void Main(string[] args)
        {
            DateTime DateOfTime = new DateTime(2020, 02, 22);
            Console.WriteLine(DateOfTime.IsDaylightSavingTime());
            Console.WriteLine(DateOfTime.IsLeapYear(DateOfTime.Year));

        }
    }
}
```

Output:

```
False
True
Press any key to continue . . .
```