برنامه نویسی پیشرفته C#

۲۱ مهر ۹۸

ملکی مجد

topics

- String.Split Method
- File (Section 3.6 C# Programming Yellow Book)
  - Stream reading and writing
- Struct and enum (Chapter 9 Microsoft Visual C# Step By Step)

String.Split Method

- is used to break a delimited string into substrings.
- Namespace : System

- Overloads
  - Table in next slide

Split(Char[], Int32, StringSplitOptions)	Splits a string into a maximum number of substrings based on the characters in an array.
Split(String[], Int32, StringSplitOptions)	Splits a string into a maximum number of substrings based on the strings in an array. You can specify whether the substrings include empty array elements.
Split(String[], StringSplitOptions)	Splits a string into substrings based on the strings in an array. You can specify whether the substrings include empty array elements.
Split(Char[], StringSplitOptions)	Splits a string into substrings based on the characters in an array. You can specify whether the substrings include empty array elements.
Split(Char[], Int32)	Splits a string into a maximum number of substrings based on the characters in an array. You also specify the maximum number of substrings to return.
Split(Char[])	Splits a string into substrings that are based on the characters in an array.

Reading multiple numbers from single input line

```
int size = int.Parse(Console.ReadLine());
string input= Console.ReadLine();
string[] inputs = input.Split(' ','\t');
int[] numbers=new int[Math.Max(inputs.Length,size)];
int index = 0; ;
foreach (string str in inputs)
{
    numbers[index++] = int.Parse(str);
}
```

File

- Need
  - a way of storing data when it is not running
- Files are looked after by the operating system of the computer.
  - Use C# to tell the operating system to create files and let us access them
- C# makes use of a thing called a stream to allow programs to work with files

Stream

- A stream is a link between your program and a data resource
- Data can flow up or down your stream, so that streams can be used to read and write to files
- A C# program can contain an object representing a particular stream that a programmer has created and connected to a file
- C# has a range of different stream types which you use depending on what you want to do. All of the streams are used in exactly the same way

Console as stream

- you are already familiar with how streams are used, since the Console class, which connects a C# program to the user, is implemented as a stream.

- The ReadLine and WriteLine methods are commands you can give any stream that will ask it to read and write data.

StreamWriter and StreamReader

- two stream types which let programs use files

Create an output stream

- create a stream object just like you would create any other one
- by using new. When the stream is created it can be passed the name of the file that is to be opened

```
StreamWriter writer ;
writer = new StreamWriter("test.txt");
```

Create an output stream(2)

```
StreamWriter writer ;
writer = new StreamWriter("test.txt");
```

- variable writer refers to the stream that you want to write into
- When the new StreamWriter is created
- the program will open a file called test.txt for output
- and connect the stream to it

- If the action fail
  - Throw an appropriate exception

Create an output stream(3)

```
StreamWriter writer ;
writer = new StreamWriter("test.txt");
```

- Note that this code does not have a problem if the file test.txt already exists.
- a brand new, empty, file is created in place of what was there.
- potentially dangerous
  - destroy the contents of an existing file

Writing to a Stream

- Once the stream has been created it can be written to by calling the write methods it provides

```
writer.WriteLine("hello world");
```

- Each time you write a line to the file it is added onto the end of the lines that have already been written

Writing to a Stream (1)

- If your program got stuck writing in an infinite loop it is possible that it might fill up the storage device. If this happens, and the write cannot be performed successfully, the call of WriteLine will throw an exception

- A properly written program should probably make sure that any exceptions like this (they can also be thrown when you open a file) are caught and handled correctly.

Closing a Stream

- When your program has finished writing to a stream it is very important that the stream is explicitly closed using the Close method:

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

- When the Close method is called the stream will write out any text to the file that is waiting to be written and disconnect the program from the file
- Any further attempts to write to the stream will fail with an exception

After closing

- Once a file has been closed it can then be accessed by other programs on the computer,
  - use the Notepad program to open test.txt and take a look at what is inside it.

- Forgot to close:
  - some of the data that you wrote into the file will not be there.
  - If your program has a stream connected to a file other programs may not be able to use that file (impossible to move or rename the file).
  - An open stream consumes a small, but significant, part of operating resource (creates lots of streams but does not close them this might lead to problems opening other files later on)

Streams and Namespaces

- this object is defined in the System.IO *namespace*
  - using System.IO;

- System.Console.WriteLine("Hello World");
- using System;
  - Console.WriteLine("Hello World");

Sample code 20

```
using System;
using System.IO;

class FileWriteDemo
{

    public static void Main()
    {
        StreamWriter writer;
        writer = new StreamWriter("test.txt");
        writer.WriteLine("hello world");
        writer.Close();
    }
}
```

Reading from a File

```
StreamReader reader = new StreamReader("Test.txt");
string line = reader.ReadLine();
Console.WriteLine (line);
reader.Close();
```

Detecting the End of an Input File

- Repeated calls of ReadLine will return successive lines of a file.
  - reaches the end of the file the ReadLine method will return an empty string each time it is called

- property EndOfStream
  - determine when the end of the file has been reached

Using EndOfStream

```
StreamReader reader = new StreamReader("Test.txt");
while (reader.EndOfStream == false)
{
    string line = reader.ReadLine();
    Console.WriteLine(line);
}
reader.Close();
```

```
using System;
using System.IO;
class FileWriteandReadDemo
{
    public static void Main()
    {
        StreamWriter writer;
        writer = new StreamWriter("test.txt");
        writer.WriteLine("hello world");
        writer.Close();

        StreamReader reader = new StreamReader("Test.txt");
        while (reader.EndOfStream == false)
        {
            string line = reader.ReadLine();
            Console.WriteLine(line);
        }
        reader.Close();
    }
}
```

File Paths in C#

- The location of a file on a computer is often called the *path* to the file.
- The path to a file can be broken into two parts, the location of the folder and the name of the file itself.

- If you don't give a folder location when you open a file (as we have been doing with the file Test.txt) then the system assumes the file that is being used is stored in the same folder as the program which is running.

File Paths in C# (2)

- If you want to use a file in a different folder (which is a good idea, as data files are hardly ever held in the same place as programs run from) you can add path information to a filename:

```
string path;
path = @"c:\data\2009\November\sales.txt";
```

struct and enum

- *types*
  - *value types* and *reference types*

- you'll learn how to create your own value types

Creating value types with enumerations and structures

- Declare an enumeration type.
- Create and use an enumeration type.
- Declare a structure type.
- Create and use a structure type.
- Explain the differences in behavior between a structure and a class.

enum

- Suppose that you want to represent the seasons of the year in a program
  - could use the integers 0, 1, 2, and 3 to represent spring, summer, fall, and winter, respectively
  - it wouldn't be obvious that a particular 0 represented spring.
  - there is nothing to stop you from assigning it any legal integer value outside the set 0, 1, 2, or 3

- C# offers a better solution. You can
  - create an enumeration whose values are limited to a set of symbolic names.

Declaring and using an enumeration

- After you have declared an enumeration, you can use it in exactly the same way you do any other type

```
enum Season { Spring, Summer, Fall, Winter }

class Example
{
    public void Method(Season parameter) // method parameter example
    {
        Season localVariable; // local variable example
        ...
    }

    private Season currentSeason; // field example
}
```

enum value

- Before you can read the value of an enumeration variable,
  - it must be assigned a value.
  - You can assign a value that is defined by the enumeration only to an enumeration variable

```
Season colorful = Season.Fall;
                  //you have to write Season.Fall rather than just Fall
Console.WriteLine(colorful); // writes out 'Fall'
```

Nullable version

- you can create a nullable version of an enumeration variable by using the *?* modifier.
- You can then assign the *null* value, as well as the values defined by the enumeration, to the variable

- Season? colorful = null;

.tostring()

```
Season colorful = Season.Fall;
string name = colorful.ToString();
Console.WriteLine(name); // also writes out 'Fall'
```

enumeration literal values

- an enumeration type associates an integer value with each element of the enumeration
  - By default, the numbering starts at 0 for the first element and goes up in steps of 1

- Casting
  - enum Season { Spring, Summer, Fall, Winter }
  - ...
  - Season colorful = Season.Fall;
  - Console.WriteLine((int)colorful); // writes out '2'

enumeration literal values(2)

- you can associate a specific integer constant (compile-time constant value such as 1) with an enumeration literal
  - enum Season { Spring = 1, Summer, Fall, Winter }

- give more than one enumeration literal the same underlying value
  - enum Season { Spring, Summer, Fall, Autumn = Fall, Winter }

struct

- classes define reference types that are always created on the heap
  - In some cases, the class can contain so little data that the overhead of managing the heap becomes disproportionate.

- A structure is a value type.
  - Because structures are stored on the stack,
  - as long as the structure is reasonably small, the memory management overhead is often reduced

- Like a class, a structure can have its own fields, methods, and (with one important exception) constructors.

Common structure types

- In C#, the primitive numeric types *int, long*, and *float* are aliases for the structures *System.Int32, System.Int64*, and *System.Single*, respectively.
  - These structures have fields and methods, and you can actually call methods on variables and literals of these types.

- For example, all these structures provide a *ToString* method

```
int i = 55;
Console.WriteLine(i.ToString());
Console.WriteLine(55.ToString());
float f = 98.765F;
Console.WriteLine(f.ToString());
Console.WriteLine(98.765F.ToString());
```

Common structure types(2)

- the static *int.Parse* method
  - are actually doing is invoking the *Parse* method of the *Int32* structure

```
string s = "42";
int i = int.Parse(s); // exactly the same as Int32.Parse
```

- These structures also include some useful static fields
  - *Int32.MaxValue* is the maximum value that an *int* can hold, and *Int32.MinValue* is the minimum value that you can store in an *int*.

Declaring a structure

```
struct Time
{
    public int hours, minutes, seconds;
}
```

Syntactically, the process is similar to declaring a class.

```
struct Time
{
    private int hours, minutes, seconds;
    ...
    public Time(int hh, int mm, int ss)
    {
        this.hours = hh % 24;
        this.minutes = mm % 60;
        this.seconds = ss % 60;
    }

    public int Hours()
    {
        return this.hours;
    }
 }
```

Understanding differences between structures and classes

- You can't declare a default constructor (a constructor with no parameters) for a structure.

The following example would compile if *Time* were a class, but because *Time* is a structure it does not:

```
struct Time
{
        public Time() { ... } // compile-time error
        ...
}
```

- can initialize fields to different values
  - your nondefault constructor must explicitly initialize all fields
    - the default initialization no longer occurs.

```
struct Time
{
    private int hours, minutes, seconds;
    ...
    public Time(int hh, int mm)
    {
        this.hours = hh;
        this.minutes = mm;
    }   // compile-time error: seconds not initialized
}
```

Understanding differences between structures and classes(2)

- In a class, you can initialize instance fields at their point of declaration. In a structure, you cannot.

```
struct Time
{
        private int hours = 0; // compile-time error
        private int minutes;
        private int seconds;
        ...
}
```

- compile if *Time* were a class, but it causes a compile-time error because *Time* is a structure
- (other differences are in inheritance topics!)

Declaring structure variables

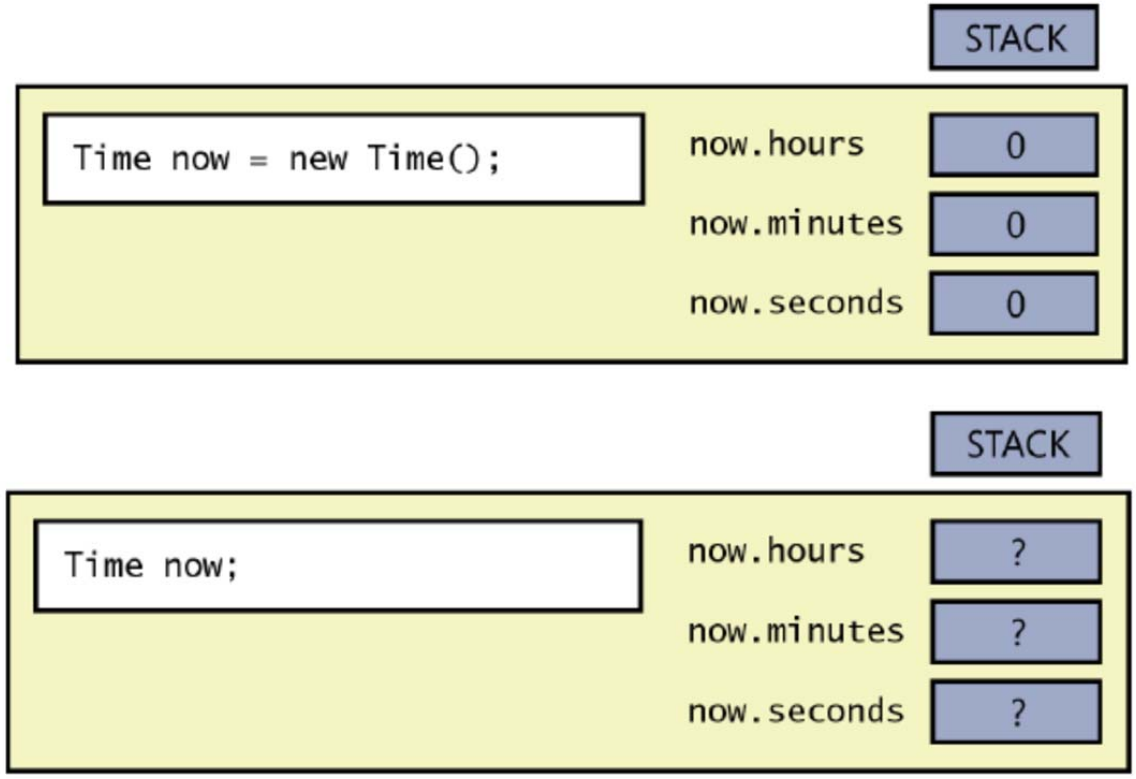
```
struct Time
{
    private int hours, minutes, seconds;
    ...
}

class Example
{
    private Time currentTime;

    public void Method(Time parameter)
    {
        Time localVariable;
        ...
    }
}
```

nullable

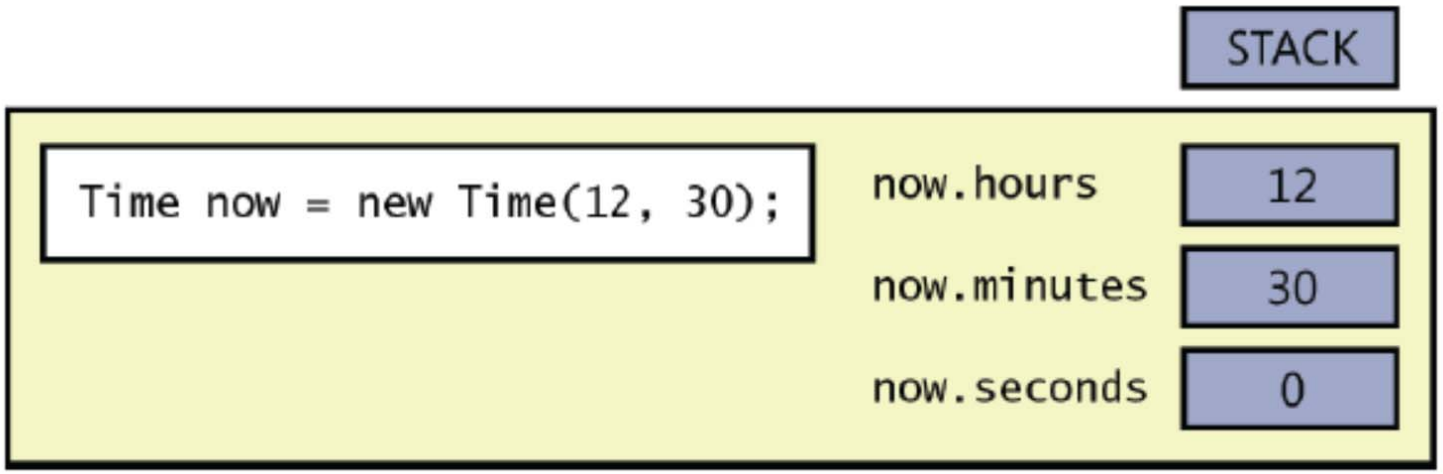
- As with enumerations, you can create a nullable version of a structure variable by using the *?* modifier. You can then assign the *null* value to the variable:

```
Time? currentTime = null;
```

Understanding structure initialization



```
struct Time
{
    private int hours, minutes, seconds;
    ...

    public Time(int hh, int mm)
    {
        hours = hh;
        minutes = mm;
        seconds = 0;
    } }
```



Copying structure variables

- You're allowed to initialize or assign one structure variable to another structure variable
  - only if the structure variable on the right side is completely initialized

- The following example fails to compile because *now* is not initialized

```
Date now;
Date copy = now; // compile-time error: now has not been assigned
```

Copying structure variables

```
Date now = new Date(2012, Month.March, 19);
Date copy = now;
```