

# **Chương 2:**

# **NGÔN NGỮ**

# **C#**

# Giới thiệu

- ▶ **C# (C-Sharp)** là ngôn ngữ lập trình hiện đại, hướng đối tượng do **Microsoft** phát triển.
- ▶ Xuất hiện lần đầu năm **2000**, cùng với nền tảng **.NET Framework**.
- ▶ Người tạo chính: **Anders Hejlsberg** (cũng là tác giả của Turbo Pascal và kiến trúc sư chính của Delphi).
- ▶ C# được thiết kế để vừa mạnh mẽ như C/C++ vừa dễ sử dụng như Java.

# Đặc điểm nổi bật

- ▶ **Hướng đối tượng (OOP)**: hỗ trợ kế thừa, đa hình, đóng gói.
- ▶ **Type-safe**: hạn chế lỗi truy cập bộ nhớ không an toàn như C/C++.
- ▶ **Tích hợp tốt với .NET**: có thư viện phong phú (I/O, Networking, GUI, Database...).
- ▶ **Đa nền tảng**: ban đầu chạy trên Windows, nhưng từ khi có **.NET Core** và nay là **.NET 8**, C# có thể chạy trên Windows, Linux, macOS.
- ▶ **Hỗ trợ lập trình hiện đại**:
  - Generics
  - LINQ (Language Integrated Query)
  - Async/Await (lập trình bất đồng bộ)
  - Lambda expression, delegates, events
  - Pattern matching

# Đánh giá hiện nay

- ▶ C# hiện là ngôn ngữ chính trên **.NET 8/9** (2023–2024)
- ▶ Trình biên dịch **C#** là một trong những trình biên dịch hiệu quả nhất trong dòng sản phẩm .NET.
- ▶ Theo **TIOBE Index (2025)**: C# thường nằm trong **top 5 ngôn ngữ phổ biến nhất**.
- ▶ Được cộng đồng hỗ trợ mạnh mẽ nhờ Microsoft, GitHub, Stack Overflow.

# Ứng dụng thực tế

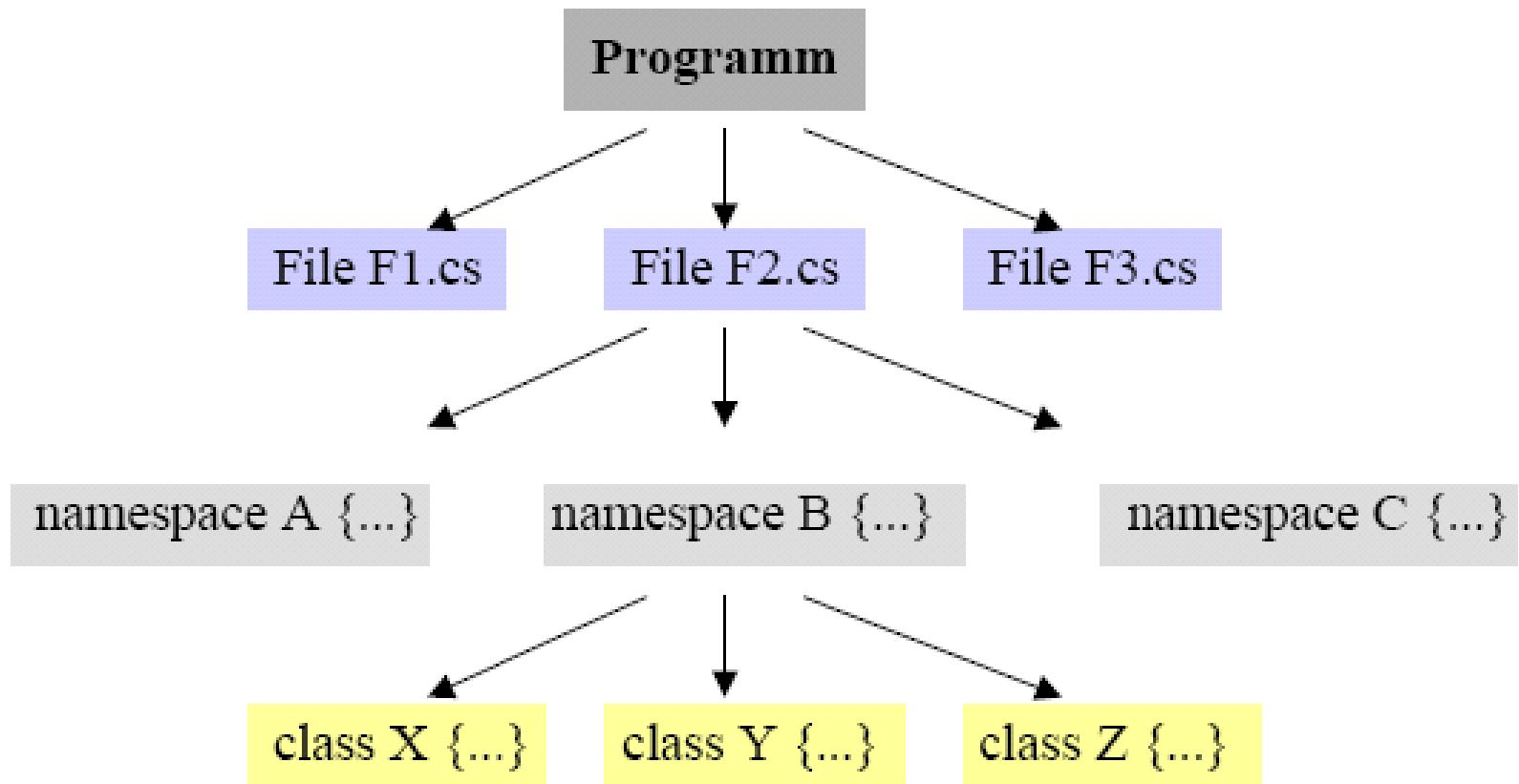
C# hiện được dùng trong nhiều lĩnh vực:

- ▶ **Ứng dụng Desktop**
  - Windows Forms, WPF, UWP
- ▶ **Web Application**
  - ASP.NET / ASP.NET Core
- ▶ **Mobile Application**
  - Xamarin, .NET MAUI
- ▶ **Game Development**
  - Unity (engine phổ biến nhất hiện nay)
- ▶ **Cloud & Microservices**
  - Azure, Docker, Kubernetes với .NET Core
- ▶ **AI & Machine Learning**
  - ML.NET

# Đặc điểm của ngôn ngữ C#

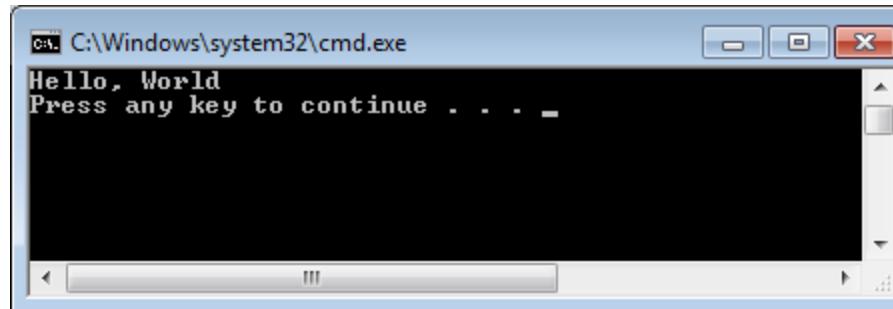
- ▶ Khoảng 80 từ khóa
- ▶ Hỗ trợ lập trình cấu trúc, lập trình hướng đối tượng, hướng thành phần (Component oriented)
- ▶ Có từ khóa khai báo dành cho thuộc tính (property)
- ▶ Cho phép tạo dữ liệu trực tiếp bên trong mã nguồn (dùng tool mã nguồn mở [NDoc](#) phát sinh ra dữ liệu)
- ▶ Hỗ trợ khái niệm interface (tương tự java)
- ▶ Cơ chế tự động dọn rác (tương tự java)
- ▶ Truyền tham số kiểu: [out](#), [ref](#)

# Cấu trúc chương trình C#



# Hello World

```
01  using System;  
02  
03  class Hello  
04  {  
05      public static void Main()  
06      {  
07          Console.WriteLine("Hello, World");  
08      }  
09  }
```



# Namespace

- ▶ Namespace cung cấp cho cách tổ chức quan hệ giữa các lớp và các kiểu khác.
- ▶ Namespace là cách mà .NET tránh né việc các tên lớp, tên biến, tên hàm trùng tên giữa các lớp.

namespace CustomerPhoneBookApp

{

    using System;

    public struct Subscriber

    { // Code for struct here... }

}

# Namespace

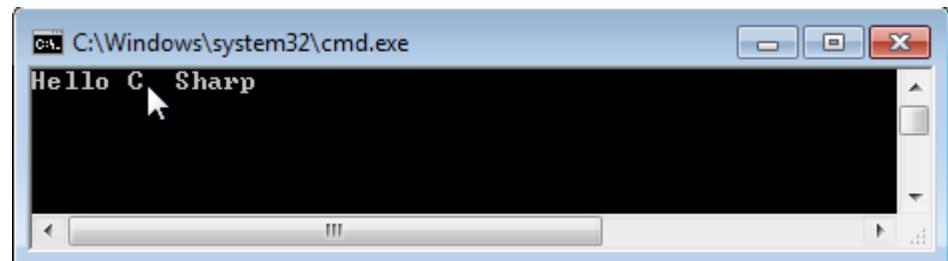
- ▶ Từ khoá using giúp giảm việc phải gõ những namespace trước các hàm hành vi hoặc thuộc tính
  - using Wrox.ProCSharp;
- ▶ Ta có thể gán bí danh cho namespace
  - Cú pháp :  
using *alias* = *NamespaceName*;

# Ví dụ

```
01  /* Chương trình cơ bản của C#*/
02
03  class Hello
04  {
05      static void Main(string[] args)
06      {
07          System.Console.WriteLine("Hello C Sharp");
08          System.Console.ReadLine();
09      }
10 }
```

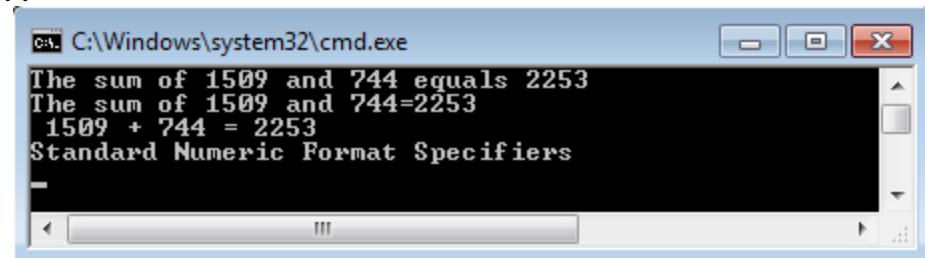
Để biên dịch từng Class, có thể sử dụng tập tin csc.exe trong cửa sổ Command Prompt với khai báo như sau:

D:\csc CSharp\ Hello.cs



# Console.WriteLine

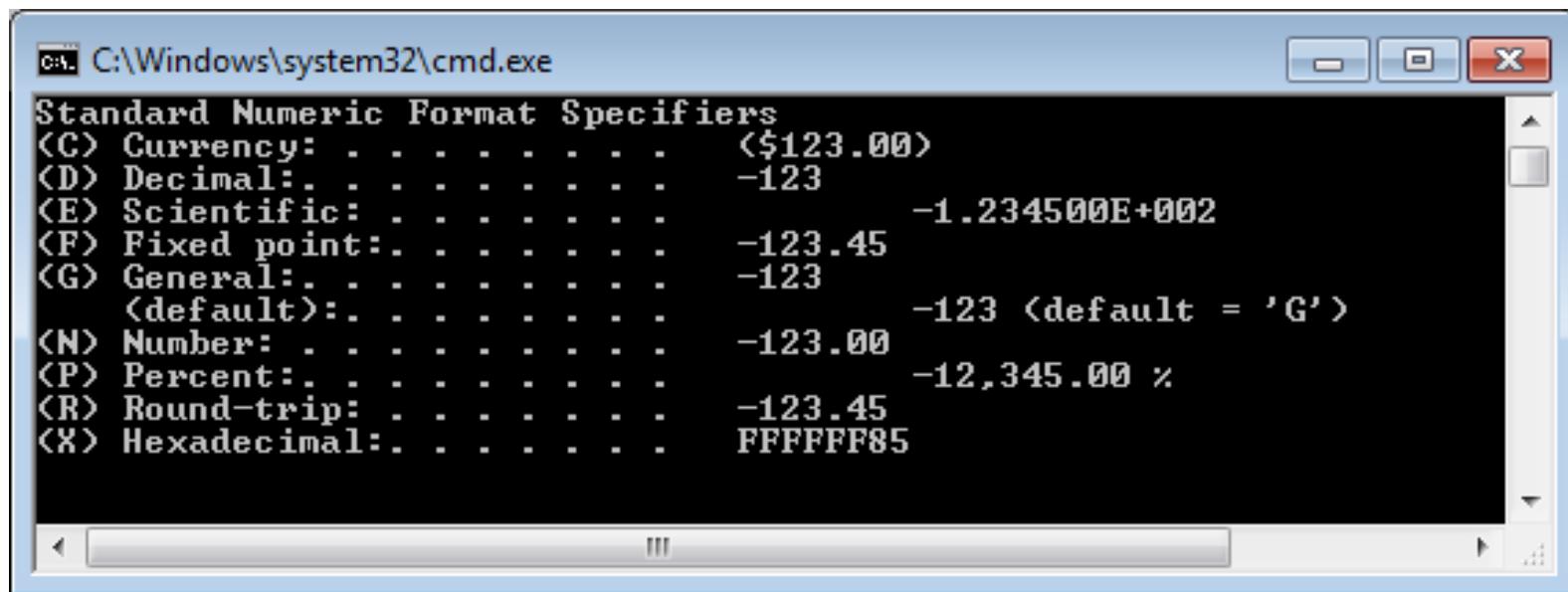
```
public static void Main() {  
    int a = 1509; int b = 744; int c = a + b;  
    Console.Write("The sum of ");  
    Console.Write(a);  
    Console.Write(" and ");  
    Console.Write(b);  
    Console.Write(" equals ");  
    Console.WriteLine(c);  
    Console.WriteLine("The sum of " + a + " and  
" + b + "=" + c);  
    Console.WriteLine("{0} + {1} = {2}", a, b,  
c);  
    Console.ReadLine();  
}
```



# Console.WriteLine

```
Console.WriteLine("Standard Numeric Format  
Specifiers");  
Console.WriteLine(  
    " (C) Currency: . . . . . . . . . . {0:C}\n" +  
    " (D) Decimal: . . . . . . . . . . {0:D}\n" +  
    " (E) Scientific: . . . . . . . . . . {1:E}\n" +  
    " (F) Fixed point: . . . . . . . . . . {1:F}\n" +  
    " (G) General: . . . . . . . . . . {0:G}\n" +  
    "     (default): . . . . . . . . . . {0}  
(default = 'G')\n" +  
    " (N) Number: . . . . . . . . . . {0:N}\n" +  
    " (P) Percent: . . . . . . . . . . {1:P}\n" +  
    " (R) Round-trip: . . . . . . . . . . {1:R}\n" +  
    " (X) Hexadecimal: . . . . . . . . . . {0:X}\n",  
    -123, -123.45f);
```

# Console.WriteLine



The screenshot shows a Windows Command Prompt window titled "cmd C:\Windows\system32\cmd.exe". The window displays a list of standard numeric format specifiers, each followed by an example value:

Specifier	Value
(C) Currency: . . . . .	<\$123.00>
(D) Decimal: . . . . .	-123
(E) Scientific: . . . . .	-1.234500E+002
(F) Fixed point: . . . . .	-123.45
(G) General: . . . . .	-123
(default): . . . . .	-123 <default = 'G'>
(N) Number: . . . . .	-123.00
(P) Percent: . . . . .	-12,345.00 %
(R) Round-trip: . . . . .	-123.45
(X) Hexadecimal: . . . . .	FFFFFFFFFF85

# Console.WriteLine

```
Console.WriteLine("Standard DateTime Format Specifiers");
Console.WriteLine(
    "(d) Short date: . . . . .          {0:d}\n" +
    "(D) Long date:.. . . . . . . . . {0:D}\n" +
    "(t) Short time: . . . . . . . . {0:t}\n" +
    "(T) Long time:.. . . . . . . . {0:T}\n" +
    "(f) Full date/short time: . . {0:f}\n" +
    "(F) Full date/long time:.. . . {0:F}\n" +
    "(g) General date/short time:.. {0:g}\n" +
    "(G) General date/long time:.. {0:G}\n" +
    "      (default):. . . . . . . . {0} (default
= 'G')\n" +
    "(M) Month:.. . . . . . . . . {0:M}\n" +
    "(R) RFC1123:.. . . . . . . . . {0:R}\n" +
    "(s) Sortable: . . . . . . . . {0:s}\n" +
    "(u) Universal sortable: . . . {0:u} (invariant)\n"
+
    "(U) Universal sortable: . . . {0:U}\n" +
    "(Y) Year: . . . . . . . . . {0:Y}\n",
thisDate);
```

# Console.WriteLine

```
C:\Windows\system32\cmd.exe
Standard DateTime Format Specifiers
<d> Short date: . . . . .           11/20/2011
<D> Long date: . . . . .           Sunday, November 20, 2011
<t> Short time: . . . . .           5:54 PM
<T> Long time: . . . . .           5:54:04 PM
<f> Full date/short time: . . .   Sunday, November 20, 2011 5:54 PM
<F> Full date/long time: . . . .  Sunday, November 20, 2011 5:54:04 PM
<g> General date/short time: .   11/20/2011 5:54 PM
<G> General date/long time: . .  11/20/2011 5:54:04 PM
    <default>: . . . . .           11/20/2011 5:54:04 PM <default = 'G'>
<M> Month: . . . . .             November 20
<R> RFC1123: . . . . .           Sun, 20 Nov 2011 17:54:04 GMT
<s> Sortable: . . . . .           2011-11-20T17:54:04
<u> Universal sortable: . . . .  2011-11-20 17:54:04Z <invariant>
<U> Universal sortable: . . . .  Sunday, November 20, 2011 10:54:04 AM
<Y> Year: . . . . .             November, 2011
```

# Console.ReadLine()

```
public static string ReadLine ()
```

- ↳ Convert.ToBoolean();

- ↳ Convert.ToByte();

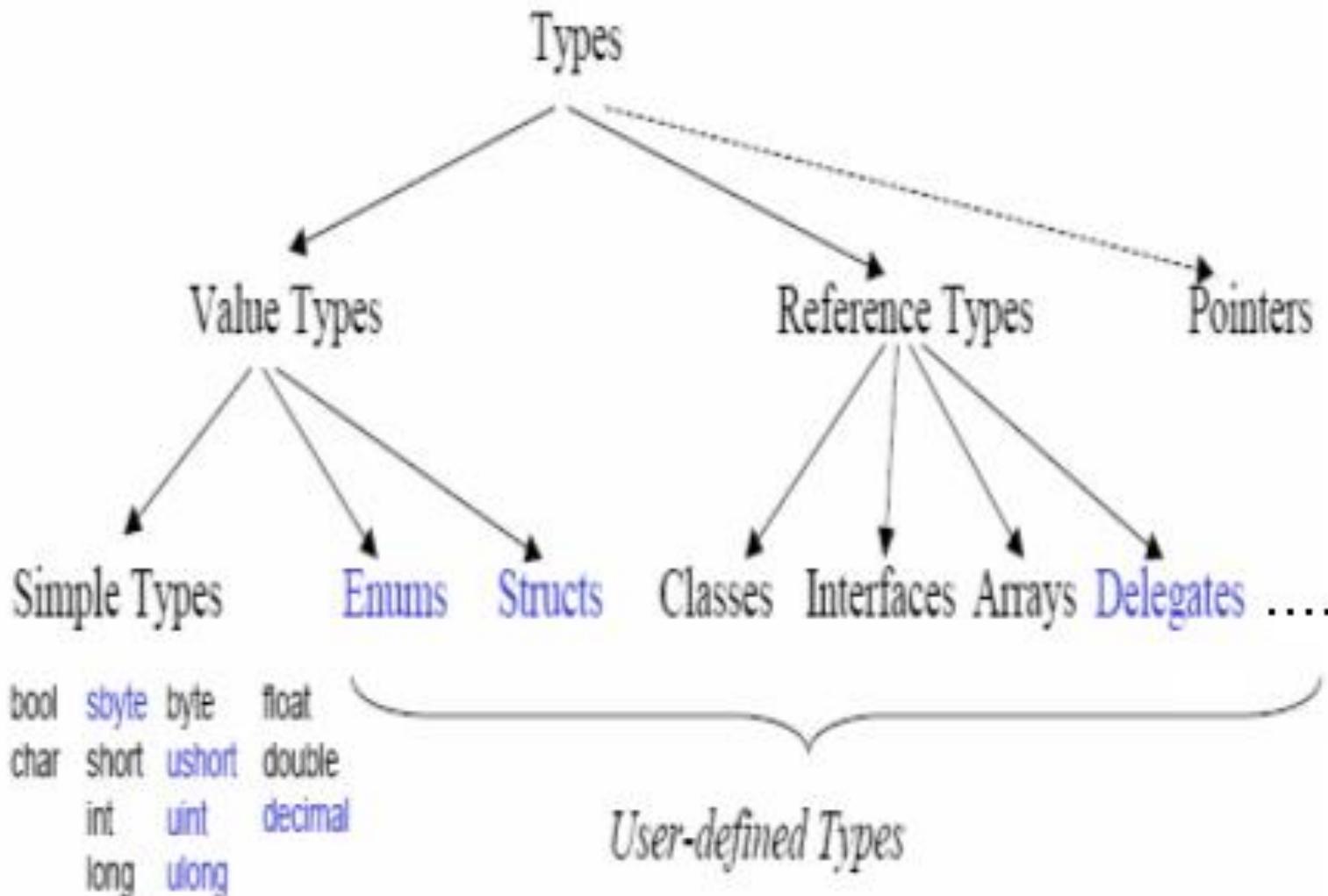
- ↳ Convert.ToInt16();

- ↳ Byte.Parse();

- ↳ Int64.Parse();

- ↳ Double.Parse()

# Kiểu dữ liệu trong C#



# Kiểu dữ liệu định sẵn

Kiểu C#	Số byte	Kiểu .NET	Mô tả
byte	1	Byte	Số nguyên dương không dấu từ 0-255
char	2	Char	Kí tự Unicode
bool	1	Boolean	Giá trị logic true/ false
sbyte	1	Sbyte	Số nguyên có dấu (từ -128 đến 127)
short	2	Int16	Số nguyên có dấu giá trị từ -32768 đến 32767
ushort	2	UInt16	Số nguyên không dấu 0 – 65.535

# Kiểu dữ liệu định sẵn

Kiểu C#	Số byte	Kiểu .NET	Mô tả
int	4	Int32	Số nguyên có dấu - 2.147.483.647 đến 2.147.483.647
uint	4	Uint32	Số nguyên không dấu 0 – 4.294.967.295
float	4	Single	Kiểu dấu chấm động, 3,4E-38 đến 3,4E+38, với 7 chữ số có nghĩa..
double	8	Double	Kiểu dấu chấm động có độ chính xác gấp đôi 1,7E-308 đến 1,7E+308 với 15,16 chữ số có nghĩa.

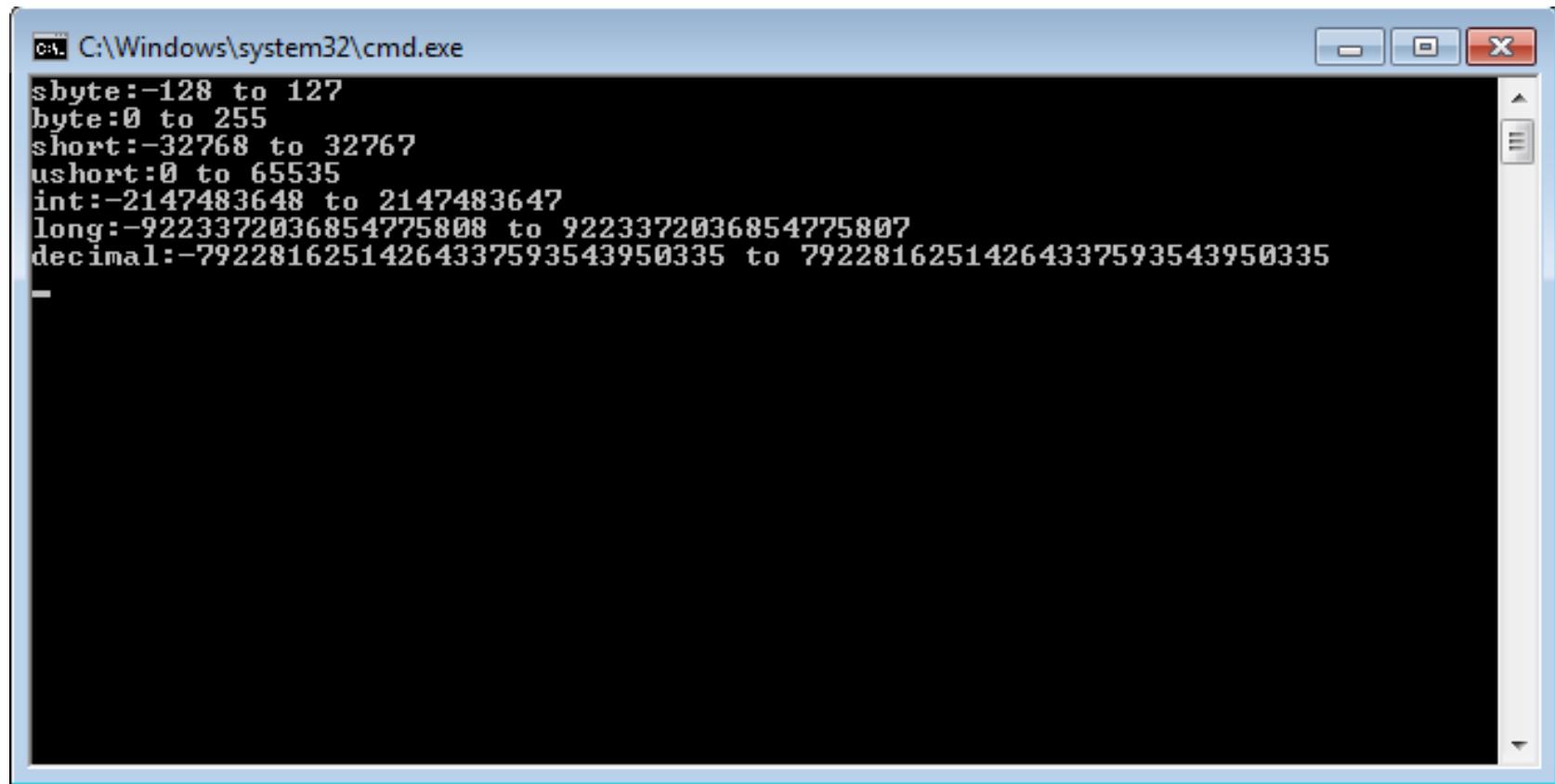
# Kiểu dữ liệu định sẵn

Kiểu C#	Số byte	Kiểu .NET	Mô tả
decimal	8	Decimal	Có độ chính xác đến 28 con số dùng trong tính toán tài chính phải có hậu tố “m” hay “M” theo sau giá trị
long	8	Int64	Kiểu số nguyên có dấu -9.223.370.036.854.775.808 đến 9.223.372.036.854.775.807
ulong	8	Uint64	Số nguyên không dấu từ 0 đến 0xffffffffffffffff

# Kiểu dữ liệu định sẵn

- ← Console.WriteLine("sbyte:{0} to {1} ",sbyte.MinValue,sbyte.MaxValue);
- ← Console.WriteLine("byte:{0} to {1}", byte.MinValue, byte.MaxValue);
- ← Console.WriteLine("short:{0} to {1}", short.MinValue, short.MaxValue);
- ← Console.WriteLine("ushort:{0} to {1}", ushort.MinValue,  
                  ushort.MaxValue);
- ← Console.WriteLine("int:{0} to {1}", int.MinValue, int.MaxValue);
- ← Console.WriteLine("long:{0} to {1}", long.MinValue, long.MaxValue);
- ← Console.WriteLine("decimal:{0} to {1}", decimal.MinValue,  
                  decimal.MaxValue);
- ← Console.ReadLine();

# Kiểu dữ liệu định sẵn



A screenshot of a Windows Command Prompt window titled "C:\Windows\system32\cmd.exe". The window displays the following text:

```
sbyte:-128 to 127
byte:0 to 255
short:-32768 to 32767
ushort:0 to 65535
int:-2147483648 to 2147483647
long:-9223372036854775808 to 9223372036854775807
decimal:-79228162514264337593543950335 to 79228162514264337593543950335
```

# Chuyển đổi kiểu dữ liệu

- ← Chuyển đổi dữ liệu là cho phép một biểu thức của kiểu dữ liệu này được xem xét như một kiểu dữ liệu khác.
- ← Chuyển đổi có thể: ẩn – ngầm định (**implicit**) hay tường minh (**explicit**),
- ← ví dụ,

```
int a = 123;  
long b = a;  
// từ int sang long (implicit)  
int c = (int) b;  
// từ long sang int (explicit)
```

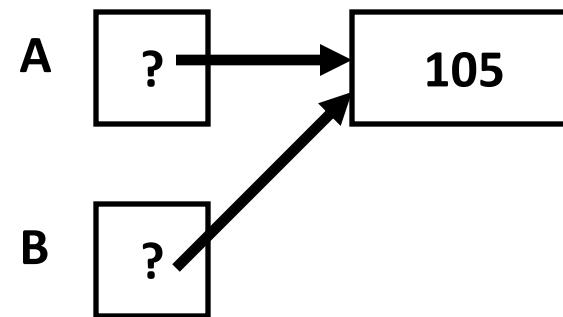
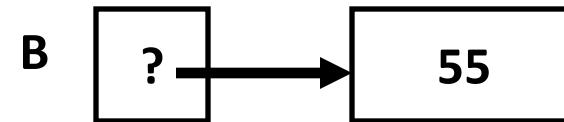
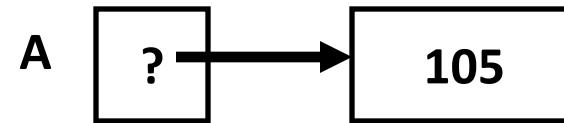
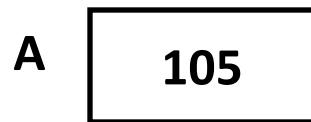
# Enum(eration) - kiểu tập hợp

```
enum Days {Sat, Sun, Mon, Tue, Wed, Thu, Fri};  
...  
Days d = Days.Mon;  
...  
switch (d) {  
    case Days.Tue: ...  
    case Days.Wed: ...  
}
```

**Rõ hơn cách dùng hằng truyền thống của C**

```
const int Sat = 1;  
...  
const int Fri = 6;
```

# Value type vs reference type



# struct

- struct : value type (class : reference type)
- Dùng để cho các đối tượng “nhỏ” như Point, Rectangle, Color,...

```
public struct MyPoint {  
    public int x, y;  
    public MyPoint(int p1, int p2) {  
        x = p1;  
        y = p2;  
    }  
}
```

# Box và Unbox

- Đổi qua lại giữa value type và reference type.
- Box: value => reference (object).
- Thường dùng trong các hàm, cấu trúc dữ liệu sử dụng tham số là kiểu **object** tổng quát.

```
int i = 123;  
object o = i; // implicit boxing  
object o = (object) i; // explicit boxing  
int j = (int) o; // unboxing
```

# Box và Unbox

On the stack

i



```
int i=123;
```

On the heap

o



```
object o=i;
```

(i boxed)



j



```
int j=(int) o;
```

# Các nhóm toán tử trong C#

Nhóm toán tử	Toán tử
Toán học	+ - * / %
Logic	&   ^ ! ~ &&    true false
Ghép chuỗi	+
Tăng, giảm	++, --
Dịch bit	<< >>
Quan hệ	== != < > <= >=
Gán	= += -= *= /= %= &=  = ^= <<= >>=
Chỉ số	[ ]
Ép kiểu	( )
Indirection và Address	* -> [ ] &

# Thứ tự ưu tiên của toán tử

Nhóm toán tử	Toán tử
Primary	{x} x.y f(x) a[x] x++ x--
Unary	+ - ! ~ ++x -x (T)x
Nhân	* / %
Cộng	+ -
Dịch bit	<< >>
Quan hệ	< > <= >= is
Bằng	== !=
Logic trên bit AND	&
XOR	^
OR	
Điều kiện AND	&&
Điều kiện OR	
Điều kiện	?:
Assignment	= *= /= %= += -= <<= >>= &= ^=  =

# Kiểu mảng

- ▶ 1 mảng là 1 tập các điểm dữ liệu (của cùng kiểu cơ sở), được truy cập dùng 1 số chỉ mục
- ▶ Các mảng trong C# phát sinh từ lớp cơ sở **System.Array**
- ▶ Mảng có thể chứa bất cứ kiểu nào mà C# định nghĩa, bao gồm các mảng đối tượng, các giao diện, hoặc các cấu trúc
- ▶ Mảng có thể 1 chiều hay nhiều chiều, và được khai báo bằng dấu ngoặc vuông ( [ ] ) đặt sau kiểu dữ liệu của mảng
- ▶ VD: **int [] a;**

# Kiểu mảng

**Khai báo biến mảng có hai cách như sau**

1) Khai báo và khởi tạo mảng

```
int[] yourarr = new int[ptu];
```

2) Khai báo sau đó khởi tạo mảng

```
int[] myarr;  
myarr=new int[ptu];
```

Khai báo mảng với số phần tử cho trước và khởi tạo giá trị cho các phần tử của mảng:

```
int[] me={1,2,3,4,5};
```

```
float[] arr = { 3.14f, 2.17f, 100 };
```

```
float[] arr = new float [3] { 3.14f, 2.17f, 100 };
```

# Kiểu mảng

arr.length: số phần tử của mảng

Khai báo mảng 2 chiều:

```
int[,] Mang2chieu;
```

```
Mang2chieu = new int[3,4]
```

Khai báo mảng của mảng:

```
int [][] M=new int[2][];
```

```
M[0]=new int[4];
```

```
M[1]= new int[30];
```

# Kiểu string

- ▶ Kiểu string là 1 kiểu dữ liệu tham chiếu trong C#
- ▶ **System.String** cung cấp các hàm tiện ích như:  
Concat(), CompareTo(), Copy(), Insert(), ToUpper(),  
ToLower(), Length, Replace(), ...
- ▶ Các toán tử == và != được định nghĩa để so sánh  
các giá trị của các đối tượng chuỗi, chứ không phải  
là bộ nhớ mà chúng tham chiếu đến
- ▶ Toán tử & là cách tốc ký thay cho Concat()
- ▶ Có thể truy cập các ký tự riêng lẻ của 1 chuỗi dùng  
toán tử chỉ mục ([ ])

# Kiểu pointer

- ▶ Kiểu pointer được khai báo với dấu \* ngay sau loại dữ liệu và trước tên biến cùng với từ khoá **unsafe**.
- ▶ Biên dịch ứng dụng C# có sử dụng kiểu dữ liệu pointer:

D:\csc pointer.cs **/unsafe**

# Kiểu pointer

- ▶ Không giống như hai kiểu dữ liệu value và reference, kiểu pointer không chịu sự kiểm soát của **garbage collector**
- ▶ Garbage collector không dùng cho kiểu dữ liệu này do chúng không biết dữ liệu mà con trỏ trỏ đến
- ▶ Vì vậy, pointer không cho phép tham chiếu đến **reference** hay một **struct** có chứa các kiểu references và kiểu tham chiếu của pointer thuộc loại kiểu không quản lý (**unmanaged-type**).

# Tham số

- ▶ **Tham trị:** tham số có giá trị không thay đổi trước và sau khi thực hiện phương thức
- ▶ **Tham biến:** tham số có giá trị thay đổi trước và sau khi thực hiện phương thức, có thể đi sau các từ khóa: ref, out, params
  - **ref:** tham số đi theo sau phải khởi tạo trước khi truyền vào phương thức
  - **out:** tham số không cần khởi tạo trước khi truyền vào phương thức
  - **params:** tham số nhận đối số mà số lượng đối số là biến, từ khoá này thường sử dụng tham số là mảng.

# Tùy Khoa ref

```
void MyMethod()
{
    int num1 = 7, num2 = 9;
    Swap(ref num1, ref num2);
    // num1 = 9, num2 = 7
}

void Swap(ref int x, ref int y)
{
    int temp = x; x = y; y = temp;
}
```

# Keyword out

```
void MyMethod()
{
    int num1 = 7, num2;
    Subtraction(num1, out num2);
    // num1 = 7, num2 = 5
}
void Subtraction(int x, out int y)
{
    y = x - 2;
    // y must be assigned a value
}
```

*uninitialised*

# Keyword params

```
void MyMethod()
{
    int sum = Addition(1, 2, 3); // sum = 6
}

int Addition(params int[] integers)
{
    int result = 0;
    for (int i = 0; i < integers.Length; i++)
        result += integers[i];
    return result;
}
```

# Keyword var

- ▶ Thay vì sử dụng khai báo biến thông thường ta có thể sử dụng từ khóa var để khai báo biến kiểu ngầm định.
- ▶ Một biến được khai báo bằng từ khóa này vẫn có kiểu cụ thể, chỉ khác là kiểu này trình biên dịch xác định từ giá trị được gán.

Ví dụ:

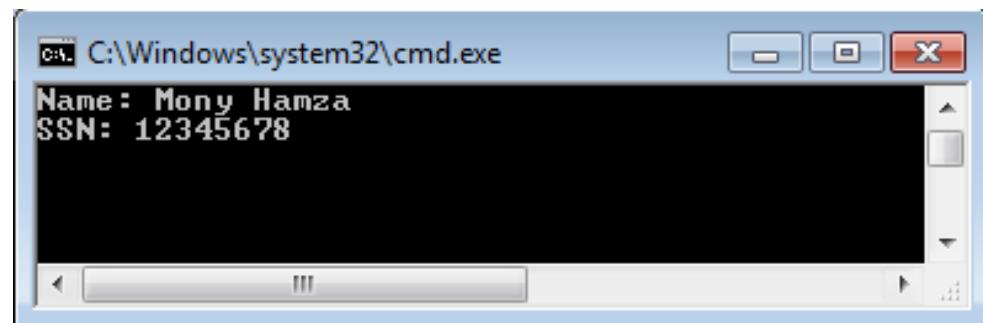
- `var int_variable = 6; // int_variable is compiled as an int`
- `var string_variable = "Mony"; // string_variable is compiled as a string`
- `var int_array = new[] { 0, 1, 2 }; // int_array is compiled as int[]`

# Anonymous Type

Tạo ra 1 kiểu mới trong lúc chạy (không tồn tại ở cấp độ source code). Kiểu mới bao gồm một tập những thuộc tính read-only.

Ví dụ:

```
var obj1 = new { Name = "Mony Hamza", SSN ="12345678" };
// Display the contents:
Console.WriteLine("Name: {0}\nSSN: {1}",
obj1.Name,obj1.SSN);
Console.ReadLine();
```



# Phát biểu chọn

Phát biểu chọn (selection statement) trong C# bao gồm các phát biểu (if, if...else..., switch...case...).

**Phát biểu if**

**if (expression)**

**statement**

**if (expression)**

**{**

**statement1**

**statement1**

**}**

**Phát biểu if...else...**

**if (expression)**

**statement1**

**else**

**statement2**

# Phát biểu switch...case...

Phát biểu switch...case... là phát biểu điều khiển nhiều chọn lựa bằng cách truyền điều khiển đến phát biểu case bên trong.

**switch (expression)**

{

**case constant-expression:**

**statement**

**jump-statement**

**[default:**

**statement**

**jump-statement]**

}

# Phát biểu lặp

Phát biểu vòng lặp trong C# bao gồm do, for, foreach, while.

**Vòng lặp do**

```
do  
    statement  
while (expression);
```

**Vòng lặp while**

```
while (expression)  
    statement
```

# Phát biểu lặp

## Vòng lặp for

```
for ([initializers]; [expression]; [iterators])  
    statement
```

## Vòng lặp foreach ... in

```
foreach (type identifier in expression)  
    statement
```

- ▶ Vòng lặp foreach lặp lại một nhóm phát biểu cho mỗi phần tử trong mảng hay tập đối tượng.
- ▶ Phát biểu dùng để duyệt qua tất cả các phần tử trong mảng hay tập đối tượng và thực thi một tập lệnh

# Phát biểu nhảy

- ← Phát biểu nhảy sẽ được sử dụng khi chương trình muốn chuyển đổi điều khiển.
- ← Phát biểu nhảy: break, continue, default, goto, return

# Tóm tắt

Statement	Example
Local variable declaration	<pre>static void Main() {     int a;     int b = 2, c = 3;     a = 1;     Console.WriteLine(a + b + c); }</pre>
Local constant declaration	<pre>static void Main() {     const float pi = 3.1415927f;     const int r = 25;     Console.WriteLine(pi * r * r); }</pre>
Expression statement	<pre>static void Main() {     int i; i = 123;     Console.WriteLine(i);     i++; // tăng i lên 1     Console.WriteLine(i); }</pre>

# Tóm tắt

Statement	Example
if statement	<pre>static void Main(string[] args) {     if (args.Length == 0) {         Console.WriteLine("No arguments");     } else { Console.WriteLine("One or more arguments"); } }</pre>
switch statement	<pre>static void Main(string[] args) {     int n = args.Length;     switch (n) {         case 0: Console.WriteLine("No arguments");                   break;         case 1: Console.WriteLine("One argument");                   break;         default: Console.WriteLine("{0} arguments", n);                   break;     } }</pre>

# Tóm tắt

Statement	Example
while statement	<pre>static void Main(string[] args) {     int i = 0;     while (i &lt; args.Length) {         Console.WriteLine(args[i]);         i++;     } }</pre>
do statement	<pre>static void Main() {     string s;     do {         s = Console.ReadLine();         if (s != null) Console.WriteLine(s);     } while (s != null); }</pre>
for statement	<pre>static void Main(string[] args) {     for (int i = 0; i &lt; args.Length; i++)         Console.WriteLine(args[i]); }</pre>

# Tóm tắt

Statement	Example
foreach statement	<pre>static void Main(string[] args) {     string s;     foreach (s in args) Console.WriteLine(s); }</pre>
break statement	<pre>static void Main() {     while (true) {         string s = Console.ReadLine();         if (s == null) break;         Console.WriteLine(s);     } }</pre>
continue statement	<pre>static void Main(string[] args) {     for (int i = 0; i &lt; args.Length; i++) {         if (args[i].StartsWith("/")) continue;         Console.WriteLine(args[i]);     } }</pre>

# Tóm tắt

Statement	Example
goto statement	<pre>static void Main(string[] args) {     int i = 0;     goto check;     loop: Console.WriteLine(args[i++]);     check: if (i &lt; args.Length) goto loop; }</pre>
return statement	<pre>static int Add(int a, int b) { return a + b; } static void Main() {     Console.WriteLine(Add(1, 2));     return; }</pre>
checked and unchecked statements	<pre>static void Main() {     int i = int.MaxValue;     checked { Console.WriteLine(i + 1); // Exception }     unchecked { Console.WriteLine(i + 1); // Overflow } }</pre>

# Tóm tắt

Statement	Example
lock statement	<pre>class Account {     decimal balance;     public void Withdraw(decimal amount) {         lock (this) {             if (amount &gt; balance)                 throw new Exception("Insufficient funds");             balance -= amount;         }     } }</pre>
using statement	<pre>static void Main() {     using (TextWriter w = File.CreateText("test.txt")) {         w.WriteLine("Line one");         w.WriteLine("Line two");         w.WriteLine("Line three");     } }</pre>

# Tóm tắt

Statement	Example
throw and try statements	<pre>static double Divide(double x, double y) {     if (y == 0) throw new DivideByZeroException();     return x / y; }  static void Main(string[] args) {     try {         if (args.Length != 2)             throw new Exception("Two numbers required");         double x = double.Parse(args[0]);         double y = double.Parse(args[1]);         Console.WriteLine(Divide(x, y));     } catch (Exception e) {         Console.WriteLine(e.Message);     } }</pre>

# OOP in C#

# Khai báo lớp

[Thuộc tính] [Bổ từ truy cập]

**class <Định danh lớp> [: Lớp cơ sở]**

{

<Phần thân của lớp:  
các thuộc tính  
phương thức >

}

# Thuộc tính truy cập

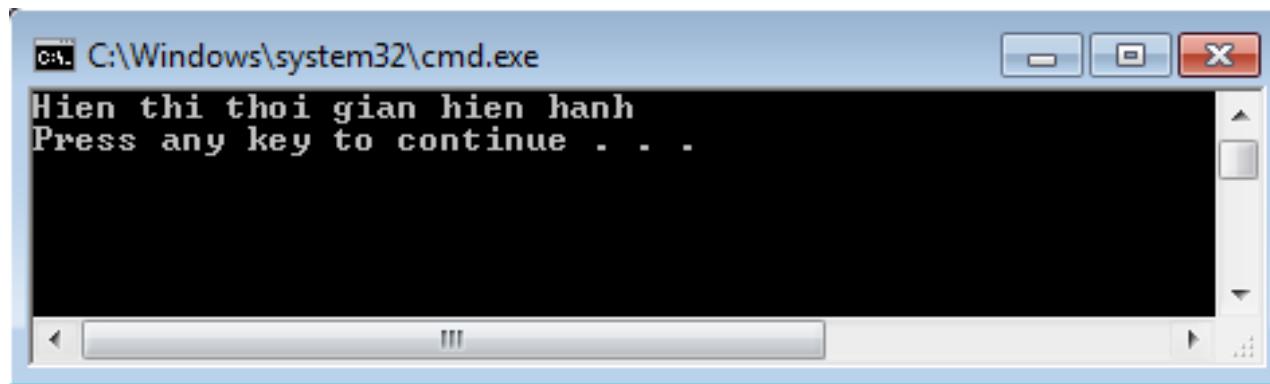
Thuộc tính	Giới hạn truy cập
public	Không hạn chế
private	Chỉ trong lớp (mặc định)
protected	Trong lớp và lớp con(lớp dẫn xuất)
internal	Trong chương trình
protected internal	Trong chương trình và trong lớp con

# Ví dụ

```
01  using System;
02  public class ThoiGian
03  {
04      public void ThoiGianHienHanh()
05      {
06          Console.WriteLine("Hiển thị thời gian
hien hanh");
07      }
08      // Các biến thành viên
09      int Nam;
10      int Thang;
11      int Ngay;
12      int Gio;
13      int Phut;
14      int Giay;
15  }
```

# Ví dụ

```
16  public class Tester  
17  {  
18      static void Main()  
19      {  
20          ThoiGian t = new ThoiGian();  
21          t.ThoiGianHienHanh();  
22      }  
23 }
```



# Khởi tạo giá trị cho thuộc tính

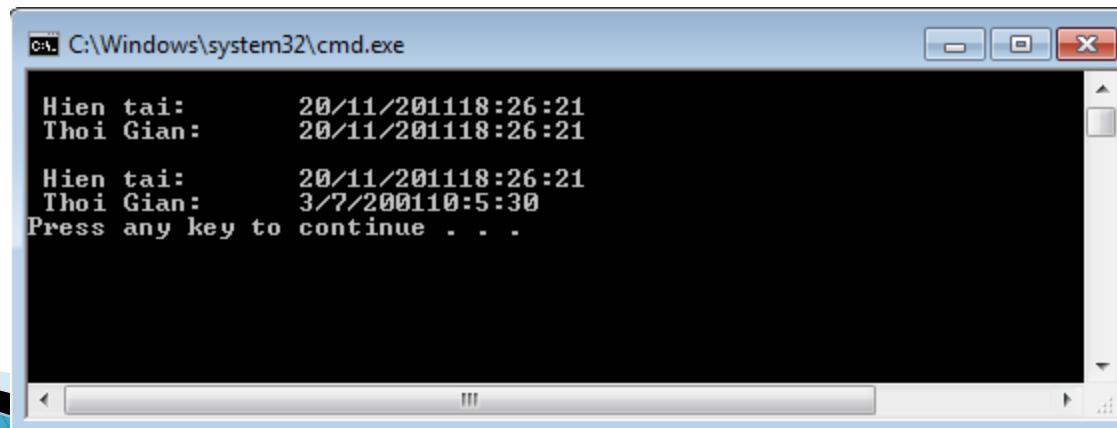
```
01  public class ThoiGian
02  {
03      public void ThoiGianHienHanh()
04      {
05          System.DateTime now = System.DateTime.Now;
06          System.Console.WriteLine("\n Hien tai: \t
07              {0}/{1}/{2}{3}:{4}:{5}",
08              now.Day, now.Month, now.Year, now.Hour,
09              now.Minute, now.Second);
10          System.Console.WriteLine(" Thoi Gian:\t
11              {0}/{1}/{2}{3}:{4}:{5}",
12              Ngay, Thang, Nam, Gio, Phut, Giay);
13      }
14      public ThoiGian( System.DateTime dt)
15      {
16          Nam = dt.Year;Thang = dt.Month;Ngay = dt.Day;
17          Gio = dt.Hour;Phut = dt.Minute;
18          Giay = dt.Second;
19      }
```

# Khởi tạo giá trị cho thuộc tính

```
17     public ThoiGian(int Year, int Month, int Date,  
18         int Hour, int Minute)  
19     {  
20         Nam = Year;Thang = Month;Ngay = Date;  
21         Gio = Hour;Phut = Minute;  
22     }  
23     private int Nam;  
24     private int Thang;  
25     private int Ngay;  
26     private int Gio;  
27     private int Phut;  
28     private int Giay = 30 ; // biến được khởi tạo.  
 }
```

# Khởi tạo giá trị cho thuộc tính

```
29  public class Tester
30  {
31      static void Main()
32      {
33          System.DateTime currentTime =
34              System.DateTime.Now;
35          ThoiGian t1 = new ThoiGian( currentTime );
36          t1.ThoiGianHienHanh();
37          ThoiGian t2 = new ThoiGian(2001,7,3,10,5);
38          t2.ThoiGianHienHanh();
39      }
}
```



# Phương thức khởi tạo

Hàm tạo mặc định: giống C++

Hàm tạo có đối số: tương tự C++

```
public class MyClass
{
    public MyClass() // zero-parameter constructor
    {
        // construction code
    }
    public MyClass(int number) // another overload
    {
        // construction code
    }
}
```

# **Phương thức khởi tạo sao chép**

← C# không cung cấp phương thức khởi tạo sao chép

```
public ThoiGian( ThoiGian tg)
```

```
{
```

```
    Nam = tg.Nam;  
    Thang = tg.Thang;  
    Ngay = tg.Ngay;  
    Gio = tg.Gio;  
    Phut = tg.Phut;  
    Giay = tg.Giay;
```

```
}
```

# Object Initializer

Tính năng này giúp ta giảm thiểu sự dài dòng khi khai báo mới một đối tượng. Thay vì dùng cách định giá trị member thông qua instance của đối tượng, ta có thể định trực tiếp ngay khi vừa khai báo đối tượng

Ví dụ:

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
}

public class Program
{
    static void Main(string[] args)
    {
        var person = new Person() {Name = "John", Age = 25};
    }
}
```

## Ví dụ

- ▶ Giả sử Công ty có hai loại nhân viên: Nhân viên văn phòng và Nhân viên sản xuất. Viết chương trình quản lý và tính lương cho từng nhân viên của công ty:
- ▶ Mỗi nhân viên cần quản lý các thông tin sau: Họ tên, ngày sinh, lương
- ▶ Công ty cần tính lương cho nhân viên như sau:
  - Đối với nhân viên sản xuất:
    - Lương = lương căn bản + số sản phẩm \* 5.000
  - Đối với nhân viên văn phòng:
    - Lương = số ngày làm việc \* 100.000
- ▶ Viết chương trình nhập vào N nhân viên, xuất ra N nhân viên vừa nhập, tổng lương công ty phải trả và nhân viên có lương cao nhất.

# Named Arguments và Optional Parameters

**Optional parameters:** Một tham số được khai báo là tùy chọn bằng cách đặt một giá trị cho nó:

Ví dụ:

```
public MyClass(int x, int y = 5, int z = 7)
```

```
{
```

```
.....
```

```
}
```

```
MyClass myClass = new MyClass(1, 2, 3); // gọi constructor của lớp MyClass
```

```
MyClass myClass = new MyClass(1, 2); // thiếu tham số z
```

```
MyClass myClass = new MyClass(1); // thiếu tham số y và z
```

# Named Arguments và Optional Parameters

## Named and optional arguments:

- C# không cho phép để trống các tham số ở giữa ví dụ MyClass(1, , 3).
- Nếu muốn bỏ trống các tham số ở giữa, mỗi tham số cần phải gán với một cái tên.

## Ví dụ:

```
MyClass myClass = new MyClass(1, z: 3); // passing z by name
```

```
MyClass myClass = new MyClass(x: 1, z: 3);
```

hay

```
MyClass myClass = new MyClass(z: 3, x: 1);
```

# Phương thức hủy bỏ

- ▶ C# cung cấp cơ chế thu dọn (garbage collection) và do vậy **không cần** phải khai báo tường minh các phương thức hủy.
- ▶ Phương thức **Finalize** sẽ được gọi bởi cơ chế thu dọn khi đối tượng bị hủy.
- ▶ Phương thức kết thúc chỉ giải phóng các tài nguyên mà đối tượng nắm giữ, và không tham chiếu đến các đối tượng khác

# Phương thức hủy bỏ

```
~Class1()  
{  
    // Thực hiện một số công việc  
}
```

```
Class1.Finalize()  
{  
    // Thực hiện một số công việc  
    base.Finalize();  
}
```

# Hàm hủy

```
class MyClass : IDisposable
{
    public void Dispose()
    {
        // implementation
    }
}
```

# Hàm hủy

- ▶ Lớp sẽ thực thi giao diện *System.IDisposable*, tức là thực thi phương thức *IDisposable.Dispose()*.
- ▶ Không biết trước được khi nào một Destructor được gọi.
- ▶ Có thể chủ động gọi thu dọn rác bằng cách gọi phương thức *System.GC.Collect()*.
- ▶ *System.GC* là một lớp cơ sở .NET mô tả bộ thu gom rác và phương thức *Collect()* dùng để gọi bộ thu gom rác.

# Con trỏ this

- ▶ Từ khóa **this** dùng để tham chiếu đến thể hiện hiện hành của một đối tượng

```
public void SetYear( int Nam)  
{  
    this.Nam = Nam;  
}
```

- ▶ Tham chiếu this này được xem là con trỏ ẩn đến tất cả các phương thức **không có thuộc tính tĩnh** trong một lớp

# Thành viên static

- ▶ Thành viên tĩnh được xem như một phần của lớp.
- ▶ Có thể truy cập đến thành viên tĩnh của một lớp thông qua tên lớp
- ▶ **Không có friend**
- ▶ Phương thức tĩnh hoạt động ít nhiều giống như phương thức toàn cục

# Thuộc tính (property)

Thuộc tính cho phép tạo ra các field read-only, write-only.

Thuộc tính cho phép tạo ra các field “ảo” với “bên ngoài”

```
class Student {  
    protected DateTime _Birthday;  
  
    public int Age {  
        get {  
            return DateTime.Today().Year - _Birthday.Year;  
        }  
    }  
}  
...  
Console.WriteLine("Age: {0}", chau.Age);
```

# Thuộc tính (property)

Cho phép “filter” các giá trị được ghi vào field mà không phải dùng “cơ chế” hàm **set\_xxx** như C++.

Bên ngoài có thể dùng như field (dùng trong biểu thức)

```
class Student {  
    protected DateTime _Birthday;  
    public DateTime Birthday {  
        get {  
            return _Birthday;  
        }  
        set {  
            if (...) ...  
                throw new ...  
            _Birthday = value;  
        }  
    }  
}  
chau.Birthday = new DateTime(2007,09,23);  
Console.WriteLine("Birthday: {0}", chau.Birthday);
```

# Thuộc tính (property)

```
protected string foreName; //foreName là attribute  
của một lớp  
public string ForeName //ForeName là một Property  
{  
    get  
    {  
        return foreName;  
    }  
    set  
    {  
        if (value.Length > 20)  
            // code here to take error recovery action  
            // (eg. throw an exception)  
        else  
            foreName = value;  
    }  
}
```

# Thuộc tính (property)

- ▶ Nếu câu lệnh Property chỉ có đoạn lệnh get -> thuộc tính chỉ đọc (Read Only)
- ▶ Nếu câu lệnh Property chỉ có đoạn lệnh set -> thuộc tính chỉ ghi (Write Only)

# Thuộc tính đọc và ghi

Cho phép gán (set) giá trị vào thuộc tính hay lấy (get) giá trị ra từ thuộc tính.

```
public <fieldType> LiA
{
    get
    {
        return LiA;
    }
    set
    {
        LiA = value; // value là từ khóa
    }
}
```

# Thuộc tính chỉ đọc

Nếu muốn thuộc tính chỉ đọc, chỉ sử dụng phương thức get

**public int liA**

{

**get**

{

**return LiA;**

}

}

# Thuộc tính chỉ đọc

```
01 using System;
02 public class BankAccount
03 {
04     protected string ID;
05     protected string Owner;
06     protected decimal _Balance;
07     public BankAccount(string ID, string Owner)
08     {
09         this.ID = ID;
10         this.Owner = Owner;
11         this._Balance = 0;
12     }
13     public void Deposit(decimal Amount)
14     {
15         _Balance += Amount;
16     }
17     public void Withdraw(decimal Amount)
18     {
19         _Balance -= Amount; // what if Amount > Balance?
20     }
21     public decimal Balance
22     {
23         get
24         {
25             return _Balance;
26         }
27     }
28 }
```

}

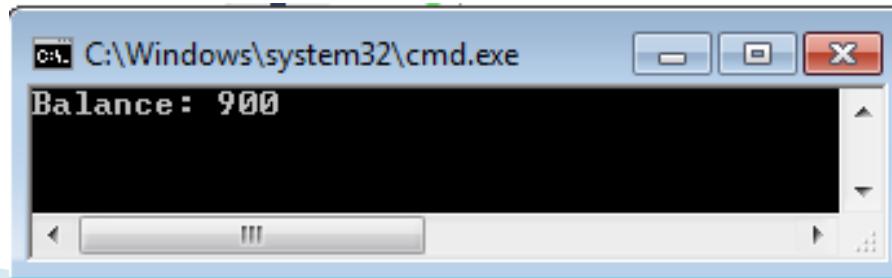
**Fields**

**Thuộc tính chỉ đọc**

**Read-only property**

# Thuộc tính chỉ đọc

```
29 class Program
30 {
31     static void Main(string[] args)
32     {
33         BankAccount myAcct = new BankAccount("100120023003",
34                                         "Nguyen Van An");
35         myAcct.Deposit(1000);
36         myAcct.Withdraw(100);
37         Console.WriteLine("Balance: {0}", myAcct.Balance);
38         //myAcct.Balance=10000;
39         Console.ReadLine();
40     }
41 }
```



# Auto-Implemented Properties

Khi không cần cài đặt gì đặc biệt cho get và set của property, Auto-Implemented Properties làm cho việc khai báo property ngắn gọn hơn.

Ví dụ:

```
class Employee
{
    public int ID{ get; private set; } // read-only
    public string FirstName { get; set; }
    public int LastName { get; set; }
}
```

# Collection Initializers

**Đơn giản hóa việc bổ sung các phần tử của danh sách thay vì dùng phương thức Add**

Ví dụ:

```
public class Person
{
    string _Name;
    List _Interset = new List();
    public string Name {
        get { return _Name; }
        set { _Name = value; } }
    public List Interests {
        get { return _Interset; }
    }
}
```

# Collection Initializers

Thay vì viết:

```
List PersonList = new List();
Person p1 = new Person();
p1.Name = "Mony Hamza";
p1.Interests.Add("Reading");
p1.Interests.Add("Running");
PersonList.Add(p1);
Person p2 = new Person();
p2.Name = "John Luke";
p2.Interests.Add("Swimming");
PersonList.Add(p2);
```

Ta có thể viết

```
var PersonList = new List{
    new Person{ Name = "Mony Hamza", Interests = { "Reading", "Running" } },
    new Person { Name = "John Luke", Interests = { "Swimming" } }
};
```

# Indexer

Cho phép tạo ra các thuộc tính giống như array (nhưng cách cài đặt bên trong không nhất thiết dùng array). Lưu ý là chỉ mục không nhất thiết phải là integer.

Có thể có nhiều chỉ mục

vd: Marks[string SubjectID, string SemesterID]

```
class Student {  
    protected string StudentID;  
    protected Database MarkDB;  
    public double this [string SubjectID] {  
        get {  
            return MarkDB.GetMark(StudentID, SubjectID);  
        }  
        set {  
            MarDB.UpdateMark(StudentID, value);  
        }  
    }  
}  
  
...  
Console.WriteLine("Physic mark: {0}", chau ["physic"]);
```

# Chồng hàm (overload)

- ▶ Không chấp nhận hai phương thức chỉ khác nhau về kiểu trả về.
- ▶ Không chấp nhận hai phương thức chỉ khác nhau về đặc tính của một thông số đang được khai báo như *ref* hay *out*.

# Sự kế thừa

- ▶ 1 class chỉ có thể kế thừa từ **1 class** cơ sở
- ▶ 1 class có thể kế thừa từ **nhiều** Interface
- ▶ Từ khóa **sealed** được dùng trong trường hợp khai báo class mà không cho phép class khác kế thừa.

# Đơn thừa kế

```
class MyDerivedClass : MyBaseClass  
{  
    // functions and data members here  
}
```

# Đơn thừa kế

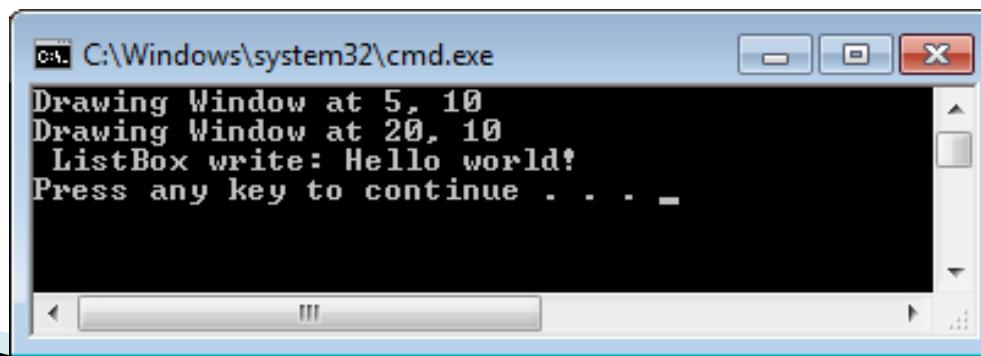
```
public class Window
{
    // Hàm khởi dựng lấy hai số nguyên chỉ đến vị trí của cửa sổ trên console
    public Window( int top, int left)
    {
        this.top = top;
        this.left = left;
    }
    public void DrawWindow()          // mô phỏng vẽ cửa sổ
    {
        Console.WriteLine("Drawing Window at {0}, {1}", top, left);
    }
    // Có hai biến thành viên private do đó hai biến này sẽ không
    // thấy bên trong lớp dẫn xuất.
    private int top;
    private int left;
}
```

# Đơn thừa kế

```
public class ListBox: Window
{
    // Khởi động có tham số
    public ListBox(int top, int left, string theContents) : base(top, left)
        //gọi khởi động của lớp cơ sở
    {
        mListBoxContents = theContents;
    }
    // Tạo một phiên bản mới cho phương thức DrawWindow
    // vì trong lớp dẫn xuất muốn thay đổi hành vi thực hiện
    // bên trong phương thức này
    public new void DrawWindow()
    {
        base.DrawWindow();
        Console.WriteLine(" ListBox write: {0}", mListBoxContents);
    }
    // biến thành viên private
    private string mListBoxContents;
}
```

# Đơn thừa kế

```
public class Tester
{
    public static void Main()
    {
        // tạo đối tượng cho lớp cơ sở
        Window w = new Window(5, 10);
        w.DrawWindow();
        // tạo đối tượng cho lớp dẫn xuất
        ListBox lb = new ListBox( 20, 10, "Hello world!");
        lb.DrawWindow();
    }
}
```



# Đa hình

- ▶ Để tạo một phương thức hỗ trợ đa hình: khai báo khóa **virtual** trong phương thức của lớp cơ sở
- ▶ Để định nghĩa lại các hàm **virtual**, hàm tương ứng lớp dẫn xuất phải có từ khóa **Override**

# Phương thức Override

```
class MyBaseClass
{
    public virtual string VirtualMethod()
    {
        return "This method is virtual and defined in MyBaseClass";
    }
}

class MyDerivedClass : MyBaseClass
{
    public override string VirtualMethod()
    {
        return "This method is an override defined in MyDerivedClass";
    }
}
```

# Phương thức Override

## Lớp Window

```
public virtual void DrawWindow()      // mô phỏng vẽ cửa sổ  
{  
    Console.WriteLine("Drawing Window at {0}, {1}", top, left);  
}
```

## Lớp Listbox

```
public override void DrawWindow()  
{  
    base.DrawWindow();  
    Console.WriteLine(" ListBox write: {0}", mListBoxContents);  
}
```

# Gọi các hàm của lớp cơ sở

- Cú pháp : *base.<methodname>()*

```
class CustomerAccount
{
    public virtual decimal CalculatePrice()
    {
        // implementation
    }
}

class GoldAccount : CustomerAccount
{
    public override decimal CalculatePrice()
    {
        return base.CalculatePrice() * 0.9M;
    }
}
```

# Ví dụ

```
Window[] winArray = new Window[3];
winArray[0] = new Window( 1, 2 );
winArray[1] = new ListBox( 3, 4, "List box is array");
winArray[2] = new Button( 5, 6 );
```

```
for( int i = 0; i < 3 ; i++)
{
    winArray[i].DrawWindow();
}
```

# Lớp cơ sở trừu tượng

abstract class Building

```
{  
    public abstract decimal CalculateHeatingCost();  
    // abstract method  
}  
  
← Một lớp abstract không được thể hiện và một phương thức abstract không được thực thi mà phải được overriden trong bất kỳ lớp thừa hưởng không abstract nào  
← Nếu một lớp có phương thức abstract thì nó cũng là lớp abstract  
← Một phương thức abstract sẽ tự động được khai báo virtual
```

# Abstract class

```
public abstract class BankAccount {  
    ...  
    public abstract bool IsSufficientFund(decimal Amount);  
    public abstract void AddInterest();  
    ...  
}
```

**Không thể new một abstract class**

**Chỉ có lớp abstract mới có thể chứa abstract method**

# Lớp cô lập (sealed class)

- ← Một lớp cô lập thì không cho phép các lớp dẫn xuất từ nó
- ← Để khai báo một lớp cô lập dùng từ khóa **sealed**

# Extension Methods

Tính năng này cho phép ta thêm method vào một lớp được xây dựng sẵn mà không làm ảnh hưởng đến cấu trúc của lớp.

//Lớp bị niêm phong, không thể kế thừa

**sealed class Person**

{

**public string Name { get; set; }**

.....

}

**static class Utility**

{

    //Extension method thêm vào lớp Person

**static public void ExMethod(this Person person)**

{

**person.Name = "John";**

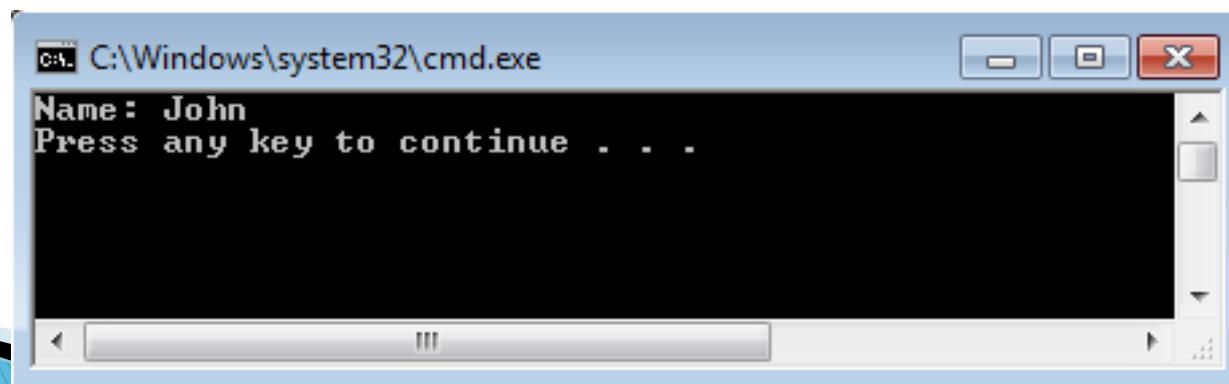
**Console.WriteLine("Name: " + person.Name);**

}

}

# Extension Methods

```
class Program
{
    static void Main(string[] args)
    {
        //Tạo instance của lớp Person
        var person = new Person();
        //Gọi Extension Method
        person.ExMethod();
    }
}
```



# Lớp Object

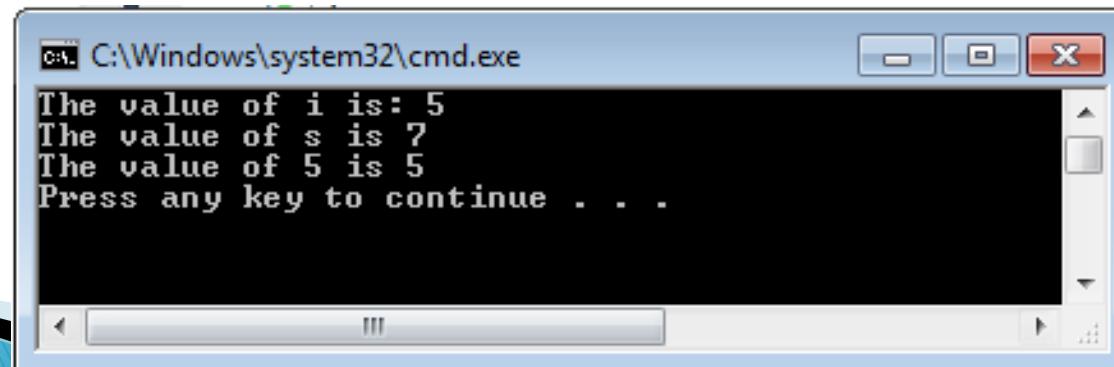
Phương thức	Chức năng
Equal( )	So sánh bằng nhau giữa hai đối tượng
GetHashCode( )	Cho phép những đối tượng cung cấp riêng những hàm băm cho sử dụng tập hợp.
GetType( )	Cung cấp kiểu của đối tượng
ToString( )	Cung cấp chuỗi thể hiện của đối tượng
Finalize( )	Dọn dẹp các tài nguyên
MemberwiseClone( )	Tạo một bản sao từ đối tượng.

# Lớp Object

```
01  using System;
02  public class SomeClass
03  {
04      public SomeClass(int val)
05      {
06          value = val;
07      }
08      public override string ToString()
09      {
10          return value.ToString();
11      }
12      private int value;
13 }
```

# Lớp Object

```
14 public class Tester  
15 {  
16     static void Main()  
17     {  
18         int i = 5;  
19         Console.WriteLine("The value of i is: {0}",  
20                           i.ToString());  
21         SomeClass s = new SomeClass(7);  
22         Console.WriteLine("The value of s is {0}",  
23                           s.ToString());  
24         Console.WriteLine("The value of 5 is {0}",  
25                           5.ToString());  
26     }  
27 }
```

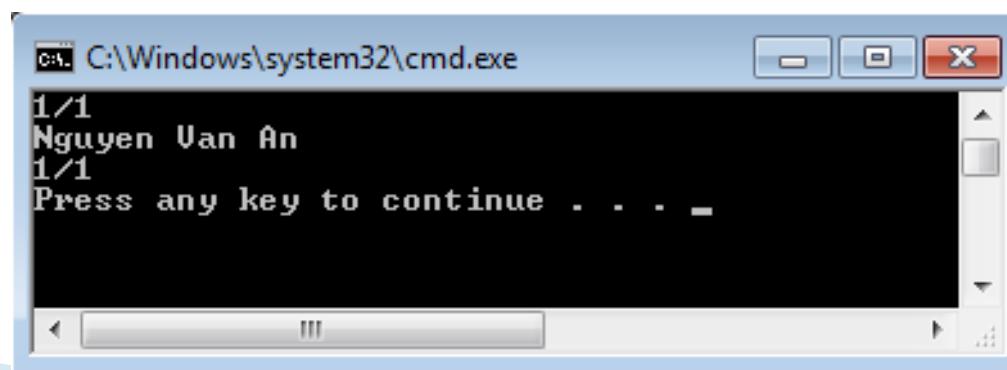


# Lớp trong lớp

```
01  using System;
02  class Nguoi
03  {
04      public class Date
05      {
06          private int ngay;
07          private int thang;
08          public Date() { ngay = 1; thang = 1; }
09          public void Xuat() { Console.WriteLine(ngay + "/" +
thang); }
10      }
11      private string ten;
12      private string ho;
13      private Date ns;
14      public Nguoi() { ten = "An"; ho = "Nguyen Van"; ns = new
Date(); }
15      public void Xuat()
16      {
17          ns.Xuat(); Console.WriteLine(ho + " " + ten);
18      }
19 }
```

# Lớp trong lớp

```
20 class Progarm
21 {
22
23     static void Main(string[] args)
24     {
25         Nguoi a = new Nguoi();
26         a.Xuat();
27         Nguoi.Date ns = new Nguoi.Date();
28         ns.Xuat();
29     }
30 }
```



# Lớp trong lớp

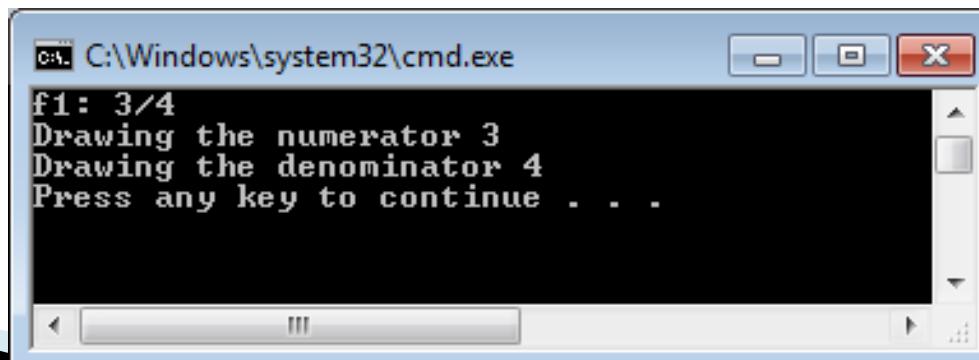
```
public class Fraction
{
    public Fraction( int numerator, int denominator)
    {
        this.numerator = numerator;
        this.denominator = denominator;
    }
    public override string ToString()
    {
        StringBuilder s = new StringBuilder();
        s.AppendFormat("{0}/{1}",numerator, denominator);
        return s.ToString();
    }
    internal class FractionArtist
    {.....}
    private int numerator;
    private int denominator;
}
```

# Lớp trong lớp

```
internal class FractionArtist
{
    public void Draw( Fraction f)
    {
        Console.WriteLine("Drawing the numerator {0}",
                          f.numerator);
        Console.WriteLine("Drawing the denominator {0}",
                          f.denominator);
    }
}
```

# Lớp trong lớp

```
public class Tester
{
    static void Main()
    {
        Fraction f1 = new Fraction( 3, 4);
        Console.WriteLine("f1: {0}", f1.ToString());
        Fraction.FractionArtist fa = new
            Fraction.FractionArtist();
        fa.Draw( f1 );
    }
}
```



# Overload Operator

**public static Fraction operator + ( Fraction lhs, Fraction rhs)**

**firstFraction + secondFraction**



**Fraction.operator+(firstFraction, secondFraction)**

nạp chồng toán tử (+) thì nên cung cấp một phương thức **Add()** cũng làm cùng chức năng là cộng hai đối tượng

# Overload Operator

- ← Overload == thì phải overload !=
- ← Overload > thì phải overload <
- ← Overload >= thì phải overload <=
- ← Phải cung cấp các phương thức thay thế cho toán tử được nạp chồng

# Overload Operator

Biểu tượng	Tên phương thức thay thế
+	Add
-	Subtract
*	Multiply
/	Divide
==	Equals
>	Compare

# Phương thức Equals

←public override bool Equals( object o )

```
public override bool Equals( object o )
{
    if ( !(o is Phanso) )
    {
        return false;
    }
    return this == (Phanso) o;
}
```

# Toán tử chuyển đổi

```
int myInt = 5;  
long myLong;  
myLong = myInt; // ngầm định  
myInt = (int) myLong; // tường minh
```

# Toán tử chuyển đổi

```
01 using System;
02 public class Phanso
03 {
04     public Phanso(int ts, int ms)
05     {
06         this.ts = ts;
07         this.ms = ms;
08     }
09     public Phanso(int wholeNumber)
10     {
11         ts = wholeNumber;
12         ms = 1;
13     }
14     public static implicit operator Phanso(int theInt)
15     {
16         return new Phanso(theInt);
17     }
```

# Toán tử chuyển đổi

```
18     public static explicit operator int(Phanso thePhanso)
19     {
20         return thePhanso.ts / thePhanso.ms;
21     }
22     public static bool operator ==(Phanso lhs, Phanso rhs)
23     {
24         if (lhs.ts == rhs.ts && lhs.ms == rhs.ms)
25         {
26             return true;
27         }
28         return false;
29     }
30     public static bool operator !=(Phanso lhs, Phanso rhs)
31     {
32         return !(lhs == rhs);
33     }
```

# Toán tử chuyển đổi

```
34     public override bool Equals(object o)
35     {
36         if (!(o is Phanso))
37         {
38             return false;
39         }
40         return this == (Phanso)o;
41     }
42     public static Phanso operator +(Phanso lhs, Phanso
rhs)
43     {
44         if (lhs.ms == rhs.ms)
45         {
46             return new Phanso(lhs.ts + rhs.ts, lhs.ms);
47         }
48         int firstProduct = lhs.ts * rhs.ms;
49         int secondProduct = rhs.ts * lhs.ms;
50         return new Phanso(firstProduct + secondProduct,
lhs.ms * rhs.ms);
51     }
```

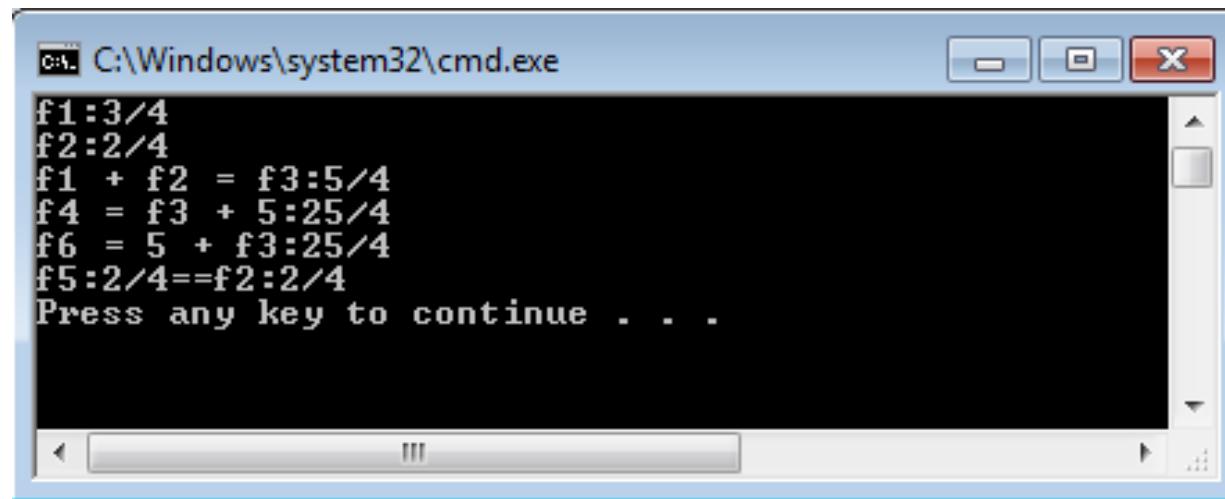
# Toán tử chuyển đổi

```
52     public override string ToString()
53     {
54         string s = ts.ToString() + "/" + ms.ToString();
55         return s;
56     }
57     private int ts;
58     private int ms;
59 }
```

# Toán tử chuyển đổi

```
60 public class Tester
61 {
62     static void Main()
63     {
64         Phanso f1 = new Phanso(3, 4);
65         Console.WriteLine("f1:{0}", f1.ToString());
66         Phanso f2 = new Phanso(2, 4);
67         Console.WriteLine("f2:{0}", f2.ToString());
68         Phanso f3 = f1 + f2;
69         Console.WriteLine("f1 + f2 = f3:{0}", f3.ToString());
70         Phanso f4 = f3 + 5;
71         Console.WriteLine("f4 = f3 + 5:{0}", f4.ToString());
72         Phanso f6 = 5+ f3 ;
73         Console.WriteLine("f6 = 5 + f3:{0}", f6.ToString());
74         Phanso f5 = new Phanso(2, 4);
75         if (f5 == f2)
76         {
77             Console.WriteLine("f5:{0}==f2:{1}", f5.ToString(),
78 f2.ToString());
79         }
80     }
```

# Toán tử chuyển đổi



```
C:\Windows\system32\cmd.exe
f1:3/4
f2:2/4
f1 + f2 = f3:5/4
f4 = f3 + 5:25/4
f6 = 5 + f3:25/4
f5:2/4==f2:2/4
Press any key to continue . . .
```

# Interface(giao diện)

## ► What is an Abstract Class?

Lớp trừu tượng đơn giản được xem như một class cha cho tất cả các Class có *cùng bản chất*. Do đó mỗi lớp dẫn xuất (lớp con) *chỉ có thể kế thừa từ một lớp trừu tượng*. Bên cạnh đó nó không cho phép tạo instance, nghĩa là sẽ không thể tạo được các đối tượng thuộc lớp đó.

## ► What is an Interface?

Lớp này được xem như một mặt nạ cho tất cả các Class *cùng cách thức hoạt động* nhưng có thể khác nhau về bản chất. Từ đó lớp dẫn xuất có thể kế thừa từ *nhiều lớp Interface* để bổ sung đầy đủ cách thức hoạt động của mình (*đa kế thừa - Multiple inheritance*).

# Interface(giao diện)

- ▶ Giống mà không giống abstract class!
- ▶ **Interface** chỉ có method hoặc property, **KHÔNG** có field (Abstract có thể có tất cả)
- ▶ Tất cả member của interface **KHÔNG** được phép cài đặt, chỉ là khai báo (Abstract class có thể có một số phương thức có cài đặt)
- ▶ Tên các interface nên bắt đầu bằng I ...
  - Ví dụ: **ICollection**, **ISortable**

# Interface(giao diện)

- ▶ Cú pháp để định nghĩa một giao diện:  
[thuộc tính] [bổ từ truy cập] interface <tên giao diện> [:  
danh sách cơ sở]

```
{  
    <phần thân giao diện>  
}
```

# Interface(giao diện)

- ▶ Một giao diện thì không có Constructor
- ▶ Một giao diện thì không cho phép chứa các phương thức nạp chồng.
- ▶ Nó cũng không cho phép khai báo những bổ túc trên các thành phần trong khi định nghĩa một giao diện.
- ▶ Các thành phần bên trong một giao diện luôn luôn là public và không thể khai báo virtual hay static.

# Interface(giao diện)

- ▶ Khi một class đã khai báo là implement một interface, nó phải **implement tất cả method hoặc thuộc tính** của interface đó
- ▶ Nếu hai interface có trùng tên method hoặc property, trong class phải chỉ rõ (explicit interface)  
**Ví dụ: IMovable và IEngine đều có thuộc tính MaxSpeed**

```
class ToyotaCar: Car, IMovable, IEngine {  
    public IMovable.MaxSpeed {  
        ...  
    }  
    public IEngine.MaxSpeed {  
    }  
}
```

# Ví dụ

- ▶ Tạo một giao diện nhằm mô tả những phương thức và thuộc tính của một lớp cần thiết để
  - lưu trữ
  - truy cậptừ một cơ sở dữ liệu hay các thành phần lưu trữ dữ liệu khác như là một tập tin

```
interface IStorable
{
    void Read();
    void Write(object);
}
```

```
public class Document : IStorable
{
    public void Read()
    {
        ....
    }
    public void Write()
    {
        ....
    }
}
```

# Xử lý lỗi

- ▶ Chương trình nào cũng có khả năng gặp phải các tình huống không mong muốn
  - người dùng nhập dữ liệu không hợp lệ
  - đĩa cứng bị đầy
  - file cần mở bị khóa
  - đối số cho hàm không hợp lệ
- ▶ Xử lý như thế nào?
  - Một chương trình không quan trọng có thể dừng lại
  - Chương trình điều khiển không lưu? điều khiển máy bay?

# Xử lý lỗi truyền thông

- ▶ Xử lý lỗi truyền thông thường là mỗi hàm lại thông báo trạng thái thành công/thất bại qua một mã lỗi
  - biến toàn cục (chẳng hạn **errno**)
  - giá trị trả về
    - **int remove ( const char \* filename );**
  - tham số phụ là tham chiếu
    - **double MyDivide(double numerator,  
double denominator, int& status);**

# Exception

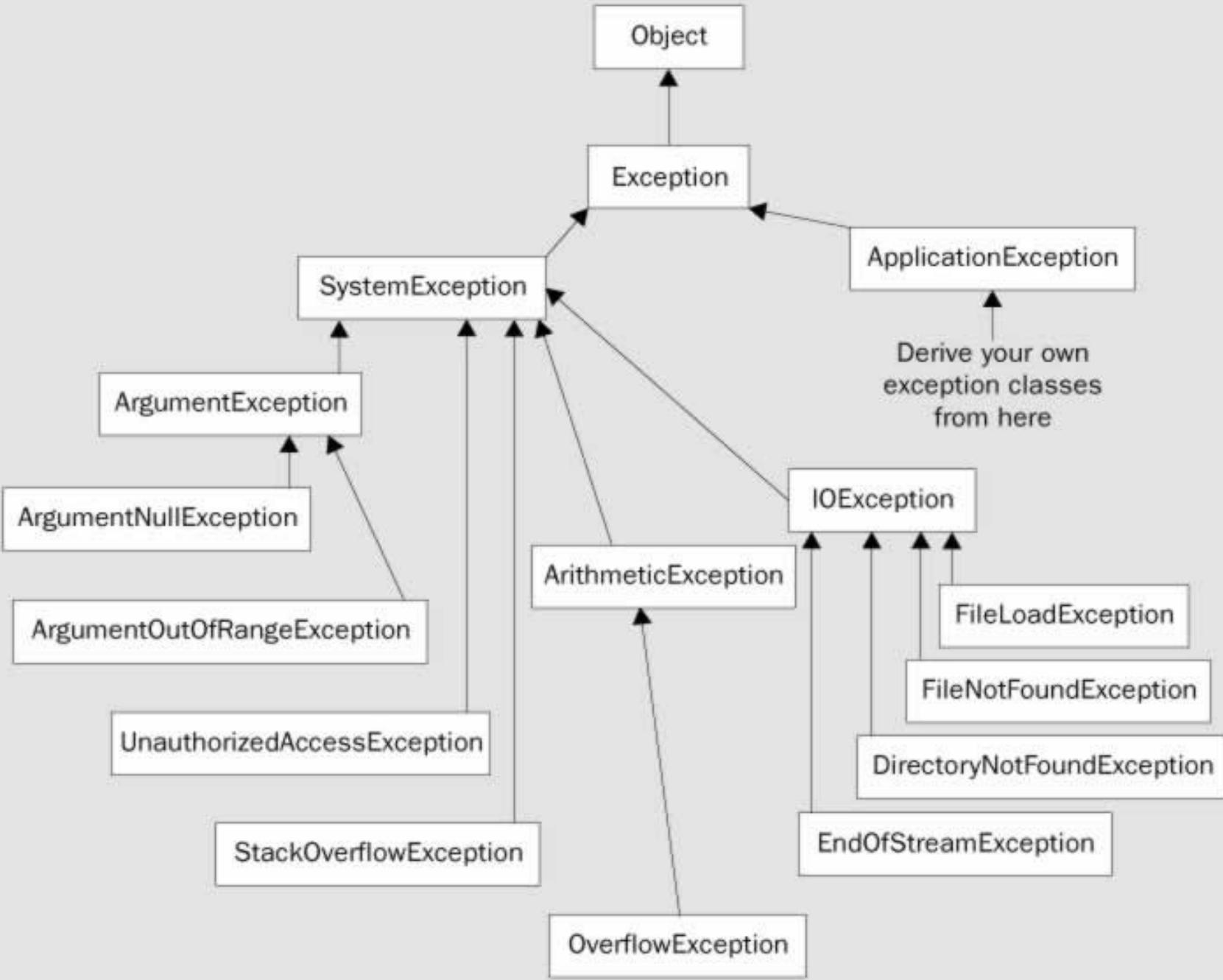
- ▶ Exception – ngoại lệ là cơ chế thông báo và xử lý lỗi giải quyết được các vấn đề kể trên
- ▶ Tách được phần xử lý lỗi ra khỏi phần thuật toán chính
- ▶ cho phép 1 hàm thông báo về nhiều loại ngoại lệ
  - Không phải hàm nào cũng phải xử lý lỗi nếu có một số hàm gọi thành chuỗi, ngoại lệ chỉ lần được xử lý tại một hàm là đủ
- ▶ không thể bỏ qua ngoại lệ, nếu không, chương trình sẽ kết thúc
- ▶ Tóm lại, cơ chế ngoại lệ mềm dẻo hơn kiểu xử lý lỗi truyền thống

# Xử lý ngoại lệ

- ▶ C# cho phép xử lý những lỗi và các điều kiện không bình thường với **những ngoại lệ**.
- ▶ Ngoại lệ là một đối tượng đóng gói những thông tin về sự cố của một chương trình không bình thường
- ▶ Khi một chương trình gặp một tình huống ngoại lệ  tạo một ngoại lệ. Khi một ngoại lệ được tạo ra, việc thực thi của các chức năng hiện hành sẽ bị treo cho đến khi nào việc xử lý ngoại lệ tương ứng được tìm thấy
- ▶ Một trình xử lý ngoại lệ là một khối lệnh chương trình được thiết kế xử lý các ngoại lệ mà chương trình phát sinh

# Xử lý ngoại lệ

- ▶ nếu một ngoại lệ được bắt và được xử lý:
  - chương trình có thể sửa chữa được vấn đề và tiếp tục thực hiện hoạt động
  - in ra những thông điệp có ý nghĩa



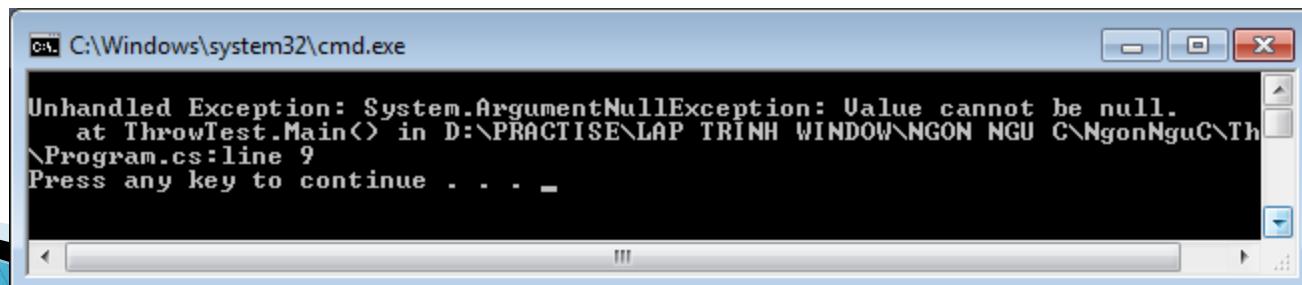
# Phát biểu throw

Phát biểu throw dùng để phát ra tín hiệu của sự cố bất thường trong khi chương trình thực thi với cú pháp:

**throw [expression];**

# Phát biểu throw

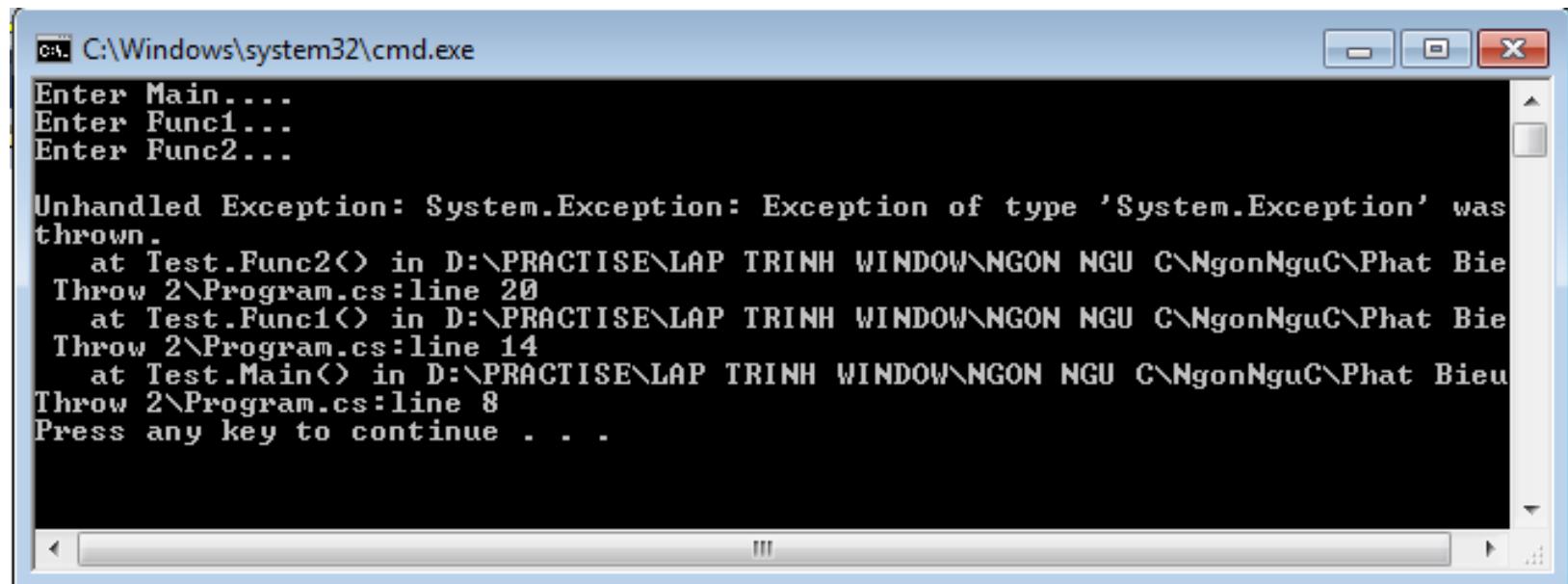
```
01  using System;
02  public class ThrowTest
03  {
04      public static void Main()
05      {
06          string s = null;
07          if (s == null)
08          {
09              throw (new ArgumentNullException());
10          }
11          Console.WriteLine("The string s is null");
12          // not executed
13      }
14 }
```



# Phát biểu throw

```
01  using System;
02  public class Test
03  {
04      public static void Main()
05      {
06          Console.WriteLine("Enter Main....");
07          Test t = new Test();
08          t.Func1();
09          Console.WriteLine("Exit Main....");
10     }
11     public void Func1()
12     {
13         Console.WriteLine("Enter Func1...");
14         Func2();
15         Console.WriteLine("Exit Func1...");
16     }
17     public void Func2()
18     {
19         Console.WriteLine("Enter Func2...");
20         throw new System.Exception();
21         Console.WriteLine("Exit Func2...");
22     }
23 }
```

# Phát biểu throw

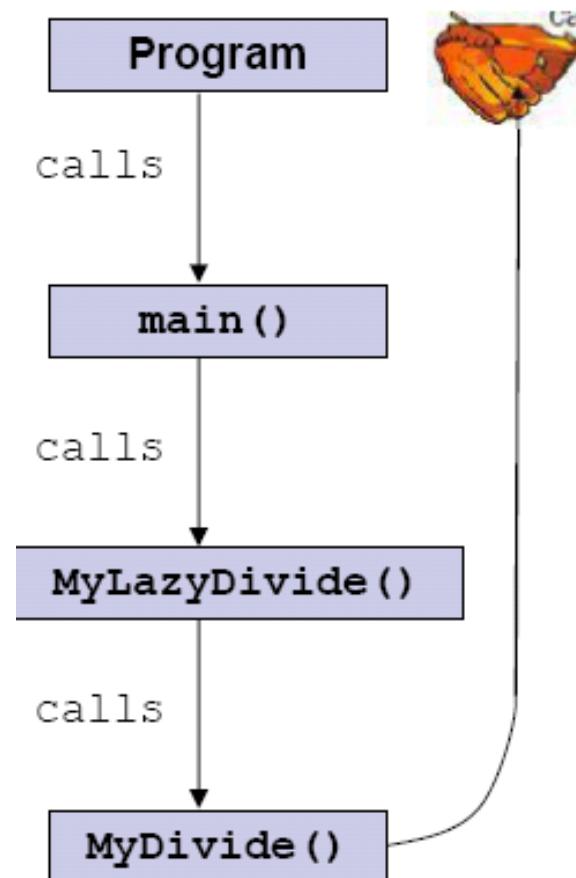


```
C:\Windows\system32\cmd.exe
Enter Main....
Enter Func1...
Enter Func2...

Unhandled Exception: System.Exception: Exception of type 'System.Exception' was
thrown.
   at Test.Func2() in D:\PRACTISE\LAP TRINH WINDOW\NGON NGU C\NgonNguC\Phat Bieu
   Throw 2\Program.cs:line 20
   at Test.Func1() in D:\PRACTISE\LAP TRINH WINDOW\NGON NGU C\NgonNguC\Phat Bieu
   Throw 2\Program.cs:line 14
   at Test.Main() in D:\PRACTISE\LAP TRINH WINDOW\NGON NGU C\NgonNguC\Phat Bieu
   Throw 2\Program.cs:line 8
Press any key to continue . . .
```

# Phát biểu try catch

- Trong C#, một trình xử lý ngoại lệ hay một đoạn chương trình xử lý các ngoại lệ được gọi là một khối catch và được tạo ra với từ khóa catch..
- Ví dụ: câu lệnh throw được thực thi bên trong khối try, và một khối catch được sử dụng để công bố rằng một lỗi đã được xử lý



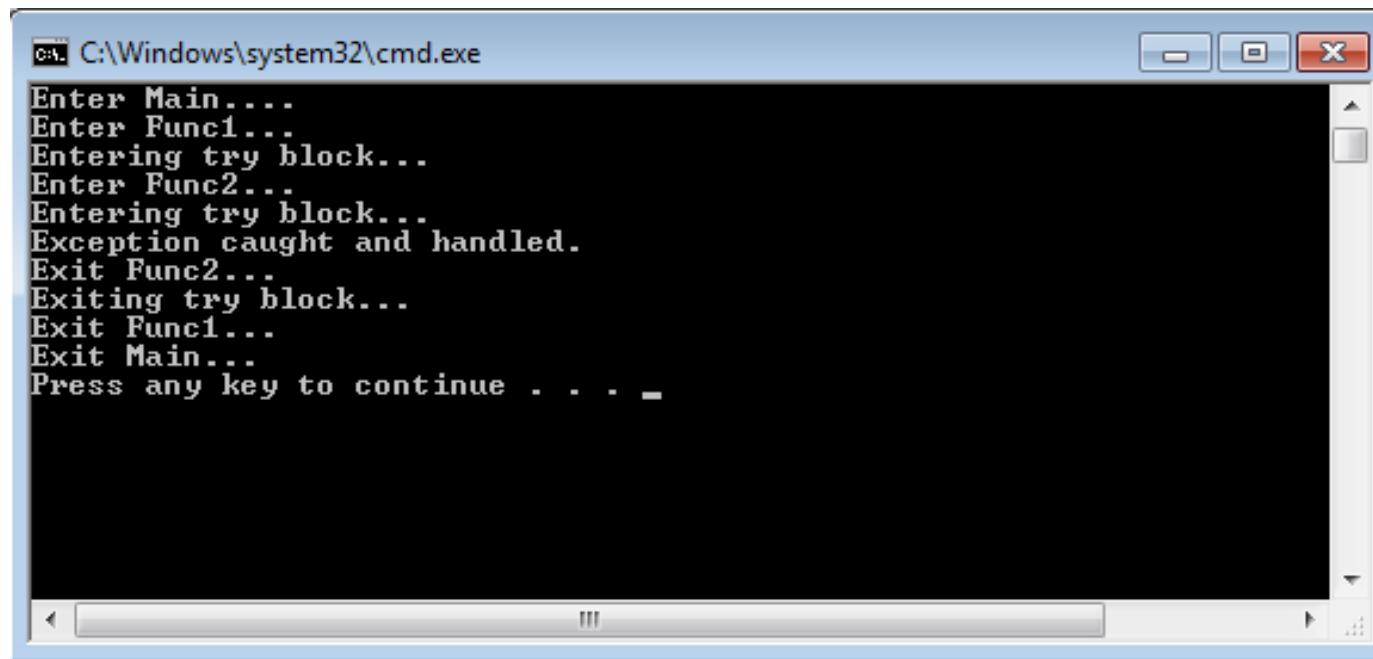
# Phát biếu try catch

```
public void Func2()
{
    Console.WriteLine("Enter Func2...");
    try
    {
        Console.WriteLine("Entering try block...");
        throw new System.Exception();
        Console.WriteLine("Exiting try block...");
    }
    catch
    {
        Console.WriteLine("Exception caught and handled.");
    }
    Console.WriteLine("Exit Func2...");
}
```

# Phát biếu try catch

```
public void Func1()
{
    Console.WriteLine("Enter Func1...");
    try
    {
        Console.WriteLine("Entering try block...");
        Func2();
        Console.WriteLine("Exiting try block...");
    }
    catch
    {
        Console.WriteLine("Exception caught and handled.");
    }
    Console.WriteLine("Exit Func1...");
}
```

# Phát biếu try catch



```
cmd C:\Windows\system32\cmd.exe
Enter Main...
Enter Func1...
Entering try block...
Enter Func2...
Entering try block...
Exception caught and handled.
Exit Func2...
Exiting try block...
Exit Func1...
Exit Main...
Press any key to continue . . .
```

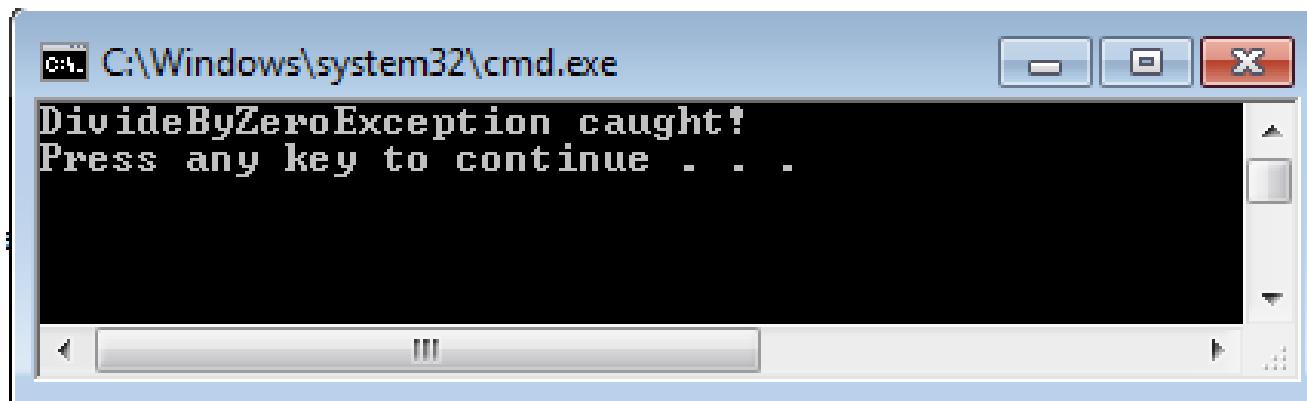
# Ví dụ

```
01 using System;
02 class Test
03 {
04     static void Main(string[] args)
05     {
06         Test t = new Test();
07         t.TestFunc();
08     }
09     public double DoDivide(double a, double b)
10     {
11         if (b == 0)
12             throw new System.DivideByZeroException();
13         if (a == 0)
14             throw new System.ArithmeticException();
15         return a / b;
16     }
}
```

# Ví dụ

```
17     public void TestFunc()
18     {
19         try
20         {
21             double a = 5;
22             double b = 0;
23             Console.WriteLine("{0} / {1} = {2}", a, b,
DoDivide(a, b));
24         }
25         catch (System.DivideByZeroException)
26         {
27             Console.WriteLine("DivideByZeroException caught!");
28         }
29         catch (System.ArithmetricException)
30         {
31             Console.WriteLine("ArithmetricException caught!");
32         }
33         catch
34         {
35             Console.WriteLine("Unknown exception caught");
36         }
37     }
38 }
```

# Ví dụ



# Câu lệnh finally

Đoạn chương trình bên trong khối ***finally*** được đảm bảo thực thi mà không quan tâm đến việc khi nào thì một ngoại lệ được phát sinh

**try**

**try-block**

**catch**

**catch-block**

**finally**

**finally-block**

# Câu lệnh finally

1. Dòng thực thi bước vào khối try.
2. Nếu không có lỗi xuất hiện,
  - tiến hành một cách bình thường xuyên suốt khối try, và khi đến cuối khối try, dòng thực thi sẽ nhảy đến khối finally ( bước 5),
  - nếu một lỗi xuất hiện trong khối try, thực thi sẽ nhảy đến khối catch ( bước tiếp theo)
3. Trạng thái lỗi được xử lí trong khối catch
4. vào cuối của khối catch , việc thực thi được chuyển một cách tự động đến khối finally
5. khối finally được thực thi

# Tạo riêng ngoại lệ

← phải được dẫn xuất từ System.ApplicationException

# Dynamic Binding

Với dynamic binding, khi nhận được một đối tượng, chúng ta không cần phải quan tâm kiểu của đối tượng đó. Môi trường thực thi sẽ quyết định phương thức hay phép toán nào sẽ áp dụng cho đối tượng nào. Điều này tạo tạo sự linh hoạt và đơn giản khi code

**Ví dụ:**

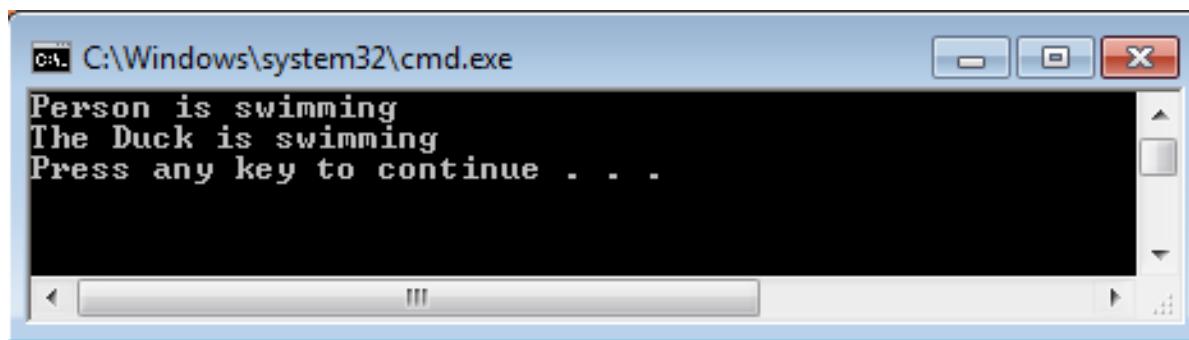
```
01 using System;
02 class Person
03 {
04     public string Name { get; set; }
05     public void Swim()
06     {
07         Console.WriteLine("Person is swimming");
08     }
09 }
```

# Dynamic Binding

```
10 class Duck
11 {
12     public string Weight { get; set; }
13     public void Swim()
14     {
15         Console.WriteLine("The Duck is swimming");
16     }
17 }
18 class Program
19 {
20     //Hàm có paramater là một object dynamic
21     static void InvokeSwim(dynamic obj)
22     {
23         obj.Swim();
24     }
}
```

# Dynamic Binding

```
25     static void Main(string[] args)
26     {
27         //Khai báo 2 object dynamic khác nhau
28         dynamic person = new Person();
29         dynamic duck = new Duck();
30         //Truyền 2 object khác kiểu vào cùng 1 hàm
31         InvokeSwim(person);
32         InvokeSwim(duck);
33     }
34 }
```



# Dynamic Binding

Dynamic hỗ trợ tốt các Operator +, -, \*, /

```
static dynamic Sum(dynamic obj1, dynamic obj2)
{
    return obj1 + obj2;
}

static void Main(string[] args)
{
    Console.WriteLine(Sum(5, 10));
    Console.WriteLine(Sum(5.2, 10.2));
}
```

# Delegates

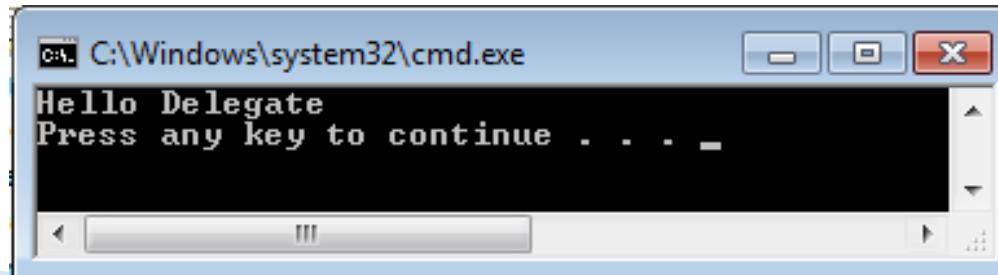
Một đối tượng kiểu delegate sẽ chứa các thông tin về method mà nó trả lời. Như vậy delegate cũng giống như con trả hàm trong C++.

Ví dụ:

```
01  using System;
02  delegate void MethodDelegate();
03  class Person
04  {
05      public void Hello()
06      {
07          Console.WriteLine("Hello Delegate");
08      }
09  }
```

# Delegates

```
10 class Program
11 {
12     static void Main(string[] args)
13     {
14         Person person = new Person();
15         MethodDelegate helloDelegate =
16             new MethodDelegate(person.Hello);
17         helloDelegate(); //gọi hàm
18     }
}
```

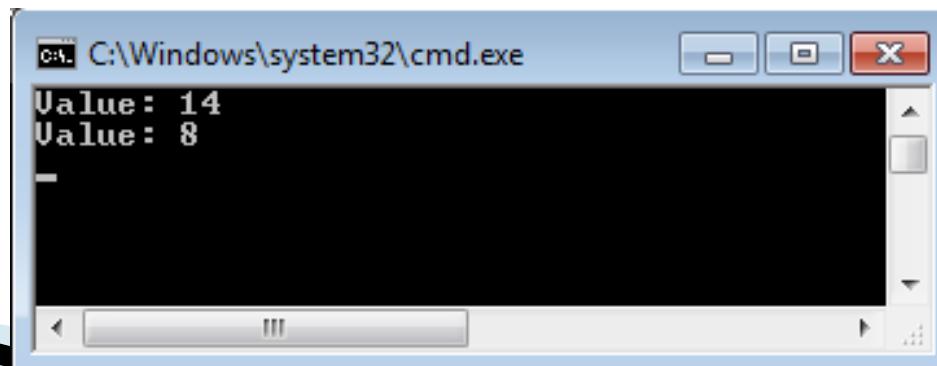


# Delegates

```
01 using System;
02 delegate void FunctionToCall(ref int x);
03 class Delegate2
04 {
05     public static void Add2(ref int x)
06     {
07         x += 2;
08     }
09     public static void Add3(ref int x)
10     {
11         x += 3;
12     }
13     static void Main(string[] args)
14     {
15         // Khai báo đồng thời gán bằng Add2
16         FunctionToCall functionDelegate = Add2;
17         functionDelegate += Add3;
18         functionDelegate += Add2;
19         functionDelegate += Add2;
```

# Delegates

```
20     int x = 5;
21     functionDelegate(ref x); // Gọi delegate
22     Console.WriteLine("Value: {0}", x);
23     int y = 5;
24     functionDelegate = Add2;
25     functionDelegate += Add3;
26     functionDelegate -= Add2;
27     functionDelegate(ref y); // Gọi delegate
28     Console.WriteLine("Value: {0}", y);
29     Console.ReadLine();
30 }
31 }
```

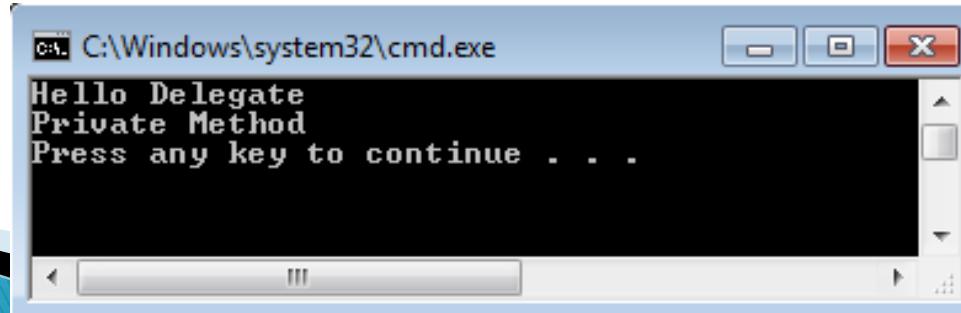


# Gọi private static method từ lớp khác

```
01  using System;
02  delegate void MethodDelegate();
03  class Person
04  {
05      public void Hello(MethodDelegate mDelegate)
06      {
07          Console.WriteLine("Hello Delegate");
08          //Gọi Method được truyền vào
09          if (mDelegate != null)
10              mDelegate();
11      }
12  }
13  class Program
14  {
15      private static void priMethod()
16      {
17          Console.WriteLine("Private Method");
18      }
}
```

# Gọi private static method từ lớp khác

```
19     static void Main(string[] args)
20     {
21         Person person = new Person();
22
23         //Khai báo delegate trả về privateMethod
24         MethodDelegate helloDelegate = new
25             MethodDelegate(priMethod);
26
27         //Truyền delegate vào method Hello như một đối
28         //số
29         person.Hello(helloDelegate);
30     }
```



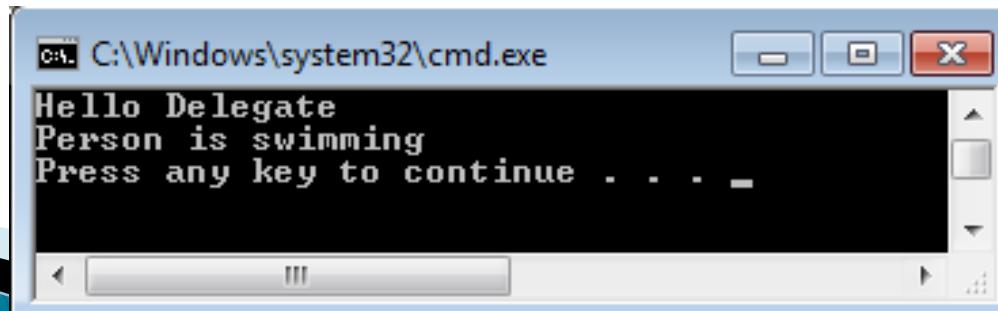
# Multicasting

**Không những đại diện cho một method, delegate còn có khả năng trả tới nhiều method cùng lúc.**

```
01  using System;
02  delegate void MethodDelegate();
03  class Person
04  {
05      public void Hello()
06      {
07          Console.WriteLine("Hello Delegate");
08      }
09      public void Swim()
10      {
11          Console.WriteLine("Person is swimming");
12      }
13 }
```

# Multicasting

```
14 class Program
15 {
16     static void Main(string[] args)
17     {
18         Person person = new Person();
19         MethodDelegate multicastDelegate = null;
20
21         //Multicasting delegate
22         multicastDelegate += new
23             MethodDelegate(person.Hello);
24
25         multicastDelegate += new
26             MethodDelegate(person.Swim);
27     }
}
```



# Events

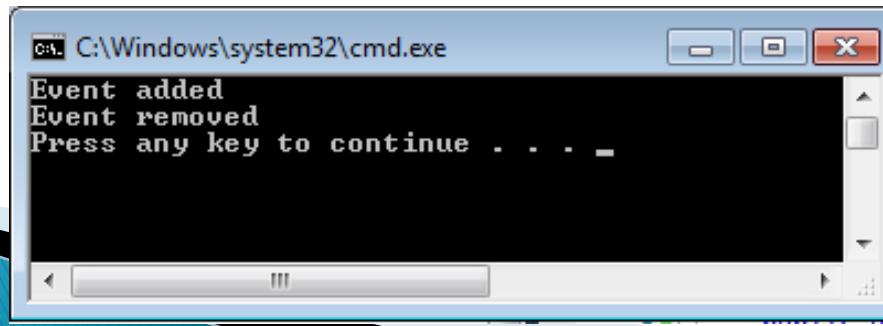
**Event là các sự kiện xảy ra khi chạy chương trình (sự kiện click của button, sự kiện giá trị của comboBox thay đổi,...). Event giúp xử lý code linh hoạt và đơn giản hơn. Khi sử dụng Event thì chúng ta không cần quan tâm đến việc khi nào thì đặt hàm xử lý vì khi event phát sinh nó sẽ tự động gọi hàm xử lý ra để thực hiện.**

**Ví dụ:**

```
01  using System;
02  internal delegate voidTextChanged();
03  class Person
04  {
05      public eventTextChanged TextChanged
06      {
07          add { Console.WriteLine("Event added"); }
08          remove { Console.WriteLine("Event removed"); }
09      }
10  }
```

# Events

```
11 class Program
12 {
13     static void Main(string[] args)
14     {
15         Person person = new Person();
16         person.TextChanged += new
17             TextChanged(person_TextChanged);
18     }
19
20     private static void person_TextChanged()
21     {
22         Console.WriteLine("Event Called");
23     }
24 }
```



# Anonymous Method

Thay vì khai báo một event như sau

```
person.TextChanged += new  
TextChanged(person_TextChanged);
```

Event đó trở tới method

```
private static void person_TextChanged()  
{  
    Console.WriteLine("Event Called");  
}
```

Bây giờ với tính năng anonymous method, ta có thể đơn giản hóa như sau

```
person.TextChanged += delegate()  
{  
    Console.WriteLine("Event Called");  
};
```

# Lambda Expressions

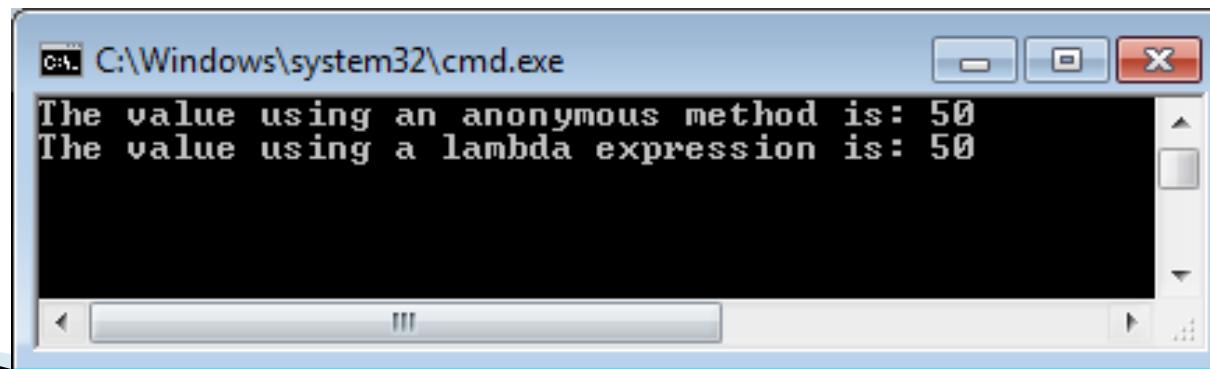
Lambda Expression được dùng để viết những phương thức anonymous ngắn gọn dùng để tạo ra các delegate

Ví dụ:

```
01  using System;
02  public delegate int MyDelegate(int n);
03  class LambdaExpresion
04  {
05      static void Main()
06      {
07          // Anonymous method that returns the argument
08          // multiplied by 10:
09          MyDelegate Obj1 = new MyDelegate(delegate(int
10              n) { return n * 10; });
11          // Display the result:
12          Console.WriteLine("The value using an
13          anonymous method is: {0}", Obj1(5));
```

# Lambda Expressions

```
11      // Using lambda expression to do the
same job:
12      MyDelegate Obj2 = (int n) => n * 10;
13      // Display the result:
14      Console.WriteLine("The value using a
lambda expression is: {0}", Obj2(5));
15      Console.ReadLine();
16  }
17 }
```



# Giao diện tập hợp

Môi trường .NET cung cấp những giao diện chuẩn cho việc liệt kê, so sánh, và tạo các tập hợp.

IEnumerable	Khi một lớp cài đặt giao diện này, đối tượng thuộc lớp đó có thể dùng trong câu lệnh foreach.
ICollection	Thực thi bởi tất cả các tập hợp để cung cấp phương thức CopyTo() cũng như các thuộc tính Count, ISReadOnly, ISSynchronized, và SyncRoot.
IComparer	So sánh giữa hai đối tượng lưu giữ trong tập hợp để sắp xếp các đối tượng trong tập hợp.
IList	Sử dụng bởi những tập hợp mảng được chỉ mục
IDictionary	Dùng trong các tập hợp dựa trên key và value
IDictionaryEnumerator	Cho phép liệt kê dùng câu lệnh foreach qua tập hợp hỗ trợ IDictionary.

# Interface IEnumerable

- ▶ Interface này chỉ có một phương thức duy nhất là `GetEnumerator()`, công việc của phương thức là trả về một sự thực thi đặc biệt của `IEnumerator`.
- ▶ Mục đích của interface `IEnumerable` là cho phép chúng ta có thể sử dụng từ khóa **foreach** trên đối tượng của class cài đặt interface này.

# Interface IEnumarator

Interface IEnumarator bao gồm 2 phương thức quan trọng là **MoveNext**, **Reset** và thuộc tính **Current**.

- ▶ Thuộc tính **Current** trả về phần tử hiện tại đang được duyệt tới trong danh sách.
- ▶ **MoveNext** dùng để đi đến phần tử tiếp theo trong danh sách (hay nói cách khác thay đổi giá trị của Property Current). Phương thức này trả về giá trị true nếu như việc di chuyển đến đối tượng tiếp theo thành công, trả về false nếu thất bại (trong trường hợp đã đến cuối danh sách).
- ▶ **Reset** dùng để đưa con trỏ hiện tại về vị trí ban đầu. Vị trí ban đầu này là vị trí nằm ngay trước phần tử đầu tiên trong danh sách.

# Ví dụ

```
01  using System;
02  using System.Collections;
03  public class Person
04  {
05      public Person(string fName, string lName)
06      {
07          this.firstName = fName;
08          this.lastName = lName;
09      }
10      public string firstName;
11      public string lastName;
12  }
13  public class People : IEnumerable
14  {
15      private Person[] _people;
16      // Khởi tạo
17      public People(Person[] pArray)
18      {
19          _people = new Person[pArray.Length];
20
21          for (int i = 0; i < pArray.Length; i++)
22          {
23              _people[i] = pArray[i];
24          }
25      }
```

# Ví dụ

```
26     IEnumarator IEnumarable.GetEnumerator()
27     {
28         return (IEnumarator)GetEnumerator();
29     }
30     public PeopleEnum GetEnumerator()
31     {
32         return new PeopleEnum(_people);
33     }
34 }
35
36 public class PeopleEnum : IEnumarator
37 {
38     public Person[] _people;
39     int position = -1;
40     // Khởi tạo
41     public PeopleEnum(Person[] list)
42     {
43         _people = list;
44     }
45     // Tăng vị trí
46     public bool MoveNext()
47     {
48         position++;
49         return (position < _people.Length);
50     }
}
```

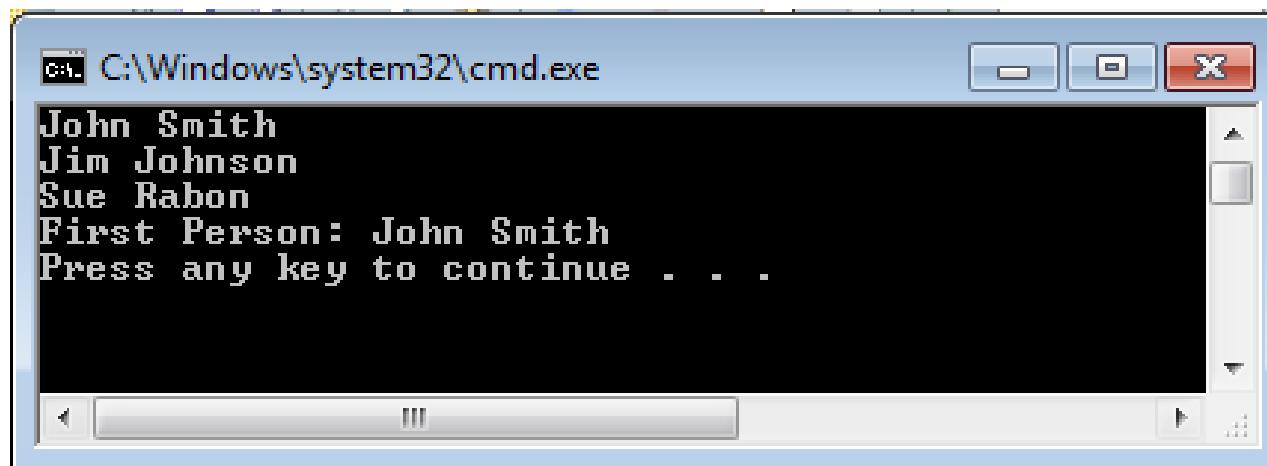
# Ví dụ

```
51     public void Reset()
52     {
53         position = -1;
54     }
55     object IEnumerator.Current
56     {
57         get
58         {
59             return Current;
60         }
61     }
62     public Person Current
63     {
64         get
65         {
66             try
67             {
68                 return _people[position];
69             }
70             catch (IndexOutOfRangeException)
71             {
72                 throw new InvalidOperationException();
73             }
74         }
75     }
76 }
```

# Ví dụ

```
77 class Program
78 {
79     static void Main()
80     {
81         Person[] peopleArray = new Person[3]
82         {
83             new Person("John", "Smith"),
84             new Person("Jim", "Johnson"),
85             new Person("Sue", "Rabon"),
86         };
87         People peopleList = new People(peopleArray);
88         // liệt kê danh sách dùng foreach
89         foreach (Person p in peopleList)
90             Console.WriteLine(p.firstName + " " + p.lastName);
91         PeopleEnum peopleEnum = peopleList.GetEnumerator();
92         peopleEnum.Reset();
93         // lấy người đầu tiên trong danh sách
94         peopleEnum.MoveNext();
95         Person firstPerson = peopleEnum.Current;
96         Console.WriteLine("First Person: {0} {1}",
97                         firstPerson.firstName, firstPerson.lastName);
98     }
}
```

# Ví dụ



# Interface ICollection

- ▶ Interface ICollection cung cấp các thuộc tính: Count, IsSynchronized, và SyncRoot.
- ▶ Ngoài ra ICollection cũng cung cấp một phương thức CopyTo().
- ▶ Thuộc tính thường được sử dụng là Count, thuộc tính này trả về số thành phần trong tập hợp.

# Interface IComparable

Vai trò của **IComparable** là cung cấp một phương pháp `CompareTo()` dùng để so sánh hai đối tượng.

# Interface IComparable

```
01  using System;
02  using System.Collections;
03  public class Employee : IComparable
04  {
05      private int empID;
06      public Employee(int empID)
07      {
08          this.empID = empID;
09      }
10      public override string ToString()
11      {
12          return empID.ToString();
13      }
14      public int EmpID
15      {
16          get
17          {
18              return empID;
19          }
20          set
21          {
22              this.empID = value;
23          }
24      }
```

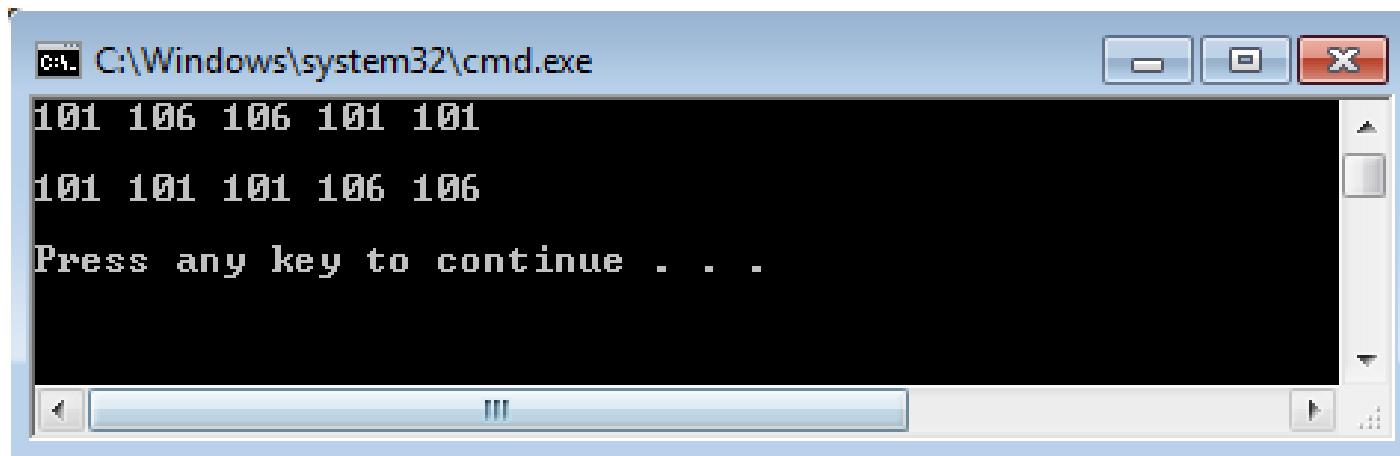
# Interface IComparable

```
25     public int CompareTo(Object o)
26     {
27         Employee r = (Employee)o;
28         return this.empID.CompareTo(r.empID) ;
29     }
30 }
31 public class Tester
32 {
33     static void Main()
34     {
35         ArrayList empArray = new ArrayList();
36         Random r = new Random();
37         for (int i = 0; i < 5; i++)
38         {
39             empArray.Add(new Employee(r.Next(10) + 100));
40         }
41 }
```

# Interface IComparable

```
41     // In tất cả nội dung của mảng
42     for (int i = 0; i < empArray.Count; i++)
43     {
44         Console.WriteLine("{0} ", empArray[i].ToString());
45     }
46     Console.WriteLine("\n");
47     // Sắp xếp lại mảng Employee dựa theo phương thức
48     CompareTo()
49     empArray.Sort();
50     // Hiển thị tất cả nội dung của mảng Employee
51     for (int i = 0; i < empArray.Count; i++)
52     {
53         Console.WriteLine("{0} ", empArray[i].ToString());
54     }
55 }
56 }
```

# Interface IComparable



# Interface IComparer

- ▶ Interface IComparer cung cấp phương thức Compare(), để so sánh hai phần tử trong một tập hợp có thứ tự.
- ▶ Phương thức Compare() thường được thực thi bằng cách gọi phương thức CompareTo() của một trong những đối tượng.
- ▶ Nếu chúng ta muốn tạo ra những lớp có thể được sắp xếp bên trong một tập hợp thì chúng ta cần thiết phải thực thi Comparable.

# Interface IComparer

```
001  using System;
002  using System.Collections;
003  public class Employee: IComparable
004  {
005      private int empID;
006      private int yearsOfSvc = 1;
007      public Employee(int empID)
008      {
009          this.empID = empID;
010      }
011      public Employee(int empID, int yearsOfSvc)
012      {
013          this.empID = empID;
014          this.yearsOfSvc = yearsOfSvc;
015      }
016      public override string ToString()
017      {
018          return "ID: " + empID.ToString() + ". Years of Svc: " +
yearsOfSvc.ToString();
019      }
020      // Phương thức tĩnh để nhận đối tượng Comparer
021      public static EmployeeComparer GetComparer()
022      {
023          return new Employee.EmployeeComparer();
024      }
```

# Interface IComparer

```
025     public int CompareTo(Object rhs)
026     {
027         Employee r = (Employee)rhs;
028         return this.empID.CompareTo(r.empID);
029     }
030     // Thực thi đặc biệt được gọi bởi custom comparer
031     public int CompareTo(Employee rhs,
032                           Employee.EmployeeComparer.ComparisionType which)
033     {
034         switch (which)
035         {
036             case Employee.EmployeeComparer.ComparisionType.EmpID:
037                 return this.empID.CompareTo(rhs.empID);
038             case Employee.EmployeeComparer.ComparisionType.Yrs:
039                 return this.yearsOfSvc.CompareTo(rhs.yearsOfSvc);
040         }
041         return 0;
042     }
043     // Lớp bên trong thực thi IComparer
044     public class EmployeeComparer : IComparer
045     {
046         private Employee.EmployeeComparer.ComparisionType
whichComparision;
```

# Interface IComparer

```
047     // Định nghĩa kiểu liệt kê
048     public enum ComparisionType
049     {
050         EmpID, Yrs
051     };
052     // Yêu cầu những đối tượng Employee tự so sánh với nhau
053     public int Compare(object lhs, object rhs)
054     {
055         Employee l = (Employee)lhs;
056         Employee r = (Employee)rhs;
057         return l.CompareTo(r, WhichComparision);
058     }
059     public Employee.EmployeeComparer.ComparisionType
WhichComparision
060     {
061         get
062         {
063             return whichComparision;
064         }
065         set
066         {
067             whichComparision = value;
068         }
069     }
070 }
071 }
072 }
```

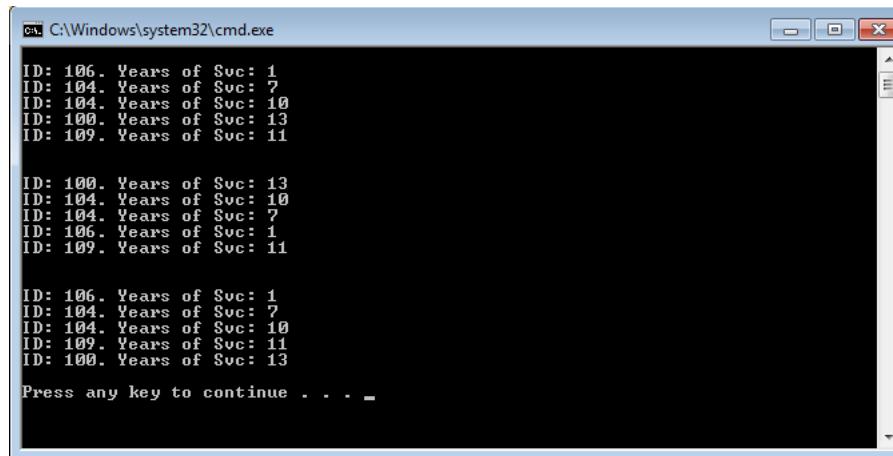
# Interface IComparer

```
073 public class Teser
074 {
075     static void Main()
076     {
077         ArrayList empArray = new ArrayList(); Random r = new
Random();
078         for (int i = 0; i < 5; i++)
079         {
080             empArray.Add(new Employee(r.Next(10) + 100,
r.Next(20)));
081         }
082         // Hiển thị tất cả nội dung của mảng Employee
083         for (int i = 0; i < empArray.Count; i++)
084         {
085             Console.WriteLine("\n{0} ", empArray[i].ToString());
086         }
087         Console.WriteLine("\n");
088         // Sắp xếp mảng theo empID
089         Employee.EmployeeComparer c = Employee.GetComparer();
090         c.WhichComparision =
Employee.EmployeeComparer.ComparisionType.EmpID;
091         empArray.Sort(c);
```

# Interface IComparer

```
092     // Hiển thị nội dung của mảng
093     for (int i = 0; i < empArray.Count; i++)
094     {
095         Console.WriteLine("\n{0} ", empArray[i].ToString());
096     }
097     Console.WriteLine("\n");
098     // Sắp xếp mảng theo yearsOfSvc
099     c.WhichComparision =
100     Employee.EmployeeComparer.ComparisionType.Yrs;
101     empArray.Sort(c);
102     // Hiển thị nội dung của mảng
103     for (int i = 0; i < empArray.Count; i++)
104     {
105         Console.WriteLine("\n{0} ", empArray[i].ToString());
106     }
107 }
108 }
```

# Interface IComparer



```
ID: 106. Years of Svc: 1
ID: 104. Years of Svc: 7
ID: 104. Years of Svc: 10
ID: 100. Years of Svc: 13
ID: 109. Years of Svc: 11

ID: 100. Years of Svc: 13
ID: 104. Years of Svc: 10
ID: 104. Years of Svc: 7
ID: 106. Years of Svc: 1
ID: 109. Years of Svc: 11

ID: 106. Years of Svc: 1
ID: 104. Years of Svc: 7
ID: 104. Years of Svc: 10
ID: 109. Years of Svc: 11
ID: 100. Years of Svc: 13

Press any key to continue . . .
```

# Interface IDictionary

Interface IDictionary là Interface đại diện chung cho kiểu tập hợp dùng cặp key và value

IDictionary cung cấp một thuộc tính public là Item. Thuộc tính Item cho phép truy cập phần tử trong tập hợp thông qua toán tử chỉ mục [] giống như truy cập mảng.

# IDictionaryEnumerator

Những đối tượng IDictionary cũng hỗ trợ vòng lặp **foreach** bằng việc thực thi phương thức Get Enumerator(), phương thức này trả về một IDictionaryEnumerator.

# ArrayList

Lớp ArrayList là một kiểu dữ liệu giống như kiểu mảng nhưng kích thước của nó có thể được thay đổi động theo yêu cầu.

Thuộc tính	Mô tả
Capacity	Thuộc tính để get hay set số phần trong ArrayList.
Count	Thuộc tính dùng để xác định số phần tử có trong ArrayList
IsFixedSize	Thuộc tính kiểm tra xem kích thước của ArrayList có cố định hay không
IsReadOnly	Thuộc tính kiểm tra xem ArrayList có thuộc tính chỉ đọc hay không.

# ArrayList

Phương thức	Mô tả
Add()	Phương thức public để thêm một đối tượng vào ArrayList
AddRange()	Phương thức public để thêm nhiều thành phần của một ICollection vào cuối của ArrayList
Clear()	Xóa tất cả các thành phần từ ArrayList
Clone()	Tạo một bản copy
Contains()	Kiểm tra một thành phần xem có chứa trong mảng hay không
CopyTo()	Phương thức public nạp chồng để sao chép một ArrayList đến một mảng một chiều.
GetEnumerator()	Phương thức public nạp chồng trả về một enumerator dùng để lặp qua mảng
Item()	Thiết lập hay truy cập thành phần trong mảng tại vị trí xác định. Đây là bộ chỉ mục cho lớp ArrayList.

# ArrayList

Phương thức	Mô tả
IndexOf()	Phương thức public nạp chồng trả về chỉ mục vị trí đầu tiên xuất hiện giá trị.
Insert()	Chèn một thành phần vào trong ArrayList
InsertRange(0)	Chèn một dãy tập hợp vào trong ArrayList
LastIndexOf()	Phương thức public nạp chồng trả về chỉ mục trị trí cuối cùng xuất hiện giá trị.
Remove()	Xóa sự xuất hiện đầu tiên của một đối tượng xác định.
RemoveAt()	Xóa một thành phần ở vị trí xác định.
RemoveRange()	Xóa một dãy các thành phần.
Reverse()	Đảo thứ tự các thành phần trong mảng.
SetRange()	Sao chép những thành phần của tập hợp qua dãy những thành phần trong ArrayList.
Sort()	Sắp xếp ArrayList.
ToArray()	Sao chép những thành phần của ArrayList đến một mảng mới.
TrimToSize()	Thiết lập kích thước thật sự chứa các thành phần trong ArrayList

# ArrayList

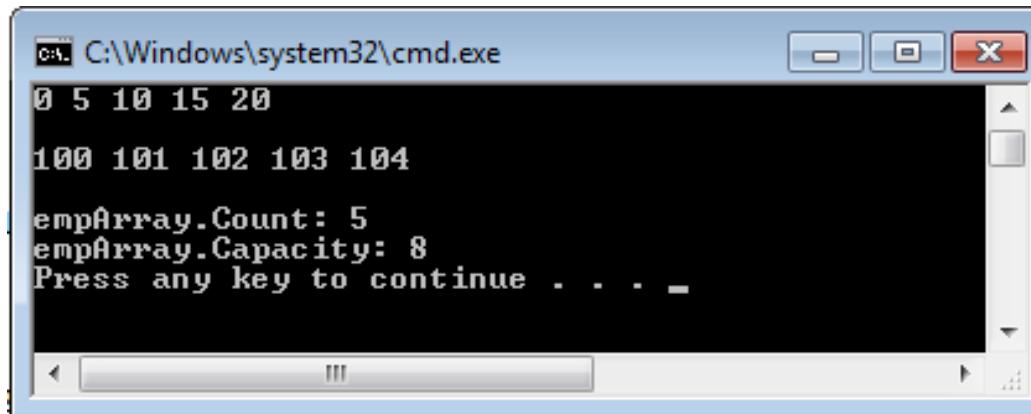
```
01  using System;
02  using System.Collections;
03  public class Employee
04  {
05      private int empID;
06      public Employee(int empID)
07      {
08          this.empID = empID;
09      }
10      public override string ToString()
11      {
12          return empID.ToString();
13      }
14      public int EmpID
15      {
16          get
17          {
18              return empID;
19          }
20          set
21          {
22              empID = value;
23          }
24      }
25 }
```

# ArrayList

```
26 class Program
27 {
28     static void Main(string[] args)
29     {
30         ArrayList empArray = new ArrayList();
31         ArrayList intArray = new ArrayList();
32         // đưa vào mảng
33         for (int i = 0; i < 5; i++)
34         {
35             empArray.Add(new Employee(i + 100));
36             intArray.Add(i * 5);
37         }
38         // in tất cả nội dung
39         for (int i = 0; i < intArray.Count; i++)
40         {
41             Console.WriteLine("{0} ", intArray[i].ToString());
42         }
43         Console.WriteLine("\n");
```

# ArrayList

```
44     // in tất cả nội dung của mảng
45     for (int i = 0; i < empArray.Count; i++)
46     {
47         Console.WriteLine("{0} ", empArray[i].ToString());
48     }
49     Console.WriteLine("\n");
50     Console.WriteLine("empArray.Count: {0}", empArray.Count);
51     Console.WriteLine("empArray.Capacity: {0}",
52                     empArray.Capacity);
53 }
```



# Queue (Hàng đợi)

Hàng đợi là một tập hợp trong đó có thứ tự vào trước và ra trước (FIFO).

Thuộc tính	Mô tả
Count	Thuộc tính trả về số thành phần trong hàng đợi
IsReadOnly	Thuộc tính xác định hàng đợi là chỉ đọc
IsSynchronized	Thuộc tính xác định hàng đợi được đồng bộ
SyncRoot	Thuộc tính trả về đối tượng có thể được sử dụng để đồng bộ truy cập Queue.

# Queue (Hàng đợi)

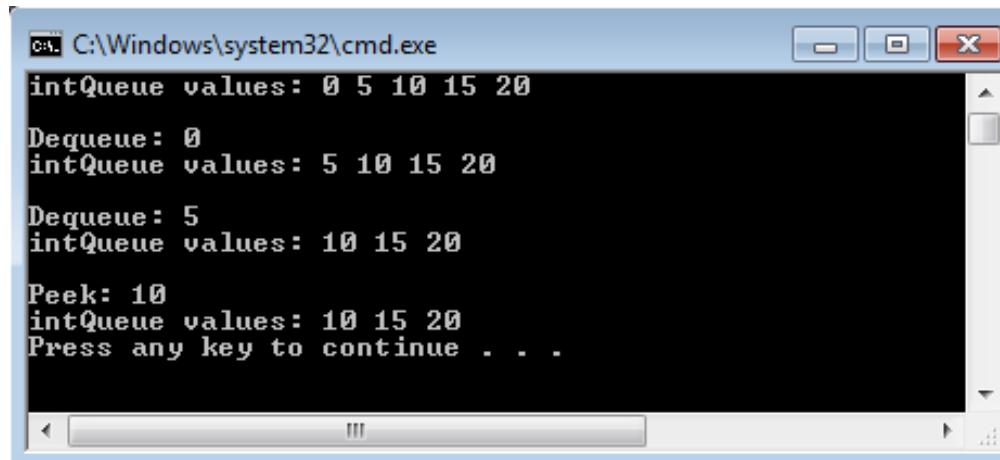
Phương thức	Mô tả
Clear()	Xóa tất cả các thành phần trong hàng đợi
Clone()	Tạo ra một bản sao
Contains()	Xác định xem một thành phần có trong hàng đợi.
CopyTo()	Sao chép những thành phần của hàng đợi đến mảng một chiều đã tồn tại
Dequeue()	Xóa và trả về thành phần bắt đầu của hàng đợi.
Enqueue()	Thêm một thành phần vào hàng đợi.
GetEnumerator()	Trả về một enumerator cho hàng đợi.
Peek()	Trả về phần tử đầu tiên của hàng đợi và không xóa nó.
ToArray()	Sao chép những thành phần qua một mảng mới

# Queue (Hàng đợi)

```
01  using System;
02  using System.Collections;
03  class Program
04  {
05      public static void Main()
06      {
07          Queue intQueue = new Queue();
08          // đưa vào trong mảng
09          for(int i=0; i <5; i++)
10          {
11              intQueue.Enqueue(i * 5);
12          }
13          // hiển thị hàng đợi
14          Console.Write("intQueue values: "); PrintValues(
15              intQueue);
16          // xóa thành phần ra khỏi hàng đợi
17          Console.WriteLine("\nDequeue: {0}", intQueue.Dequeue());
18          // hiển thị hàng đợi
19          Console.Write("intQueue values: "); PrintValues(intQueue);
20          // xóa thành phần khỏi hàng đợi
21          Console.WriteLine("\nDequeue: {0}", intQueue.Dequeue());
22          // hiển thị hàng đợi
23          Console.Write("intQueue values: "); PrintValues(intQueue);
24          // xóa thành phần đầu tiên trong hàng đợi.
25          Console.WriteLine("\nPeek: {0}", intQueue.Peek());
```

# Queue (Hàng đợi)

```
25     // hiển thị hàng đợi
26     Console.WriteLine("intQueue values: "); PrintValues(intQueue);
27 }
28 public static void PrintValues(IEnumerable myCollection)
29 {
30     foreach (Object obj in myCollection)
31         Console.Write("{0} ", obj);
32     Console.WriteLine();
33 }
34 }
```



# Stack (Ngăn xếp)

- ← Ngăn xếp là một tập hợp mà thứ tự là vào trước ra sau hay vào sao ra trước (LIFO)
- ← Hai phương thức chính cho việc thêm và xóa từ stack là Push và Pop, ngoài ra ngăn xếp cũng đưa ra phương thức Peek tương tự như Peek trong hàng đợi.

Thuộc tính	Mô tả
Count	Thuộc tính trả về số thành phần trong ngăn xếp
IsReadOnly	Thuộc tính xác định ngăn xếp là chỉ đọc
IsSynchronized	Thuộc tính xác định ngăn xếp được đồng bộ
SyncRoot	Thuộc tính trả về đối tượng có thể được sử dụng để đồng bộ truy cập Stack.

# Stack (Ngăn xếp)

Phương thức	Mô tả
Clear()	Xóa tất cả các thành phần trong ngăn xếp
Clone()	Tạo ra một bản sao
Contains()	Xác định xem một thành phần có trong ngăn xếp.
CopyTo()	Sao chép những thành phần của ngăn xếp đến mảng một chiều đã tồn tại
Pop()	Xóa và trả về phần tử đầu Stack
Push()	Đưa một đối tượng vào đầu ngăn xếp
Peek()	Trả về phần tử đầu tiên của ngăn xếp và không xóa nó.
ToArray()	Sao chép những thành phần qua một mảng mới

# Stack (Ngăn xếp)

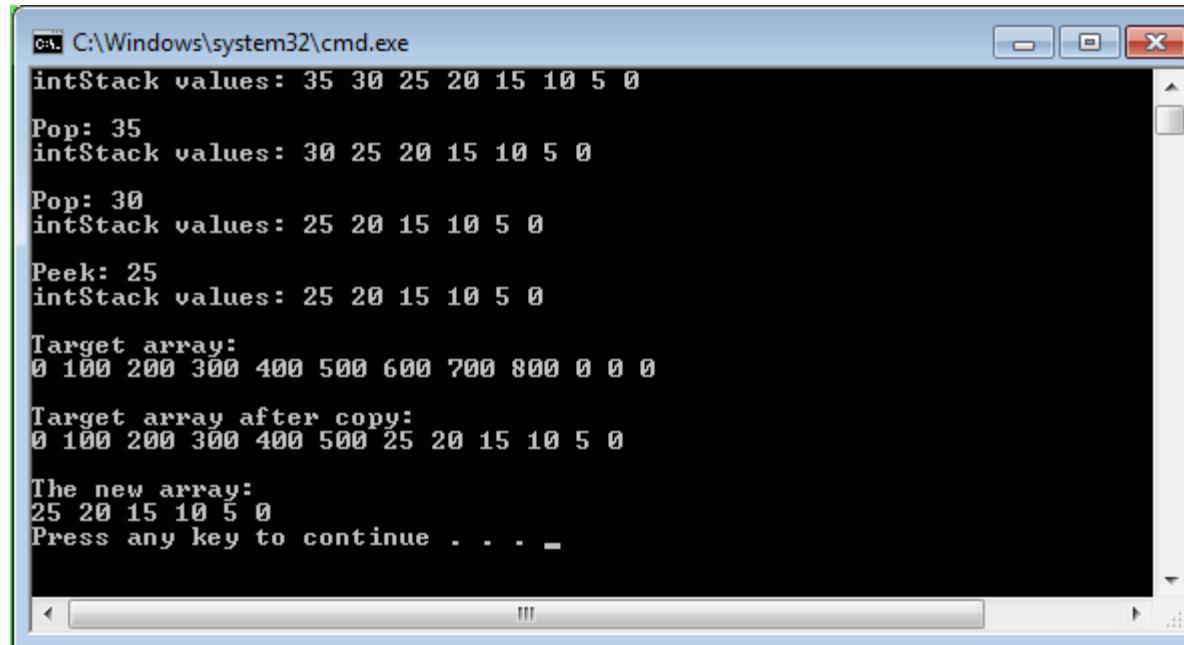
```
01  using System;
02  using System.Collections;
03  public class Program
04  {
05      static void Main()
06      {
07          Stack intStack = new Stack();
08          // đưa vào ngăn xếp
09          for (int i = 0; i < 8; i++)
10          {
11              intStack.Push(i * 5);
12          }
13          // hiển thị stack
14          Console.Write("intStack values: "); PrintValues( intStack );
15          // xóa phần tử đầu tiên
16          Console.WriteLine("\nPop: {0}", intStack.Pop());
17          // hiển thị stack
18          Console.Write("intStack values: "); PrintValues( intStack );
19          // xóa tiếp phần tử khác
20          Console.WriteLine("\nPop: {0}", intStack.Pop());
21          // hiển thị stack
22          Console.Write("intStack values: "); PrintValues( intStack );
        );
```

# Stack (Ngăn xếp)

```
23     // xem thành phần đầu tiên stack
24     Console.WriteLine("\nPeek: {0}", intStack.Peek());
25     // hiển thị stack
26     Console.Write("intStack values: "); PrintValues( intStack );
27     // khai báo mảng với 12 phần tử
28     Array targetArray = Array.CreateInstance(typeof(int), 12);
29     for (int i = 0; i <= 8; i++)
30     {
31         targetArray.SetValue(100 * i, i);
32     }
33     // hiển thị giá trị của mảng
34     Console.WriteLine("\nTarget array: "); PrintValues(
targetArray );
35     // chép toàn bộ stack vào mảng tại vị trí 6
36     intStack.CopyTo( targetArray, 6 );
37     // hiển thị giá trị của mảng sau copy
38     Console.WriteLine("\nTarget array after copy: ");
PrintValues( targetArray );
39     // chép toàn bộ stack vào mảng mới
40     Object[] myArray = intStack.ToArray();
41     // hiển thị giá trị của mảng mới
42     Console.WriteLine("\nThe new array: "); PrintValues(
myArray );
43 }
```

# Stack (Ngăn xếp)

```
44     public static void PrintValues(IEnumerable myCollection)
45     {
46         foreach (Object obj in myCollection)
47             Console.Write("{0} ", obj);
48         Console.WriteLine();
49     }
50 }
```



# Hashtables

- ← Hashtable là một kiểu từ điển trong đó có hai thành phần chính liên hệ với nhau là key và value
- ← Hashtable là kiểu từ điển đã được tối ưu cho phép việc truy cập nhanh chóng.

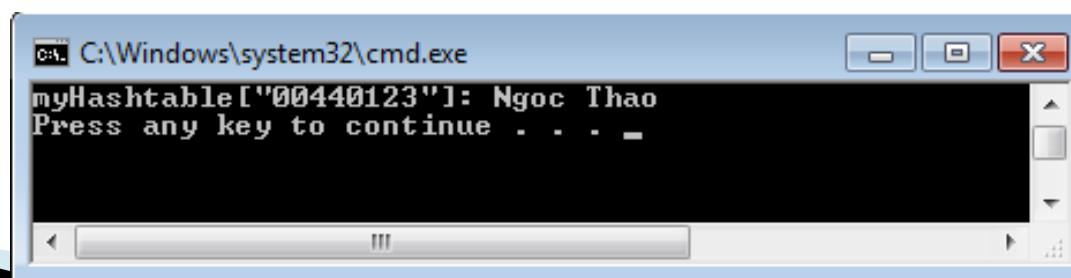
Thuộc tính	Mô tả
Count	Thuộc tính trả về số thành phần trong hashtable
IsReadOnly	Thuộc tính xác định hashtable là chỉ đọc
Keys	Thuộc tính trả về một ICollection chứa những khóa trong hashtable.
Values	Thuộc tính trả về một ICollection chứa những giá trị trong hashtable.

# Hashtables

Phương thức	Mô tả
Add()	Thêm một thành phần mới với khóa và giá trị xác định.
Clear()	Xóa tất cả đối tượng trong hashtable.
Item()	Chỉ mục cho hastable
Clone()	Tạo ra một bản sao
Contains()	Xác định xem một thành phần có trong hashtable.
ContainsKey()	Xác định xem hashtable có chứa một khóa xác định
CopyTo()	Sao chép những thành phần của hashtable đến mảng một chiều đã tồn tại
GetEnumerator()	Trả về một enumerator cho hashtable.
Remove()	Xóa một thành phần với khóa xác định.

# Hashtables

```
01  using System;
02  using System.Collections;
03  public class Tester
04  {
05      static void Main()
06      {
07          // tạo và khởi tạo hashtable
08          Hashtable hashTable = new Hashtable();
09          hashTable.Add("00440123", "Ngoc Thao");
10          hashTable.Add("00123001", "My Tien");
11          hashTable.Add("00330124", "Thanh Tung");
12          // truy cập qua thuộc tính Item
13          Console.WriteLine("myHashtable[\"00440123\"]: {0}",
14              hashTable["00440123"]);
15      }
16  }
```



# System.IO

- ← Thư viện System.IO cung cấp nhiều lớp dùng cho việc đọc, ghi file cũng như việc thao tác với file và thư mục
- ← Một số lớp chính của System.IO

# DriveInfo

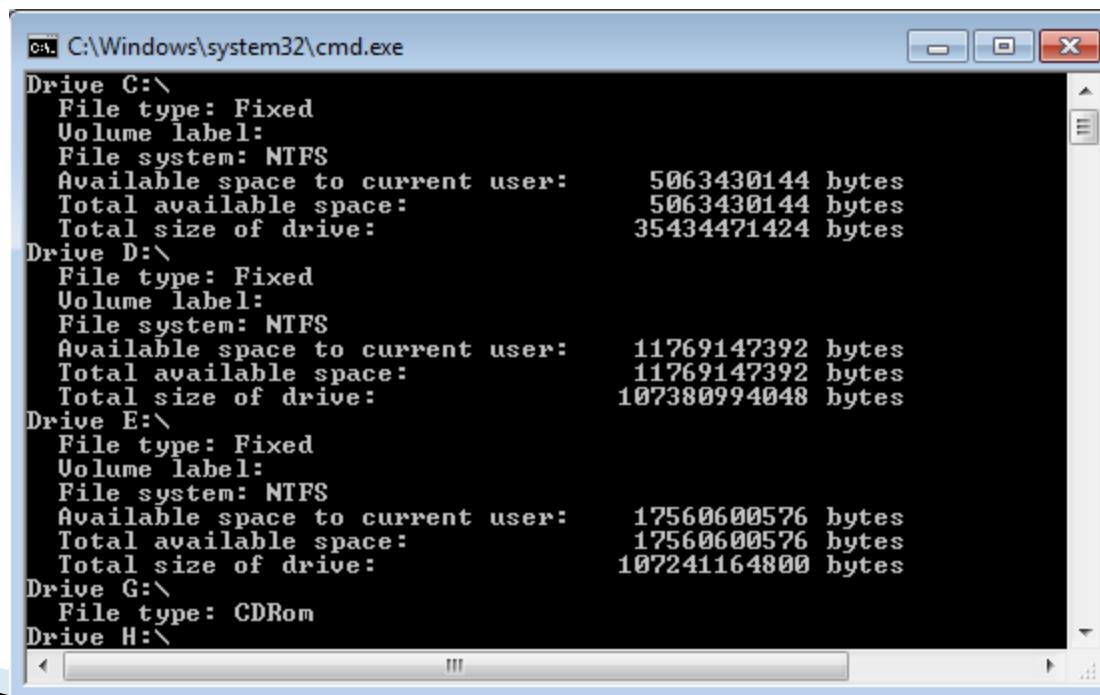
Thuộc tính/ Phương thức cơ bản	Mô tả
DriveFormat	Tên của file system, ví dụ: NTFS/ FAT32
DriveType	Cho biết loại ổ đĩa. Kiểu dữ liệu trả về là DriveType: CDRom, Fixed, Network, Ram, Removable, Unknown
IsReady	Cho biết ổ đĩa đã sẵn sàng Read/Write.
Name	Tên ổ đĩa
TotalFreeSpace	Xem dung lượng đĩa trống
TotalSize	Xem tổng dung lượng đĩa
GetDrives()	Lấy danh sách ổ đĩa hiện có

# DriveInfo

```
01  using System;
02  using System.IO;
03  class Test
04  {
05      public static void Main()
06      {
07          DriveInfo[] allDrives = DriveInfo.GetDrives();
08          foreach (DriveInfo d in allDrives)
09          {
10              Console.WriteLine("Drive {0}", d.Name);
11              Console.WriteLine(" File type: {0}", d.DriveType);
12              if (d.IsReady == true)
13              {
14                  Console.WriteLine(" Volume label: {0}",
15                  d.VolumeLabel);
16                  Console.WriteLine(" File system: {0}",
17                  d.DriveFormat);
18                  Console.WriteLine(
19                      " Available space to current user:{0, 15}
20                      bytes",
21                      d.AvailableFreeSpace);
```

# DriveInfo

```
19     Console.WriteLine(
20         " Total available space:      {0, 15} bytes",
21         d.TotalFreeSpace);
22     Console.WriteLine(
23         " Total size of drive:      {0, 15} bytes",
24         d.TotalSize);
25 }
26 }
27 }
28 }
```



# DirectoryInfo

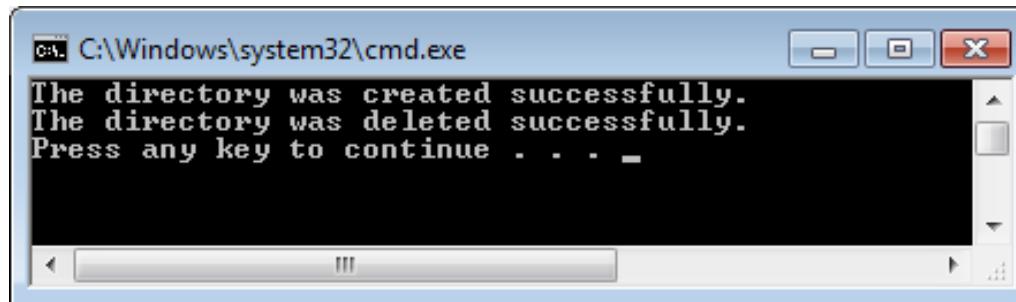
Thuộc tính/ phương thức cơ bản	Mô tả
CreationTime	Xem hoặc thiết lập thời gian tạo thư mục
Exists	Kiểm tra thư mục tồn tại trên ổ đĩa
FullName	Lấy đường dẫn của tới thư mục
LastAccessTime	Cho biết thời gian cuối cùng thư mục (file) được truy cập
Name	Cho biết tên của thư mục
Parent	Trả về thư mục cha.
FileAttributes Attributes	Cho biết thuộc tính của thư mục (file) FileAttributes là 1 enum gồm các giá trị như: Directory, Readonly, Hidden,...
Create()	Tạo thư mục
Delete()	Xóa thư mục
MoveTo()	Di chuyển thư mục
GetDirectories()	Lấy các thư mục con trong thư mục
GetFiles	Lấy tất cả các tập tin trong thư mục

# DirectoryInfo

```
01  using System;
02  using System.IO;
03  class Test
04  {
05      public static void Main()
06      {
07          // Specify the directories you want to manipulate.
08          DirectoryInfo di = new DirectoryInfo(@"c:\MyDir");
09          try
10          {
11              // Determine whether the directory exists.
12              if (di.Exists)
13              {
14                  // Indicate that the directory already exists.
15                  Console.WriteLine("That path exists already.");
16                  return;
17              }
18              // Try to create the directory.
19              di.Create();
20              Console.WriteLine("The directory was created
successfully.");
```

# DirectoryInfo

```
21          // Delete the directory.  
22          di.Delete();  
23          Console.WriteLine("The directory was deleted  
24          successfully.");  
25      }  
26      catch (Exception e)  
27      {  
28          Console.WriteLine("The process failed: {0}",  
29          e.ToString());  
30      }  
31  }
```



# FileInfo

Thuộc tính/Phương thức cơ bản	Mô tả
CreationTime	Xem hoặc thiết lập thời gian tạo thư mục
Exists	Kiểm tra thư mục tồn tại trên ổ đĩa
Directory	Trả về đối tượng thư mục cha
DirectoryName	Trả về chuỗi đường dẫn (full path) của thư mục cha
Extension	Trả về tên đuôi file (txt,bat,exe,...)
Name	Cho biết tên của file
Attributes	Cho biết thuộc tính của file
CopyTo()	Copy file đến 1 nơi khác
Create()	Tạo file
Delete()	Xóa file
MoveTo()	Di chuyển file hoặc đổi tên file
CreateText	Tạo StreamWriter để ghi file
OpenText	Tạo StreamReader để đọc file
ReplaceFile	Thay đổi nội dung file

# FileInfo

```
01  using System;
02  using System.IO;
03  class Test
04  {
05      public static void Main()
06      {
07          string path = Path.GetTempFileName();
08          FileInfo fil = new FileInfo(path);
09          //Create a file to write to.
10          using (StreamWriter sw = fil.CreateText())
11          {
12              sw.WriteLine("Hello");
13              sw.WriteLine("And");
14              sw.WriteLine("Welcome");
15          }
16          //Open the file to read from.
17          using (StreamReader sr = fil.OpenText())
18          {
19              string s = "";
20              while ((s = sr.ReadLine()) != null)
21              {
22                  Console.WriteLine(s);
23              }
24          }
25      }
26  }
```

# FileInfo

```
25      try
26      {
27          string path2 = Path.GetTempFileName();
28          FileInfo fi2 = new FileInfo(path2);
29          //Ensure that the target does not exist.
30          fi2.Delete();
31          //Copy the file.
32          fil.CopyTo(path2);
33          Console.WriteLine("{0} was copied to {1}.", path,
34                           path2);
35          //Delete the newly created file.
36          fi2.Delete();
37          Console.WriteLine("{0} was successfully deleted.",
38                           path2);
39      }
40      catch (Exception e)
41      {
42          Console.WriteLine("The process failed: {0}",
43                           e.ToString());
44      }
45 }
```

# Xử lý đọc/ghi file

- ← Đọc và viết dữ liệu sẽ được thực hiện thông qua lớp Stream.
- ← Stream là 1 luồng dữ liệu, nó đưa dữ liệu từ điểm bắt đầu đến điểm cuối.
- ← System.IO.Stream là một lớp abstract định nghĩa một số thành viên có khả năng hỗ trợ việc đọc/viết đồng bộ (synchronous) hoặc không đồng bộ (asynchronous) đối với khối trữ tin

# Stream Class

Thuộc tính/Phương thức cơ bản	Mô tả
CanRead	Luồng có hỗ trợ đọc
CanSeek	Luồng có hỗ trợ di chuyển con trỏ
CanTimeOut	Xác định xem luồng có timeout hay không
CanWrite	Luồng có hỗ trợ ghi
Length	Chiều dài (theo bytes) của luồng
ReadTimeout	Thiết lập timeout cho phương thức Read
WriteTimeout	Thiết lập timeout cho phương thức Write
Position	Lấy hoặc xác lập vị trí con trỏ trong luồng
Close()	Đóng luồng và giải phóng tài nguyên
Flush()	Đẩy toàn bộ dữ liệu buffer trong luồng lên trên thiết bị
Read()	Thực thi phương thức đọc mảng byte trên luồng.
Seek()	Di chuyển vị trí con trỏ đọc
Write()	Ghi mảng byte lên trên luồng

# FileStream Class

Lớp FileStream là lớp dẫn xuất từ lớp Stream. FileStream có một số phương thức và thuộc tính riêng.

Thuộc tính/Phương thức cơ bản	Mô tả
Name	Lấy tên của file
Lock()	Khóa file, tránh truy xuất đồng thời lên File
Unlock	Mở khóa file, có thể truy xuất đồng thời lên file

# StreamReader

StreamReader có thể dùng để đọc văn bản

Thuộc Tính	Mô tả
BaseStream	Trả về luồng đọc
CurrentEncoding	Lấy thông tin định dạng của luồng đang sử dụng
EndOfStream	Xác định con trỏ đọc đến cuối luồng chưa

Phương thức	Mô tả
Close	Đóng luồng và giải phóng tài nguyên
Peek	Trả về giá trị kí tự tiếp theo trong luồng, không di chuyển con trỏ đọc.
Read	Thực thi phương thức đọc mảng các kí tự trên luồng.
ReadBlock	Đọc khối kí tự tiếp theo trên luồng.
ReadLine	Đọc nguyên dòng trên luồng
ReadToEnd	Đọc tất cả các kí tự tới cuối luồng

# Ví dụ

```
01  using System;
02  using System.IO;
03  class Test
04  {
05      public static void Main()
06      {
07          try
08          {
09              // Tạo một StreamReader để đọc file
10              using (StreamReader sr = new StreamReader("TestFile.txt"))
11              {
12                  string line;
13                  // Đọc từng dòng của File
14                  while ((line = sr.ReadLine()) != null)
15                  {
16                      Console.WriteLine(line);
17                  }
18              }
19          }
20          catch (Exception e)
21          {
22              // Hiển thị thông điệp lỗi
23              Console.WriteLine("The file could not be read:");
24              Console.WriteLine(e.Message);
25          }
26      }
27  }
```

# **StreamWriter**

**StreamWriter có thể dùng để ghi văn bản**

<b>Thuộc tính</b>	<b>Mô tả</b>
<b>AutoFlush</b>	<b>Thiết lập cơ chế tự động Flush, sau mỗi lệnh Write</b>
<b>BaseStream</b>	<b>Trả về luồng bên dưới</b>
<b>Encoding</b>	<b>Lấy chế độ mã hóa hiện hành của luồng</b>

<b>Phương thức</b>	<b>Mô tả</b>
<b>Close</b>	<b>Đóng luồng và giải phóng tài nguyên</b>
<b>Write</b>	<b>Ghi vào luồng</b>
<b>WriteLine</b>	<b>Ghi một chuỗi kí tự vào luồng và xuống hàng</b>

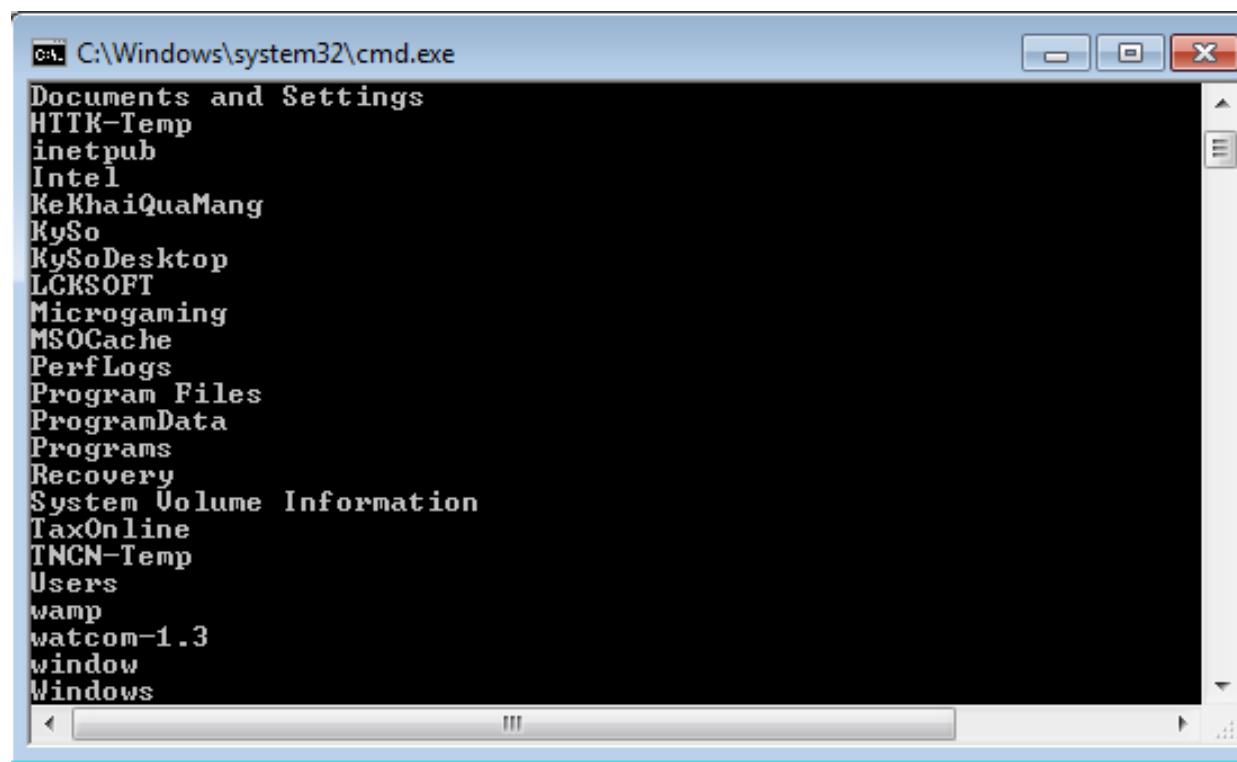
# Ví dụ

```
01  using System;
02  using System.IO;
03  class Program
04  {
05      static void Main(string[] args)
06      {
07          // Lấy các thư mục hiện hành trên ổ đĩa
08          DirectoryInfo[] cDirs = new
09          DirectoryInfo(@"c:\").GetDirectories();
10          // Viết tên các thư mục vào file
11          using (StreamWriter sw = new
12          StreamWriter("CDriveDirs.txt"))
13          {
14              foreach (DirectoryInfo dir in cDirs)
15              {
16                  sw.WriteLine(dir.Name);
17              }
18          }
19      }
20  }
```

# Ví dụ

```
18     // Đọc và hiển thi tên thư mục từ file
19     string line = "";
20     using (StreamReader sr = new
21         StreamReader("CDriveDirs.txt"))
22     {
23         while ((line = sr.ReadLine()) != null)
24         {
25             Console.WriteLine(line);
26         }
27     }
28 }
29 }
```

# Ví dụ



# BinaryReader và BinaryWriter

Tương tự như StreamReader và StreamWriter,  
BinaryReader và BinaryWriter có thể dùng để đọc file nhị  
phân

```
FileStream theFile =
File.Open(@"c:\somefile.bin", FileMode.Open);
BinaryReader reader = new BinaryReader(theFile);
long number = reader.ReadInt64(); byte[] bytes =
reader.ReadBytes(4); string s = reader.ReadString();
reader.Close();

FileStream theFile = File.Open(@"c:\somefile.bin",
    FileMode.OpenOrCreate, FileAccess.Write);
BinaryWriter writer = new BinaryWriter(theFile);
long number = 100;

byte[] bytes = new byte[] { 10, 20, 50, 100 };
string s = "Toi di hoc";
writer.Write(number); writer.Write(bytes);
writer.Write(s);
```

# **BufferedStream**

**BufferedStream thường được sử dụng để tăng hiệu quả đọc ghi dữ liệu**

```
FileStream newFile = File.Create(@"c:\test.txt");
BufferedStream buffered = new BufferedStream(newFile);
StreamWriter writer = new StreamWriter(buffered);
writer.WriteLine("Some data");
streamWriter.Close();
bufferedStream.Close();
fileStream.Close();
```

# Bài tập

Xây dựng giao diện ISoSanh có tính năng so sánh. Xây dựng lớp Sinh viên (Họ tên, ĐTB) hiện thực giao diện ISoSanh, biết 2 sinh viên so sánh với nhau dựa trên ĐTB.

Viết chương trình nhập, xuất danh sách sinh viên. Xuất danh sách sinh viên theo điểm tăng dần.