

LOGO

NGÔN NGỮ C#

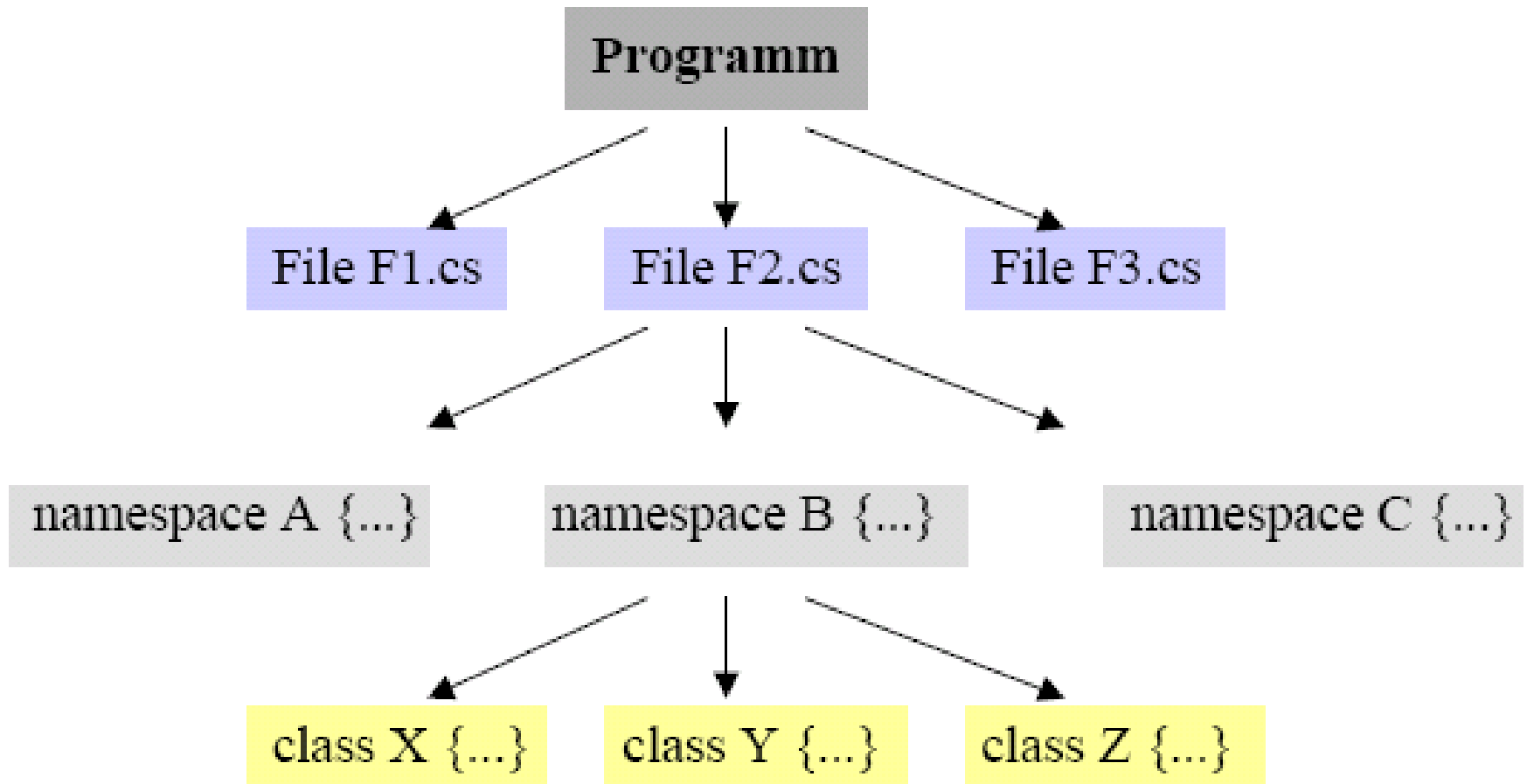
C#

- ❖ Ngôn ngữ lập trình “thuần” hướng đối tượng
- ❖ 70% Java, 10% C++, 5% Visual Basic, 15% mới
- ❖ Trình biên dịch C# là một trong những trình biên dịch hiệu quả nhất trong dòng sản phẩm .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 sơ liệu trực tiếp bên trong mã nguồn (dùng tool mã nguồn mở **NDoc** phát sinh ra sơ liệu)
- ❖ Hỗ trợ khái niệm interface (tương tự java)
- ❖ Cơ chế tự động dọn rác (tương tự java)
- ❖ Truyền tham số kiểu: in(ø), out, ref

Cấu trúc chương trình C#



Hello World

```
using System;

class Hello
{
    public static void Main()
    {
        Console.WriteLine("Hello, World");
    }
}
```

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 1

```
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

Ví dụ 2

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

Ví dụ 3

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

```
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);
```

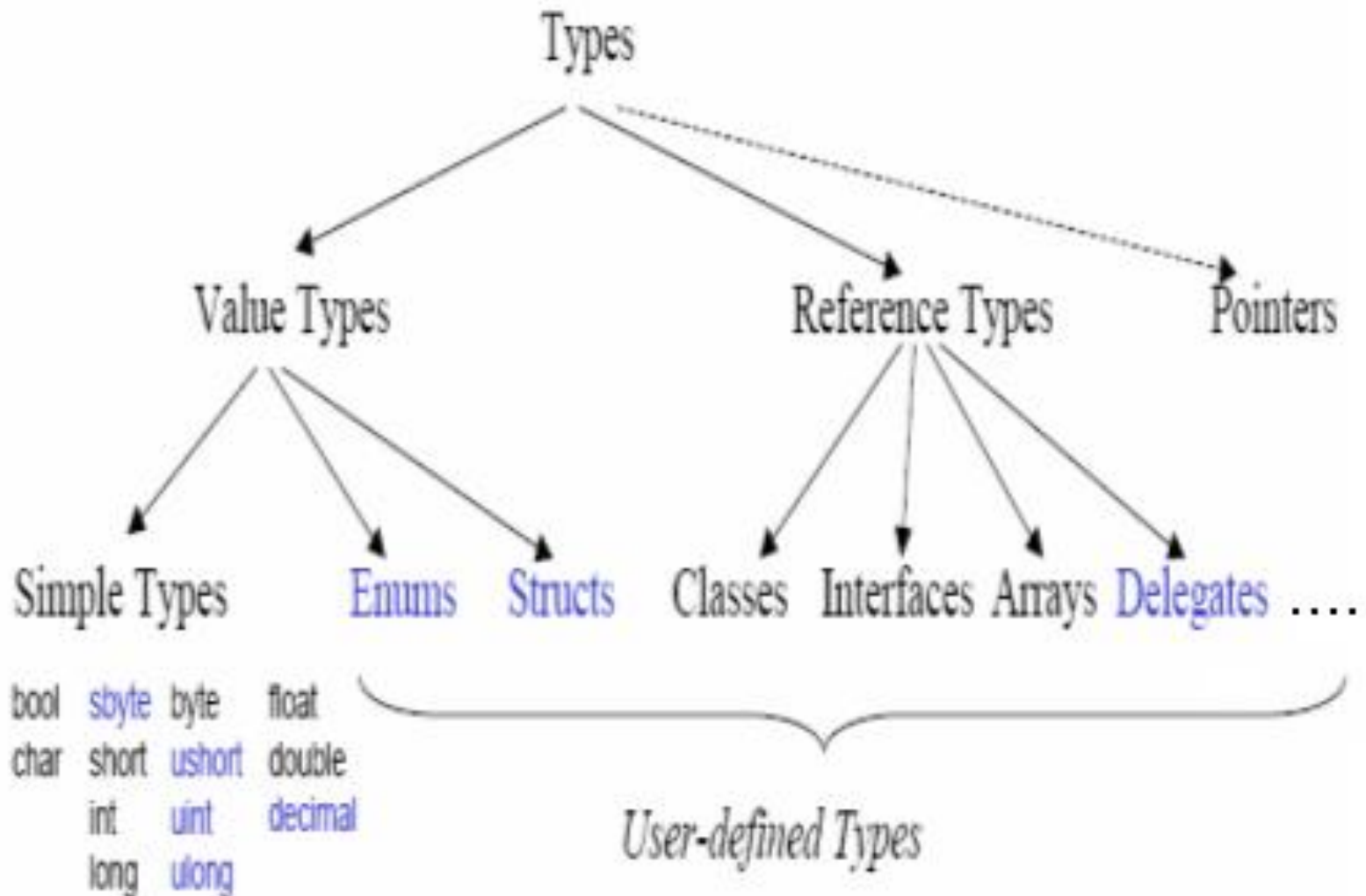
Console.ReadLine()

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();
```

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

A 105

B 55

A ? → 105

B ? → 55

A ? → 105

B ? ↗ 105

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

On the heap

i



`int i=123;`

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 tắt 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 khóa này thường sử dụng tham số là mảng.

Từ Khóa 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  
}
```



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;  
}
```

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

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 < 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 < args.Length; i++) Console.WriteLine(args[i]); }</pre>

Tóm tắt

Statement	Example
foreach statement	<pre>static void Main(string[] args) { foreach (string 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 < 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 < 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 > 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 >

}

Ví dụ

```
using System;
public class ThoiGian
{
    public void ThoiGianHienHanh()
    {
        Console.WriteLine("Hien thi thoi gian hien hanh");
    }
    // Các biến thành viên
    int Nam;
    int Thang;
    int Ngay;
    int Gio;
    int Phut;
    int Giay;
}
```

```
public class Tester
{
    static void Main()
    {
        ThoiGian t = new ThoiGian();
        t.ThoiGianHienHanh();
    }
}
```

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

Khởi tạo giá trị cho thuộc tính

```
public class ThoiGian
{
    public void ThoiGianHienHanh()
    {
        System.DateTime now = System.DateTime.Now;
        System.Console.WriteLine("\n Hien tai: \t {0}/{1}/{2}
                                   {3}:{4}:{5}",now.Day, now.Month, now.Year,
                                   now.Hour, now.Minute, now.Second);
        System.Console.WriteLine(" Thoi Gian:\t {0}/{1}/{2}
                                   {3}:{4}:{5}",
                                   Ngay, Thang, Nam, Gio, Phut, Giay);
    }
    public ThoiGian( System.DateTime dt)
    {
        Nam = dt.Year; Thang = dt.Month;Ngay = dt.Day;
        Gio = dt.Hour;Phut = dt.Minute;
        Giay = dt.Second;
    }
}
```

Khởi tạo giá trị cho thuộc tính

```
public ThoiGian(int Year, int Month, int Date, int Hour, int Minute)
{
    Nam = Year;Thang = Month;Ngay = Date;
    Gio = Hour;Phut = Minute;
}
private int Nam;
private int Thang;
private int Ngay;
private int Gio;
private int Phut;
private int Giay = 30 ; // biến được khởi tạo.
}
```


Khởi tạo giá trị cho thuộc tính

```
public class Tester
{
    static void Main()
    {
        System.DateTime currentTime =
            System.DateTime.Now;
        ThoiGian t1 = new ThoiGian( currentTime );
        t1.ThoiGianHienHanh();
        ThoiGian t2 = new ThoiGian(2001,7,3,10,5);
        t2.ThoiGianHienHanh();
    }
}
```

Phương thức khởi tạo

Hàm tạo mặc định: giống C++

Hàm tạo có đối số: tương tự C++ nhưng **không có tham số mặc định**

```
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;
}
```

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 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
- ❖ C# không cho phép truy cập đến các phương thức tĩnh và các biến thành viên tĩnh thông qua một thể hiện.
- ❖ **Không có friend**
- ❖ Phương thức tĩnh hoạt động ít nhiều giống như phương thức toàn cục

Thành viên static

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 int 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 int 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;
    }
}
```


Hướng đối tượng

```

public class BankAccount {
    protected string ID;
    protected string Owner;
    protected decimal _Balance;
    public BankAccount(string ID, string Owner) {
        this.ID = ID;
        this.Owner = Owner;
        this._Balance = 0;
    }
    public void Deposit(decimal Amount) {
        _Balance+=Amount;
    }
    public void Withdraw(decimal Amount) {
        _Balance-=Amount; // what if Amount > Balance?
    }
    public decimal Balance {
        get {
            return _Balance;
        }
    }
};

```

Fields

Thuộc tính chỉ đọc
Read-only property

Hướng đối tượng

```
class Program
{
    static void Main(string[] args)
    {
        BankAccount myAcct = new Account(
            "100120023003", "Nguyen Van An");
        myAcct.Deposit(1000);
        myAcct.Withdraw(100);
        Console.WriteLine("Balance: {0}", myAcct.Balance);
        //myAcct.Balance=10000;
        Console.ReadLine();
    }
}
```

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 Marks[string SubjectID] {
        get {
            return MarkDB.GetMark(StudentID, SubjectID);
        }
        set {
            MarDB.UpdateMark(StudentID, value);
        }
    }
}
```

...

```
Console.WriteLine("Physic mark: {0}", chau.Marks["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
}
```

```

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;
}

```



```

public class ListBox: Window
{
    // Khởi dựng có tham số
    public ListBox(int top, int left, string theContents) : base(top, left)
        //gọi khởi dựng 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;
}

```

```
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 ảo tính 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 overridden 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**

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.

```
public class SomeClass
{
    public SomeClass(int val)
    {
        value = val;
    }
    public override string ToString()
    {
        return value.ToString();
    }
    private int value;
}
```

```
public class Tester
{
    static void Main()
    {
        int i = 5;
        Console.WriteLine("The value of i is: {0}", i.ToString());
        SomeClass s = new SomeClass(7);
        Console.WriteLine("The value of s is {0}", s.ToString());
        Console.WriteLine("The value of 5 is {0}", 5.ToString());
    }
}
```

Lớp trong lớp

```
class Nguoi
```

```
{  
    public class Date {  
        private int ngay;  
        private int thang;  
        public Date() { ngay = 1; thang = 1; }  
        public void Xuat() {Console.WriteLine(ngay + "/" + thang); }  
    }  
    private string ten;  
    private string ho;  
    private Date ns;  
    public Nguoi() { ten = "An"; ho = "Nguyen Van"; ns = new Date(); }  
    public void Xuat()  
    {  
        ns.Xuat();Console.WriteLine(ho + " " + ten);  
    }  
}
```

Lớp trong lớp

```
class Program
{
    static void Main(string[] args)
    {
        Nguoi a=new Nguoi();
        a.Xuat();
        ConsoleApplication7.Nguoi.Date ns = new
            ConsoleApplication7.Nguoi.Date();
        ns.Xuat();
    }
}
```

```
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;
}
```



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

```
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
```

```
public class Phanso
{
    public Phanso(int ts, int ms)
    {
        this.ts = ts;
        this.ms = ms;
    }
    public Phanso(int wholeNumber)
    {
        ts = wholeNumber;
        ms = 1;
    }
    public static implicit operator Phanso(int theInt)
    {
        return new Phanso(theInt);
    }
}
```



```
public static explicit operator int(Phanso thePhanso)
{
    return thePhanso.ts / thePhanso.ms;
}
public static bool operator ==(Phanso lhs, Phanso rhs)
{
    if (lhs.ts == rhs.ts && lhs.ms == rhs.ms)
    {
        return true;
    }
    return false;
}
public static bool operator !=(Phanso lhs, Phanso rhs)
{
    return !(lhs == rhs);
}
```

```

public override bool Equals(object o)
{
    if (!(o is Phanso))
    {
        return false;
    }
    return this == (Phanso)o;
}
public static Phanso operator +(Phanso lhs, Phanso rhs)
{
    if (lhs.ms == rhs.ms)
    {
        return new Phanso(lhs.ts + rhs.ts, lhs.ms);
    }
    int firstProduct = lhs.ts * rhs.ms;
    int secondProduct = rhs.ts * lhs.ms;
    return new Phanso(firstProduct + secondProduct, lhs.ms * rhs.ms);
}
}

```

```
public override string ToString()
{
    string s = ts.ToString() + "/" + ms.ToString();
    return s;
}
private int ts;
private int ms;
}
```

```
public class Tester
```

```
{
```

```
    static void Main()
```

```
{
```

```
    Phanso f1 = new Phanso(3, 4);
```

```
    Console.WriteLine("f1:{0}", f1.ToString());
```

```
    Phanso f2 = new Phanso(2, 4);
```

```
    Console.WriteLine("f2:{0}", f2.ToString());
```

```
    Phanso f3 = f1 + f2;
```

```
    Console.WriteLine("f1 + f2 = f3:{0}", f3.ToString());
```

```
    Phanso f4 = f3 + 5;
```

```
    Console.WriteLine("f4 = f3 + 5:{0}", f4.ToString());
```

```
    Phanso f6 = 5+ f3 ;
```

```
    Console.WriteLine("f6 = 5 + f3:{0}", f6.ToString());
```

```
    Phanso f5 = new Phanso(2, 4);
```

```
    if (f5 == f2)
```

```
{
```

```
        Console.WriteLine("f5:{0}==f2:{1}",f5.ToString(), f2.ToString());
```

```
}
```

```
}
```

```
}
```

Interface(giao diện)

- ❖ **Interface** là ràng buộc, giao ước đảm bảo cho các lớp hay các cấu trúc sẽ thực hiện một điều gì đó.
- ❖ Khi một lớp thực thi một giao diện, thì lớp này báo cho các thành phần client biết rằng lớp này có **hỗ trợ các phương thức, thuộc tính, sự kiện và các chỉ mục khai báo trong giao diện.**
- ❖ Giao diện chính là phần đặc tả (không bao hàm phần cài đặt cụ thể nội dung) của 1 lớp.
- ❖ Một lớp đối tượng có thể đưa ra cùng lúc nhiều giao diện để các chương trình bên ngoài truy xuất

Interface(giao diện)

- ❖ Giống mà không giống abstract class! (khó phân biệt)
- ❖ **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 bộ từ 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 cập

từ 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()  
    {  
        ....  
    }  
}
```

Interface(giao diện)

❖ Thực thi nhiều giao diện:

```
public class Document : IStorable, Icompressible
```

❖ Mở rộng giao diện

```
interface ILoggedCompressible : ICompressible  
{  
    void LogSavedBytes();  
}
```

❖ Kết hợp các giao diện:

```
interface IStorableCompressible : IStoreable, ILoggedCompressible  
{  
    void LogOriginalSize();  
}
```

Interface(giao diện)

❖ Toán tử **is**

- **<biểu thức> is <kiểu dữ liệu>**

❖ Toán tử **as**

- **<biểu thức> as <kiểu dữ liệu>**

Interface vs Abstract

Abstract: phản ánh tất cả đặc điểm (kể cả **cấu trúc nội tại**) chung nhất của một tập đối tượng nào đó.

Interface: phản ánh một phần đặc điểm (bên ngoài) của một loại đối tượng. Hai đối tượng về mặt bản chất có thể rất khác nhau nhưng vẫn có thể có chung một phần đặc điểm nào đó giống nhau.

Ví dụ: xe hơi Toyota và học sinh đều có tính chất chung là di chuyển được

```
interface IMovable
{
    public 3DPoint Position { get; set; }
    public double MaxSpeed { get; set; }
    public MoveTo(3DPoint newPosition);
}
```

Interface vs Abstract

Một class chỉ được thừa kế từ một lớp cơ sở

Nhưng được phép **implement** (cài đặt, hiện thực hóa) nhiều loại **interface** khác nhau.

```
public class ToyotaCar: Car, IMovable, IUsePower
{
    ...
}
```

```
public class Student: People, IMovable, IBreathable
{
    ...
}
```

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 thống

- ❖ Xử lý lỗi truyền thống 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

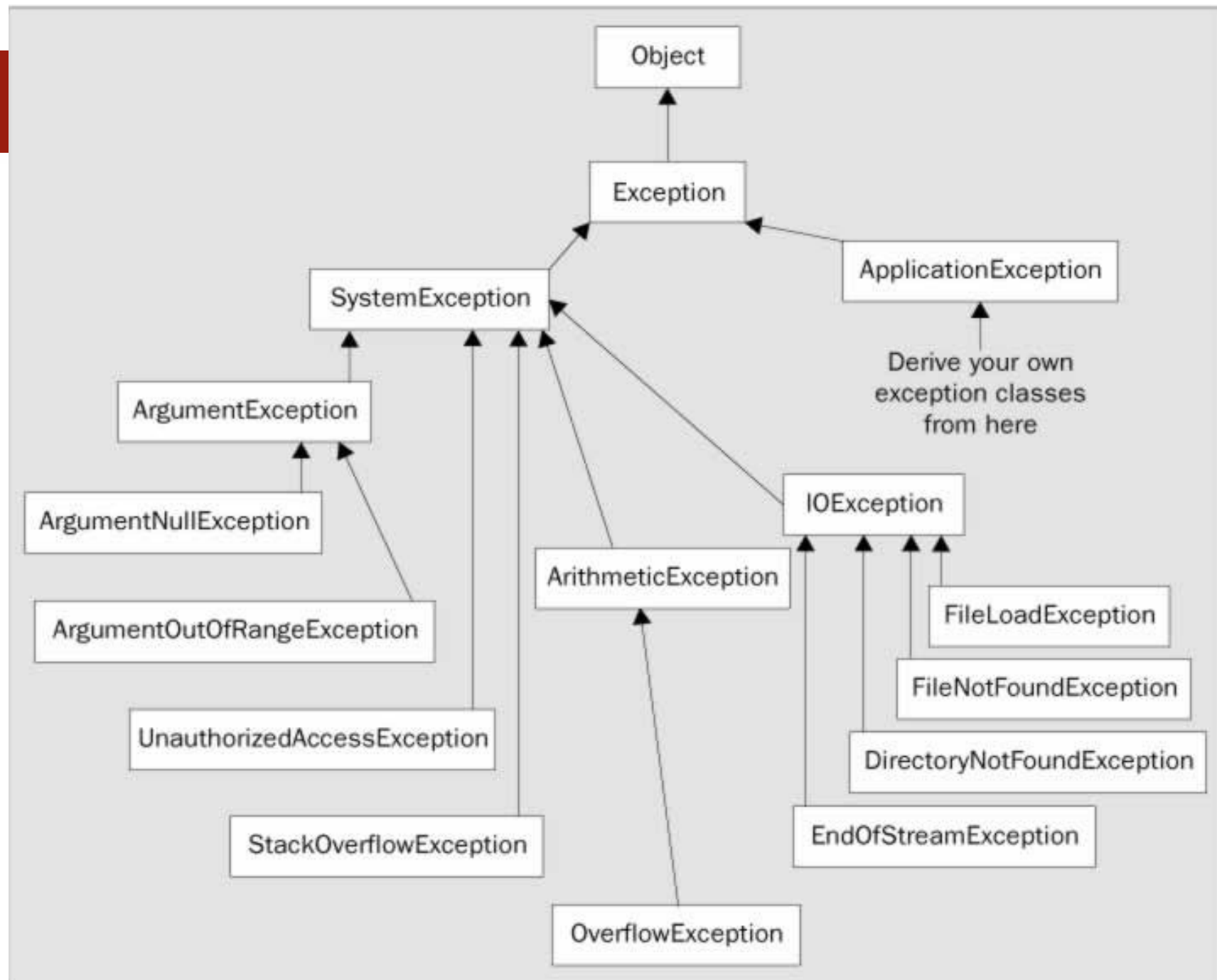
- ❖ Exception – ngoại lệ là cơ chế thông báo và xử lý lỗi giải quyết được các vấn đề kể trên
- ❖ Tách được phần xử lý lỗi ra khỏi phần thuật toán chính
- ❖ cho phép 1 hàm thông báo về nhiều loại ngoại lệ
 - Không phải hàm nào cũng phải xử lý lỗi nếu có một số hàm gọi thành chuỗi, ngoại lệ chỉ lần được 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

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];

```
using System;
public class ThrowTest
{
    public static void Main()
    {
        string s = null;
        if (s == null)
        {
            throw(new ArgumentNullException());
        }
        Console.WriteLine("The string s is null");
        // not executed
    }
}
```

```
public class Test
```

```
{
```

```
    public static void Main()
```

```
    {
```

```
        Console.WriteLine("Enter Main....");
```

```
        Test t = new Test();
```

```
        t.Func1();
```

```
        Console.WriteLine("Exit Main...");
```

```
    }
```

```
    public void Func1()
```

```
    {
```

```
        Console.WriteLine("Enter Func1...");
```

```
        Func2();
```

```
        Console.WriteLine("Exit Func1...");
```

```
    }
```

```
    public void Func2()
```

```
    {
```

```
        Console.WriteLine("Enter Func2...");
```

```
        throw new System.Exception();
```

```
        Console.WriteLine("Exit Func2...");
```

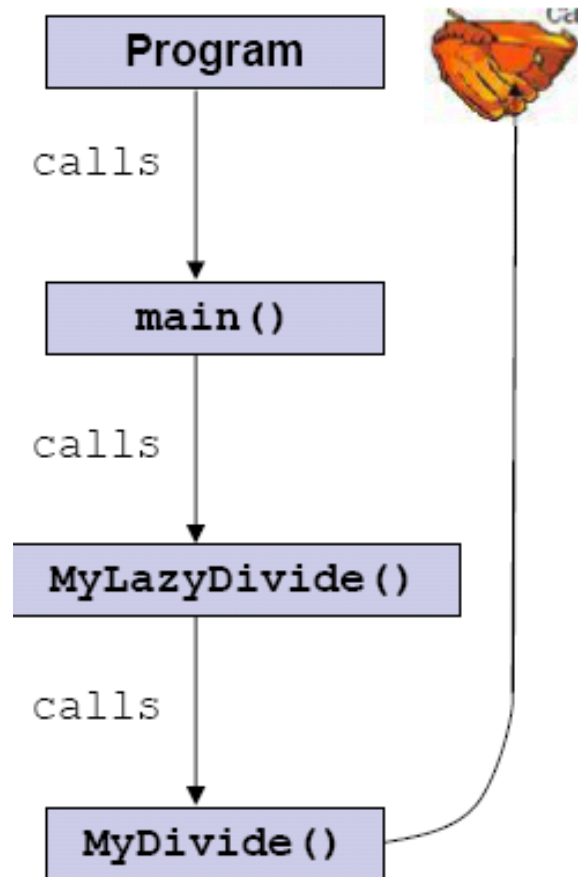
```
    }
```

```
}
```



Enter Main....
Enter Func1...
Enter Func2...
Exception occurred: System.Exception: An exception of type
System.Exception was throw.
 at Programming_CSharp.Test.Func2() in ... exception01.cs:line
26
 at Programming_CSharp.Test.Func1() in ... exception01.cs:line
20
 at Programming_CSharp.Test.Main() in ... exception01.cs:line 12

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



```
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...");
}
```



Enter Main...
Enter Func1...
Enter Func2...
Entering try block...
Exception caught and handled.
Exit Func2...
Exit Func1...
Exit Main...

```
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...");
}
```

```
public void Func2()
{
    Console.WriteLine("Enter Func2...");
    throw new System.Exception();
    Console.WriteLine("Exit Func2...");
}
```

Enter Main...
Enter Func1...
Entering try block...
Enter Func2...
Exception caught and handled.
Exit Func1...
Exit Main...

Ví dụ

```
class Test
{
    static void Main(string[] args)
    {
        Test t = new Test();
        t.TestFunc();
    }
    public double DoDivide(double a, double b)
    {
        if ( b == 0)
            throw new System.DivideByZeroException();
        if ( a == 0)
            throw new System.ArithmeticException();
        return a/b;
    }
}
```

```
{  
    try  
    {  
        double a = 5;  
        double b = 0;  
        Console.WriteLine("{0} / {1} = {2}", a, b, DoDivide(a,b));  
    }  
    catch (System.DivideByZeroException)  
    {  
        Console.WriteLine("DivideByZeroException caught!");  
    }  
    catch (System.ArithmeticException)  
    {  
        Console.WriteLine("ArithmeticException caught!");  
    }  
    catch  
    {  
        Console.WriteLine("Unknown exception caught");  
    }  
}
```


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

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

Tạo riêng ngoại lệ

- ❖ phải được dẫn xuất từ `System.ApplicationException`