

Bài 3: Kiến thức cơ bản về C# (tt)

Array, Collection

Lương Trần Hy Hiến

FIT, HCMUP

Lập trình Windows Form với C#

Nội dung

- **Array, Indexer và Collection**
- **Collection**
- **Cấu trúc (Struct)**
- **Xử lý lỗi & exception (biệt lệ)**

Mảng

- **Mảng 1 chiều**
- **Mảng 2 chiều**
- **Mảng nhiều chiều**
- **Mảng Jagged Array**

Mảng

- Mảng là tập hợp các biến có cùng kiểu dữ liệu, cùng tên nhưng có chỉ số khác nhau.
- Trong C#, mảng có chỉ số bắt đầu là 0 và luôn luôn là mảng động (mảng có khả năng thay đổi kích thước).
- Ví dụ: Mảng nguyên 5 phần tử

| 0 | 1 | 2 | 3 | 4 | ← Chỉ số |
|----|----|----|----|------|-----------|
| 10 | 15 | 20 | 27 | 2009 | ← Giá trị |

Mảng 1 chiều

- Cú pháp:

Kieu[] tenMang;

tenMang = **new Kieu**[so_phan_tu];

Hoặc:

Kieu[] tenMang = **new Kieu**[so_phan_tu];

- Ví dụ:

int [] mang = **new** int[5];

double[] B = **new** double[3];

string[] C = **new** string[4];

Thao tác với mảng

- Lấy độ dài mảng:
`int dodai = mang.Length;`
- Truy cập phần tử thứ **index**: `mang[index]`
`int kq = mang[1] + mang[2];`
- Duyệt qua các phần tử trong mảng:

```
for(int i = 0; i < mang.Length; i++)  
{  
    //xử lý phần tử mang[i]  
}
```

Làm việc với mảng 1 chiều

- Lấy kích thước mảng:

```
int arrayLength = myIntegers.Length;
```

- Sắp xếp mảng số nguyên:

```
Array.Sort(myIntegers);
```

- Đảo ngược mảng:

```
Array.Reverse(myArray);
```

- Duyệt mảng:

Mảng 2 chiều

- Cú pháp:

`type[,] array-name;`

- Ví dụ:

```
int[,] myRectArray = new int[2,3];
```

```
int[,] myRectArray = new int[,] {  
    {1,2},{3,4},{5,6},{7,8}}; //mảng 4 hàng 2 cột  
string[,] beatleName = { {"Lennon","John"},  
    {"McCartney","Paul"},  
    {"Harrison","George"},  
    {"Starkey","Richard"} };
```


Làm việc với mảng 2 chiều

- Duyệt mảng:

```
double [, ] matrix = new double[10, 10];
```

```
for (int i = 0; i < 10; i++)
```

```
{
```

```
    for (int j=0; j < 10; j++)
```

```
        matrix[i, j] = 4;
```

```
}
```

Mảng nhiều chiều

- Ví dụ:

```
string[, ] my3DArray;
```

Mảng jagged - Mảng răng cưa

- Một loại thứ 2 của mảng nhiều chiều trong C# là Jagged array.
- Là mảng mà số phần tử trong mỗi chiều có thể khác nhau
- Ví dụ:

```
int[][] a = new int[3][];  
a[0] = new int[4];  
a[1] = new int[3];  
a[2] = new int[1];
```

Làm việc với Jagged Array

- Khởi tạo ma trận $n \times m$ Jagged Array:

```
int[][] a = new int[n][];  
for(int i = 0; i < n; i++)  
{  
    a[i] = new int[m];  
    for (int j = 0; j < m; j++)  
    {  
        a[i][j] = i*n + j;  
    }  
}
```

Một số lưu ý khi sử dụng mảng

- Sử dụng thuộc tính **Length** của mảng thay vì phải nắm số phần tử trong mảng
- Cấu trúc lặp **foreach** hữu hiệu hơn là dùng **for !!!**
- Lấy số chiều 1 mảng : sử dụng thuộc tính **rank**

Collections

HIENLTH, FIT of HCMUP

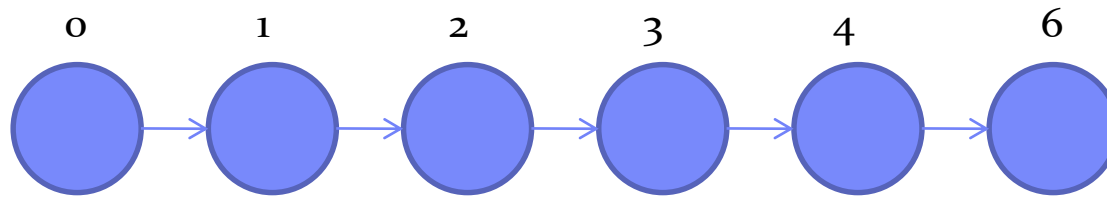
Lập trình Windows Form với C#

Collections – Tập hợp

- Là cấu trúc lưu trữ các phần tử có kiểu dữ liệu khác nhau và không hạn chế số lượng phần tử.
- Các lớp có kiểu tập hợp nằm trong namespace System.Collections bao gồm:
 - List
 - ArrayList
 - HashTable
 - Queue
 - Stack

MẢNG ĐỘNG

- ▶ **Mảng động hay còn gọi là danh sách được sử dụng để quản lý danh sách các phần tử có thứ tự và hỗ trợ các thao tác thêm, xóa, sửa, tìm kiếm,...**
- ▶ **Hình ảnh danh sách như sau**



- ▶ **2 lớp sau đây được sử dụng để tạo mảng động**
 - **List<Kiểu dữ liệu>**: danh sách có kiểu
 - **ArrayList**: danh sách không kiểu

Mảng động – List

- **List<kieu_DL> mylist = new List<kieu_DL>();**
- **Phải chỉ định rõ kiểu dữ liệu khi dùng. VD:**
List<string> mylist = new List<string>();
 - ▶ Tạo mảng rỗng chứa số nguyên
 - **List<int> ages = new List<int>();**
 - ▶ Tạo mảng số thực khởi đầu 3 phần tử
 - **List<double> salaries = new List<double>()**
{1.2, 3.4, 5.6};

Mảng động – List

| Method / Property | Diễn giải |
|-------------------|--|
| Add() | Thêm 01 phần tử |
| Capacity | Tổng số phần tử tối đa |
| Clear() | Xóa tất cả phần tử |
| Contains() | Xác định 1 phần tử có trong danh sách hay không? |
| Count | Số lượng phần tử thật sự |
| Insert() | Chèn 1 phần tử vào vị trí cụ thể |
| RemoveAt() | Xóa phần tử tại vị trí |
| Sort() | Sắp xếp các phần tử |

THAO TÁC MẢNG ĐỘNG

Thao tác thông thường

- **Add(<kiểu> element)**: thêm phần tử vào mảng
- **Remove(<kiểu> element)**: xóa phần tử khỏi mảng
- **[index]**: truy xuất
- **Count**: lấy số phần tử trong mảng

Tìm và kiểm tra

- **Int IndexOf(<kiểu> element)**: tìm vị trí phần tử từ đầu
- **Int LastIndexOf(<kiểu> element)**: tìm vị trí phần tử từ cuối
- **Bool Contains(<kiểu>element)**: kiểm tra sự tồn tại

Các thao tác khác

- **Clear()**: xóa sạch
- **Reverse()**: đảo ngược
- **Sort()**: sắp xếp

VÍ DỤ: MẢNG ĐỘNG - LIST

```
// Tạo mảng rỗng và thêm vào 3 phần tử
List<int> ages = new List<int>();
ages.Add(77);
ages.Add(33);
ages.Add(88);
Console.WriteLine("Số phần tử: {0}", ages.Count);
// Sửa phần tử thứ 2
ages[1] = 55;
// Tìm vị trí phần tử 88
int index = ages.IndexOf(88);
Console.WriteLine("Vị trí của 88: {0}", index);
// Xóa phần tử 77
ages.Remove(77);
// Duyệt và xuất tất cả các phần tử trong mảng
foreach (int age in ages)
{
    Console.WriteLine(" >> Phần tử: {0}", age);
}
// Xóa sạch các phần tử trong mảng
ages.Clear();
```

Thêm các phần tử vào mảng

Mảng động – ArrayList

- Không cần chỉ định kiểu khi khai báo
- Các phần tử có thể có kiểu dữ liệu khác nhau.
 - ▶ Tạo mảng rỗng chứa phần tử kiểu bất kỳ
 - **ArrayList items = new ArrayList();**
 - ▶ Tạo mảng rỗng có khởi đầu 3 phần tử
 - **ArrayList student = new ArrayList()**
{“Ngoc Thanh”, true, 80};
 - ArrayList birds = new ArrayList();
birds.Add(“cat”);
birds.Add(10952);
foreach (Bird b in birds)

ArrayList Members

■ Some Methods:

- BinarySearch()
- IndexOf()
- Sort()
- ToArray()
- Remove()
- RemoveAt()
- Insert()

■ Some Properties:

- Count
- Capacity
- IsFixedSize
- IsReadOnly
- IsSynchronized

Lớp Stack<T>

- **Mảng động, kiểu tùy ý, kích thước tự động, thêm/bớt xảy ra tại đỉnh (LIFO)**
- **Phương thức:**
 - **Push(T)** – thêm phần tử vào đỉnh stack
 - **Pop()** – xóa phần tử ở đỉnh stack và trả về giá trị đó

Stack<T> – Ví dụ

■ Sử dụng **Push()**, **Pop()** và **Peek()**

```
static void Main()
{
    Stack<string> stack = new Stack<string>();
    stack.Push("1. Ivan");
    stack.Push("2. Nikolay");
    stack.Push("3. Maria");
    stack.Push("4. George");
    Console.WriteLine("Top = {0}", stack.Peek());
    while (stack.Count > 0)
    {
        string personName = stack.Pop();
        Console.WriteLine(personName);
    }
}
```


Lớp Queue<T

25

- **Mảng động, kiểu tùy ý, kích thước động; thêm/bớt xảy ra 2 chiều (FIFO)**
- **Phương thức:**
 - **Enqueue(T)** – thêm phần tử vào cuối mảng
 - **Dequeue()** – lấy phần tử đầu mảng và trả về giá trị đó

Ví dụ Queue

26

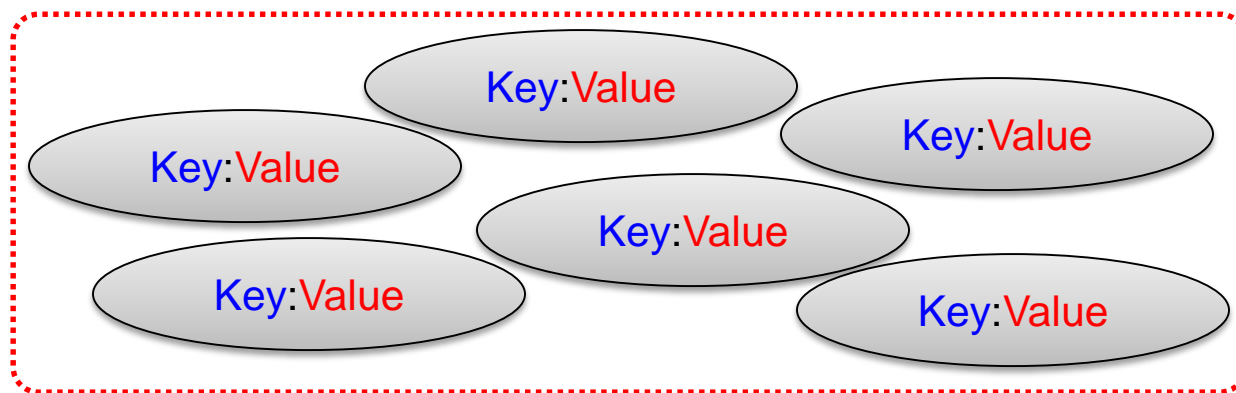
```
static void Main()
{
    Queue<string> queue = new
Queue<string>();

    queue.Enqueue("Message One");
    queue.Enqueue("Message Two");
    queue.Enqueue("Message Three");
    queue.Enqueue("Message Four");
    while (queue.Count > 0)
    {
        string message = queue.Dequeue();
        Console.WriteLine(message);
    }
}
```

TỪ ĐIỂN

Dùng quản lý tập hợp các phần tử không phân biệt thứ tự. Mỗi phần tử gồm 2 phần (khóa và giá trị). Để truy xuất giá trị của một phần tử ta phải biết khóa của nó.

Hình ảnh



2 lớp thường dùng

- **Dictionary<Kiểu khóa, Kiểu giá trị>**: có kiểu
- **Hashtable**: không kiểu

THAO TÁC TỪ ĐIỂN

Truy xuất

[Key]: truy xuất giá trị của phần tử

Thuộc tính thường dùng

Count: lấy số phần tử

Keys: lấy tập hợp khóa

Values: lấy tập hợp giá trị

Phương thức thường dùng

Add(Key, Value): thêm một phần tử

Remove(Key): xóa một phần tử

Clear(): xóa sách

ContainsKey(Key): kiểm tra sự tồn tại của khóa

ContainsValue(Value): kiểm tra sự tồn tại của giá trị

HashTable

- Là Kiểu từ điển
- Mỗi phần tử bao gồm 01 cặp [key-value]
- Truy xuất nhanh.
- Các cặp key không trùng nhau

| Key | Value |
|---------|--------|
| masp | SP001 |
| soluong | 10 |
| gia | 123.45 |

```
Hashtable ht = new Hashtable();  
ht.Add("masp", "SP001");  
ht.Add("soluong", 10);  
ht.Add("gia", 123.45);  
foreach (DictionaryEntry de in ht)  
{  
    s += string.Format("kqy={0}, value = {1}", de.Key,  
de.Value);  
}
```

KHỞI TẠO TỪ ĐIỂN

```
// Tạo từ điển không kiểu, thêm vào 3 phần tử
Hashtable student = new Hashtable();
student.Add("Name", "Nguyen Van Teo");
student.Add("Age", 30);
student.Add("Gender", true);
// Tạo từ điển có kiểu, thêm 4 phần tử
Dictionary<String, Double> emp = new Dictionary<String, Double>();
emp.Add("Nguyen Thanh Tin", 8.5);
emp.Add("Pham Thi Hoa", 7);
emp.Add("Ngo Quoc Bao", 6.5);
emp.Add("Luong Van Thanh", 9);
// Tạo từ điển có kiểu, khởi đầu 3 phần tử
Dictionary<String, String> words = new Dictionary<String, String>()
{
    {"Love", "Yêu"},
    {"One", "Một"},
    {"School", "Trường học"}
};
```

VÍ DỤ TỪ ĐIỂN

// Tạo và khởi đầu từ điển

```
Dictionary<String, Double[]> marks = new Dictionary<String, Double[]>()
{
    {"Tuấn", new Double[]{7.0, 8.5, 9.5}},
    {"Hoa", new Double[]{5, 7, 4}},
    {"Hồng", new Double[]{6, 8, 10}}
};
// Sửa phần tử thứ hai trong mảng có khóa là "Hoa"
marks["Hoa"][2] = 5.5;
//Duyệt và xuất ra thông tin của mỗi phần tử
foreach (String Name in marks.Keys)
{
    Double[] MA = marks[Name];
    Console.WriteLine("{0}, {1}, {2}, {3}", Name, MA[0], MA[1], MA[2]);
}
Console.WriteLine("Số phần tử: {0}", marks.Count);
```

Struct

- **Struct là kiểu Value Type không phải là Reference Type → có thể không cần sử dụng từ khóa new.**
- **Trong Struct có thể định nghĩa các phương thức (giống Class).**
- **Trong Struct, trình biên dịch luôn luôn cung cấp một constructor không tham số mặc định, và không cho phép thay thế.**
- **Struct không hỗ trợ thừa kế.**

Struct

```
struct StrHocSinh
```

```
{
```

```
    public int MaSo;
```

```
    public string HoTen;
```

```
    public double Toan;
```

```
    public double Van;
```

```
    public double DTB;
```

```
    public StrHocSinh(int ms, string ht, double t, double v)
```

```
    {
```

```
        MaSo = ms;
```

```
        HoTen = ht;
```

```
        Toan = t;
```

```
        Van = v;
```

```
        DTB = (t+ v)/2;
```

```
    }
```

```
}
```

Cấu Trúc

```
using System;
public struct Location
{
    public Location(int xCoordinate, int yCoordinate)
    {
        xVal = xCoordinate;
        yVal = yCoordinate;
    }
    public int x
    {
        get{ return xVal; }
        set{ xVal = value; }
    }
    public int y
    {
        get{ return yVal; }
        set{ yVal = value; }
    }
    public override string ToString( )
    {
        return (String.Format("{0}, {1}", xVal,yVal));
    }
    private int xVal;
    private int yVal;
}
```

Cách dùng :

Location hpt

= new Location(200,300);

Console.WriteLine("KQ = {0}", hpt);

Xử lý lỗi & exception (ngoại lệ)

- Exception chứa các thông tin về sự cố bất thường của chương trình
- Phân biệt **bug**, **error** và **exception**
- Chương trình dù đã không còn bug hay error vẫn có thể cho ra các exception (truy cập, bộ nhớ)
- Có thể dùng các đối tượng exception có sẵn, tự tạo exception, hay bắt exception trong exception (trong trường hợp sửa lỗi)

Xử lý lỗi & exception (ngoại lệ)

■ Cấu trúc xử lý lỗi

```
try
{
    Console.WriteLine("Open file here");
    double a = 5;
    double b = 0;
    Console.WriteLine ("{0} / {1} = {2}",
                        a, b, DoDivide(a,b));
    Console.WriteLine ("This line may or may not print");
}
catch (System.DivideByZeroException)
{
    Console.WriteLine("DivideByZeroException caught!");
}
catch
{
    Console.WriteLine("Unknown exception caught");
}
finally
{
    Console.WriteLine ("Close file here.");
}
```

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 (ngoại lệ)

- **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**

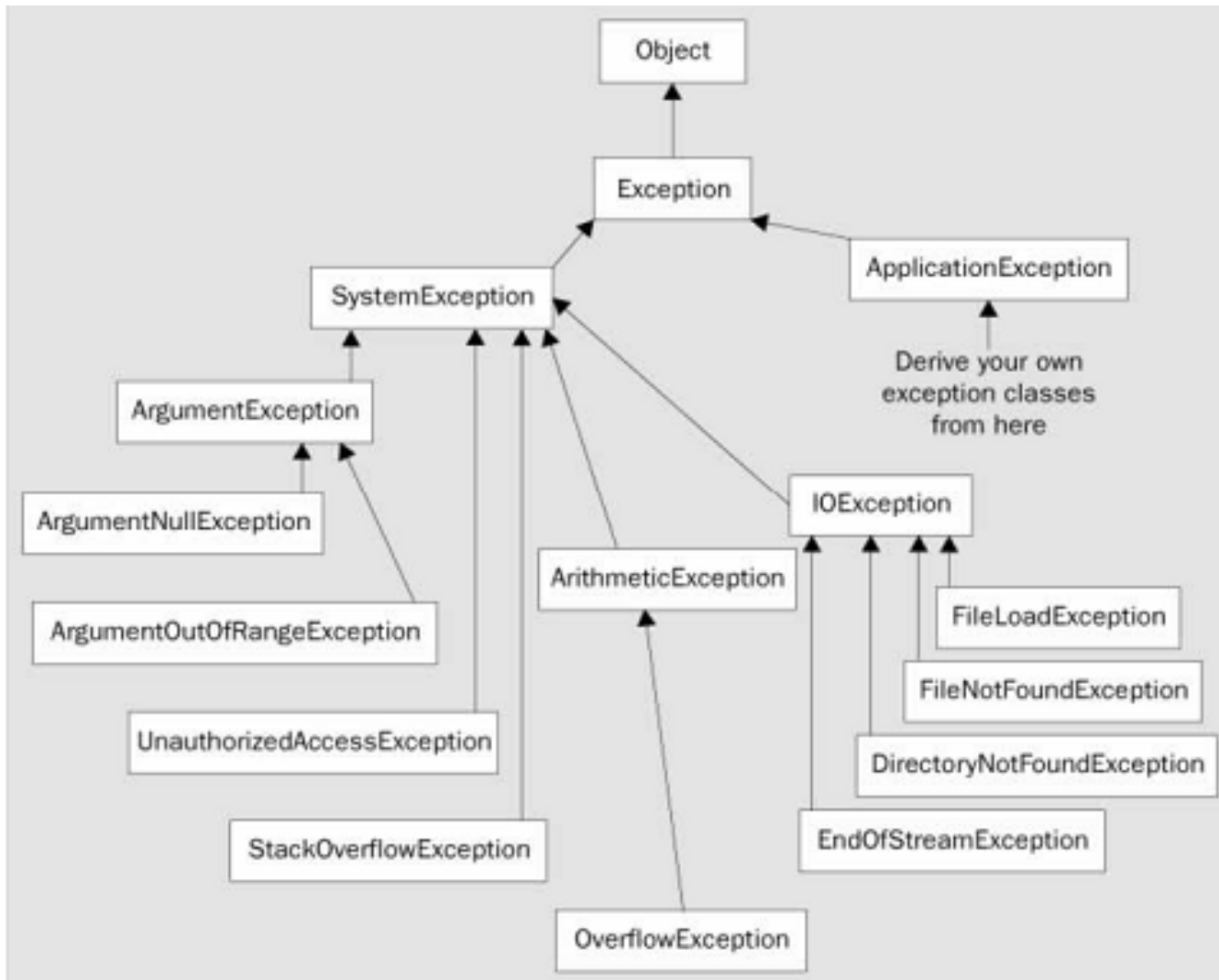
Xử lý ngoại lệ

- 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 → 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 in ra những thông điệp có ý nghĩa

Exception



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 (tt)

■ Ví dụ 01:

```
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
    }
}
```

Ví dụ 02 - Throw

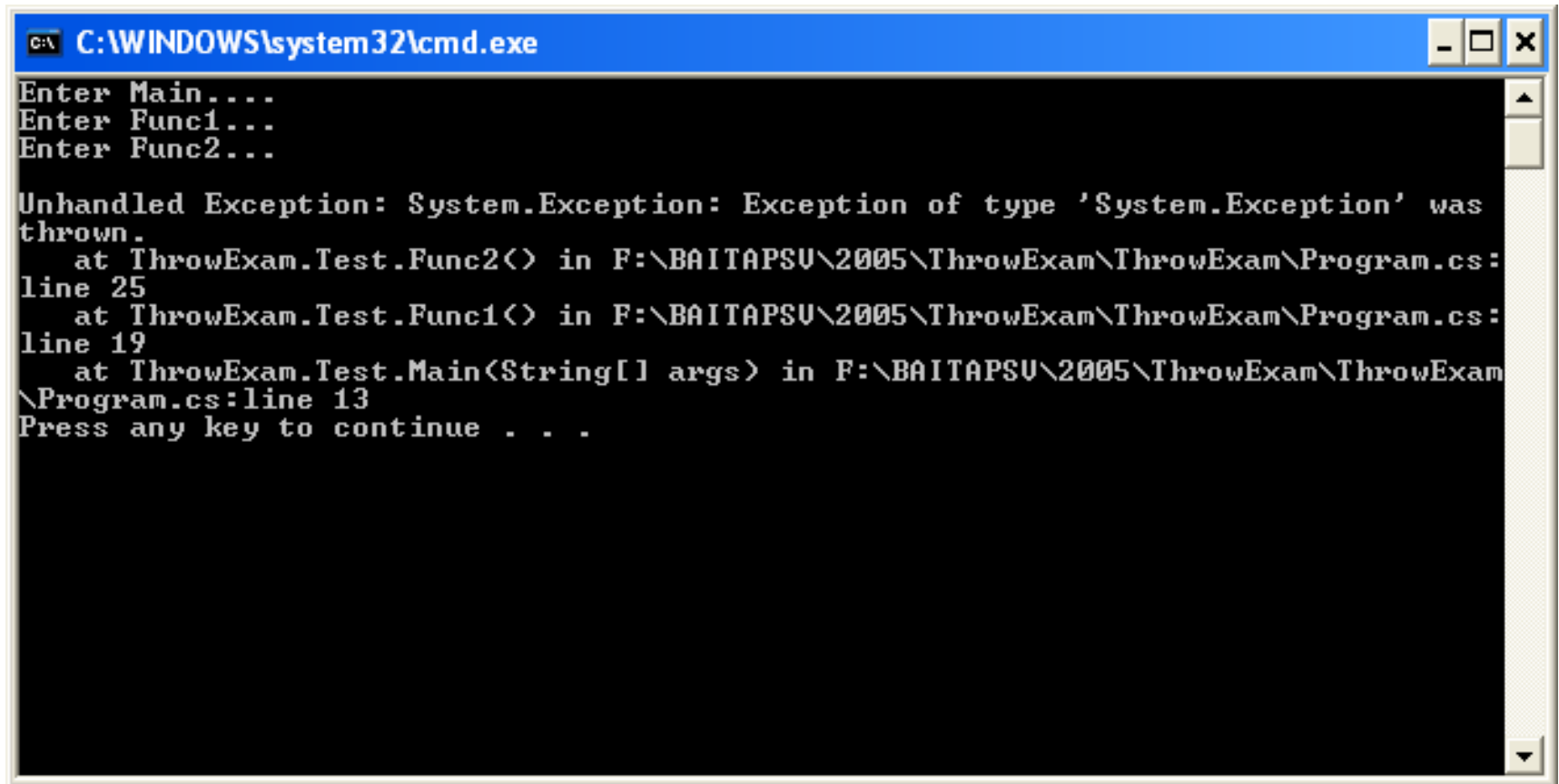
```
using System;
using System.Collections.Generic;
using System.Text;

namespace ThrowExam
{
    class Test
    {
        static void Main(string[] args)
        {
            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...");
        }
    }
}
```

Ví dụ 02 – Throw (tt)



The screenshot shows a Windows command prompt window titled "C:\WINDOWS\system32\cmd.exe". The window has a blue title bar and standard Windows window controls (minimize, maximize, close). The command prompt displays the following text:

```
Enter Main....  
Enter Func1...  
Enter Func2...  
  
Unhandled Exception: System.Exception: Exception of type 'System.Exception' was  
thrown.  
    at ThrowExam.Test.Func2() in F:\BAITAPSV\2005\ThrowExam\ThrowExam\Program.cs:  
line 25  
    at ThrowExam.Test.Func1() in F:\BAITAPSV\2005\ThrowExam\ThrowExam\Program.cs:  
line 19  
    at ThrowExam.Test.Main(String[] args) in F:\BAITAPSV\2005\ThrowExam\ThrowExam  
\Program.cs:line 13  
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 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

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;
    }
    //Còn tiếp
}
```


Ví dụ (tt)

```
public void TestFunc() {  
    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

Câu lệnh finally (tt)

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, dòng 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

Xử lý lỗi & exception (ngoại lệ)

■ Đối tượng Exception :

```
catch (System.DivideByZeroException e)
{
    Console.WriteLine(
        "\nDivideByZeroException! Msg: {0}", e.Message);
    Console.WriteLine("\nHelpLink: {0}", e.HelpLink);
    Console.WriteLine(
        "\nHere's a stack trace: {0}\n", e.StackTrace);
}
```

■ Tạo ngoại lệ :

```
public class MyCustomException : System.ApplicationException
{
    public MyCustomException(string message) : base(message)
    {
    }
}

catch (MyCustomException e)
```

