



Chương 3: Lập trình C# nâng cao

Giảng viên: Tạ Việt Phương
E-mail: phuongtv@uit.edu.vn

Nội dung



Generics



Collections



Anonymous, nullable, dynamic



Exceptions



Generics

Generics

- ❏ Lập trình tổng quát (generic program hoặc generics) là dạng lập trình mà **kiểu dữ liệu** của biến hoặc phương thức chỉ được xác định khi khởi tạo và sử dụng.
- ❏ Là dạng **tham số hóa kiểu dữ liệu**.
- ❏ Sử dụng generic ở giai đoạn khai báo là một **kiểu dữ liệu giả (tham số kiểu – type parameter)**; khi sử dụng, kiểu dữ liệu giả này được thay thế bằng **kiểu dữ liệu thật**.
- ❏ Tham số kiểu thường được ký hiệu là **T**.
- ❏ Có thể sử dụng với kiểu type, class, phương thức, giao diện (interface), đại diện (delegate) hoặc tập hợp (collections).
 - Đối với class và interface: tác dụng trong toàn bộ class; và sử dụng cho biến thành viên, thuộc tính, kiểu trả về của phương thức, tham số truyền vào của phương thức.

Generics

- Đối với phương thức: tác dụng trong phạm vi code của phương thức; sử dụng cho biến cục bộ, kiểu trả về và tham số truyền vào.
- Đối với delegate: được sử dụng cho tham số và kiểu kết quả trả về.

Đặc điểm của generic:

- Kiểu dữ liệu giả đặt trong cặp dấu <>; tên thường là một chữ cái in hoa (T, U, V).
- Phương thức/ lớp tổng quát cho phép lựa chọn kiểu dữ liệu ở giai đoạn sử dụng.
- Trong code có thể sử dụng kiểu dữ liệu giả này tương tự như bất kỳ kiểu dữ liệu thật.
- Không giới hạn số lượng kiểu dữ liệu giả; và phân cách nhau bởi dấu phẩy.
- Có thể giới hạn loại kiểu dữ liệu của kiểu giả.

Generics

- **Khi nào nên sử dụng generics?**
- Khi lập trình, nếu gặp một trong hai tình huống dưới đây thì hãy nghĩ ngay đến generics:
 - Nếu có sự trùng lặp code về mặt logic và cách xử lý dữ liệu, chỉ khác biệt về kiểu dữ liệu: hãy nghĩ đến generics để tránh lặp code. Chúng ta sẽ gặp tình huống này ngay trong phần giới thiệu về generic method dưới đây.
 - Nếu lúc xây dựng class chưa xác định được kiểu dữ liệu của các biến thành viên, thuộc tính hoặc biến cục bộ (của phương thức) thì cần sử dụng lập trình generic. Chúng ta sẽ gặp tình huống này khi xem xét generic class ở phần sau.

Generics

Generic phương thức:

- Xét phương thức SwapInt đổi giá trị 2 biến kiểu int

```
public static void SwapInt(ref int a, ref int b)
{
    int temp = a;
    a = b;
    b = temp;
}
```

- Phương thức Swap với generic dùng đổi cho nhiều kiểu giá trị.

```
public static void Swap<T>(ref T a, ref T b)
{
    T temp = a;
    a = b;
    b = temp;
}
```

Generics

○ Kết quả:

```
static void Main(string[] args)
{
    Console.OutputEncoding = Encoding.Unicode;
    int a = 10, b = 20;
    double x = 10.5, y = 20.1;

    Console.WriteLine($"Trước: a = {a}, b = {b}");
    Swap<int>(ref a, ref b);
    Console.WriteLine($"Sau: a = {a}, b = {b}");
    Console.WriteLine($"Trước: x = {x}, y = {y}");
    Swap<double>(ref x, ref y);
    Console.WriteLine($"Sau: x = {x}, y = {y}");
}
```


Generics

- Khi logic của phương thức giống nhau và chỉ khác biệt nhau về kiểu dữ liệu thì có thể cài đặt phương thức theo generic.
- Sử dụng **<T>** để xác định phương thức generic. Nếu hai tham biến cùng kiểu dữ liệu chỉ cần khai báo một kiểu dữ liệu giả T.
- Nhiều kiểu giả: **<T1, T2, T3>**
- Thao tác trên kiểu dữ liệu **<T>** giống như là một kiểu dữ liệu bình thường.
- Trong generic method, kiểu giả chỉ có ý nghĩa và sử dụng trong thân phương thức.
- Kiểu giả có thể được sử dụng làm kiểu trả về của phương thức.
- Khi sử dụng phương thức generic thì thay thế kiểu giả T bằng một kiểu dữ liệu cụ thể.

Generics

Generic class

```
class DSInt
{
    private int[] _data;
    public int Count => _data.Length;
    public DSInt(int size) => _data = new int[size];
    public void Set(int index, int value)
    {
        if (index >= 0 && index < _data.Length) _data[index] = value;
    }

    public int Get(int index)
    {
        if (index >= 0 && index < _data.Length) return _data[index];
        return default(int);
    }
}
```

Generics

```
class DSChar
{
    private char[] _data;
    public int Count => _data.Length;
    public DSChar(int size) => _data = new char[size];
    public void Set(int index, char value)
    {
        if (index >= 0 && index < _data.Length) _data[index] = value;
    }

    public char Get(int index)
    {
        if (index >= 0 && index < _data.Length) return _data[index];
        return default(char);
    }
}
```

Generics

```
class DS<T>
{
    private T[] _data;
    public int Count => _data.Length;
    public DS(int size) => _data = new T[size];
    public void Set(int index, T value)
    {
        if (index >= 0 && index < _data.Length) _data[index] = value;
    }

    public T Get(int index)
    {
        if (index >= 0 && index < _data.Length) return _data[index];
        return default(T);
    }
}
```

Generics

```
static void Main(string[] args)
{
    Console.OutputEncoding = Encoding.Unicode;
    var dsInt = new DS<int>(10);
    for (int i = 0; i < dsInt.Count; i++)
        Console.WriteLine($"{dsInt.Get(i)}t");

    var dsChar = new DS<char>(10);
    for (int i = 0; i < dsChar.Count; i++)
        Console.WriteLine($"{dsChar.Get(i)}t");
}
```

Generics

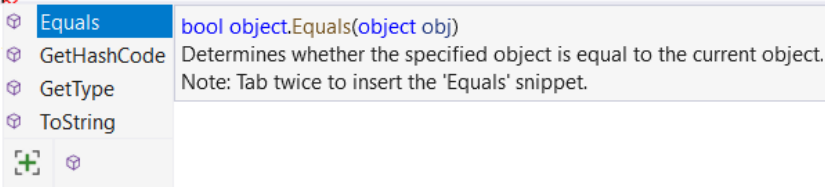
- Khi xây dựng class, chưa xác định được kiểu dữ liệu của biến thành viên, thuộc tính hoặc biến cục bộ của phương thức.
- Khi nhiều class có cùng chung về mặt logic (các biến, các phương thức) chỉ khác biệt về kiểu dữ liệu.
- Sử dụng **<T>** sau tên class để xác định class generic.
- Trong class có thể khai báo và sử dụng nhiều kiểu giả: **<T1, T2, T3>**
- **VD:** Biến private `_data` có kiểu dữ liệu là một kiểu giả T.
- Phạm vi và ý nghĩa của kiểu giả T sẽ là trong toàn class.

Generics

Đặc điểm của kiểu dữ liệu T

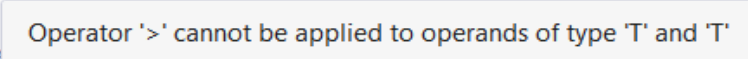
- Biến thuộc kiểu T có đúng những thuộc tính và phương thức của kiểu Object.

```
0 references
public static void Swap<T>(ref T a, ref T b)
{
    if (a.)
    {
        T
        a
        b
    }
}
```



- Không cho phép so sánh (bằng các phép so sánh thường) trên các biến thuộc kiểu dữ liệu T

```
0 references
public static void Swap<T>(ref T a, ref T b)
{
    if (a > b)
    {
        T temp;
        a = b;
        b = temp;
    }
}
```



Generics

- Như vậy, sẽ có giới hạn kiểu T để biến thuộc kiểu T có những đặc điểm mà ta mong muốn.

```
public static void Swap<T>(ref T a, ref T b) where T : IComparable
{
    if (a.CompareTo(b) > 0)
    {
        T temp = a;
        a = b;
        b = temp;
    }
}
```

- Cho phép kiểu T có thể nhận được kiểu dữ liệu thực theo mong muốn.
- Một số loại giới hạn kiểu:
 - Kiểu class: `where T : class`.
 - Kiểu value: `where T : struct`. Bắt buộc là các kiểu value (int, double, struct, enum...)
 - Kiểu constructor: `where T : new()`. Yêu cầu lớp thay thế cho T phải có hàm tạo không tham số.
 - Kiểu lớp con: `where T : <base class name>`. VD: `where T : Animal`.
 - Nếu nhiều kiểu giả, mỗi kiểu giả có giới hạn khác nhau:
`where T : class where U : struct`

Generics

- Generics collections nằm trong **System.Collections.Generic**
- Một số **generic collections** phổ biến:

Lớp	Mô tả
List<T>	Là mảng động, có khả năng tự tăng giảm kích thước; thay thế cho ArrayList.
Stack<T>	Lưu trữ và thao tác theo cấu trúc ngăn xếp (LIFO), thay thế cho Stack
Queue<T>	Lưu trữ và thao tác theo cấu trúc hàng đợi FIFO, thay thế cho Queue
LinkedList<T>	Lưu trữ và thao tác theo danh sách liên kết, thay thế cho LinkedList
Dictionary<TKey, TValue>	Lưu trữ và thao tác theo cấu trúc dữ liệu từ điển dưới dạng cặp Key-Value, thay thế cho Hashtable



Collections

Collections

- 📖 Là kiểu dữ liệu đặc biệt, có cấu trúc để lưu trữ một nhóm dữ liệu.
- 📖 Cung cấp các phép toán để thao tác trên tập dữ liệu.
- 📖 Các phép toán cơ bản: thêm, xóa, cập nhật, gán và lấy giá trị.
- 📖 Collection luôn xác định một tập thuộc tính đặc trưng và các phép toán để thao tác trên dữ liệu.
- 📖 Ví dụ: thuộc tính **Count** (đếm số lượng phần tử), các phép toán (phương thức): **Add** (thêm phần tử), **Insert** (chèn phần tử), **Remove** (xóa phần tử)....

Collections

📖 Có 2 loại Collections:

- **Tuyến tính** (linear): danh sách các phần tử có thể được sắp xếp theo thứ tự (1,2,3...). VD: mảng (array). Gồm 3 loại:
 - *Truy xuất trực tiếp* (direct access)
 - *Truy xuất tuần tự* (sequential access)
 - *Dùng chỉ số* (indexed).
- **Phi tuyến tính** (nonlinear): lưu trữ danh sách phần tử mà không thể sắp xếp được theo thứ tự. VD: sơ đồ tổ chức, sơ đồ cây, heap, đồ thị hoặc tập hợp. Gồm 2 loại:
 - *Phân cấp* (hierarchical)
 - *Phân nhóm* (grouped)

Collections

- 📖 Tuyến tính (linear): truy xuất trực tiếp (direct access)
 - Mảng (array):
 - Là kiểu dữ liệu dựng sẵn (class Array)
 - Truy xuất trực tiếp phần tử trong mảng qua chỉ số (index).
 - Có 2 loại: mảng tĩnh (cố định các phần tử trong mảng) và mảng động (số lượng phần tử trong mảng có thể thay đổi được).
 - Phù hợp với việc lưu trữ nhóm giá trị có số lượng cố định
 - String:
 - Là một kiểu array thuộc loại bất biến (immutable)
 - Struct:
 - Là kiểu chứa nhiều phần tử thuộc nhiều kiểu dữ liệu khác nhau và cho phép truy xuất trực tiếp qua tên phần tử.

Collections

- 📖 Tuyến tính (linear): truy xuất tuần tự (sequential access)
 - Để truy xuất một phần tử, bắt buộc phải duyệt qua các phần tử trước nó.
 - Duyệt theo một chiều (từ phần tử đầu đến cuối hoặc từ phần tử cuối lên đầu) hoặc theo cả hai chiều.
 - Các phần tử trong danh sách có thể được sắp xếp (ordered list) hoặc chưa được sắp xếp (unordered list)
 - Có 2 loại:
 - Ngăn xếp (Stack): chứa và truy xuất dữ liệu theo mô hình LIFO.
 - Hàng đợi (Queue): chứa và truy xuất dữ liệu theo mô hình FIFO.

Collections



Tuyến tính (linear): dùng chỉ số(indexed)

- Dữ liệu được lưu trữ dưới dạng cặp khóa – giá trị.
- Có 2 loại:
 - Bảng băm (Hash table): sử dụng hàm băm (Hash function) để biến giá trị của khóa thành chỉ số (số nguyên) và dùng chỉ số này để truy xuất giá trị.
 - Từ điển (Dictionary): giống từ điển ngôn ngữ, chứa cặp khóa - giá trị; khóa có vai trò như từ gốc, giá trị tương đương nghĩa của từ.

Collections



Phi tuyến tính (nonlinear): phân cấp (hierarchical):

- Các phần tử được phân chia thành các cấp độ và có quan hệ với nhau.
- Loại phổ biến là dạng cây (tree):
 - Cây sẽ có một phần tử cao nhất (root), sau đó phân chia thành các nhánh.
 - Mỗi phần tử được gọi là node, ở cuối mỗi nhánh gọi là lá (leaf).

Collections



Phi tuyến tính (nonlinear): phân nhóm (grouped):

- Các phần tử không thể sắp xếp theo trật tự nào.
- Có 3 loại:
 - Tập hợp (set): chứa các giá trị không trùng lặp.
 - Đồ thị (graph): là tập hợp các nút (node) và cạnh (edge); cạnh nối các nút.
 - Mạng (network): là dạng đặc biệt của đồ thị; mỗi cạnh được gán một trọng số.

List

- 📖 Là một dạng mảng nhưng cho phép tăng giảm số lượng phần tử trong mảng một cách linh động.
- 📖 Có nhiều loại list được xây dựng sẵn trong C# và .NET Framework.
- 📖 Xét 3 loại:
 - ArrayList.
 - List<T>.
 - SortedList.

ArrayList

- 📖 Là lớp thực thi cho một loại danh sách (list) có thể:
 - Chứa dữ liệu thuộc bất cứ kiểu dữ liệu nào
 - Dễ dàng thêm, bớt, tìm kiếm phần tử trong danh sách.
 - Các phần tử có thể có kiểu dữ liệu khác nhau.
 - Thuộc **System.Collections**
- 📖 Khai báo và khởi tạo ArrayList có 3 cách:
 - Cách 1: khởi tạo một ArrayList rỗng

```
ArrayList list = new ArrayList();
```
 - Cách 2: khởi tạo và cung cấp số lượng phần tử ban đầu

```
ArrayList list = new ArrayList(10);
```
 - Cách 3: khởi tạo và chứa các phần tử được sao chép từ một Collections khác hoặc cung cấp các phần tử ban đầu.

```
ArrayList myList = new ArrayList(list); hoặc  
ArrayList myList = new ArrayList(new object[] { "abc",  
1, true });
```

ArrayList

Một số phương thức của ArrayList:

○ Thêm phần tử vào cuối danh sách:

- Thêm giá trị value: `Add(object value)`

```
list.Add("XYZ");
```

- Thêm giá trị là mảng các object: `AddRange(ICollection c)`

```
list.AddRange(new[] { "XYZ", "JQK" });
```

```
list.AddRange(new object[] { "a", 2 });
```

○ Chèn phần tử vào vị trí bất kỳ trong danh sách:

- Chèn value vào vị trí index: `Insert(int index, object value)`

```
list.Insert(2, 13);
```

- Chèn mảng các object vào vị trí index: `InsertRange(int index, ICollection c)`

```
list.InsertRange(3, new[] { "a", "b" });
```

```
list.InsertRange(3, new object[] { "a", 1 });
```

ArrayList

- Xóa phần tử:

- Xóa phần tử có giá trị value xuất hiện đầu tiên trong ArrayList :

`Remove(object value)`

`list.Remove("a");`

- Xóa phần tử ở vị trí xác định: `RemoveAt(int index)`

`list.RemoveAt(3);`

- Xóa bỏ nhóm phần tử theo số lượng count tính từ vị trí index :

`RemoveRange(int index, int count)`

`list.RemoveRange(1, 3);`

- Xóa bỏ tất cả các phần tử: `Clear()`

`list.Clear();`

ArrayList

- Truy xuất phần tử dùng chỉ số tương tự mảng, và phải ép kiểu từ object sang kiểu dữ liệu cụ thể.

```
var i = (int)list[1];  
string s = list[0] as string;
```

- Duyệt ArrayList dùng vòng lặp tương tự mảng hoặc sử dụng foreach

```
for (int i = 0; i < list.Count; i++)  
{  
    Console.WriteLine($"{list[i]}");  
}
```

Hoặc

```
foreach (object item in list)  
{  
    Console.WriteLine($"{item}");  
}
```

ArrayList

📖 Một số thuộc tính và phương thức khác:

- `int Count { get; }`: trả về số lượng phần tử hiện có trong ArrayList.
- `int Capacity { get; set; }`: trả về hoặc thiết lập sức chứa của ArrayList, nếu thêm mới phần tử vào vượt quá sức chứa thì ArrayList sẽ tự tăng lên.
- `bool Contains(object item)`: kiểm tra xem một giá trị có trong ArrayList không.
- `int IndexOf(object value)`: trả về vị trí đầu tiên xuất hiện của giá trị value trong ArrayList.
- `void Sort()`: sắp xếp các phần tử trong ArrayList tăng dần.
- `object[] ToArray()`: chuyển ArrayList về thành mảng

ArrayList



Ưu điểm:

- Khả năng thêm mới, xóa bỏ, chèn phần tử rất linh hoạt.
- Không giới hạn kiểu dữ liệu của các phần tử.
- Sử dụng đơn giản và không khác biệt nhiều so với mảng.



Nhược điểm:

- Các phần tử đều thuộc kiểu object, khi lưu trữ các phần tử kiểu giá trị (int, float, char...) sẽ xảy ra qua trình boxing/unboxing.
- Thêm phần tử phải chuyển về kiểu object.
- Lấy phần tử ra phải ép kiểu về dạng ban đầu của dữ liệu, nếu không sẽ báo lỗi.

List<T>

- 📖 Là một kiểu dữ liệu.
- 📖 Tương tự như mảng và ArrayList.
- 📖 Có các thuộc tính và phương thức với tên gọi và cách sử dụng giống với ArrayList.
- 📖 Cách thức truy xuất phần tử trong List<T> tương tự ArrayList.
- 📖 Kiểu dữ liệu cơ sở của các phần tử được xác định ở giai đoạn khởi tạo(theo kiểu generic).Do đó, các phần tử bắt buộc phải cùng kiểu dữ liệu T.
- 📖 Khi truy xuất, không cần phải ép kiểu.
- 📖 Thuộc **System.Collections.Generic**
- 📖 Có thể sử dụng với thư viện LINQ để truy vấn dữ liệu.
- 📖 List<T> được sử dụng rộng rãi trong C#

List<T>

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

```
enum Country
{
    US, VN, UK, DE, FR, PO
}
```

List<T>

```
Console.Title = "List with custom type";
List<Dinosaurs> dinosaurs = new List<Dinosaurs>
{
    new Dinosaurs { Name = "Tyrannosaurus", Age = 10, Country
=Country.US },
    new Dinosaurs { Name = "Amargasaurus", Age = 20, Country
= Country.DE },
    new Dinosaurs { Name = "Deinonychus", Age = 10, Country =
Country.UK }
};
dinosaurs.Add(new Dinosaurs { Name = "Mamenchisaurus", Age =
15, Country = Country.FR });

foreach (var d in dinosaurs)
{
    Console.WriteLine($"{d.Name}, {d.Age} years old, from
{d.Country}");
}
```

SortedList

- 📖 Lưu trữ dữ liệu dưới dạng các cặp khóa - giá trị (Key - Value).
- 📖 Là một Hashtable, giá trị được sắp xếp theo khóa (tự động).
- 📖 Có thể định nghĩa cách sắp xếp của Key
- 📖 Trong danh sách, khóa không được trùng và null
- 📖 Truy xuất các phần tử thông qua khóa hoặc chỉ số phần tử (như ArrayList).
- 📖 Là sự kết hợp giữa ArrayList và Hashtable.
- 📖 Thuộc **System.Collections**

SortedList

📖 Khai báo và khởi tạo có 5 cách:

- Cách 1: khởi tạo một SortedList rỗng

```
SortedList SL = new SortedList();
```

- Cách 2: khởi tạo và cung cấp số lượng phần tử ban đầu

```
SortedList SL = new SortedList(10);
```

- Cách 3: khởi tạo và chứa các phần tử được sao chép từ một SortedList.

```
SortedList SL1 = new SortedList(SL);
```

- Cách 4: khởi tạo và chỉ ra cách sắp xếp của key

```
SortedList SL = new SortedList(new Comparer());
```

- Cách 5: khởi tạo và chứa các phần tử được sao chép từ một SortedList đồng thời chỉ ra cách sắp xếp của key

```
SortedList SL1 = new SortedList(SL, new Comparer());
```

SortedList

📖 Một số thuộc tính và phương thức:

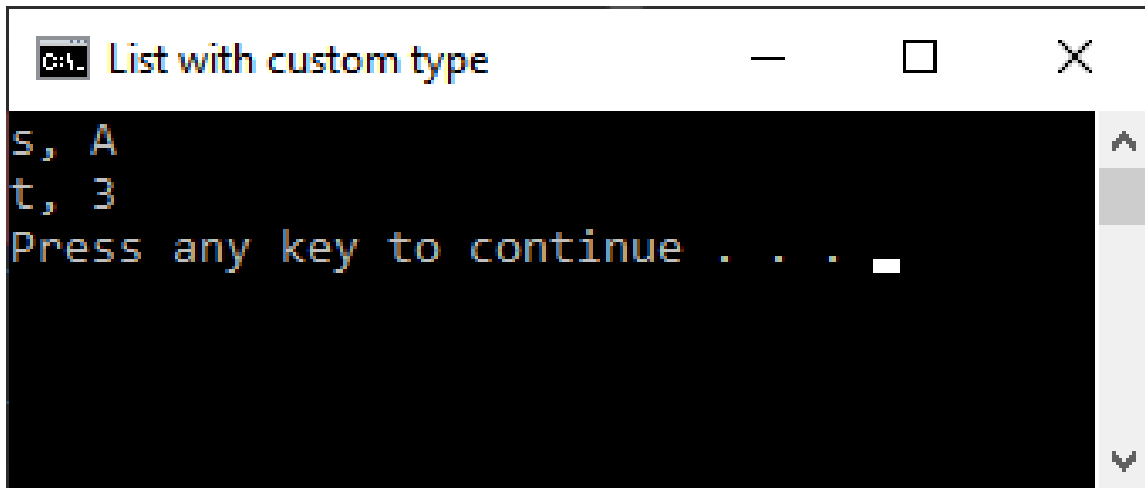
- `int Count { get; }`: trả về số lượng phần tử hiện có trong `SortedList`.
- `int Capacity { get; set; }`: trả về, hoặc thiết lập sức chứa của `SortedList`.
- `ICollection Keys { get; }`: trả về danh sách các key
- `ICollection Values { get; }`: trả về danh sách các value
- `Add(object key, object value)`: thêm giá trị.
- `Remove(object key)`: xóa phần tử với key xác định.
- `RemoveAt(int index)`: xóa phần tử tại vị trí index.
- `Clear()`: Xóa tất cả các phần tử trong `SortedList`
- `Clone()`: tạo bản sao từ `SortedList` hiện tại
- `CopyTo(Array array, int arrayIndex)`: sao chép các phần tử trong `SortedList` sang mảng một chiều từ index

SortedList

- `bool ContainsKey(object key)`: kiểm tra xem có khóa trong SortedList không
- `bool ContainsValue(object value)`: kiểm tra xem có giá trị value trong SortedList không
- `object GetKey(int index)`: trả về giá trị khóa tại index.
- `object GetByIndex(int index)`: trả về giá trị value tại index.
- `IList GetKeyList()`: trả về danh sách các khóa.
- `IList GetValueList()`: trả về danh sách các giá trị.
- `int IndexOfKey(object key)`: trả về chỉ số index của khóa.
- `int IndexOfValue(object value)`: trả về chỉ số index của giá trị.

SortedList

```
var sort = new SortedList();  
sort.Add("t", 3);  
sort.Add("s", "A");  
  
for (int i = 0; i < sort.Count; i++)  
{  
    Console.WriteLine($"{sort.GetKey(i)}, {sort.GetByIndex(i)}");  
}
```

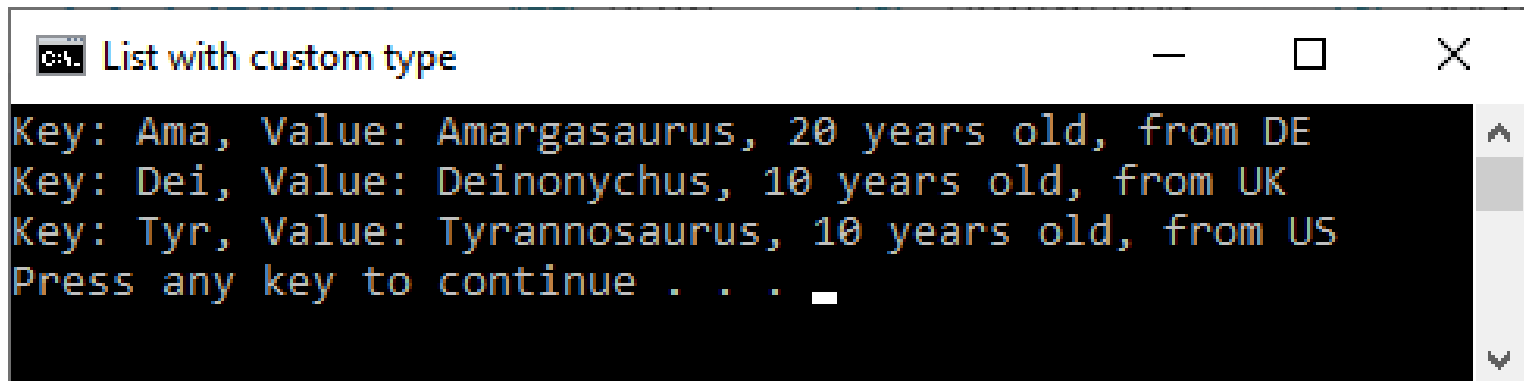


```
C:\> List with custom type  
s, A  
t, 3  
Press any key to continue . . .
```


SortedList<TKey, TValue>

```
var sortL = new SortedList<string, Dinosaurs>{  
    {"Tyr", new Dinosaurs { Name = "Tyrannosaurus", Age = 10,  
Country = Country.US } },  
    {"Ama", new Dinosaurs { Name = "Amargasaurus", Age = 20,  
Country = Country.DE } },  
    { "Dei", new Dinosaurs { Name = "Deinonychus", Age = 10,  
Country = Country.UK } }  
};
```

```
foreach (var d in sortL){  
    Console.WriteLine($"Key: {d.Key}, Value: {d.Value.Name},  
{d.Value.Age} years old, from {d.Value.Country}");  
}
```



```
Key: Ama, Value: Amargasaurus, 20 years old, from DE  
Key: Dei, Value: Deinonychus, 10 years old, from UK  
Key: Tyr, Value: Tyrannosaurus, 10 years old, from US  
Press any key to continue . . .
```



Anonymous, nullable, dynamic

Anonymous

- 📖 Anonymous type (kiểu dữ liệu vô danh) là loại kiểu dữ liệu tạm thời khi compiler tự suy đoán cấu trúc khi object của nó được khởi tạo.
- 📖 Khởi tạo thông qua cú pháp khởi tạo object theo thuộc tính.
- 📖 Được dùng chứa dữ liệu và là dữ liệu chỉ đọc.
- 📖 Khi object của anonymous được khởi tạo thì giá trị không thay đổi được.
- 📖 Không thể dùng cách thức khai báo biến thông thường cho Anonymous, thay vào đó dùng từ khóa **var**.
- 📖 Dùng phổ biến trong LINQ với phương thức Select.

Anonymous

📖 **VD:** tạo một object với 3 thuộc tính

```
var dino = new { Name = "T-Rex", Age = 10, Country = "US" };
```

📖 **VD:** Khai báo biến trước và dùng để tạo object

```
string name = "T-Rex";  
int age = 10;  
var dino = new { name, age };
```

📖 Truy xuất như là truy xuất object bình thường

```
Console.WriteLine($"Name: {dino.Name}, Age: {dino.Age}");
```

Anonymous

- 📖 **VD:** khai báo trước 3 biến và dùng chúng để tạo object

```
int id = 0;  
string name = "Lucky";  
int age = 15;
```

- 📖 Tạo object của kiểu vô danh từ 3 biến, C# tự suy đoán ra tên property

```
var obj2 = new { id, name, age };  
Console.WriteLine ($"ID: {obj2.id}, Name: {obj2.name}, Age:  
{obj2.age}");
```

Nullable

- ❏ Đối với kiểu dữ liệu thuộc nhóm reference type, biến có thể nhận giá trị null (biến không có giá trị)
- ❏ Nhưng với biến kiểu value thì không thể nhận giá trị null.
- ❏ Nullable type là kiểu dữ liệu value type có thể nhận giá trị null mang ý nghĩa biến không có giá trị.
- ❏ VD: kiểu bool có 2 giá trị `true`, `false`; thì nullable bool sẽ có 3 giá trị: `true`, `false` và `null`
- ❏ Dùng modifier ? Đặt sau **value type** để chuyển thành nullable type:

```
bool? flag = null;
```

```
int? i = null;
```

- ❏ Hai thuộc tính đặc trưng của nullable type:
 - `bool` HasValue { `get`; }: Trả về true nếu biến khác null
 - `T` Value { `get`; }: trả về giá trị value type của biến

Nullable



Một số phép toán:

- Null coalescing: `<variable> ?? <value>` Nếu variable khác null biểu thức sẽ nhận giá trị của variable; ngược lại biểu thức nhận giá trị value

```
int? i = null;
```

```
int i1 = i ?? 10;    //KQ: i1= 10;
```

- Null conditional: khi truy xuất vào giá trị null sẽ gây lỗi

```
> int? i = null;
```

```
> i.Value
```

```
System.InvalidOperationException: Nullable object must have a value.
```

```
+ System.ThrowHelper.ThrowInvalidOperationException(System.ExceptionResource)
```

```
+ Nullable<T>.get_Value()
```

```
> string s = null;
```

```
> s.ToUpper();
```

```
System.NullReferenceException: Object reference not set to an instance of an object.
```

```
>
```

Nullable

- Do đó, cần phải kiểm tra giá trị của biến trước khi thực hiện bất kỳ thao tác gì. Nếu biến có giá trị khác null mới thực hiện thao tác đó. Ví dụ:

```
if (str != null) str.ToLower();
```

- Dùng dấu **?** sau tên biến và trước phương thức truy xuất để không lỗi, và phương thức chỉ thực hiện khi biến không null

```
> s?.ToUpper()  
null
```

- Dấu **?** sau tên biến và trước phép toán truy xuất phần tử có tên gọi là **null conditional operator**, giúp kiểm tra xem biến có bằng null không. Nếu biến khác null thì mới thực hiện phép toán truy xuất phần tử.

Định kiểu tĩnh và động

- 📖 Với JavaScript hay PHP, có một sự khác biệt rất lớn về việc khai báo và sử dụng biến. Khi khai báo biến, không cần chỉ định kiểu dữ liệu cụ thể của nó. Giá trị gán cho một biến không được xác định và kiểm tra ở giai đoạn dịch mà là ở giai đoạn thực thi. Đặc thù như vậy của ngôn ngữ gọi là định kiểu động (dynamically typed).
- 📖 Ngược lại, C# bắt buộc kiểu của biến phải được xác định và kiểm tra ở giai đoạn dịch. Đặc thù đó của C# được gọi là định kiểu tĩnh (statically typed). Ngôn ngữ định kiểu tĩnh như C# rất an toàn khi viết code. Nhưng đồng thời, nó lại thiếu đi sự linh hoạt của ngôn ngữ định kiểu động.
- 📖 Bắt đầu từ C# 4.0, một kiểu dữ liệu đặc biệt được đưa vào để hỗ trợ những nhu cầu về định kiểu động trong lập trình: kiểu **dynamic**.

Dynamic

- 📖 Dynamic (kiểu động): là một kiểu dữ liệu đặc biệt trong C#
- 📖 Cho phép gán bất kỳ giá trị nào vào biến.
- 📖 Cho phép sử dụng thuộc tính và phương thức như biến bình thường.
- 📖 Tuy nhiên, trong Visual Studio, Intellisense sẽ không hoạt động với biến dynamic.
- 📖 C# không xác định được kiểu biến cụ thể ở giai đoạn viết code và dịch.
- 📖 Do đó, khi viết phải xác định đây là biến thuộc kiểu gì và xác định kiểu đó có các phương thức và thuộc tính gì.
- 📖 Nếu viết sai thuộc tính và phương thức của kiểu dữ liệu thì vẫn không báo lỗi khi viết code và biên dịch, nhưng sẽ báo lỗi khi thực thi.

Dynamic



VD

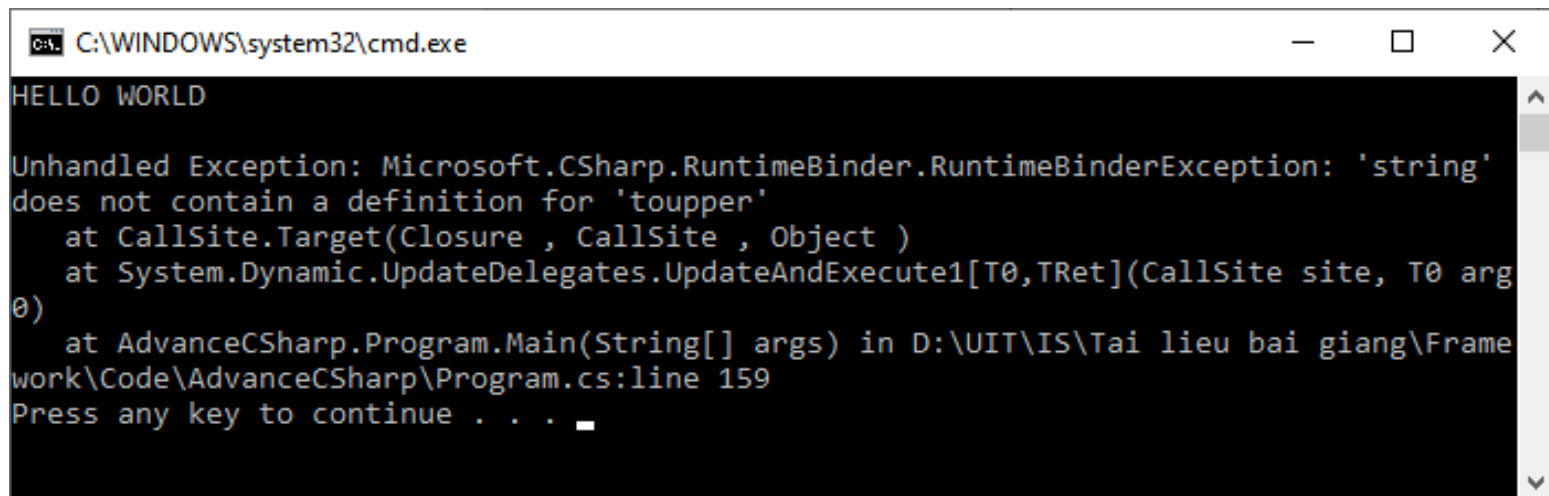
```
> dynamic d = "Hello";  
> d  
"Hello"  
> d.GetType()  
[System.String]  
> d = true;  
> d.GetType()  
[System.Boolean]  
>
```

Dynamic

- 📖 Khi viết code, hàm toupper không bị báo lỗi

```
dynamic str = "Hello world";  
Console.WriteLine(str.ToUpper());  
Console.WriteLine(str.toupper());
```

- 📖 Khi thực thi chương trình sẽ bị báo lỗi



The screenshot shows a Windows command prompt window titled "C:\WINDOWS\system32\cmd.exe". The output of the program is as follows:

```
HELLO WORLD  
Unhandled Exception: Microsoft.CSharp.RuntimeBinder.RuntimeBinderException: 'string'  
does not contain a definition for 'toupper'  
    at CallSite.Target(Closure , CallSite , Object )  
    at System.Dynamic.UpdateDelegates.UpdateAndExecute1[T0,TRet](CallSite site, T0 arg  
0)  
    at AdvanceCSharp.Program.Main(String[] args) in D:\UIT\IS\Tai lieu bai giang\Fram  
ework\Code\AdvanceCSharp\Program.cs:line 159  
Press any key to continue . . .
```



Exceptions

Exceptions

- ❏ Exception (ngoại lệ): là một vấn đề phát sinh trong quá trình thực thi chương trình.
- ❏ Khi ngoại lệ xảy ra, nếu không xử lý thì chương trình sẽ kết thúc.
- ❏ Vd: khi chia với mẫu số bằng 0, sai đường dẫn file, vượt kích thước của mảng.
- ❏ C# xử lý ngoại lệ bằng:

- Khối lệnh **try-catch-finally**
- Sử dụng từ khóa **throw**: phát ra thông báo ngoại lệ

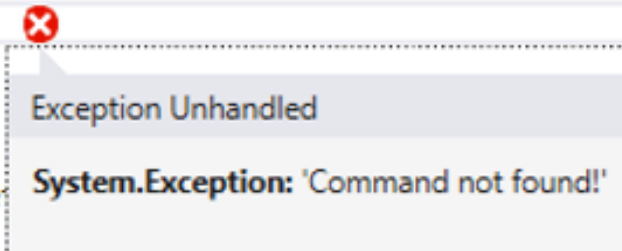
throw new Exception("thông tin về lỗi");

Cấu trúc này khởi tạo một object của lớp Exception và gửi object này cho cơ chế thông báo lỗi của .NET framework. Khi gọi lệnh throw, luồng điều khiển của chương trình sẽ thay đổi.

Exceptions

- 📖 Khi một ngoại lệ được phát ra ở một vị trí bất kỳ trong chương trình, việc thực thi của chương trình sẽ dừng lại. Nếu chương trình đang chạy ở chế độ Debug, trình soạn thảo code sẽ được mở ra và đoạn code bị lỗi sẽ được đánh dấu giúp cho người lập trình xác định vị trí và nguyên nhân gây lỗi.

```
var req = new Request(request);  
if (!_routingTable.ContainsKey(req.Route))  
    throw new Exception("Command not found!");  
if (req.Parameter == null)  
    _routingTable[req.Route]?.Invoke();  
else  
    _routingTable[req.Route]?.Invoke(req.Parameter);
```

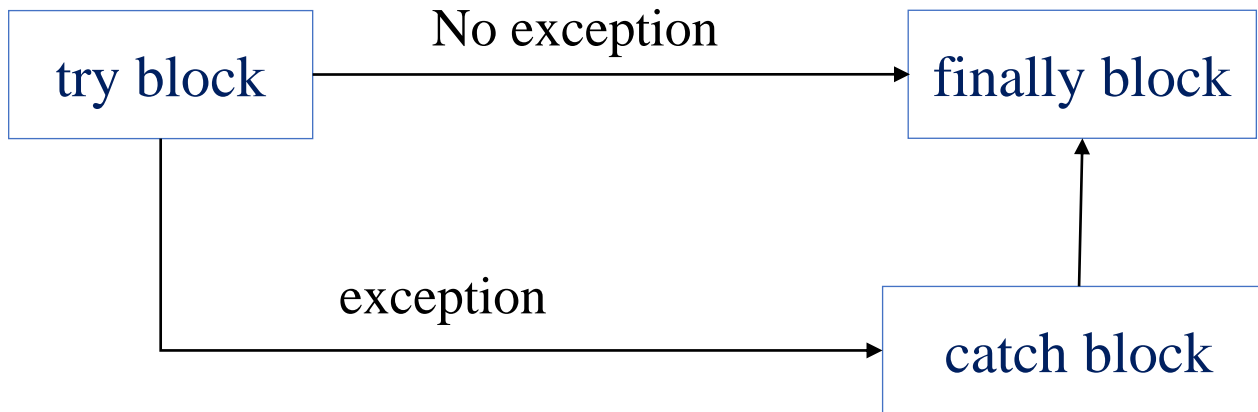


- 📖 Tuy nhiên, cơ chế thông báo lỗi mặc định của .NET framework tương đối không thân thiện với người dùng.
- 📖 .NET cung cấp cho các chương trình tính năng bắt và xử lý ngoại lệ để tự mình xác định xem khi xảy ra lỗi (ngoại lệ) thì sẽ làm gì.

Exceptions

❏ Khối `try-catch-finally`:

- `try`: để xác định một khối lệnh có khả năng xảy ra ngoại lệ
- `catch`: được dùng để bắt ngoại lệ phát sinh ở `try` và xử lý ngoại lệ này. Mỗi `catch` chỉ bắt được một ngoại lệ.
- `finally`: dùng để đảm bảo khối lệnh được thực thi dù ngoại lệ có xảy ra hay không. VD: Đóng file, đóng connection.



Exceptions

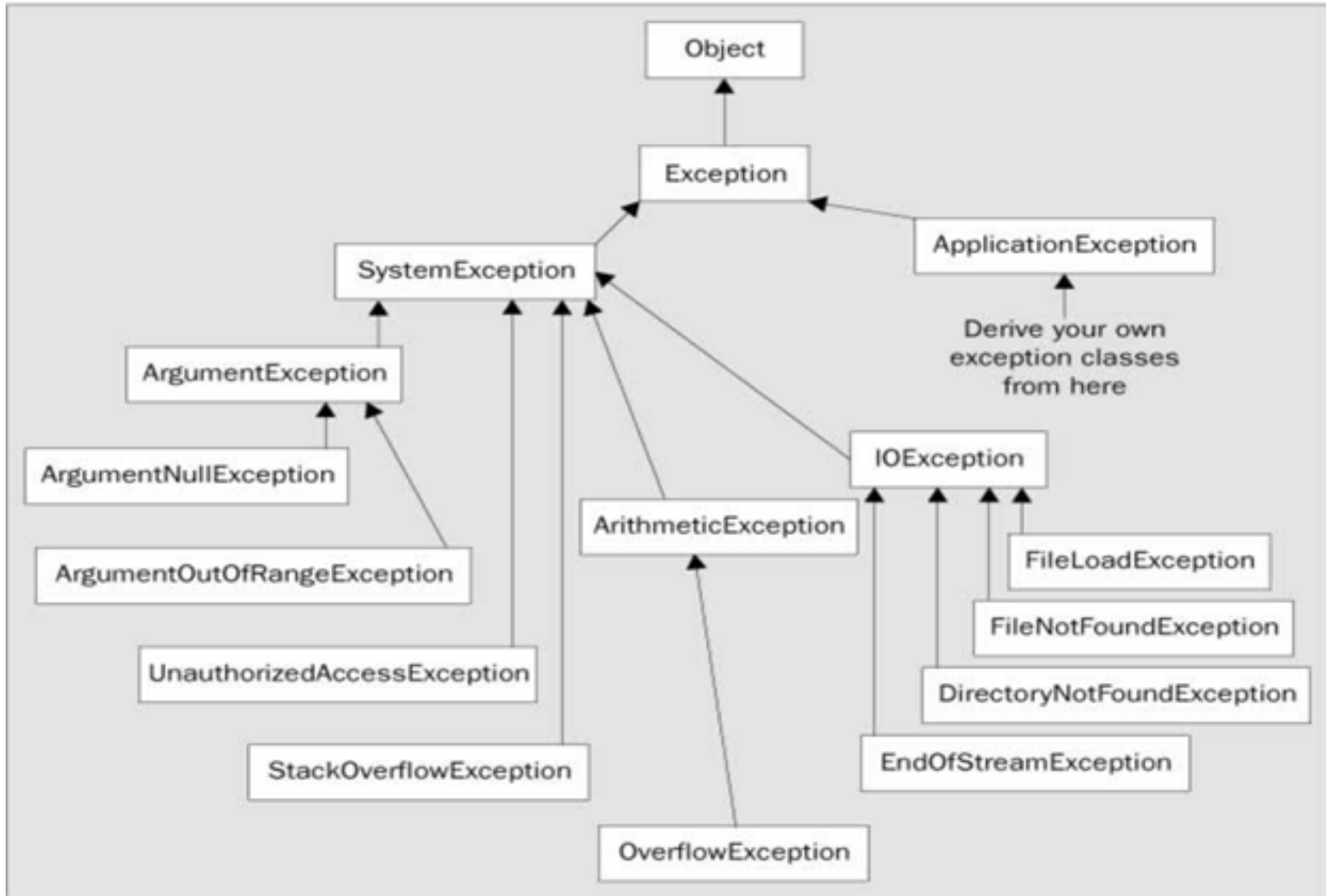
📖 Cú pháp:

```
try
{
    // statements causing exception
}
catch( ExceptionName e1 )
{
    // error handling code
}
catch( ExceptionName e2 )
{
    // error handling code
} finally {
    // statements to be executed
}
```

Exceptions



Một số lớp xử lý ngoại lệ của C#



Exceptions

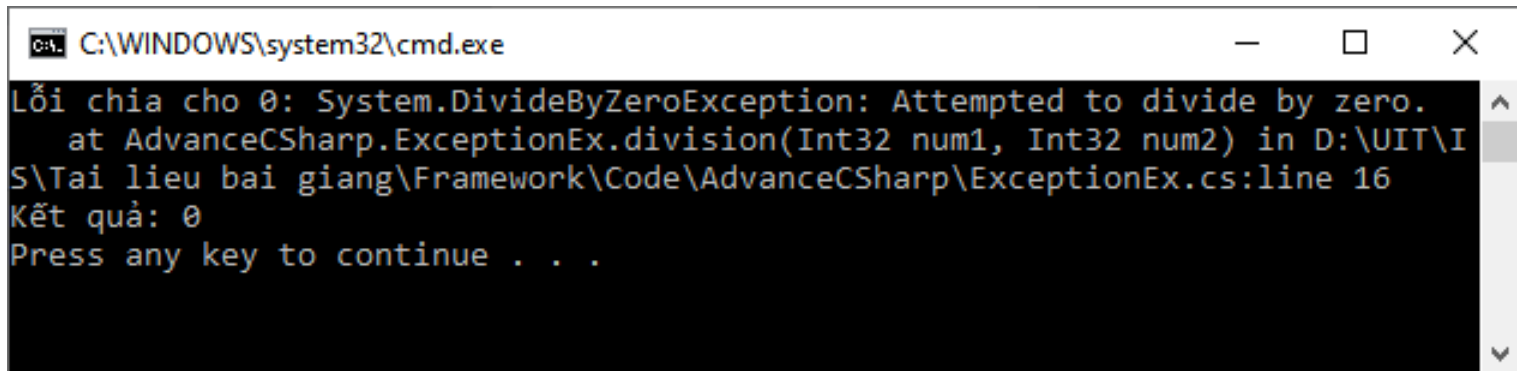


Một số lớp ngoại lệ thuộc `System.SystemException`

Lớp ngoại lệ	Mô tả
<code>System.IO.IOException</code>	Xử lý lỗi nhập xuất
<code>System.IndexOutOfRangeException</code>	Xử lý các lỗi vượt chỉ số của mảng.
<code>System.ArrayTypeMismatchException</code>	Xử lý các lỗi không phù hợp với các kiểu mảng.
<code>System.NullReferenceException</code>	Xử lý các lỗi đối tượng null.
<code>System.DivideByZeroException</code>	Xử lý các lỗi chia cho số không.
<code>System.InvalidCastException</code>	Xử lý các lỗi phát sinh trong quá trình phân loại.
<code>System.OutOfMemoryException</code>	Xử lý các lỗi vượt bộ nhớ
<code>System.StackOverflowException</code>	Xử lý các lỗi phát sinh từ tràn stack.

Exceptions

```
try
{
    result = 20 / 0;
}
catch (DivideByZeroException e)
{
    Console.WriteLine("Lỗi chia cho 0: {0}", e);
}
finally
{
    Console.WriteLine("Kết quả: {0}", result);
}
```



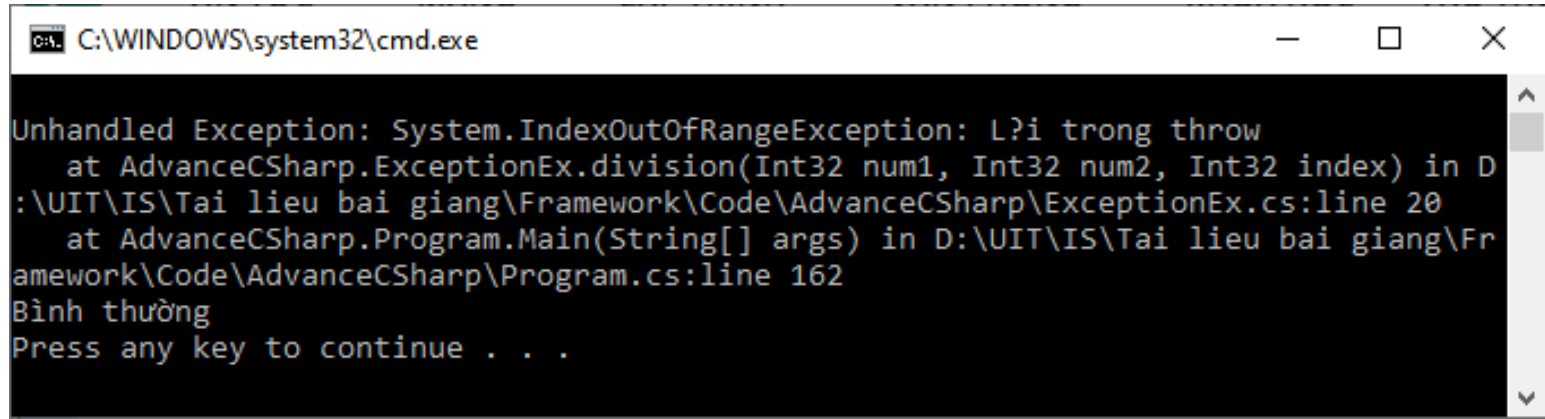
```
C:\WINDOWS\system32\cmd.exe
Lỗi chia cho 0: System.DivideByZeroException: Attempted to divide by zero.
   at AdvanceCSharp.ExceptionEx.division(Int32 num1, Int32 num2) in D:\UIT\I
S\Tai lieu bai giang\Framework\Code\AdvanceCSharp\ExceptionEx.cs:line 16
Kết quả: 0
Press any key to continue . . .
```

Exceptions

```
try
{
    int index = 5;
    if (index < 10)
        throw new IndexOutOfRangeException("Trong throw");
}
catch (IndexOutOfRangeException e)
{
    Console.WriteLine("Bắt lỗi trong catch");
}
finally
{
    Console.WriteLine("Bình thường");
}
```

Exceptions

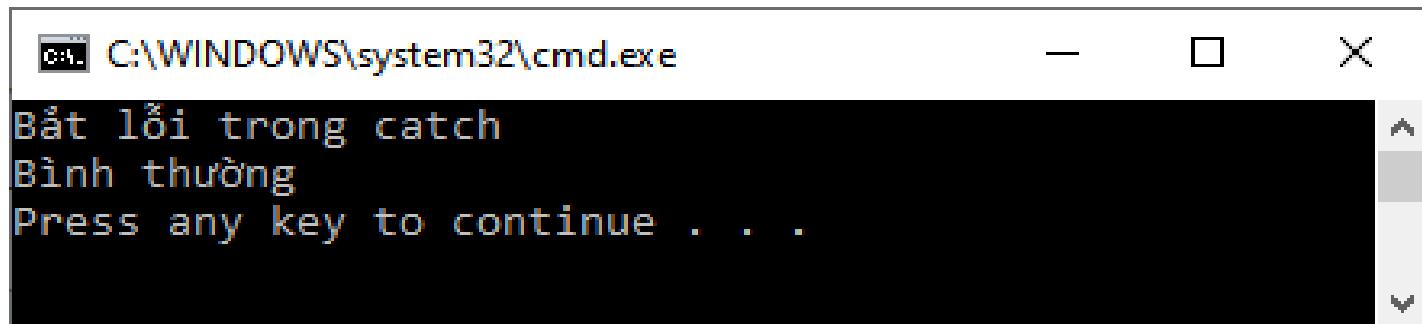
Trường hợp không có catch



```
C:\WINDOWS\system32\cmd.exe

Unhandled Exception: System.IndexOutOfRangeException: Lỗi trong throw
   at AdvanceCSharp.ExceptionEx.division(Int32 num1, Int32 num2, Int32 index) in D:\UIT\IS\Tai lieu bai giang\Framework\Code\AdvanceCSharp\ExceptionEx.cs:line 20
   at AdvanceCSharp.Program.Main(String[] args) in D:\UIT\IS\Tai lieu bai giang\Framework\Code\AdvanceCSharp\Program.cs:line 162
Bình thường
Press any key to continue . . .
```

Trường hợp có catch



```
C:\WINDOWS\system32\cmd.exe

Bắt lỗi trong catch
Bình thường
Press any key to continue . . .
```

Exceptions

📖 Ngoại lệ tự định nghĩa

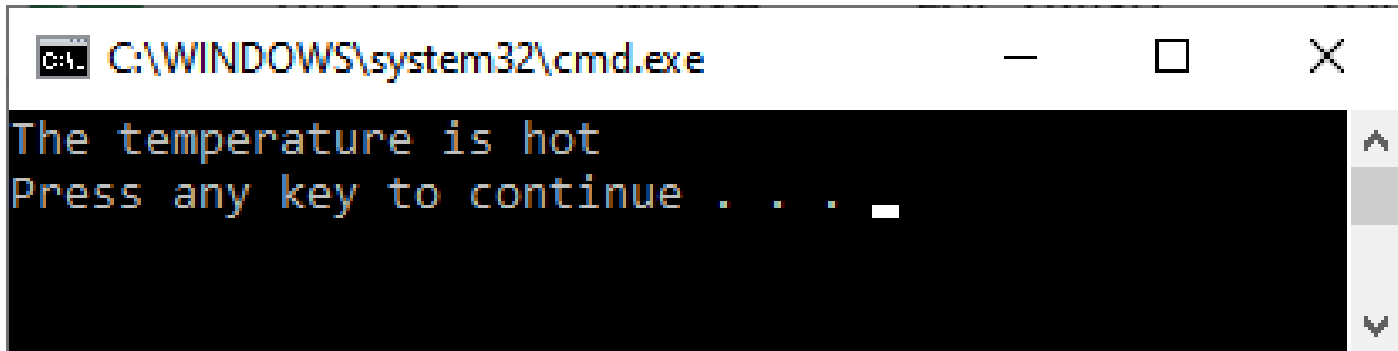
```
class TempIsHotException : Exception
{
    public TempIsHotException(string mess) : base(mess) { }
}

class Temperature {
    public void showTemp(int temp)
    {
        if (temp >= 30) throw (new TempIsHotException("hot"));
        else Console.WriteLine("Temperature is {0}",temp);
    }
}
```

Exceptions

Hàm Main:

```
Temperature temp = new Temperature();  
try  
{  
    temp.showTemp(35);  
}  
catch(TempIsHotException e)  
{  
    Console.WriteLine("The temperature is {0}" , e.Message);  
}
```



The screenshot shows a Windows command prompt window titled "C:\WINDOWS\system32\cmd.exe". The window has a black background with white text. The text displayed is "The temperature is hot" followed by "Press any key to continue . . .". A white cursor is visible at the end of the second line. The window has standard Windows window controls (minimize, maximize, close) in the title bar.



Q & A

Giảng viên: Tạ Việt Phương
E-mail: phuongtv@uit.edu.vn