Microsoft®
Visual C#®

CHUYÊN ĐỀ NGÔN NGỮ LẬP TRÌNH 1

Người biên soạn: Hồ Quang Thái (MSCB: 2299)
BM. Công Nghệ Phần Mềm, Khoa CNTT&TT
Email: hqthai@cit.ctu.edu.vn
Số tín chỉ: 2 (20 LT + 20TH)

Chương 3

LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG C#

Các phương pháp lập trình truyền thống

***Lập trình tuyến tính*: Chương trình sẽ được thực hiện tuần tự từ đầu đến cuối, lệnh này kế tiếp lệnh kia cho đến khi kết thúc chương trình.**

Đặc trưng là đơn giản và đơn luồng.

- ***Ưu điểm*: Chương trình đơn giản, dễ hiểu. Ứng dụng cho các chương trình đơn giản.**
- ***Nhược điểm***: Với các ứng dụng phức tạp, người ta không thể dùng lập trình tuyến tính để giải quyết.

Các phương pháp lập trình truyền thống

***Lập trình cấu trúc*:** Chương trình chính được chia nhỏ thành các chương trình con và mỗi chương trình con thực hiện một công việc xác định. Chương trình chính sẽ gọi đến chương trình con theo một giải thuật, hoặc một cấu trúc được xác định trong chương trình chính.

- Ngôn ngữ: Pascal, C, C++, …

  Đặc trưng : ***Chương trình = Cấu trúc DL + Giải thuật***

  - Cấu trúc dữ liệu là cách tổ chức dữ liệu, cách mô tả bài toán dưới dạng ngôn ngữ lập trình.
  - Giải thuật là một quy trình để thực hiện một công việc xác định.

Các phương pháp lập trình truyền thống

Ưu điểm:

- Chương trình sáng sủa, dễ hiểu, dễ theo dõi.
- Tư duy giải thuật rõ ràng.

Nhược điểm:

- Lập trình cấu trúc không hỗ trợ việc sử dụng lại mã nguồn: Giải thuật luôn phụ thuộc chặt chẽ vào cấu trúc dữ liệu, do đó, khi thay đổi cấu trúc dữ liệu, phải thay đổi giải thuật, nghĩa là phải viết lại chương trình.
- Không phù hợp với các phần mềm lớn: tư duy cấu trúc với các giải thuật chỉ phù hợp với các bài toán nhỏ, nằm trong phạm vi một module của chương trình.

Phương pháp lập trình hướng đối tượng

Để khắc phục được hai hạn chế này khi giải quyết các bài toán lớn, người ta xây dựng một phương pháp tiếp cận mới, là phương pháp lập trình hướng đối tượng, với hai mục đích chính:

- Đóng gói dữ liệu để hạn chế sự truy nhập tự do vào dữ liệu, không quản lí được.
- Cho phép sử dụng lại mã nguồn, hạn chế việc phải viết lại mã từ đầu cho các chương trình.

Phương pháp lập trình hướng đối tượng

Đặc trưng:

- ***Đóng gói dữ liệu***: dữ liệu luôn được tổ chức thành các thuộc tính của lớp đối tượng. Việc truy nhập đến dữ liệu phải thông qua các phương thức của đối tượng lớp.
- ***Sử dụng lại mã nguồn***: việc sử dụng lại mã nguồn được thể hiện thông qua cơ chế kế thừa. Cơ chế này cho phép các lớp đối tượng có thể kế thừa từ các lớp đối tượng khác.

Phương pháp lập trình hướng đối tượng

Ưu điểm :

- Không còn nguy cơ dữ liệu bị thay đổi tự do trong chương trình. Vì dữ liệu đã được đóng gói vào các đối tượng. Phải thông qua các phương thức cho phép của đối tượng.
- Khi thay đổi cấu trúc dữ liệu của một đối tượng, không cần thay đổi các đổi mã nguồn của các đối tượng khác, mà chỉ cần thay đổi một số hàm thành phần của đối tượng bị thay đổi.
- Có thể sử dụng lại mã nguồn, tiết kiệm tài nguyên.
- Phù hợp với các dự án phần mềm lớn, phức tạp.

Đối tượng

- **Đối tượng** là một thực thể hoạt động khi chương trình đang chạy. Một đối tượng được xác định bằng ba yếu tố:
  - ***Định danh đối tượng***: xác định duy nhất cho mỗi đối tượng trong hệ thống, nhằm phân biệt các đối tượng với nhau.
  - ***Trạng thái của đối tượng***: là sự tổ hợp của các giá trị của các thuộc tính mà đối tượng đang có.
  - ***Hoạt động của đối tượng***: là các hành động mà đối tượng có khả năng thực hiện được.

Ví dụ về đối tượng

Họ và tên: Nguyễn Thị B
Giới tính: Nữ
Nghề nghiệp: Sinh viên
Ngày sinh: 20/12/1987

Hành động: Ăn, nói, đọc, viết, đi, ….

Trạng thái: Vui, buồn, hờn, giận …

Lớp đối tượng

- ***Lớp đối tượng*** là *một khái niệm trừu tượng* dùng để chỉ tập hợp các đối tượng.
- Lớp được dùng để biểu diễn đối tượng, cho nên lớp cũng có thuộc tính và phương thức:
  - **Thuộc tính** của lớp tương ứng với thuộc tính của các đối tượng.
  - **Phương thức** của lớp tương ứng với các hành động của đối tượng.

Lớp đối tượng

Lưu ý: Một lớp có thể có một trong các khả năng sau:

- Hoặc chỉ có thuộc tính, không có phương thức.
- Hoặc chỉ có phương thức, không có thuộc tính.
- Hoặc có cả thuộc tính và phương thức, trường hợp này là phổ biến nhất.
- Đặc biệt, lớp không có thuộc tính và phương thức nào là các lớp trừu tượng. Các lớp này không có đối tượng tương ứng.

Tạo một lớp mới

- Sử dụng từ khóa **class**:

```
class <Tên lớp>{
        ...
}
```

- ***Ví dụ về một lớp đối tượng***

```
class MyRectangle
{
        int x, y;
        int GetArea() { return x * y; }
}
```

Tạo đối tượng từ lớp

- Việc tạo đối tượng chính là việc **khai báo và khởi tạo** một biến từ lớp đối tượng.
- Một đối tượng được gọi là một ***thể hiện*** (*instance*) của lớp. Đối tượng sẽ mang một bộ các thành viên giống như lớp đối tượng đó và lưu các giá trị riêng của riêng đối tượng đó.
- ***Ví dụ:*** *Tạo đối tượng r thuộc lớp MyRectangle*:

```
class MyClass {
      static void Main(){
        // Tạo ra một thể hiện của lớp MyRectangle
        MyRectangle r1 = new MyRectangle();
        MyRectangle r2 = new MyRectangle() { x = 10, y = 5 };
} }
```

Truy cập dữ liệu thành viên từ lớp

- Sau khi đã khai báo và khởi tạo lớp, ta có thể truy cập đến dữ liệu thành viên thể hiện của lớp bằng cách sử dụng dấu chấm (.) theo sau thể hiện
- ***Ví dụ:*** *Truy cập dữ liệu thành viên từ MyRectangle*:

```
static void Main()
{
        MyRectangle r = new MyRectangle();
        r.x = 10;
        r.y = 5;
        int a = r.GetArea(); // 50
}
```

Phương thức xây dựng

- Là phương thức được gọi thực hiện khi một đối tượng của một lớp đối tượng được khởi tạo.
- Trong **C#**, phương thức xây dựng có tên *trùng với tên lớp*. Mỗi lớp đều có một ***phương thức xây dựng mặc nhiên*** không có đối số. Tuy nhiên, nếu lớp đã có lớp do phương thức xây dựng do người dùng định nghĩa, phương thức xây dựng mặc nhiên sẽ không được tạo
- Phương thức xây dựng *không có kiểu trả về*.
- Có thể truy cập từ lớp khác nếu dùng từ khóa phạm vi truy cập là ***public***.
- Có thể nạp chồng phương thức xây dựng.

```
public MyRectangle() { x = 10; y = 5; }
```

Phương thức xây dựng

```
class MyRectangle
{
        public int x, y;

        public MyRectangle() { x = 10; y = 5; }

        public MyRectangle(int width, int height)
        {
                x = width; y = height;
        }

        static void Main()
        {
                MyRectangle r1 = new MyRectangle();
                MyRectangle r2 = new MyRectangle(20, 15);
        }
}
```

Từ khóa this

- Từ khóa ***this*** cho phép tham chiếu đến đối tượng hiện hành.
- Thông qua từ khóa ***this*** ta có thể truy cập đến các thành viên của đối tượng hiện hành.

```
class MyRectangle
{
        int x, y;
        public MyRectangle(int x, int y)
        {
                this.x = x; //Gán trường x = tham số x
                this.y = y;
        }
}
```

Từ khóa this

```
public MyRectangle()
{
        x = 10; y = 5;
}
public MyRectangle(int a)
{
        x = a; y = a;
}
public MyRectangle(int a, int b)
{
        x = a; y = b;
}
```

```
public MyRectangle() : this(10,5) {}
public MyRectangle(int a) : this(a,a) {}
public MyRectangle(int a, int b) { x = a; y = b; }
```

Phương thức hủy - Destructor

- Tổng thể, .**NET** tự động giải phóng các tài nguyên quản lý được. Tuy nhiên, khi chương trình sử dụng các tài nguyên không quản lý được như tập tin, kết nối mạng và những thành phần người dùng. Chúng ta sử dụng phương thức hủy. Phương thức hủy có tên là ký tự ~ theo sau là tên lớp.

```
class MyComponent {
        public System.ComponentModel.Component comp;
        public MyComponent(){
                comp = new System.ComponentModel.Component();
        }
        // Hàm hủy
        ~MyComponent(){
                comp.Dispose();
        }
}
```

Lớp Partial

- Một lớp có thể được chia ra nhiều tập tin nguồn bằng từ khóa **partial**.
- Những lớp này sẽ được kết hợp thành một lớp hoàn chỉnh bởi trình biên dịch.
- Các phần của lớp được chia đều phải có từ khóa **partial** và chia sẻ cùng mức độ truy cập.

```
// File1.cs
public partial class MyPartialClass {}
// File2.cs
public partial class MyPartialClass {}
```

Thừa kế

- Thừa kế là khả năng một lớp (*lớp con*) thừa hưởng những thành viên từ 1 lớp đã có (*lớp cha*).
- *Thí dụ:*
  - **Square** kế thừa từ **Rectangle**, hay **Square** còn gọi là một lớp dẫn xuất từ lớp **Rectangle**.
  - **Square** có tất cả các thành viên *(thuộc tính, phương thức)* có thể truy cập được của **Rectangle**, ngoại trừ các phương thức xây dựng và phương thức hủy.

Thừa kế

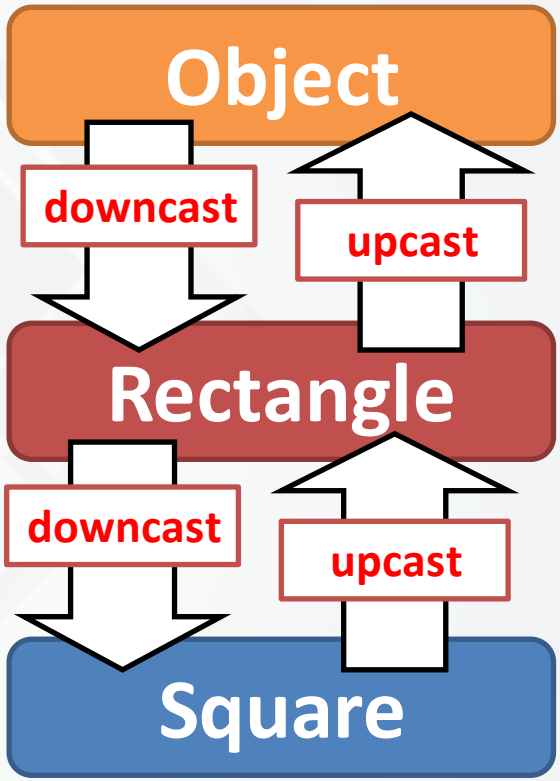
```
// Lớp cơ sở (lớp cha)
class Rectangle{
    public int x = 10, y = 10;
    public int GetArea() { return x * y; }
}
// Lớp dẫn xuất (lớp con)
class Square : Rectangle {}
```

- Một lớp có thể chỉ thừa kế từ một lớp cha. Nếu không, nó sẽ thừa kế từ lớp cơ sở gốc **Object.**
- **Object** không chỉ áp dụng cho lớp, nó còn áp dụng cho tất cả các loại kiểu tham chiếu khác.
- Lớp **Object** có một bộ các phương thức dùng chung

Ví dụ: **System.Int32** là kiểu cấu trúc. Nó có gốc là lớp **Object**

Thừa kế

- Nếu **Square** là lớp con của lớp **Rectangle** thì**:**
  - **Square** cũng là một **Object**.
  - Nó có thể được sử dụng ở bất kỳ đâu như một **Rectangle** hoặc một **Object**.
  - Nếu một *thể hiện* của **Square** được tạo ra, nó có thể ***tăng kiểu*** (*upcast*) sang **Rectangle**.
  - Khi nó được ***giảm kiểu*** (*downcast*) thì mọi thứ của lớp **Square** vẫn sẽ được bảo toàn. Nhưng điều ngược lại không đúng. Xét ví dụ sau =>



Thừa kế

```
Square s = new Square();
Rectangle r = s;
Square s2 = (Square)r; // Cho phép
Rectangle r2 = new Rectangle();
Square s3 = (Square)r2; // lỗi
```

Có thể tránh ngoại lệ trên bằng cách sử dụng:

- *Toán tử **is***: sẽ trả lại true nếu biểu thức bên trái có thể chuyển sang kiểu bên phải mà không xảy ra ngoại lệ.

```
Rectangle q = new Square(); //Khởi tạo q là Square
if (q is Square) { Square o = q; } // true
```

- *Toán tử **as***: là một cách chuyển kiểu, nhưng nếu không chuyển được trả lại ***null***

```
Rectangle r = new Rectangle();// Khai báo q là Rectangle
Square o = r as Square; // Chuyển không được, trả lại null
```

Boxing và Unboxing

- ***Boxing***: Là quá trình chuyển đổi một giá trị kiểu sơ cấp sang kiểu tham chiếu.
- **Thí dụ**:

```
int myInt = 5;
object myObj = myInt; // boxing
```

- ***Unboxing***: là quá trình ngược lại, chuyển đổi từ kiểu tham chiếu sang giá trị. Quá trình chuyển đổi phải rõ ràng, nếu đối tượng không thể được chuyển đổi được, lỗi khi thực thi sẽ xảy ra.
- **Thí dụ**:

```
myInt = (int)myObj; // unboxing
```

Định nghĩa lại thành viên

- Trong trường hợp lớp con cần thay đổi về nội dung của một thành viên nào đó nhưng giữ nguyên tên gọi, khi đó ta phải định nghĩa lại thành viên đó.
- ***Ví dụ:*** Định nghĩa lại phương thức **getArea**()

```
class Rectangle {
        public int x = 1, y = 10;
        public int GetArea() { return x * y; }
}
class Square : Rectangle {
        public int GetArea() { return 2 * x; }
}
```

Định nghĩa lại phương thức

- Khi định nghĩa lại, có 2 cách để tái định nghĩa: **ẩn** (*hiding*) hay **ghi đè** (*override*).
- Mặc định, phương thức được định nghĩa lại dùng cách **ẩn** *(hiding)*, nhưng trình biên dịch sẽ thông báo chúng ta phải định rõ cách định nghĩa lại phương thức. Để bỏ qua thông báo này, từ khóa **new** được sử dụng.

```
class Square : Rectangle
{
        public new int GetArea() { return 2 * x; }
}
```

Định nghĩa lại phương thức

- Với cách ***ghi đè*** *(override)*, từ khóa ***virtual*** và ***override*** phải được thêm vào cả lớp cha và lớp con ở cùng phương thức cần định nghĩa lại.

```
class Rectangle
{
        public int x = 1, y = 10;
        public virtual int GetArea() { return x * y; }
}

class Square : Rectangle
{
        public override int GetArea() { return 2 * x; }
}
```

Định nghĩa lại thành viên

- Nếu **Square** được *tăng kiểu* (***upcast***) đến **Rectangle** thì:
  - Cách dùng **new** cho phép truy cập lại phương thức đã ẩn của **Rectangle**.
  - Cách dùng **override** thì chỉ phương thức mới bị ghi đè được gọi.
  - Về cơ bản, từ khóa **new** chỉ định nghĩa lại phương thức trong lớp con, còn **override** thì định nghĩa lại ở cả hai lớp.

Từ khóa sealed

- Để ngăn định nghĩa lại, phương thức có thể khai báo là **sealed** đề từ chối từ khóa **virtual**

```
class MyClass {
        public sealed override int NonOverridable() {}
}
```

- Một lớp cũng có thể khai báo **sealed** để ngăn không cho lớp khác thừa kế nó

```
sealed class NonOverridable() {}
```

Từ khóa base

- Dùng để truy cập đến các thành viên của lớp cha từ nội bộ lớp con.

```
class Triangle : Rectangle {
    public override GetArea() { return base.GetArea()/2; }
}
```

- Từ khóa **base** cũng có thể dùng để gọi phương thức xây dựng

```
class Rectangle {
        public int x = 1, y = 10;
        public Rectangle(int a, int b) { x = a; y = b; }
}
class Square : Rectangle {
        public Square(int a) : base(a,a) {}
}
```

Từ khóa base

- ***Chú ý:*** Nếu phương thức xây dựng ở lớp con khai báo không có từ khóa **base** thì nó sẽ được trình biên dịch gọi ngầm phương thức xây dựng mặc định không tham số của lớp cha.

```
class Square : Rectangle {
      public Square(int a) {} // : base() được gọi ngầm
}
```

- Trong trường hợp lớp cha có phương thức xây dựng do người dùng định nghĩa, phương thức xây dựng mặc định sẽ không được tạo, khi đó sẽ xảy ra lỗi.

```
class Base { public Base(int a) {} }
class Derived : Base {} // Lỗi vì base() không được tạo
```

TỪ KHÓA ĐỊNH PHẠM VI TRUY CẬP

- Mỗi thành viên trong lớp đều có phạm vi truy cập.
- Có 5 từ khóa định phạm vi truy cập trong C#:
  - ✓ ***public***
  - ✓ ***protected***
  - ✓ ***internal***
  - ✓ ***protected internal***
  - ✓ ***private***
- Phạm vi truy cập mặc định là **private**

Từ khóa private

```
class MyBase
{
    public int myPublic; // Truy cập không hạn chế
    // Trong assembly đang định nghĩa hoặc lớp con
    protected internal int myProtInt;
    internal int myInternal; // Trong cùng assembly
    protected int myProtected; // Lớp con và lớp định nghĩa
    private int myPrivate; // Chỉ lớp đang định nghĩa

     void Test(){
            myPublic = 0; // Cho phép
            myProtInt = 0; // Cho phép
            myInternal = 0; // Cho phép
            myProtected = 0; // Cho phép
            myPrivate = 0; // Cho phép
     }
}
```

Ghi chú: assembly là không gian tên, mỗi project, ứng dụng thực thi hay thư viên liên kết động được sự là một không gian tên riêng (một assembly riêng).

Từ khóa protected

- Thành viên đã định nghĩa **protected** có thể được truy cập từ lớp con. Nhưng không thể truy cập ở các lớp khác.

```
class Derived : MyBase
{
    void Test()
    {
        myPublic = 0; // Cho phép
        myProtInt = 0; // Cho phép
        myInternal = 0; // Cho phép
        myProtected = 0; // Cho phép
        myPrivate = 0; // không thể truy cập
    }
}
```

Từ khóa internal

- Thành viên nội bộ có thể truy cập bất kỳ đâu trong **assembly** hiện tại, nhưng không thể truy cập từ **assembly** khác.
- ***Ví dụ:*** *Lớp* ***AnyClass*** *được tạo ở assembly hiện tại*

```
class AnyClass {
    void Test(MyBase m)
    {
        m.myPublic = 0; // cho phép
        m.myProtInt = 0; // cho phép
        m.myInternal = 0; // cho phép
        m.myProtected = 0; // không thể truy cập
        m.myPrivate = 0; // không thể truy cập
    }
}
```

Từ khóa protected internal

- Truy cập vừa **internal** và vừa **protected**.
- Truy cập bất kỳ đâu bên trong **assembly**, hoặc trong lớp bên ngoài **assembly** nhưng phải là lớp con của một lớp trong **assembly** hiện tại.
- ***Ví dụ:*** *Lớp* ***AnyClass*** *được tạo ở một assembly khác*

```
class Derived : MyBase {
    void Test(MyBase m) {
        m.myPublic = 0; // cho phép
        m.myProtInt = 0; // cho phép
        m.myInternal = 0; // không thể truy cập
        m.myProtected = 0; // cho phép
        m.myPrivate = 0; // không thể truy cập
    }
}
```

Từ khóa public

- Truy cập bất kỳ đâu mà thành viên có thể tham chiếu
- ***Ví dụ:*** *Lớp* ***AnyClass*** *được tạo ở một assembly khác*

```
class AnyClass
{
    void Test(MyBase m)
    {
        m.myPublic = 0; // cho phép
        m.myProtInt = 0; // không thể truy cập
        m.myInternal = 0; // không thể truy cập
        m.myProtected = 0; // không thể truy cập
        m.myPrivate = 0; // không thể truy cập
    }
}
```

Phạm vi truy cập cao nhất

- Phạm vi truy cập cao nhất (top-level) là phạm vi được khai báo bên ngoài cùng của kiểu.
- Trong **C#**, các kiểu sau đây có thể khai báo phạm vi truy cập cao nhất: **class**, **interface**, **struct**, **enum** và **delegate**.
- Khi không khai báo, phạm vi truy cập là **internal.**
- Để có thể truy cập cao nhất từ bất kỳ đâu, kiểu đó phải được định nghĩa bằng từ khóa **public.**

```
internal class MyInternalClass {}
public class MyPublicClass {}
```

Lớp inner

- Lớp có thể mang một ***lớp bên trong*** (*inner class*). Lớp trong cũng có 5 cấp độ truy cập. Nếu lớp trong không thể truy cập, nó không thể tạo thể hiện cũng như thừa kế. Mặc định, lớp trong là **private**, chỉ truy cập được từ trong lớp.

```
class MyBase {
    // Inner classes (nested classes)
    public class MyPublic {}
    protected internal class MyProtInt {}
    internal class MyInternal {}
    protected class MyProtected {}
    private class MyPrivate {}
}
```

Từ khóa static

- Từ khoá **static** dùng để khai báo thuộc tính và phương thức có thể được truy cập mà không cần tạo ra thể hiện.
- Các thành viên **static** chỉ tồn tại như một bản sao, nó thuộc chính lớp đó, khi khởi tạo thể hiện, chỉ các thành viên **non-static** được sao chép thành các thể hiện.
- Phương thức **static** không thể sử dụng các thành viên **non-static** của lớp đó, vì khi tạo thể hiện nó không là một phần của thể hiện đó.

Từ khóa static

```
class MyCircle
{
    // biến thường (được sao chép riêng khi khởi tạo)
    float r = 10;
    // biến static (chỉ một duy nhất thuộc lớp)
    static float pi = 3.14 F;
    // phương thức thường (được sao chép riêng khi khởi tạo)
    float GetArea() {
        return ComputeArea(r);
    }
    // phương thức static
    (chỉ một duy nhất thuộc lớp, không được sao chép khi khởi tạo)
    static float ComputeArea(float a){
        return pi*a*a;
    }
}
```

Truy cập thành viên static

- Để truy cập thành viên **static** bên ngoài lớp. Ta truy cập giống như ta dùng tên lớp thay vì thể hiện của nó.

```
static void Main()
{
        float f = MyCircle.ComputeArea(MyCircle.pi);
}
```

Phương thức static

- Phương thức **static** là phương thức có thể được sử dụng mà không cần tạo thể hiện.
- Chỉ có thể thực hiện các chức năng tổng quan
- Chỉ được truy cập các thành viên trong lớp nếu nó cũng được đánh dấu là **static.**
- ***Ví dụ:*** lớp System.Math, nó chứa rất nhiều những phương thức toán học mà ta chỉ cần gọi chúng khi sử dụng.

```
static void Main(){
        double pi = System.Math.PI;
}
```

Thuộc tính static

- Thuộc tính **static** có thể tồn tại trong suốt vòng đời ứng dụng
- Giá trị mặc định của biến **static** chỉ được đặt một lần trước khi sử dụng
- ***Ví dụ:*** dùng thuộc tính (biến) **static** để lưu trữ số lần phương thức được gọi.

```
static int count = 0;
public static void Dummy()
{
        count++;
}
```

Lớp static

- Một lớp **static** chỉ mang các thành viên và hằng **static**.
- Lớp **static** *không thể được* thừa kế hay tạo đối tượng thể hiện.

```
static class MyCircle {}
```

Phương thức xây dựng static

- Phương thức xây dựng **static** có thể thực hiện bất kỳ hành động nào để khởi tạo lớp
- Thông thường, chúng được sử dụng để khởi tạo các biến **static**.

```
class MyClass
{
    static int[] array = new int[5];
    static MyClass()
    {
        int i = 0;
        foreach (int element in array)
                element = i++;
    }
}
```

Phương thức mở rộng

- Tính năng mới trong **C# 3.0**, cho phép thêm phương thức trong những lớp đã có sẵn.
- Một phương thức mở rộng phải được định nghĩa là **static** trong một lớp static và từ khóa **this** sử dụng trong tham số đầu tiên để chỉ định lớp mở rộng

```
static class MyExtensions {
    // Extension method
    public static int ToInt(this string s) {
        return Int32.Parse(s);
    }
}
```

Thuộc tính

- **Thuộc tính** cho phép bảo vệ các dữ liệu thành viên của 1 lớp bằng các thao tác đọc/ghi dữ liệu thành viên thông qua khai báo thuộc tính.
- **Thuộc tính** có thể cho phép kiểm tra sự hợp lệ của dữ liệu thành viên

```
class Time{
    private int seconds;
    public int sec
    {
      get { return seconds;}
      set { seconds = value;}
    }
}
```

```
static void Main()
{
    Time t = new Time();
    int s = t.sec;
}
```

Thuộc tính

- Thuộc tính có thể được khai báo tự động như cú pháp sau đây:

```
class Time{
    private int seconds;
    public int sec
    {
      get { return seconds;}
      set { seconds = value;}
    }
}
```

```
class Time
{
    public int sec
    {
        get;
        set;
    }
}
```

Ưu điểm khi sử dụng thuộc tính

- Thuộc tính cho phép thay đổi những thành phần bổ sung bên trong mà không phá vỡ cấu trúc của chương trình nào đang sử dụng nó
- ***Ví dụ:*** Đổi kiểu dữ liệu của **seconds** từ *int* sang *byte*

```
class Time{
    private byte seconds;
    public int sec
    {
      get { return (int)seconds;}
      set { seconds = (byte)value;}
    }
}
```

Ưu điểm khi sử dụng thuộc tính

- Thuộc tính cho phép kiểm tra sự hợp lệ của dữ liệu trước khi được thay đổi.
- ***Ví dụ:*** Giá trị nhập vào **seconds** kiểm tra để không nhận giá trị âm.

```
set {
    if (value > 0)
        seconds = value;
    else
        seconds = 0;
    }
}
```

Ưu điểm khi sử dụng thuộc tính

- Thuộc tính không phải thao tác trực tiếp trên trường thật nên nó có thể cố định giá trị thực, nó cũng có thể tính toán giá trị, lấy giá trị từ bên ngoài lớp như từ cơ sở dữ liệu.
- ***Ví dụ:***.

```
public int hour {
        get{ return seconds / 3600; }
        set{
                seconds = value * 3600;
                count++;
            }
}
private int count = 0;
```

Thuộc tính Read-only và Write-only

- Một trong hai từ khóa get và set có thể bỏ qua. Không có set là thuộc tính read-only và ngược lại là write-only. Ta cũng có thể định cấp độ truy cập cho thuộc tính
- ***Ví dụ:***

```
// Read-only property
private int sec {
        public get { return seconds; }
}
// Write-only property
private int sec {
        public set { seconds = value; }
}
```

Indexer

- Chỉ mục (**indexer**) cho phép một đối tượng được xem như là một mảng. Chúng được khai báo giống như thuộc tính, nhưng dùng từ khóa **this**.
- ***Ví dụ:***

```
class MyArray {
        object[] data = new object[10];
        public object this[int i]{
                get {
                        return data[i];
                }
                set {
                        data[i] = value;
                }
        }
}
```

Indexer

- Sử dụng **indexer**, ta khai báo và khởi tạo bình thường nhưng khi đó đối tượng được tạo ra có thể được sử dụng như một mảng, có cả hàm **get** và **set**

```
static void Main()
{
    MyArray a = new MyArray();
    a[5] = "Hello World";
    object o = a[5]; // Hello World
}
```

Indexer

- Tham số dành cho **indexer** tương tự như một phương thức, ngoại trừ phải có ít nhất một tham số và không cho phép từ khóa **ref**, **out**.

```
class MyArray
{
    object[,] data = new object[10,10];
    public object this[int i, int j]
    {
        get { return data[i,j]; }
        set { data[i,j] = value; }
    }
}
```

Indexer

- Ta có thể dùng đối tượng để đưa vào tham số của **indexer** để trả lại vị trí của nó trong mảng

```
class MyArray
{
    object[] data = new object[10];
    public int this[object o]
    {
        get { return System.Array.IndexOf(data, o); }
    }
}
```

Indexer

- Ta có thể nạp chồng **indexer**

```
class MyArray {
        object[] data = new object[10];
        public int this[object o] {
            get { return System.Array.IndexOf(data, o); }
        }
        public object this[int i] {
            get { return data[i]; }
            set { data[i] = value; }
        }
}
```

Indexer

- ***Ví dụ:*** Viết một **indexer** và kiểm tra dữ liệu để không xảy ra ngoại lệ

```
public object this[int i] {
    get {
        return (i >= 0 && i < data.Length) ? data[i] : null;
    }

    set {
        if (i >= 0 && i < data.Length)
                data[i] = value;
    }
}
```

Giao tiếp (interface)

- Là một kiểu để mô tả những chức năng hoặc hành động mà lớp sử dụng nó có được mở rộng.

Ví dụ: Một người có thể khi ở nhà là một người con, ở trường là một sinh viên, ở lớp là một người bạn.

- Khai báo như khai báo lớp dùng từ khóa **interface**.
- Giao tiếp có thể được dùng để thể hiện sự đa kế thừa như trong C++.

```
interface MyInterface {}
```

Giao tiếp (interface)

- Phương thức trong giao tiếp chỉ được khai báo ở dạng khuôn mẫu và không có mã lệnh.

```
interface IMyInterface
{
    // Interface method
    int GetArea();
    // Interface property
    int Area { get; set; }
    // Interface indexer
    int this[int index] { get; set; }
    // Interface event
    event System.EventHandler MyEvent;
}
```

Giao tiếp (interface)

Ví dụ về giao tiếp: Viết một giao tiếp được gọi là **IComparable** có một phương thức là **compare**

```
interface IComparable {
        int compare(Object o);
}
```

Một lớp Circle sử dụng giao tiếp trên, có thuộc tính r để lưu bán kính

```
class Circle : IComparable {
        int r;
}
```

Giao tiếp (interface)

Vì **Circle** sử dụng giao tiếp **IComparable** nên nó phải định nghĩa lại phương thức ***compare***. Lớp này sẽ trả lại sự khác nhau giữa bán kính của 2 đối tượng.

```
class Circle : IComparable {
        int r;

        public int compare(Object o) {
                return r - (o as Circle).r;
        }
}
```

Chức năng của giao tiếp

Giao tiếp **IComparable** xác định chức năng mà lớp có thể có. Nó cũng làm cho việc sử dụng các thành viên trong giao tiếp mà không cần biết kiểu thực sự của nó là gì.

Ví dụ: Viết phương thức trả lại đối tượng lớn nhất giữa 2 đối tượng. Phương thức này sẽ làm việc với tất cả các lớp nào sử dụng giao tiếp **IComparable**

```
public static Object largest(IComparable a, IComparable b)
{
        return (a.compare(b) > 0) ? a : b;
}
```

Chức năng của giao tiếp

Một cách sử dụng khác của giao tiếp là đưa ra những khuôn mẫu chức năng thực sự cho lớp.

Ví dụ: Viết giao tiết **MyInterface** cho lớp **MyClass.** Giao tiếp sẽ chỉ bao gồm các chức năng, người chỉ sử dụng lớp **MyClass** nếu cần.

```
interface IMyInterface { void exposed(); }
class MyClass : IMyInterface {
    public void exposed() {}
    public void hidden() {}
}

public static void main(String[] args){
        IMyInterface m = new MyClass();
}
```

Lớp trừu tượng (Abstract)

- ***Ví dụ về lớp trừu tượng:***

```
abstract class Shape
{
    // Abstract method
    public abstract int GetArea();
    // Abstract property
    public abstract int area { get; set; }
    // Abstract indexer
    public abstract int this[int index] { get; set; }
    // Abstract event
    public delegate void MyDelegate();
    public abstract event MyDelegate MyEvent;
    // Abstract class
    public abstract class InnerShape {};
}
```

Lớp trừu tượng (Abstract)

- ***Ví dụ về lớp trừu tượng:***

```
abstract class Shape
{
    private int x = 100, y = 100;
    public abstract int GetArea();
}

class Rectangle : Shape
{
        public int GetArea() { return x * y; }
}
```

Lớp trừu tượng (Abstract)

- Một lớp con của lớp trừu tượng cũng có thể được khai báo **abstract.** Trong trường hợp này, nó không cần phải mở rộng bất kỳ thành viên trừu tượng nào của lớp cha

```
abstract class Rectangle : Shape {}
```

- Một lớp trừu tượng cũng có thể thừa kế từ một lớp không trừu tượng

```
class NonAbstract {}
abstract class Abstract : NonAbstract {}
```

Lớp trừu tượng (Abstract)

- Nếu lớp cha có thành viên có từ khóa virtual, chúng có thể được ghi và khai báo abstract để bắt buộc lớp con của nó phải mở rộng các thuộc tính trừu tượng.

```
class MyClass { void virtual Dummy() {} }
abstract class Abstract : MyClass {
        void abstract override Dummy() {} }
```

- Một lớp trừu tượng cũng được sử dụng như **interface** để giữ đối tượng từ lớp con

```
Shape s = new Rectangle();
```

- Không thể tạo một thể hiện của lớp trừu tượng

```
Shape s = new Shape(); // compile-time error
```

Phân biệt abstract và interface

Giống:

- Cả hai đều tương tự nhau, đều định nghĩa những khuôn mẫu để lớp tham chiếu phải mở rộng.

Phân biệt lớp trừu tượng và giao tiếp

Khác:

- Lớp trừu tượng có thể mang những thành viên không trừu tượng, giao tiếp thì không.
- Một lớp có thể mở rộng bao nhiêu giao tiếp tùy thích nhưng chỉ có thể thừa kế từ một lớp, lớp trừu tượng hoặc không.
- Lớp trừu tượng có thể thừa kế một lớp và sử dụng cả giao tiếp. Giao tiếp cũng có thể thừa kế từ một giao tiếp nhưng hiệu quả nhất là kết hợp hai giao tiếp thành một

Ngoại lệ

- Ngoại lệ là lỗi khi thực thi chương trình (runtime error)
- Ngoại lệ làm quá trình thực thi của chương trình bị chấm dứt bất thường

```
using System;
using System.IO;
class ErrorHandling
{
    static void Main(){
      // Run-time error
      StreamReader sr = new StreamReader("missing.txt");
    }
}
```

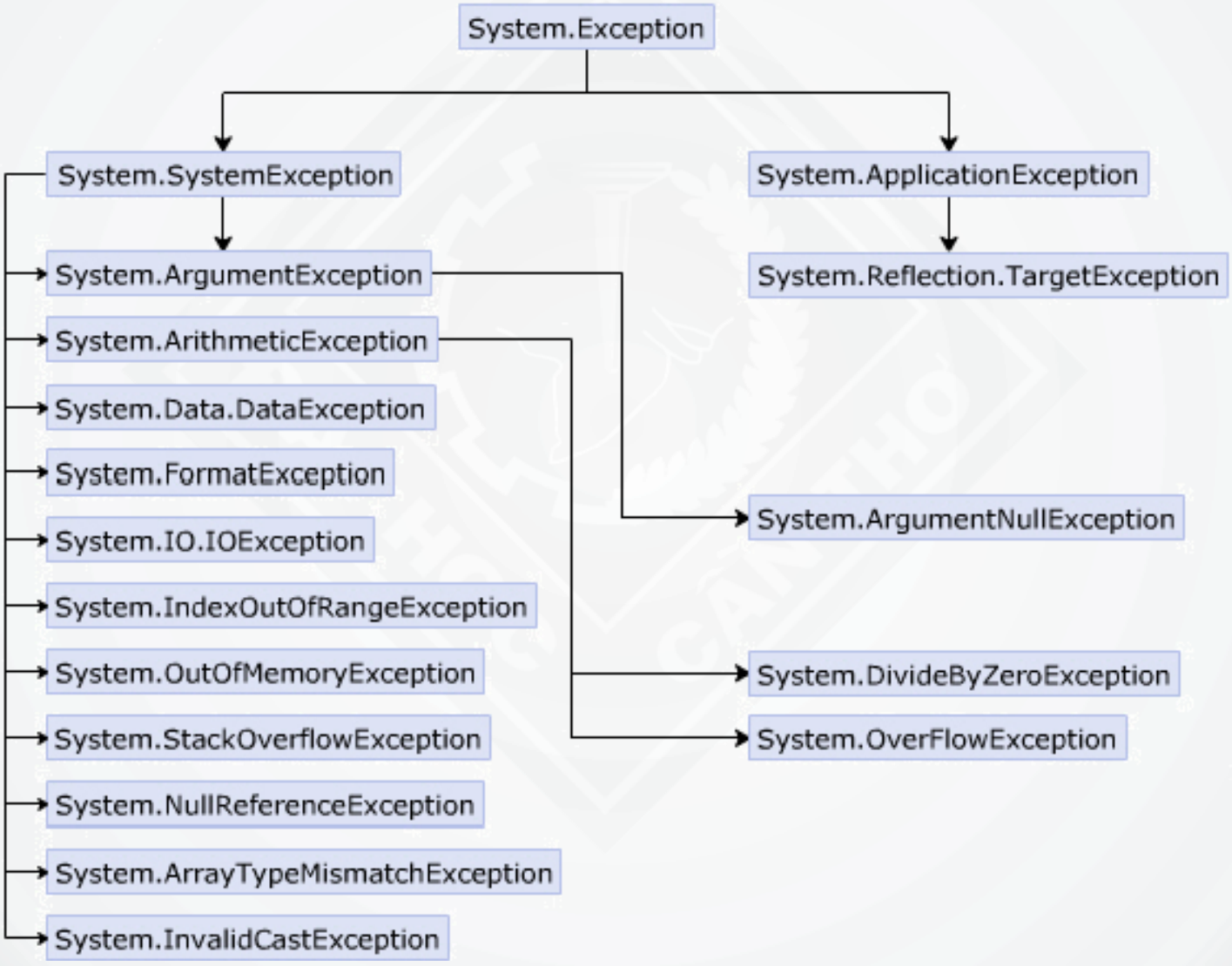
Câu lệnh try-catch

- Để chống lại chương trình xảy ra ngoại lệ, ta dung câu lệnh try-catch để bắt ngoại lệ phát sinh

```
try {
        StreamReader sr = new StreamReader("missing.txt");
} catch (exception e) {
        Console.Write("File not found");
}
```

- Lớp **Exception** trong không gian tên System là lớp gốc của mọi ngoại lệ trong .NET.

Ngoại lệ


System.Exception
System.SystemException
System.ApplicationException
System.ArgumentException
System.Reflection.TargetException
System.ArithmeticException
System.Data.DataException
System.FormatException
System.ArgumentNullException
System.IO.IOException
System.IndexOutOfRangeException
System.OutOfMemoryException
System.DivideByZeroException
System.StackOverflowException
System.OverFlowException
System.NullReferenceException
System.ArrayTypeMismatchException
System.InvalidCastException

Khối lệnh finally

- Là khối lệnh luôn được thực thi mặc cho ngoại lệ có xảy ra hay không.

```
StreamReader sr = null;
try {
        sr = new StreamReader("missing.txt");
}
catch (FileNotFoundException) {} // có thể bỏ qua
finally
{
        if (sr != null) sr.Close();
}
```

Khối lệnh using

- Là khối lệnh rút gọn của khối lệnh **try-finally**. Giúp cho việc cấp phát và thu hồi được cô đọng hơn.

```
using System.Drawing;
// ...
using (Bitmap b = new Bitmap(100, 100))
{
    System.Console.WriteLine("Width: " + b.Width +
    ", Height: " + b.Height);
}
```

Phát sinh ngoại lệ

- Khi một phương thức không thể phục hồi xảy ra ngoại lệ, chúng ta có thể phát sinh ngoại lệ để báo cho đối tượng rằng phương thức đã không thể thực khi phương thức như mong muốn

```
static void MakeError() {
        throw new System.DivideByZeroException("My Error");
}

static void Main() {
        try { MakeError(); } catch { throw; }
}
```

Delegates

- **Delegate** là một kiểu được dùng để tham chiếu đến một phương thức.
- **Delegate** được khai báo như kiểu phương thức mẫu giống như phương thức mà nó sẽ tham chiếu đến.
- Một đối tượng **delegate** có thể tham chiếu đến nhiều phương thức khác nhau ở những thời điểm khác nhau trong quá trình thực thi chương trình.

Delegates

- ***Ví dụ về delegate:***

```
delegate void MyDelegate(string s);

class MyClass
{
        static void Print(string t)
        {
                System.Console.Write(t);
        }
        static void Main()
        {
                MyDelegate d = Print;
        }
}
```

Delegates

- ***Sử dụng delegate:***

```
MyDelegate d = Print;
d("Hello");
```

- Cú pháp khác:

```
MyDelegate d = new MyDelegate(Print); // C# 2.0

MyDelegate f = delegate(string t) // Anonymous methods
{
        System.Console.Write(t);
};
```

Delegates

- Sử dụng biểu thức **lambda** (C# 3.0)

```
delegate int MyDelegate(int i);
static void Main()
{
    // Anonymous method
    del a = delegate(int x) { return x * x; };
    // Lambda expression
    del b = (int x) => x * x;
    a(5); // 25
    b(5); // 25
}
```