**Introducing interface in C#**

**Comparing Interfaces and Abstract Classes**

**Interface:**

No implementation code (inside the interface)

Implement any number of interfaces

Member automatically public

Properties

Methods

Events

Indexers

**Abstract Class**

May have implementation code

Single inheritance

Access modifiers on members

Properties

Methods

Evens

Indexers

Fields

Constructors

Destructors

**Scenario 1:** sử dụng concrete class.

Ta định nghĩa một class tên là ConcreteRegularPolygon để dùng cho các đa giác điều (đa giác điều là đa giác có các cạnh bằng nhau).

Trong class này ta có

NumberOfSides: Dùng để cho biết số cạnh của đa giác này.

SideLength: Dùng để cho biết chiều dày của một cạnh.

GetPerimeter: Dùng để tính chu vi của đa giác đó, đối với đa giác điều thì công thức tính chu vi là số cạnh nhân với chiều dài một cạnh.

GetArea: Dùng để tính diện tích của đa giác điều, ở đây ta định nghĩa phương thức này là ảo (**virtual**) để chỉ ra rằng tùy vào loại đa giác điều cụ thể nào thì nó sẽ **override** lại phương thức này đối với công thức tính diện tích của nó, nếu nó không override lại thì ta ném một ngoại lệ (**exeption**) vô mặt thằng implement.

public class ConcreteRegularPolygon

{

    public int NumberOfSides { get; set; }

    public int SideLength { get; set; }

    public ConcreteRegularPolygon(int sides, int length)

    {

        NumberOfSides = sides;

        SideLength = length;

    }

    public double GetPerimeter()

    {

        return NumberOfSides \* SideLength;

    }

    public virtual double GetArea()

    {

        throw new NotImplementedException();

    }

}

Bây giờ ta tạo một class tên là Square kế thừa từ thằng ConcreteRegularPolygon, ta biết rằng square là một thực thể cụ thể của thằng ConcreteRegularPolygon (Hình vuông là một đa giác điều, tổ tiên đã nói vậy thì đừng cãi nhé ).

Đối với class này ta làm như sau:

* Lúc khởi tạo ta chỉ ra nó có 4 cạnh và chiều dài của một cạnh.
* Ta override lại phương thức GetArea(), đối với hình vuông, công thức tính diện tích là bình phương của một cạnh.

public class Square : ConcreteRegularPolygon

{

    public Square(int length) :

        base(4, length)

    { }

    public override double GetArea()

    {

        return SideLength \* SideLength;

    }

}

Tới lúc này mọi chuyện rất haappy, Khi ta chạy chương trình, nó sẽ tính được chu vi và diện tích theo yêu cầu của ta một cách smoothly.

Vậy chuyện gì sẽ xảy ra khi ta bỏ đi phương thức GetArea() trong class Square? Khi ta biên dịch chương trình thì chả có gì bất ổn cả, compiler rất happy với cách ta implement class Square mà không override lại method GetArea().

Nhưng khi ta chạy chương trình thì nó lại phát sinh ra lỗi, bởi vì Square kế thừa từ class ConcreteRegularPolygon nhưng lại không override lại phương thức GetArea() của lớp này, do vậy ta sẽ nhận được một throw new NotImplementedException(); của anh ConcreteRegularPolygon.

**Note: Vậy từ những điều bên trên ta rút ra được là nếu ta nếu ta sử dụng cách implement theo concrete thì nó không ép buộc ta phải implement những thằng virtual khi ta biên dịch, đến khi ta chạy thì nó sẽ phát sinh ra exeception.**

**Scenario 2**: Sử dụng abstract class.

Ta định nghĩa một abstract class AbstractRegularPolygon như sau:

public abstract class AbstractRegularPolygon

{

    public int NumberOfSides { get; set; }

    public int SideLength { get; set; }

    public AbstractRegularPolygon(int sides, int length)

    {

        NumberOfSides = sides;

        SideLength = length;

    }

    public double GetPerimeter()

    {

        return NumberOfSides \* SideLength;

    }

    public abstract double GetArea();

}

Bây giờ ta định nghĩa một class khác có tên là Triangle như sau:

(tam giác điều cũng là một thực thể cụ thể của một đa giác điều)

public class Triangle : AbstractRegularPolygon

{

    public Triangle(int length) :

        base(3, length)

    { }

    public override double GetArea()

    {

        return SideLength \* SideLength \* Math.Sqrt(3) / 4;

    }

}

Khi ta chạy chương trình để tính chu vi và diện thích của một triangle, mọi thứ điều rất good và happy.

Khi ta bỏ đi method GetArea() trong class Triangle, thì ngay lập tức compiler ném một cục lỗi (compile-time error) vào mặt ta với nội dung “Ê thằng ngu! Mày không implement một thành phần abstract mà mày kế thừa kìa!”. Đây là một điểm khác biệt giữa concrete và abstract class (nếu ta được chọn lựa giữa run-time error và compile-time error thì chắc hẵng compile-time error vẫn tốt hơn chứ nhỉ?).

**Scenario 3**: Sử dụng interface.

Ta xem interface như là một contract, do vậy tất cả điều phải nhìn thấy được, nghĩa là tất cả các thành phần bên trong điều có access modifier là public, mặt khác nếu một method bên trong một interface thì nó chỉ là tên định nghĩa chứ không có phần định nghĩa thực thi, đây chính là điểm khác biệt giữa interface và abstract class.

Nếu một class implement một interface thì nó phải implement hết tất các những nhứ có bên trong interface mà nó implement.

public interface IRegularPolygon

{

    int NumberOfSides { get; set; }

    int SideLength { get; set; }

    double GetPerimeter();

    double GetArea();

}

Tiếp theo ta định nghĩa một class tên là implement từ IRegularPolygon như sau:

public class Octagon :  IRegularPolygon

{

    public int NumberOfSides { get; set; }

    public int SideLength { get; set; }

    public Octagon(int length)

    {

        NumberOfSides = 8;

        SideLength = length;

    }

    public double GetPerimeter()

    {

        return NumberOfSides \* SideLength;

    }

    public double GetArea()

    {

        return SideLength \* SideLength \* (2 + 2 \* Math.Sqrt(2));

    }

}

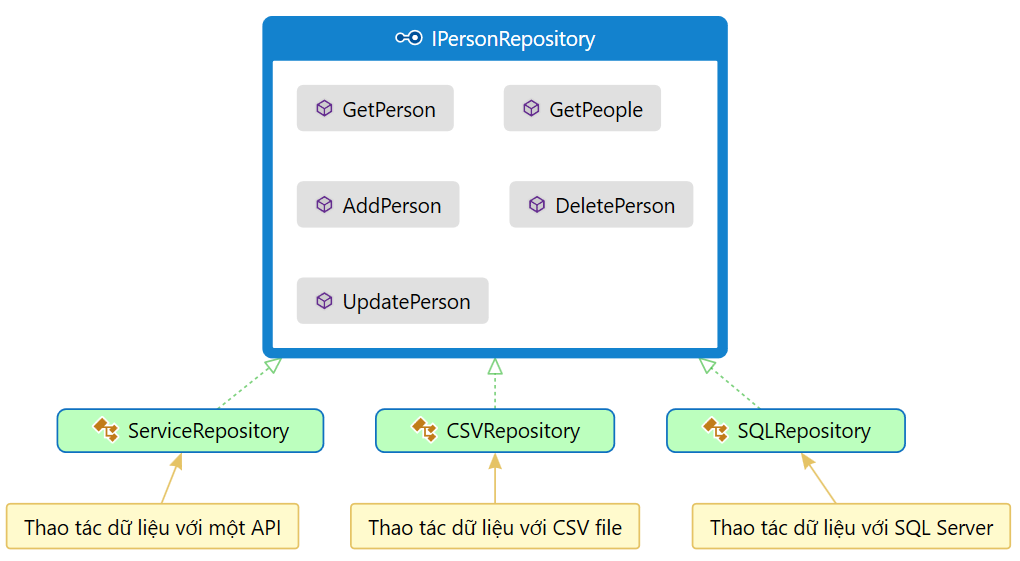
Đến lúc này khi ta run program và thực hiện tính chu vi và diện tích cho thằng octagon thì mọi thứ work well, Ở class Octagon nếu ta không implement một thành phần nào đó của interface thì compiler lập tức báo lỗi tương tự như đối với abstract class.

**Creating Interfaces to Add Extensibility**

Đầu tiên ta xem xét diagram sau:

Ta tạo ra một IPersonRepository xem đây như là một loại contract (hợp đồng) bên trong chứa các điều khoản (các methods) để thao tác với từng loại chứa dữ liệu (database or file) và tùy vào trường hợp nào mà ta sẽ có cách implement khác nhau cho từng công việc (lấy hết dữ liệu, lấy một đối tượng cụ thể, thêm, xóa, cập nhật).

Ví dụ như ServiceRepository dùng để thao tác dữ liệu với một web API khi anh implement IpersonRepository thì anh phải implement hết các method bên trong IpersonRepository, tùy vào loại cơ sở dữ liệu hay tập tin mà ta sẽ có cách thao tác dữ liệu khác nhau, nhưng phải tuân thủ một contact đã ràng buộc theo IpersonRepository.



Ta xem xét các đoạn code cho từng đối tượng cụ thể như sau:

**IPersonRepository**

Đây sẽ đóng vai trò nhưng là một standard cho việc thao tác dữ liệu của ứng dụng, mọi thằng nào một khi implement thằng này phải implement hết tất cả mọi thứ trong IPersonRepository.

public interface IPersonRepository

{

    IEnumerable<Person> GetPeople();

    Person GetPerson(int id);

    void AddPerson(Person newPerson);

    void UpdatePerson(int id, Person updatedPerson);

    void DeletePerson(int id);

}

**CSVRepository**

Chụi trách nhiệm thao tác dữ liệu với CSV file trong ứng dụng.

public class CSVRepository : IPersonRepository

{

    string path;

    public CSVRepository()

    {

        var filename = ConfigurationManager.AppSettings["CSVFileName"];

        path = AppDomain.CurrentDomain.BaseDirectory + filename;

    }

    public IEnumerable<Person> GetPeople()

    {

        var people = new List<Person>();

        if (File.Exists(path))

        {

            using (var reader = new StreamReader(path))

            {

                string line;

                while((line = reader.ReadLine()) != null)

                {

                    var elements = line.Split(',');

                    var person = new Person()

                    {

                        Id = Int32.Parse(elements[0]),

                        GivenName = elements[1],

                        FamilyName = elements[2],

                        StartDate = DateTime.Parse(elements[3]),

                        Rating = Int32.Parse(elements[4]),

                        FormatString = elements[5],

                    };

                    people.Add(person);

                }

            }

        }

        return people;

    }

    public Person GetPerson(int id)

    {

        return GetPeople().FirstOrDefault(p => p.Id == id);

    }

    public void AddPerson(Person newPerson)

    {

        throw new NotImplementedException();

    }

    public void UpdatePerson(int id, Person updatedPerson)

    {

        throw new NotImplementedException();

    }

    public void DeletePerson(int id)

    {

        throw new NotImplementedException();

    }

}

**ServiceRepository**

Thực hiện thao tác dữ liệu với một web API khác ngoài hệ thống.

public class ServiceRepository : IPersonRepository

{

    WebClient client = new WebClient();

    string baseUri = "http://localhost:9874/api/people";

    public IEnumerable<Person> GetPeople()

    {

        string result = client.DownloadString(baseUri);

        IEnumerable<Person> people =

            JsonConvert.DeserializeObject<IEnumerable<Person>>(result);

        return people;

    }

    public Person GetPerson(int id)

    {

        throw new NotImplementedException();

    }

    public void AddPerson(Person newPerson)

    {

        throw new NotImplementedException();

    }

    public void DeletePerson(int id)

    {

        throw new NotImplementedException();

    }

    public void UpdatePerson(int id, Person updatedPerson)

    {

        throw new NotImplementedException();

    }

}

**SQLRepository**

Chụi trách nhiệm thao tác dữ liệu với cở sở dữ liệu SQL Server.

public class SQLRepository : IPersonRepository

{

    DbContextOptions<PersonContext> options;

    public SQLRepository()

    {

        var optionsBuilder = new DbContextOptionsBuilder<PersonContext>();

        optionsBuilder.UseSqlite("Data Source=People.db");

        options = optionsBuilder.Options;

    }

    public IEnumerable<Person> GetPeople()

    {

        using (var context = new PersonContext(options))

        {

            return context.People.ToArray();

        }

    }

    public Person GetPerson(int id)

    {

        using (var context = new PersonContext(options))

        {

            return context.People.Where(p => p.Id == id).FirstOrDefault();

        }

    }

    public void AddPerson(Person newPerson)

    {

        using (var context = new PersonContext(options))

        {

            context.People.Add(newPerson);

            context.SaveChanges();

        }

    }

    public void UpdatePerson(int id, Person updatedPerson)

    {

        using (var context = new PersonContext(options))

        {

            var person = context.People.FirstOrDefault(p => p.Id == id);

            if (person != null)

            {

                person.Id = updatedPerson.Id;

                person.GivenName = updatedPerson.GivenName;

                person.FamilyName = updatedPerson.FamilyName;

                person.StartDate = updatedPerson.StartDate;

                person.Rating = updatedPerson.Rating;

                person.FormatString = updatedPerson.FormatString;

                context.SaveChanges();

            }

        }

    }

    public void DeletePerson(int id)

    {

        using (var context = new PersonContext(options))

        {

            var person = context.People.FirstOrDefault(p => p.Id == id);

            if (person != null)

            {

                context.People.Remove(person);

                context.SaveChanges();

            }

        }

    }

}

**RepositoryFactory**

Mặt khác để thuận tiên truy cập đến từng loại repository ta cũng implement thêm RepositoryFactory, class này sẽ chụi trách nhiệm nhận vào loại đối tượng repository nào (service, SQL, CSV) để nó tạo ra một instance cho đối tượng đó.

public static class RepositoryFactory

{

    public static IPersonRepository GetRepository(string repositoryType)

    {

        IPersonRepository repository = null;

        switch (repositoryType)

        {

            case "Service": repository = new ServiceRepository();

                break;

            case "CSV": repository = new CSVRepository();

                break;

            case "SQL": repository = new SQLRepository();

                break;

            default:

                throw new ArgumentException("Invalid repository type");

        }

        return repository;

    }

}

Ở tầng UI của ứng dụng ta định nghĩa một method như sau:

Method này nhận vào một loại repository (PopulateListBox("Service")) và sau đó thực hiện gởi loại repository đó đến RepositoryFactory để thằng này chụi trách nhiệu tạo mới đối đượng theo loại repository mà ta muốn. Sau khi có được một instance của repository mà ta muốn thì lúc này ta chỉ việc gọi đến method cần dùng của repository đó.

private void PopulateListBox(string repositoryType)

{

    ClearListBox();

    IPersonRepository repository =

        RepositoryFactory.GetRepository(repositoryType);

    var people = repository.GetPeople();

    foreach (var person in people)

        PersonListBox.Items.Add(person);

    ShowRepositoryType(repository);

}

**Explicit Interface Implementation**

**Standard Interface Implementation**

*(implement interface theo cách thông thường).*

Ta xét đoạn code sau:

public interface ISaveable

{

    void Save();

}

public class Catalog: ISaveable

{

    public void Save()

    {

        Console.WriteLine("Save");

    }

}

// Truy cập thông qua đối tượng implement

// từ interface.

Catalog catalog = new Catalog();

catalog.Save();

// Result: SaveISaveable

// Truy cập thông qua một interface

ISaveable saveable = catalog;

saveable.Save();

// Result: SaveISaveable

Ta thấy rằng bằng việc implement một interface theo cách thông thường thì ta có thể truy cập đến đối tượng được implement của interface thông qua 2 con được là đối tượng implement interface đó hoặc bằng chính interface đó.

**Explicit Interface Implementation**

Ta xét đoạn code sau:

public interface ISaveable

{

    void Save();

}

public class Catalog : ISaveable

{

    void ISaveable.Save()

    {

        Console.WriteLine("Save");

    }

}

Catalog catalog = new Catalog();

catalog.Save();

// Compiler báo lỗi.

ISaveable saveable = catalog;

saveable.Save();

// Result: Save

((ISaveable)catalog).Save();

// Result: Save

Bằng cách implement interface theo explicit, thì khi muốn truy cập một method của interface đó. Ta bị ép buộc là thằng truy cập phải thuộc kiểu interface đó mới được truy xuất, nếu ta truy xuất trực tiếp bằng kiểu của thằng implement interface đó, thì nó vẫn báo lỗi trừ phi ta ép kiểu về kiểu interface mà thằng đó implement.

Cho trường hợp này ta có thể tóm tắt lại như sau:

// Interface type

ISaveable saveable = new Catalog();

saveable.Save();

//Result: Save

// Interface not used.

Catalog catalog = new Catalog();

catalog.Save();

// Result: error, compiler error

// Interface not used.

// Kiểu ngầm định var sẽ lấy kiểu được tạo ra

// từ bên phải và declare cho bên trái

// trên trong trường hợp này varCatalog vẫn là

// kiểu Catalog.

var varCatalog = new Catalog();

varCatalog.Save();

// Result: error, compiler error

// Interface type

((ISaveable)catalog).Save();

// Result: Save

Bây giờ ta xem xét tiếp việc mixed methods của interface và của lớp mà implement interface đó.

Ta xét đoạn code sau:

public interface ISaveable

{

    void Save();

}

public class Catalog : ISaveable

{

    // Đây là Save của thằng Catalog.

    public void Save()

    {

        Console.WriteLine("Save (catalog)");

    }

    // Đây là Save của thằng interface.

    void ISaveable.Save()

    {

        Console.WriteLine("Save (interface)");

    }

}

Ta thấy rằng bên class Catalog nó có 2 phương thức **Save** có cùng tham số đầu vào và cùng kiểu trả về nhưng compiler vẫn rất happy với việc này vì 2 thằng này là 2 thằng khác nhau, một thằng là của chính bản thân class **Catalog**, còn thằng còn lại là của thằng interface.

Bên đưới là các cách ta gọi ra các method **save** và kết quả của từng loại đó.

Catalog catalog = new Catalog();

catalog.Save();

// Result: Save (catalog)

ISaveable saveable = catalog;

saveable.Save();

// Result: Save (interface)

((ISaveable)catalog).Save();

// Result: Save (interface)

**Vấn đề Conflicting Method Signatures**

Trong đoạn code bên dưới, ta thấy rằng class Catalog implements 2 interfaces là ISabeble và IDbSaver. Cả hai interface này điều có cùng một method tên là Save() nhưng lại khác kiểu trả về. Do vậy, class mà implement từ 2 interface này thì sẽ có 2 methods save và một trong 2 đứa đó phải theo kiểu explicitly, hoặc cả hai điều là explicit nếu không sẽ bị báo lỗi ngay lập tức.

public interface ISaveable

{

    void Save();

}

public interface IDbSaver

{

    string Save();

}

public class Catalog : ISaveable,IDbSaver

{

    // Catalog và ISaveable

    public void Save()

    {

        Console.WriteLine("Saved from ISaveable interface");

    }

    // IDbSaver (explicit)

    string IDbSaver.Save()

    {

        return "Saved from IDbSaver interface";

    }

}

Bây giờ ta sẽ xét một ví dụ cụ thể trong .Net Framework.

Ta xét đoạn code: **PersonListBox.ItemsSource= people**;

**Trong đó**:

**PersonListBox.ItemsSource** là một **IEnumerable**

**people** là một **IEnumberable<Person>**

Ta biết rằng **public interface IEnumerable<T> : IEnumerable**. Do vậy sẽ không có sự mismatch về kiểu của **IEnumerable** và **IEnumerable<Person>.**

Ta đi vào từng chi tiết của từng interface ta thấy rằng

public interface IEnumerable { IEnumerator GetEnumerator(); }

public interface IEnumerable<T> { IEnumerator<T> GetEnumerator(); }

Do cả hai thằng điều có method tên là **GetEnumerator**, do vậy sẽ có conflic method và để tránh điều này bất kỳ thằng nào implement lại thằng interface IEnumerable<T> phải dùng đến **explicit implementation**.

Ta xem ví demo bên dưới cho trường hợp này.

public class OddGenerator : IEnumerable<int>

{

    // IEnumerator of T

    public IEnumerator<int> GetEnumerator()

    {

        int i = 1;

        yield return i;

        while (true)

        {

            i += 2;

            yield return i;

        }

    }

    // IEnumerator

    IEnumerator IEnumerable.GetEnumerator()

    {

        return this.GetEnumerator();

    }

}

class Program

{

    static void Main(string[] args)

    {

        Console.WriteLine ("Odd Numbers:");

        var generator = new OddGenerator();

        foreach(var odd in generator)

        {

            if (odd > 50)

                break;

            Console.WriteLine(odd);

        }

        Console.Read();

    }

}

Loạt bài viết này dựa theo khóa học C# interface của Jeremy Clark