C# - Coding Conventions

I. Quy ước đặt tên (Naming conventions)

1. Pascal case

- Sử dụng pascal casing ("PascalCasing") khi đặt tên một **class**, **record**, hoặc **struct**.

```
public class DataService
{
}
```

```
public record PhysicalAddress(
    string Street,
    string City,
    string StateOrProvince,
    string ZipCode);
```

```
public struct ValueCoordinate
{
}
```

- Khi đặt tên một **interface**, hãy sử dụng cách viết hoa pascal ngoài việc đặt tiền tố tên bằng chữ **I**. Điều này chỉ ra rõ ràng cho người tiêu dùng rằng đó là một **interface**.

```
public interface IWorkerQueue
{
}
```

- Khi đặt tên cho **public** các thành viên của kiểu, chẳng hạn như **fields, properties, events, methods,** và **local functions,** hãy sử dụng cách viết hoa pascal.

```
public class ExampleEvents
{
    // A public field, these should be used sparingly
    public bool IsValid;

    // An init-only property
    public IWorkerQueue WorkerQueue { get; init; }

    // An event
    public event Action EventProcessing;

    // Method
    public void StartEventProcessing()
    {
        // Local function
        static int CountQueueItems() => WorkerQueue.Count;
        // ...
```

```
    }
}
```

- Khi viết bản ghi vị trí, hãy sử dụng cách viết hoa pascal cho các tham số vì chúng là thuộc tính công khai của bản ghi.

```
public record PhysicalAddress(
string Street,
string City,
string StateOrProvince,
string ZipCode);
```

2. Camel case

- Sử camelCasing khi đặt tên **private** hoặc **internal** các trường và đặt tiền tố chúng bằng _

```
public class DataService
{
    private IWorkerQueue _workerQueue;
}
```

- Khi làm việc với **static** các trường có **private** hoặc **internal**, hãy sử dụng **s_** tiền tố và để sử dụng tĩnh chuỗi **t_**.

```
public class DataService
{
    private static IWorkerQueue s_workerQueue;

    [ThreadStatic]
    private static TimeSpan t_timeSpan;
}
```

- Khi viết các **tham số** của **phương thức**, hãy sử dụng camelCasing.

```
public T SomeMethod<T>(int someNumber, bool isValid)
{
}
```

II. Quy ước đặt tên bổ sung (Additional naming conventions)

- Sử dụng các điều kiện về không gian tên. Nếu bạn biết rằng một không gian tên được nhập theo mặc định trong một dự án, bạn không cần phải đáp ứng đầy đủ các tên từ không gian tên đó. Tên đủ điều kiện có thể bị ngắt sau dấu chấm (.) Nếu chúng quá dài so với một dòng, như thể hiện trong ví dụ sau.

```
var currentPerformanceCounterCategory = new
System.Diagnostics.
    PerformanceCounterCategory();
```

III. Quy ước về bố cục (Layout conventions)

- Sử dụng cài đặt Trình chỉnh sửa mã mặc định (thụt lề thông minh, thụt lề bốn ký tự, các tab được lưu dưới dạng dấu cách
- Chỉ viết một câu lệnh trên mỗi dòng.
- Chỉ viết một khai báo trên mỗi dòng.
- Nếu các dòng tiếp tục không được thụt lề tự động, hãy thụt lề chúng ở một điểm dừng tab (bốn dấu cách).
- Thêm ít nhất một dòng trống giữa định nghĩa phương thức và định nghĩa thuộc tính.
- Sử dụng dấu ngoặc đơn để tạo các mệnh đề trong một biểu thức rõ ràng, như được hiển thị trong đoạn mã sau.

```
if ((val1 > val2) && (val1 > val3))
{
    // Take appropriate action.
}
```

IV. Nguyên tắc ngôn ngữ (Language guidelines)

- Các phần sau đây mô tả các phương pháp mà nhóm C # tuân theo để chuẩn bị các ví dụ và mẫu mã.

1. Kiểu dữ liệu chuỗi (string)

- Sử dụng **string interpolation** để nối các chuỗi ngắn, như được hiển thị trong đoạn mã sau.

```
string displayName = $"{nameList[n].LastName},
{nameList[n].FirstName}";
```

- Để nối các chuỗi trong các vòng lặp, đặc biệt khi bạn đang làm việc với một lượng lớn văn bản, hãy sử dụng một đối tượng **StringBuilder**

```
var phrase = "lalalalalalalalalalalalalalalalalalalalalalalalalalalalalala";
var manyPhrases = new StringBuilder();
for (var i = 0; i < 10000; i++)
{
   manyPhrases.Append(phrase);
}
//Console.WriteLine("tra" + manyPhrases);
```

2. Các biến cục bộ được nhập ngầm (Implicitly typed local variables)

- Sử dụng **implicit typing** cho các biến cục bộ khi kiểu của biến là rõ ràng ở phía bên phải của phép gán hoặc khi kiểu chính xác không quan trọng.

```
var var1 = "This is clearly a string.";
var var2 = 27;
```

- Không sử dụng **var** khi loại không rõ ràng ở phía bên phải của nhiệm vụ. Đừng cho rằng kiểu rõ ràng với tên phương thức. Một kiểu biến được coi là rõ ràng nếu đó là một **new** toán tử hoặc một kiểu ép kiểu rõ ràng.

```
int var3 = Convert.ToInt32(Console.ReadLine());
int var4 = ExampleClass.ResultSoFar();
```

- Đừng dựa vào tên biến để chỉ định kiểu của biến. Nó có thể không đúng. Trong ví dụ sau, tên biến **inputInt** bị sai lệch. Đó là một chuỗi.

```
var inputInt = Console.ReadLine();
Console.WriteLine(inputInt);
```

- Tránh sử dụng **var** thay thế cho **dynamic** .Sử dụng **dynamic** khi bạn muốn suy luận kiểu thời gian chạy.

- Sử dụng kiểu gõ ngầm để xác định kiểu của biến vòng lặp trong **for** các vòng lặp.

- Ví dụ sau sử dụng cách nhập ẩn trong một **for** câu lệnh.

```
var phrase =
"lalalalalalalalalalalalalalalalalalalalallalala";
var manyPhrases = new StringBuilder();
for (var i = 0; i < 10000; i++)
{
    manyPhrases.Append(phrase);
}
//Console.WriteLine("tra" + manyPhrases);
```

- Không sử dụng kiểu nhập ngầm để xác định kiểu của biến vòng lặp trong **foreach** các vòng lặp.
Ví dụ sau sử dụng cách nhập rõ ràng trong một foreach câu lệnh.

```
foreach (char ch in laugh)
{
   if (ch == 'h')
      Console.Write("H");
   else
      Console.Write(ch);
}
Console.WriteLine();
```

IV. Unsigned data types

1. Mảng (Array)

- Sử dụng cú pháp ngắn gọn khi bạn khởi tạo mảng trên dòng khai báo. Trong ví dụ sau, lưu ý rằng bạn không thể sử dụng **var** thay thế **string[]**.

```
string[] vowels1 = { "a", "e", "i", "o", "u" };
```

- Nếu bạn sử dụng thuyết minh rõ ràng, bạn có thể sử dụng **var**.

```
var vowels2 = new string[] { "a", "e", "i", "o", "u" };
```

- Nếu bạn chỉ định kích thước mảng, bạn phải khởi tạo từng phần tử một.

```
var vowels3 = new string[5];
vowels3[0] = "a";
vowels3[1] = "e";
// And so on.
```

2. Delegates

- Sử dụng **Func**<> và **Action**<> thay vì xác định các loại **Delegates**. Trong một lớp, xác định **delegate method**.

```
public static Action<string> ActionExample1 = x =>
Console.WriteLine($"x is: {x}");

public static Action<string, string> ActionExample2 = (x, y)
=>
    Console.WriteLine($"x is: {x}, y is {y}");

public static Func<string, int> FuncExample1 = x =>
Convert.ToInt32(x);

public static Func<int, int, int> FuncExample2 = (x, y) => x +
y;
```

- Gọi các phương pháp sử dụng các chữ ký xác định bởi các **Func**<> hay **Action**<> đại diện.

```
ActionExample1("string for x");
ActionExample2("string for x", "string for y");
Console.WriteLine($"The value is {FuncExample1("1")}");
Console.WriteLine($"The sum is {FuncExample2(1, 2)}");
```

- Nếu bạn tạo các phiên bản của kiểu delegate, hãy sử dụng cú pháp ngắn gọn. Trong một lớp, xác định kiểu delegate và một phương thức có chữ ký phù hợp.

```
public delegate void Del(string message);

public static void DelMethod(string str)
{
    Console.WriteLine("DelMethod argument: {0}", str);
```

```
}
```

- Tạo một thể hiện của kiểu delegate và gọi nó. Khai báo sau đây cho thấy cú pháp cô đọng.

```
Del exampleDel2 = DelMethod;
exampleDel2("Hey");
```

- Khai báo sau sử dụng cú pháp đầy đủ.

```
Del exampleDel1 = new Del(DelMethod);
exampleDel1("Hey");
```

V. Câu lệnh try-catch và using trong xử lý ngoại lệ

- Sử dụng câu lệnh **try-catch** cho hầu hết các xử lý ngoại lệ.

```
static string GetValueFromArray(string[] array, int index)
{
    try
    {
        return array[index];
    }
    catch (System.IndexOutOfRangeException ex)
    {
        Console.WriteLine("Index is out of range: {0}",
index);
        throw;
    }
}
```

- Đơn giản hóa mã của bạn bằng cách sử dụng câu lệnh C # **using**. Nếu bạn có câu lệnh **try-finally** trong đó mã duy nhất trong khối **finally** là lệnh gọi phương thức **Dispose**, hãy sử dụng câu lệnh **using** thay thế.

- Trong ví dụ sau, câu lệnh **try-finally** chỉ gọi **Dispose** trong khối **finally**.

```
Font font1 = new Font("Arial", 10.0f);
try
{
    byte charset = font1.GdiCharSet;
}
finally
{
    if (font1 != null)
    {
        ((IDisposable)font1).Dispose();
    }
}
```

- Bạn có thể làm điều tương tự với câu lệnh using.

```
using (Font font2 = new Font("Arial", 10.0f))
{
    byte charset2 = font2.GdiCharSet;
}
```

- Trong C # 8 và các phiên bản mới hơn, hãy sử dụng cú pháp sử dụng mới không yêu cầu dấu ngoặc nhọn:

```
using Font font3 = new Font("Arial", 10.0f);
byte charset3 = font3.GdiCharSet;
```

VI. Toán tử && và | |

- Để tránh các trường hợp ngoại lệ và tăng hiệu suất bằng cách bỏ qua các phép so sánh không cần thiết, hãy sử dụng **&&** thay cho **&** và || thay vì | khi bạn thực hiện so sánh, như được hiển thị trong ví dụ sau.

```
Console.Write("Enter a dividend: ");
int dividend = Convert.ToInt32(Console.ReadLine());

Console.Write("Enter a divisor: ");
int divisor = Convert.ToInt32(Console.ReadLine());

if ((divisor != 0) && (dividend / divisor > 0))
{
    Console.WriteLine("Quotient: {0}", dividend / divisor);
}
else
{
    Console.WriteLine("Attempted division by 0 ends up
here.");
}
```

- Nếu số chia là 0, mệnh đề thứ hai trong câu lệnh **if** sẽ gây ra lỗi run-time error. Nhưng toán tử **&&** ngắn mạch khi biểu thức đầu tiên là sai. Đó là, nó không đánh giá biểu thức thứ hai. Toán tử & sẽ đánh giá cả hai, dẫn đến lỗi thời gian chạy khi số chia là 0.

VII. Toán tử new

- Sử dụng một trong những hình thức ngắn gọn của việc khởi tạo đối tượng, như được hiển thị trong các khai báo sau. Ví dụ thứ hai cho thấy cú pháp có sẵn bắt đầu bằng C # 9.

```
var instance1 = new ExampleClass();

ExampleClass instance2 = new();
```

Các khai báo trước tương đương với khai báo sau.

```
ExampleClass instance2 = new ExampleClass();
```

- Sử dụng bộ khởi tạo đối tượng để đơn giản hóa việc tạo đối tượng, như được hiển thị trong ví dụ sau.

```
var instance3 = new ExampleClass { Name = "Desktop", ID =
37414,
    Location = "Redmond", Age = 2.3 };
```

- Ví dụ sau đặt các thuộc tính giống như ví dụ trước nhưng không sử dụng bộ khởi tạo.

```
var instance4 = new ExampleClass();
instance4.Name = "Desktop";
instance4.ID = 37414;
instance4.Location = "Redmond";
instance4.Age = 2.3;
```

VIII. Xử lý sự kiện (Event handling)

- Nếu bạn đang xác định một trình xử lý sự kiện mà bạn không cần xóa sau này, hãy sử dụng biểu thức lambda.

```
public Form2()
{
    this.Click += (s, e) =>
        {
            MessageBox.Show(
                ((MouseEventArgs)e).Location.ToString());
        };
}
```

- Biểu thức lambda rút gọn định nghĩa truyền thống sau đây.

```
public Form1()
{
    this.Click += new EventHandler(Form1_Click);
}

void Form1_Click(object sender, EventArgs e)
{
    MessageBox.Show(((MouseEventArgs)e).Location.ToString());
}
```

IX. Static members

Gọi các **static** members bằng cách sử dụng tên lớp: *ClassName.StaticMember*. Phương pháp này làm cho mã dễ đọc hơn bằng cách làm cho truy cập static trở nên rõ ràng. Không đủ điều kiện cho một static members được định nghĩa trong một lớp cơ sở với tên của một lớp dẫn

xuất. Trong khi mã đó biên dịch, khả năng đọc của mã bị sai lệch, và mã có thể bị hỏng trong tương lai nếu bạn thêm một thành viên tĩnh có cùng tên vào lớp dẫn xuất.

X. Truy vấn LINQ

- Sử dụng tên có ý nghĩa cho các biến truy vấn. Ví dụ sau sử dụng **seattleCustomers** cho khách hàng ở Seattle.

```
var seattleCustomers = from customer in customers
                       where customer.City == "Seattle"
                       select customer.Name;
```

- Sử dụng bí danh để đảm bảo rằng tên thuộc tính của các loại ẩn danh được viết hoa chính xác, sử dụng cách viết hoa Pascal.

```
var localDistributors =
    from customer in customers
    join distributor in distributors on customer.City equals
distributor.City
    select new { Customer = customer, Distributor =
distributor };
```

- Đổi tên thuộc tính khi tên thuộc tính trong kết quả sẽ không rõ ràng. Ví dụ: nếu truy vấn của bạn trả về tên khách hàng và ID nhà phân phối, thay vì để chúng là **Name** và **ID** trong kết quả, hãy đổi tên chúng để làm rõ **Name** là tên của khách hàng và **ID** là ID của nhà phân phối.

```
var localDistributors2 =
    from customer in customers
    join distributor in distributors on customer.City equals
distributor.City
    select new { CustomerName = customer.Name, DistributorID =
distributor.ID };
```

- Sử dụng kiểu nhập ngầm trong khai báo các biến truy vấn và biến phạm vi.

```
var seattleCustomers = from customer in customers
                       where customer.City == "Seattle"
                       select customer.Name;
```

- Căn chỉnh các mệnh đề truy vấn dưới mệnh đề **from**, như được hiển thị trong các ví dụ trước.

- Sử dụng mệnh đề **where** trước các mệnh đề truy vấn khác để đảm bảo rằng các mệnh đề truy vấn sau này hoạt động trên tập dữ liệu đã lọc, rút gọn.

```
var seattleCustomers2 = from customer in customers
                        where customer.City == "Seattle"
                        orderby customer.Name
                        select customer;
```

- Sử dụng nhiều mệnh đề **from** thay vì một mệnh đề **join** truy cập các bộ sưu tập bên trong. Ví dụ, một tập hợp các đối tượng **Student** có thể chứa một tập hợp các điểm kiểm tra. Khi truy vấn sau được thực hiện, nó sẽ trả về mỗi điểm trên 90, cùng với họ của học sinh đã nhận được điểm.

```
var scoreQuery = from student in students
                 from score in student.Scores
                 where score > 90
                 select new { Last = student.LastName, score
};
```

XI. Security

Thực hiện theo các nguyên tắc trong Secure Coding Guidelines.

Link tham khảo: https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/coding-style/coding-conventions?fbclid=IwAR3OGHF5pMYGoSX10vwAbOu_e5akOFjnUw5CtyJZShd4SUDsMNBylXVGMkA