C# Code Style

Layout of a class can be found here: [C# Class Layout](#)

Use of Static

Only use **static** when necessary. Please refactor cases you find where we are using static but shouldn't be or don't need to be.

Properties

It is preferred to use Properties instead of fields, unless a field is necessary.

To accommodate readonly field behavior, we can use a getter-only property. Such a property can be initialized in the constructor or inline in the definition of the property

```
//readonly field that we can make a property

private readonly int someNumber = 42;


//getter only property

//could also be set in the constructor

public int SomeNumber { get; } = 42;
```

=> operator

Using the => operator in a property is shorthand for a one line getter:

```
public string MyString {
 get {
    return "something";
    }
}


//same as above

public string MyString => "something";
```

This operator can be used in methods for a one line return from the method:

```
public string GetPaddedString(string input) {
    return $"{    }{input}";
}
```

```
//same as above
public string GetPaddedString(string input) => $"{    }{input}";
```

??= operator

This operator is used to set a value to a variable if it is null:

```
List<int> numbers = null;
int? a = null;

(numbers ??= new List<int>()).Add(5);
Console.WriteLine(string.Join(" ", numbers));  // output: 5

numbers.Add(a ??= 0);  //this adds 0 to the number array AND sets "a" to 0
Console.WriteLine(string.Join(" ", numbers));  // output: 5 0
Console.WriteLine(a);  // output: 0
```

If statements and braces

Always use braces for if statements, even if the action is one line:

```
//discouraged
if(condition) return true;

//discouraged
if(condition)
    return true;

//correct usage
if(condition) {
    return true;
}
```

Ternary statements

Ternary statements should only be used if necessary and when short and one line:

```
//short and sweet example
int num = (b>c) ? b : c;


//bad examples, too long to use this
int num = (condition1 && aReallyLongConditionOfSomeKind && (OtherCondition1 || OtherCondition2)) ?
something : somethingElse;
int num = (condition1 && aReallyLongConditionOfSomeKind && (OtherCondition1 || OtherCondition2)
        && aFewMoreConditions && aFewMoreConditions2) ? something : somethingElse;
```

Regions

Use of regions can help to organize code and make it easier to read. That being said, they can be overused. Please avoid nesting regions in regions.

It is recommended to use regions for the different sections of a class as described in [C# Class Layout](#):

```
#region Constants


const string MY_STRING = "hello";


#endregion


#region Constructors


public MyConstructor() {}


#endregion


#region Properties


public int Num1 { get; set; }
public int Num2 { get; set; }


#endregion


#region Methods


...
```

```
#endregion


etc...
```

Async methods

Async methods should return a Task instead of void (event handlers are the exception). Async methods should have "Async" at the end of their name so when people use the method, they know right away that it can/should be awaited.

```csharp
public async Task DoSomethingAsync()
{
 ...code
    await someAsyncThingAsync();
    ...code but no return;
}


public async Task<IEnumerable<object>> GetObjectsAsync()
{
 IEnumerable<object> objects = await getSomeObjectsAsync();
    ...do other stuff
    return objects;
}
```

Use of "var"

Using var can be convenient, but we should only use var when the type of the variable is apparent in that line of code:

```csharp
//type is apparent in this line
var objList = new List<object>();


//type is not necessarily known, should explicitly define the variable
var myStuff = await getMyStuffAsync();


//these will get you in trouble, as it may not return what you think...
var someResult = getMyStuffAsync();  //no await actually returns the Task, which would not get flagged as var is used
var someResult = getMyStuff();  //if this is async, but not named correctly, same issue as above...
```

Refactoring

We want to be looking for things that can/need to be refactored as we are in files writing new code or fixing others.

Things to look for would be:

- Unnecessary casts or toList
- Unnecessary use of static
- Lean toward the use of IEnumerable

Test Your Code and Code Reviews

- It is **required** that you test your code, **before** checking it in and creating a PR.
- It is **strongly recommended** that after testing your code to ensure it is functional, you **review it** and try to make it more concise and elegant.
- Please have others review your code with you (easy to do if you are pairing) even before creating a PR