C# User-Defined Types

Topics Covered

- Five C# User-Defined Types
- C# Naming Conventions
- enum Types
  - [Flags]
  - enum Base Type
- C# Classes
  - Members
  - Nested Classes
- Interfaces
  - Abstracting Interfaces
  - Implementing Interfaces
- struct
- Delegates
  - Events
  - Custom Delegates

C# User-Defined Types

Type	Definition	Example
class	A class is a construct that enables you to create your own custom types by grouping together variables of other types, methods and events. A class is like a blueprint. It defines the data and behavior of a type.	`public class FirstClass { // TODO: define members }`
enum	Enums are integers with memorable names.	`public enum Direction { Up, Down, Left, Right }`
interface	A set of methods, properties, and/or events defined without implementation.	`public interface IMovable { void Move(Direction direction); }`
struct	While very similar to classes, memory for structs is allocated on the stack and they are copied by value, not by reference. Unlike classes, structs do not support inheritance.	`public struct Point { public double X; public double Y; }`
delegate	Delegates represent methods with specific parameter lists and return type.	`delegate int HashOf(object value);`

Naming Conventions

There are well-established naming conventions in C# programming. These conventions help us to better read and understand code. These become especially important when working on multi-member teams.

Name Type	Casing	Example
Class	Pascal	PokerHandComparer
Field	Camel	pokerHand
Method	Pascal	FindBestHands
Property	Pascal	TopFiveCards
Event	Pascal	IllegalHandDealt
Interface	I+Pascal	IPokerPlayer
Delegate	Pascal+”Handler”	IllegalCardHandler
Local Variable	Camel	tableCards
Method Parameter	Camel	allHands

enum

The enum keyword is used to declare an enumeration, a distinct type that consists of a set of named constants, called the enumerator list.

I think of enums as integers with memorable names. Enums are integer types – they can safely be cast to integers. Here is an example of declaring and using an enum:

```
public enum ConsoleColor
{
    Black = 0,
    DarkBlue = 1,
    DarkGreen = 2,
    DarkCyan = 3,
    DarkRed = 4,
    DarkMagenta = 5,
    DarkYellow = 6,
    Gray = 7,
    DarkGray = 8,
    Blue = 9,
    Green = 10,
    Cyan = 11,
    Red = 12,
    Magenta = 13,
    Yellow = 14,
    White = 15
}
```

The ConsoleColor enum can be used with

Console.ForegroundColor and Console.BackgroundColor properties.

```
Console.ForegroundColor = ConsoleColor.Red;
Console.BackgroundColor = ConsoleColor.White;
Console.Clear();
Console.WriteLine("Hello Colorful World!");
Console.WriteLine();
```

```
Hello Colorful World!
```

[Flags] enum

The ConsoleColor enumeration is exclusive – the Console background cannot be both Black and White. There are situations where it is useful to define non-exclusive enums. We declare these with the [Flags] attribute. Their values are usually assigned as bitmasks. We can use them with the numeric bitwise operators | and ~.

```
[Flags]
public enum DayCombinations
{
    Sunday    = 0x0001,
    Monday    = 0x0002,
    Tuesday   = 0x0004,
    Wednesday = 0x0008,
    Thursday  = 0x0010,
    Friday    = 0x0020,
    Saturday  = 0x0040,
    AllDays   = 0x007f
}
```

```
static void UseFlags()
{
    DayCombinations weekend = DayCombinations.Saturday | DayCombinations.Sunday;
    DayCombinations weekdays = DayCombinations.AllDays & ~weekend;
    Console.WriteLine(weekend);
    Console.WriteLine(weekend.HasFlag(DayCombinations.Wednesday));
    Console.WriteLine(weekdays);
}
```

enum Base Type

By default, enums use an underlying type of integer. It is possible to declare enums using a different base-type:

```
// enums based on other integral types:
enum Small : byte { One, Two, Three, Four };
enum Medium : short { One, Two, Three, Four };
enum Large : int { One, Two, Three, Four };
enum Huge : long { One, Two, Three, Four };
```

The byte type is just 8 bits. It can accommodate up to 256 unique values, and cannot represent a value greater than 255. If used with [Flags], it can only represent 8 individual bits.

C# Classes

```
public static class Console
{
    public static int WindowWidth { get; set; }
    public static bool IsOutputRedirected { get; }
    public static bool IsErrorRedirected { get; }
    public static TextReader In { get; }
    public static TextWriter Out { get; }
    public static TextWriter Error { get; }
    public static Encoding InputEncoding { get; set; }
    public static Encoding OutputEncoding { get; set; }
    public static ConsoleColor BackgroundColor { get; set; }
    public static ConsoleColor ForegroundColor { get; set; }
    public static int BufferHeight { get; set; }
    public static int BufferWidth { get; set; }
    public static int WindowHeight { get; set; }
    public static bool TreatControlCAsInput { get; set; }
    public static int LargestWindowWidth { get; }
    public static int LargestWindowHeight { get; }
    public static int WindowLeft { get; set; }
    public static int WindowTop { get; set; }
    public static int CursorLeft { get; set; }
    public static int CursorTop { get; set; }
    public static int CursorSize { get; set; }
    public static bool CursorVisible { get; set; }
    public static string Title { get; set; }
    public static bool KeyAvailable { get; }
    public static bool NumberLock { get; }
    public static bool CapsLock { get; }
    public static bool IsInputRedirected { get; }
```

All of the user-defined types are essential to a rich C# application, but classes are arguably where 80 - 90% or more of the coding is done. Let's start with a closer look a the Console class. If you right-click on Console in Visual Studio and select "Go to Definition", you will see something like the image.

The full class definition does not fit into a screenshot. What you see here are some of the members of the Console class. You only see their declarations, not their implementations.

These members, with the get or get/set keywords are called Properties. These are some of the many Properties of the Console class.

Console is a static class. Let's look next at the String class, which is not static.

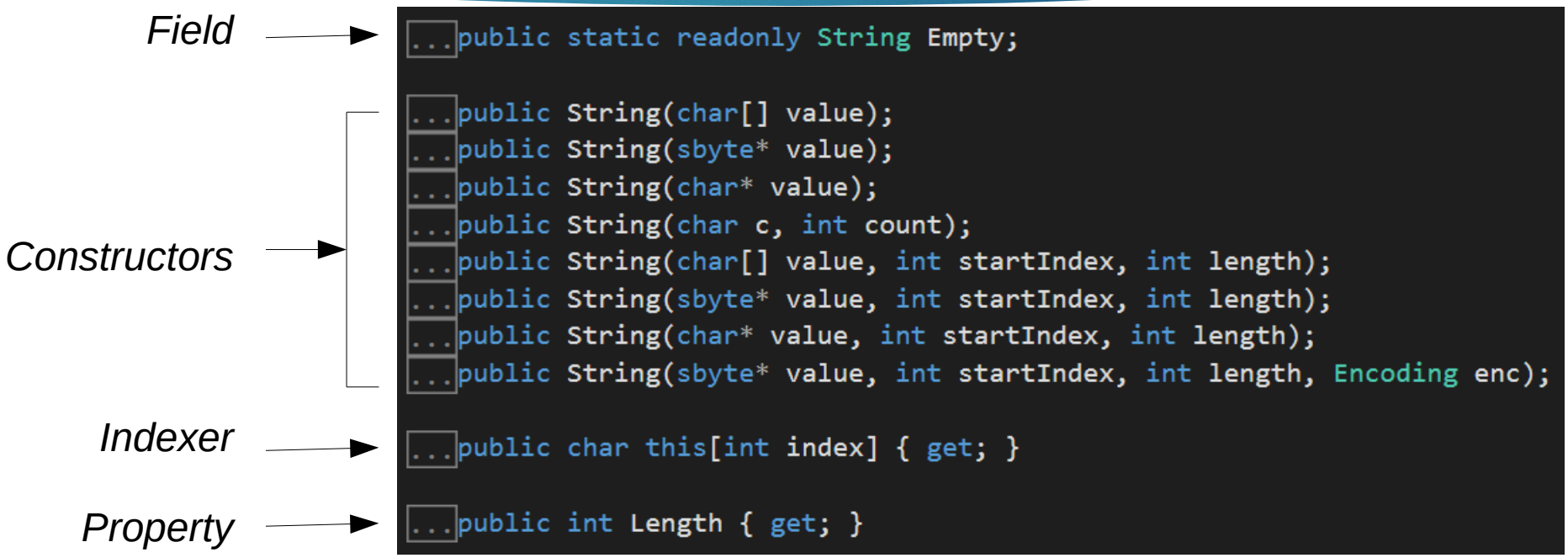
The String Class

public sealed class **String** : ICloneable, IComparable, IComparable<string>, IConvertible, IEquatable<string>, System.Collections.Generic.IEnumerable<char>

This is the declaration of the String class. Let's break it down into parts:

Word	Meaning
public	This keyword is an access modifier. public indicates that the class is visible everywhere.
sealed	The sealed keyword indicates that the String class cannot be inherited.
class	The class keyword indicates that what follows is a class definition.
String	This is the name of the class. Class names follow the same rules as other C# identifiers. There are also well-defined conventions for class names.
ICloneable, IComparable, etc.	These are Interfaces implemented by the String class.

String Class Members



Fields are like variables within a class. This one is declared readonly, meaning that after it is initialized it cannot be changed.

Constructors are special methods with the same name as the class that allow us to create objects of that class using the new keyword.

Indexers are *Properties* accessed using array-like syntax (square brackets).

static String Members

```
public static int Compare(String strA, String strB, StringComparison comparisonType);
public static int Compare(String strA, String strB);
public static int CompareOrdinal(String strA, int indexA, String strB, int indexB, int length);
public static int CompareOrdinal(String strA, String strB);
public static String Concat(String str0, String str1);
public static String Concat(String str0, String str1, String str2);
public static String Concat(object arg0);
```

Further down the list of String members we see a series of methods declared static.

We recognize these as *Methods* by their syntax:

[access modifier] [static] returnType Name(parameter list);

Many of them have the same name but unique parameter lists. These are overloaded methods.

static methods belong to the class, not to an instance of the class.

Instance String Members

```
public object Clone();
public int CompareTo(object value);
public int CompareTo(String strB);
public bool Contains(String value);
public void CopyTo(int sourceIndex, char[] destination, int destinationIndex, int count);
public bool EndsWith(String value);
public bool EndsWith(String value, StringComparison comparisonType);
public bool EndsWith(String value, bool ignoreCase, CultureInfo culture);
public bool Equals(String value);
public bool Equals(String value, StringComparison comparisonType);
public override bool Equals(object obj);
public CharEnumerator GetEnumerator();
public override int GetHashCode();
public TypeCode GetTypeCode();
public int IndexOf(char value, int startIndex, int count);
public int IndexOf(char value, int startIndex);
```

Non-static methods are *instance* methods.
We also find these two custom operators.

```
public static bool operator ==(String a, String b);
public static bool operator !=(String a, String b);
```

Nested Classes

A class can be defined within another class definition. The nested class can be declared either private, public, or internal. As in the example below, nested class have full access to the outer class' private (or protected) interface.

```
public class OuterClass
{
    static readonly Random random = new Random();
    public static readonly IComparer<OuterClass> OuterClassComparer = new InnerClass();

    private int secretValue = random.Next();

    private class InnerClass : IComparer<OuterClass>
    {
        public int Compare(OuterClass x, OuterClass y)
        {
            return Comparer<int>.Default.Compare(x.secretValue, y.secretValue);
        }
    }
}
```

One common use for nested classes is implement an interface that works with the outer class. In the example, a private nested class is invisible to the world, but the interface that it implements is exposed through a static readdonly field.

Interfaces

Think about the word “Interface”. The “interface” of a computer is the ways in which we can interact with it – the screen, the mouse, the keyboard, the speakers, the microphone. If the screen is not a touch-screen, then that part of the computer’s interface is read-only.

In general, the *interface* of a class is the means by which we interact with it. The public interface is those methods, properties, and events by which external users can utilize the class (static) or instances of the class (non-static).

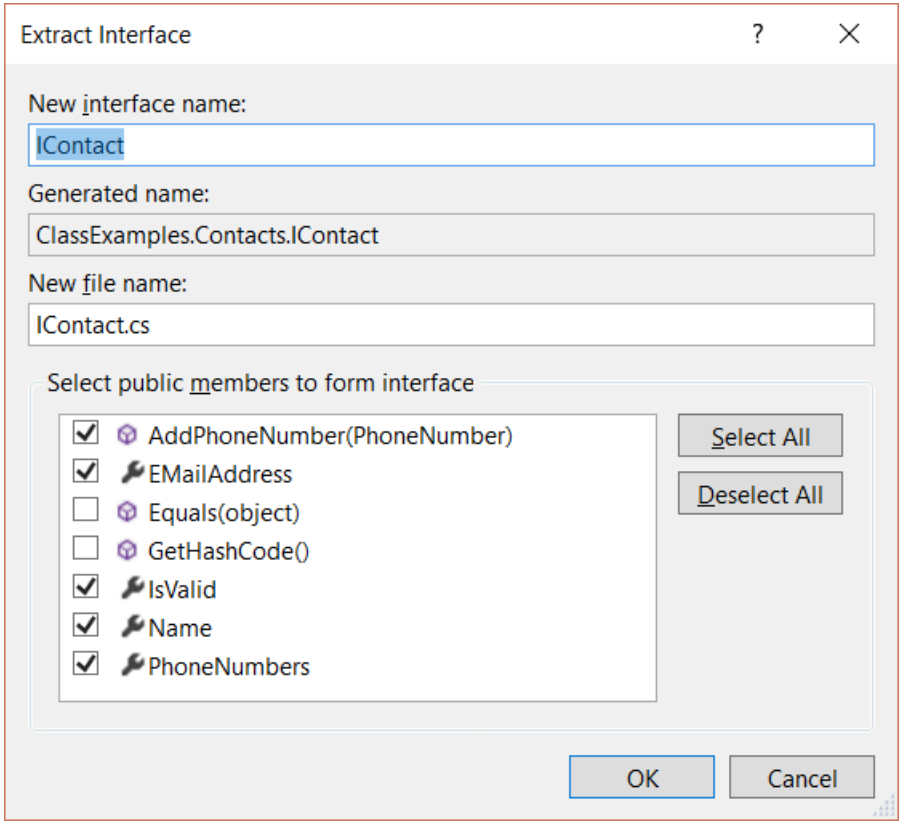
We can also speak of the internal interface (the members to which other other classes in the assembly have access), the protected interface (members through which subclasses can interact), and the private interface (members through which the class interacts with itself).

If you right-click on a class name in Visual Studio and select “Quick Actions and Refactorings...”, you can choose “Extract to Interface”. Let’s try this with the Contacts class.

Extracting the Interface

The *Extract Interface* dialog lists all of the public members of our Contact class and gives us the option of including them (or not) in our Interface definition. Here I have checked only those members defined directly on Contact and un-checked the public methods from the Object base class.

The default name of our interface follows naming conventions: “I” following by the class name. Click OK.



The IContact Interface

```
public interface IContact
{
    string EMailAddress { get; set; }
    bool IsValid { get; }
    string Name { get; set; }
    List<PhoneNumber> PhoneNumbers { get; }

    bool AddPhoneNumber(PhoneNumber phoneNumber);
}
```

Here we see a C# interface definition. Each of Contact's properties is listed with get or set keywords based on whether the property is read-only. We also see the public AddPhoneNumber method, complete with return-type and parameter.

Notice the lack of access modifiers. Notice that there are no implementations.

Implementing Interfaces

After adding our IContact interface, Visual Studio modified our Contact class definition:

```
public class Contact : IContact
```

The C# declaration for a non-static class has this form:

[access modifier] class ClassName [: [base class][, [interfaces implemented]]]

If no base class is provided, then the class inherits directly from Object.

We can say that the Contact class “implements IContact". This means that it provides a public* member corresponding to every property, method, and/or event defined on the interface.

Classes can and often do implement multiple interfaces. Revisit the String class – it implements 6 different interfaces.

Using Interfaces

IContact is now a full-fledged C# type. We can declare a variable of type IContact and access its members just like any other C# type:

```
IContact contactLikeThing = new Contact { Name = "Benny" };
Console.WriteLine(contactLikeThing.Name);
contactLikeThing.AddPhoneNumber(new PhoneNumber { Type = PhoneType.Business, Number = "11111" });
```

On the first line of this snippet, the “contactLikeThing” need not be an actual Contact instance. It could be any class that implements the IContact interface.

Suppose you’ve developed a full-featured ContactList application. A friend in HR says she’d love to have those features in their employee management application, but their software system is based on an Employee class with very simlar properties as Contact. How can you incorporate their Employee class into your ContactList application?

It is possible that Employee could implement IContact with very few code changes. What if refactoring ContactList to work with IContact objects instead of Contact instances were easy? (It is).

struct

Structs are very similar to classes, but they differ subtly with respect to where memory is allocated to store them. Also, structs do not support inheritance.

Most of the built-in C# types are, in fact, structs. Let's look at the Int32, Double, and DateTime type declarations:

```
[System.Runtime.InteropServices.ComVisible(true)]
public struct Int32 : IComparable, IComparable<int>, IConvertible, IEquatable<int>, IFormattable
```

```
[System.Runtime.InteropServices.ComVisible(true)]
public struct Double : IComparable, IComparable<double>, IConvertible, IEquatable<double>, IFormattable
```

```
public struct DateTime : IComparable, IComparable<DateTime>, IConvertible, IEquatable<DateTime>,
IFormattable, System.Runtime.Serialization.ISerializable
```

Put simply: memory for structs is allocated on the stack, while memory for classes is allocated in the heap.

Here is a nice walkthrough explaining the difference between stack and heap:

C# Heap(ing) Vs Stack(ing) in .NET: Part I

Delegates

public delegate void EventHandler(object sender, EventArgs e);

This is the declaration of a delegate type. What do you see?

If we excluded the delegate keyword, this would look exactly like a method declaration:

public void EventHandler(object sender, EventArgs e);

Delegates are types that refer to methods. We can declare a variable of a delegate type, point it to a method, and then invoke it:

The method must match the delegate signature. When we invoke the delegate, we must pass the correct number and types of arguments.

```
private static void DemoEventHandler()
{
    EventHandler handler = MyEventHandler;
    handler(new object(), EventArgs.Empty);
}

private static void MyEventHandler(object sender, EventArgs e)
{
    Console.WriteLine($"{nameof(MyEventHandler)} invoked.");
}
```

Delegates and Events

The most common use of delegates is in the declaration and implementation of events. For example, all user-interface controls in Forms applications have a Click event:

```
public event EventHandler Click;
```

We can attach event handlers to events with code like the following:

```
Button downLoadButton = new Button();
// Attach a handler to the Click event:
downLoadButton.Click += HandleDownloadButtonClicked;
```

```
private void HandleDownloadButtonClicked(object sender, EventArgs e)
{
    // TODO:  Download requested files
}
```

We do not have the source code, but somewhere in the Button (or Control) class is code that *raises* this event, something like one of these:

```
if (Click != null) Click(this, EventArgs.Empty);
Click?.Invoke(this, EventArgs.Empty);
```

Defining Delegates

EventHandler is a system delegate. We can also define our own delegates:

```
// Represents a method that calculates a mean value
public delegate double MeanCalculationStrategy(double[] values);
```

Like all other C# types, we can pass delegates as arguments and use them as property return types. This allows us to use delegates to implement a Strategy Pattern:

```
public class MeanCalculator
{
    public MeanCalculator(MeanCalculationStrategy strategy)
    {
        Strategy = strategy ?? throw new ArgumentNullException(nameof(strategy));
    }

    private MeanCalculationStrategy Strategy { get; set; }

    public double CalculateMean(params double[] values)
    {
        if (values == null || values.Length == 0) return 0;
        return Strategy(values);
    }
}
```

Generic Delegates

Just as we can defined generic methods, we can also define generic delegates. A rather useful generic system delegate is EventHandler<T>:

```
public delegate void EventHandler<T>(object sender, T e);
```

Generic delegates can use generic constraints. In the example below, ScoreItemGenerator represents a method which returns an object which implements IScoreItem:

```
public delegate T ScoreItemGenerator<T>(string name, object expectedValue, object observedValue, double weight)
  where T : IScoreItem;
```

```
public interface IScoreItem
{
    string Name { get; }
    object ExpectedValue { get; }
    object ObservedValue { get; }
    bool IsCorrect { get; }
    double Weight { get; }
    double Score { get; }
    bool IgnoreCase { get; set; }
    bool IncludeScore { get; }
}
```

Topics Covered

- Five C# User-Defined Types
- C# Naming Conventions
- enum Types
  - [Flags]
  - enum Base Type
- C# Classes
  - Members
  - Nested Classes
- Interfaces
  - Abstracting Interfaces
  - Implementing Interfaces
- struct
- Delegates
  - Events
  - Custom Delegates