# C# Overview

Cpt S 321

Washington State University

# .NET 101

- **.NET**: an open-source developer platform to build different types of applications. It includes different languages, libraries, and tools.

- Languages: C#, VisualBasic, F#

- Different implementations of .NET (also called different platforms):
  - **.NET Framework**: a platform for websites, services, desktop apps, and more on Windows. This is one option that you can chose for our Spreadsheet project.
  - **.NET Core**: a cross-platform implementation, i.e., it runs on everything (Windows, Linux, and macOS). It is open source.
  - **.NET 5, .NET 6**: newer version of .NET Core (7 and 8 are coming). Continue the efforts for cross platform support. Using .NET 6 is highly recommended
  - **Avalonia**: a cross-platform UI framework, i.e., great for apps that require windowing. This is another option that you can chose for our Spreadsheet project.
  - **Xamarin/Mono**: a .NET implementation for mobile devices (e.g., iOS and Android)
  - **.NET Standard**: a formal specification of the APIs that are common across .NET implementations. This allows the same code and libraries to run on different implementations.

- Originally came from Microsoft, but other developers and companies are contributing now

# C# Basics

- Managed language - dynamically allocated content is automatically freed after it is determined to no longer be in use.
    - Garbage collection: automatic
    - Disposal: explicitly invoked

- Pointers are not eliminated – they are simply unnecessary most of the time

- Syntax is similar (but definitely not identical) to C++ (VERY similar to Java)

- Has lots of preexisting code associated with it, through the different platforms

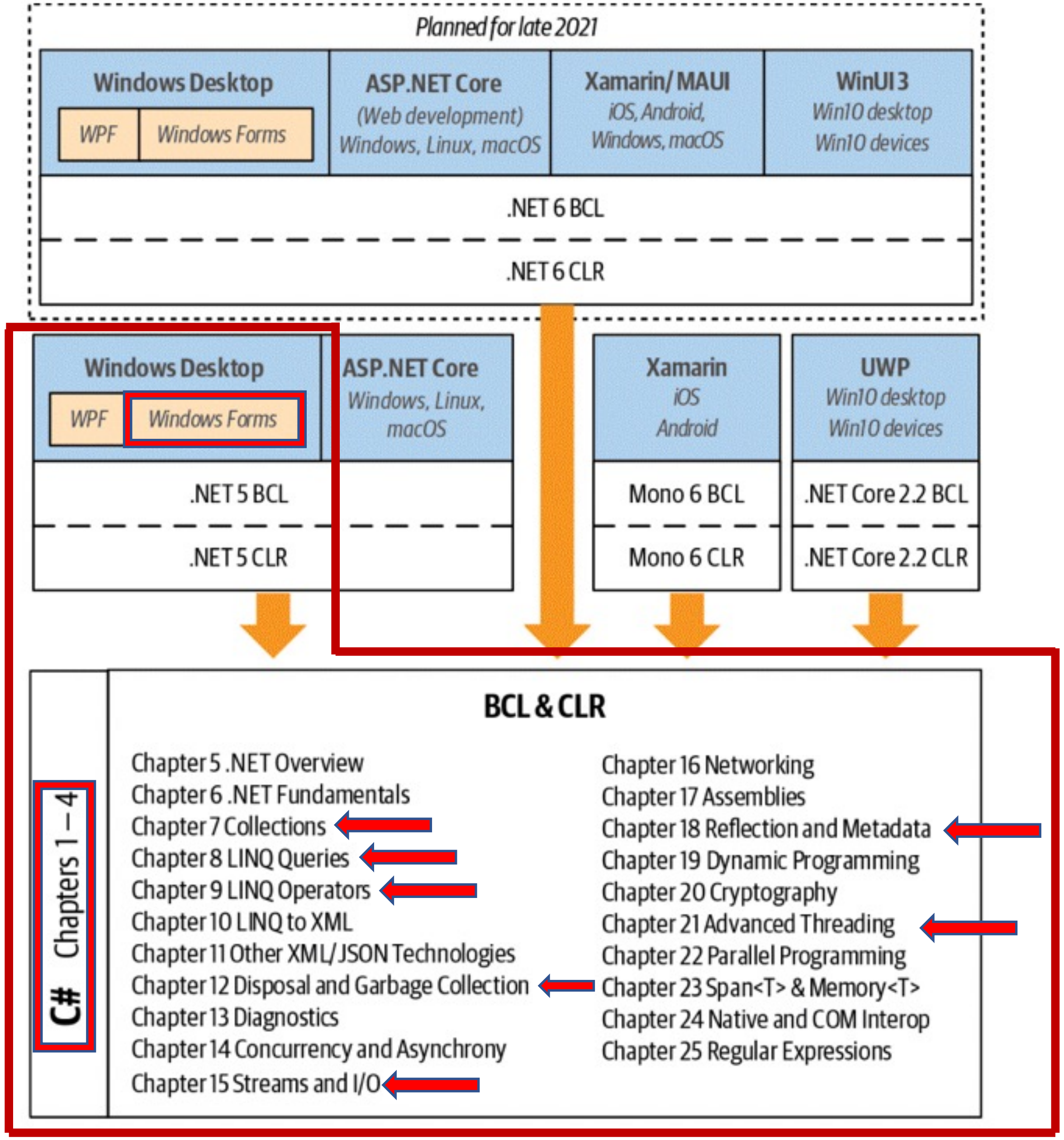
# The C# 9.0 in a Nutshell Book Topics

Framework (runtime) consists of:

- BCL
- CLR
- Application layer

**BCL**: Base Class Library (collections, I/O, text processing, etc)

**CLR**: Common Language Runtime (automatic memory management, exception handling, etc.)



*Planned for late 2021*

Windows Desktop — *WPF* | *Windows Forms*

ASP.NET Core — *(Web development)* *Windows, Linux, macOS*

Xamarin/ MAUI — *iOS, Android, Windows, macOS*

WinUI 3 — *Win10 desktop* *Win10 devices*

.NET 6 BCL

.NET 6 CLR

Windows Desktop — *WPF* | *Windows Forms*

ASP.NET Core — *Windows, Linux, macOS*

.NET 5 BCL

.NET 5 CLR

Xamarin — *iOS* *Android*

Mono 6 BCL

Mono 6 CLR

UWP — *Win10 desktop* *Win10 devices*

.NET Core 2.2 BCL

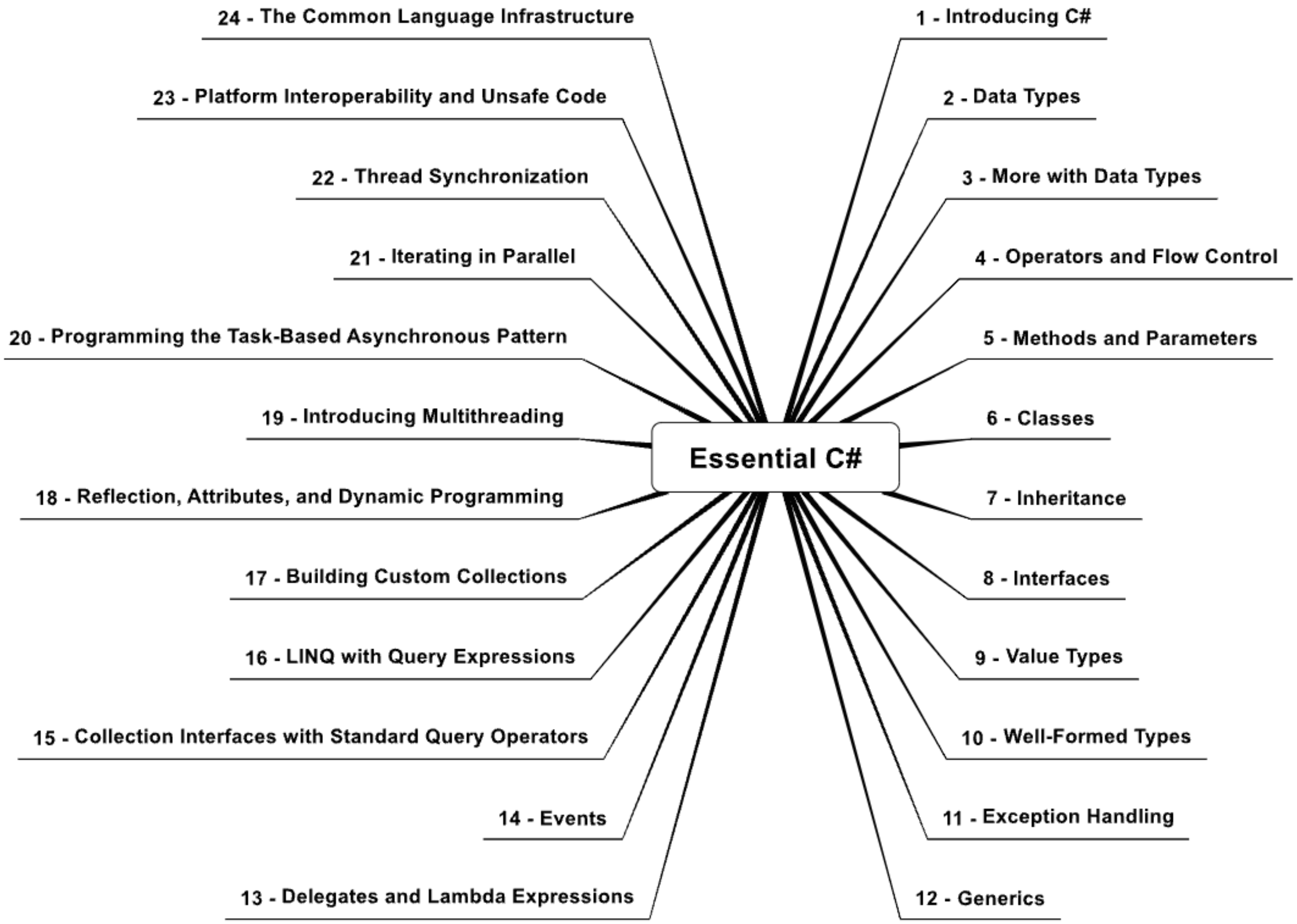
.NET Core 2.2 CLR

C# Chapters 1 – 4

**BCL & CLR**

Chapter 5 .NET Overview
Chapter 6 .NET Fundamentals
Chapter 7 Collections
Chapter 8 LINQ Queries
Chapter 9 LINQ Operators
Chapter 10 LINQ to XML
Chapter 11 Other XML/JSON Technologies
Chapter 12 Disposal and Garbage Collection
Chapter 13 Diagnostics
Chapter 14 Concurrency and Asynchrony
Chapter 15 Streams and I/O
Chapter 16 Networking
Chapter 17 Assemblies
Chapter 18 Reflection and Metadata
Chapter 19 Dynamic Programming
Chapter 20 Cryptography
Chapter 21 Advanced Threading
Chapter 22 Parallel Programming
Chapter 23 Span<T> & Memory<T>
Chapter 24 Native and COM Interop
Chapter 25 Regular Expressions

# The Essential C# Book Topics



Essential C#

1 - Introducing C#

2 - Data Types

3 - More with Data Types

4 - Operators and Flow Control

5 - Methods and Parameters

6 - Classes

7 - Inheritance

8 - Interfaces

9 - Value Types

10 - Well-Formed Types

11 - Exception Handling

12 - Generics

13 - Delegates and Lambda Expressions

14 - Events

15 - Collection Interfaces with Standard Query Operators

16 - LINQ with Query Expressions

17 - Building Custom Collections

18 - Reflection, Attributes, and Dynamic Programming

19 - Introducing Multithreading

20 - Programming the Task-Based Asynchronous Pattern

21 - Iterating in Parallel

22 - Thread Synchronization

23 - Platform Interoperability and Unsafe Code

24 - The Common Language Infrastructure

# C# Basics

- Multi-paradigm programming language with:
  - strong typing -- predominantly statically typed language; enforces static and dynamic type checking
  - imperative -- can use statements to change state
  - declarative -- can use logic flow paradigms
  - functional -- can fit as mathematical functions
  - generic -- allows templating and virtual base class
  - object-oriented -- actually, it’s *all* in objects here: Unified Type System
  - event-driven -- the flow is determined by events
  - component-oriented -- designed to help define the use of objects as services and coherent behaviors

# Similarities with C++

- Has
  - classes, structures,
  - encapsulation, information hiding (sometimes called abstraction), inheritance, and polymorphism

- Has access modifiers (public, private, protected): default versus explicitly specifying them

- Dealing with many basic types (int, short, char, bool) is the same

- For-loops, foreach-loops, and while-loops are just about the same

# Differences from C++

- Arrays are managed objects – not just a pointer to an address
- No pointers (without the "unsafe" language subset, which does support them)
- char* is no longer what we use for a string. We use the string class.
- EVERYTHING is a class or a structure. Integers (int type) are structures, strings are classes.
- Classes are reference types, structures are value types (this is an important one; more on this in a few slides)
- Structures do NOT support inheritance, only classes do
- No globals, everything is declared inside a class

# Differences from C++

- Variables cannot be used without first being initialized in C#

  ```
  int x;
  x+= 5; // Compiler error
  ```

- Dynamically allocated objects are automatically freed
- Syntax for accessing members of objects in C# is simplified. It’s most of the time just the dot (.) - no arrow (->)
  - Works for accessing methods, properties and fields of objects, regardless of whether they are classes or structures.
  - Works for access things declared in other namespaces
  - One small exception for unsafe code, but we’re not going to be dealing much with that

# Variable Declarations in C#

- Local variables in functions aren’t all that different than C++

- Member variables (also called attributes or fields) are similar as well, but each member gets its own access definition:

| C++ | C# |
|---|---|
| class MyClass<br>{<br>private:<br>    int m_number;<br>    string m_name;<br>}; | class MyClass<br>{<br>    private int number;<br>    private string name;<br>} |

# C# Access Modifiers

- Supported access modifiers in C# (for both structs and classes):
  - public
  - **internal**: can be accessed by any code in the same assembly, but not from another assembly.
  - private
- Recall that it was previously mentioned that structures have no inheritance. So **protected** wouldn’t make sense for them.
- Additional supported access modifiers for classes only:
  - protected: can be accessed by code in the same class or in a derived class.
  - **private protected**: access is limited to the containing class or types derived from the containing class <u>within the current assembly.</u>
  - **protected internal**: Access is limited to the current assembly or types derived from the containing class.
- **Default vs explicit access modifiers**: You don’t HAVE to (but you better do in this class and in general) explicitly specify an access modifier. If not explicitly specified, the default rules for modifiers are applied:
  - Internal is the default for classes and structures if no access modifier is specified.
  - Private is default for members of classes and structures.

# Reference Types vs. Value Types

- Recall from C++: there’s a significant difference between dealing with pointers and values. Passing an object by value to a function is different from passing it by reference or pointer.

- Predefined **value** types in C#
  - Numeric: integer (sbyte, short, int, long and byte, ushort, uint, ulong) and real (float, double, decimal) numbers
  - Logical (bool)
  - Character (char)

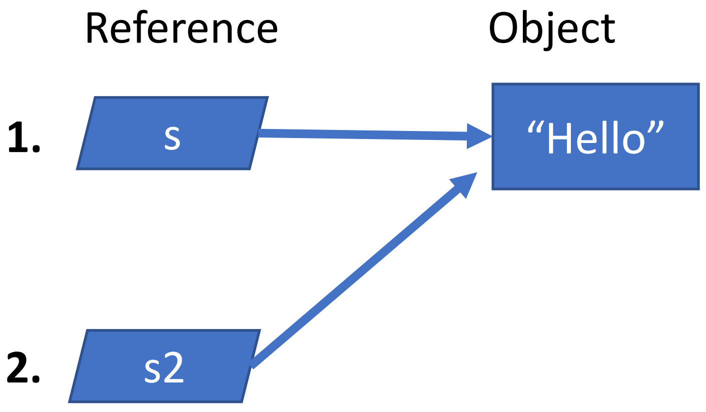
- Predefined **reference** types in C#
  - String
  - Object

- User defined types: Classes are reference types, structures are value types
  - Further reading: Choosing between Class and Struct

# Reference Types vs. Value Types - examples

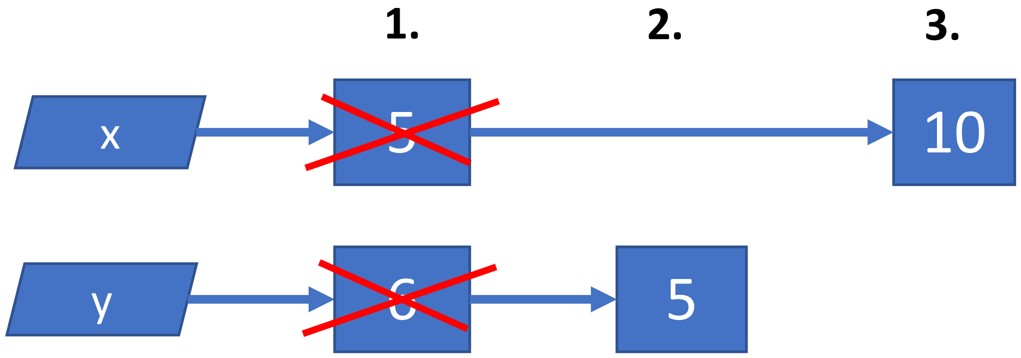
- Example of a reference type.

1. **string s = “Hello”;** // The variable s is implicitly a
   // reference to a string object
   // in this case because of the
   // fact that string is a class.
2. **string s2 = s;** // This copies a reference.
   //Now we have two variables
   // referencing the same string.



- Example of a value type.

1. **int x = 5, y = 6;**
2. **y = x;** // Copies the value of x to y
3. **x += 5;** // Does not affect y at all



# What Can We Declare in a C# Class or Structure?

- Methods, fields (also called attributes), and properties

```
public class BasicMessageClass
{
   private string message = "(default message)"; // FIELD; what would be the access modifier here if we don’t specify?

   public void ShowMessageConsole() { Console.WriteLine(this.message); } // METHOD
   // public void ShowMessageConsole() => Console.WriteLine(this.message); // Expression-bodied member
                                                                       // equivalent to the previous line

   public string Message // PROPERTY : Promotes encapsulation by acting as a wrapper around the field;
                            Acts a lot like a field to code outside of the class,
                            but it is actually pair of methods – accessor (aka getter)
                            and modifier (aka setter);

   {
           get { return message; }          // equivalent to: get => message;
           set { message = value; }         // equivalent to: set => message = value;
   }
}
```

- Somewhere outside this class:

```
BasicMessageClass someBasicMessage = new BasicMessageClass();
someBasicMessage.Message = “new message!”
```

# Properties can be more than simple accessors/modifiers

```
public class AngleClass
{
        private double angleRadians; // angle in radians

        public double AngleDegrees // PROPERTY – angle in degrees
        {
            get { return angleRadians * 180.0 / Math.PI; }
            set { angleRadians = value / 180.0 * Math.PI; }
        }
}
```

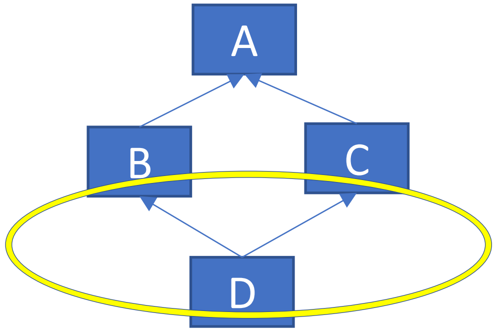
- Allows you to perform logic when setting the field. For example, say you have class that stores an angle value, in radians. You could make a property that allows you get or set that value using degrees.
- A getter (read-only property) or setter (write-only property) or both (read-write)

# Inheritance

- C# supports inheritance, but not multiple inheritance

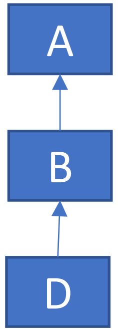
Suppose A, B, C, and D are classes; Arrows indicate inheritance and the direction matters – from child to parent

**Not allowed in C#:**



multiple inheritance

**Allowed in C#:**



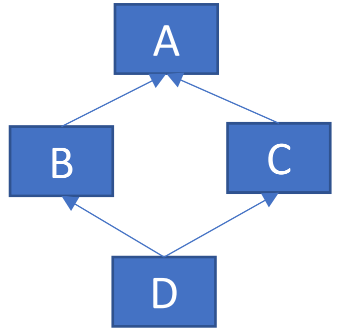
- NOT multiple inheritance;
- This is inheritance with a deeper level of inheritance tree

# Inheritance (cont.)

Not allowed in C#:

A, B, C, D: classes

Arrow: inheritance



- Have interfaces to deal with the lack of multiple inheritance. Classes and structures can implement multiple interfaces.
    - An interface defines a contract (i.e., specifies the method signature but not the implementation)
    - An interface is similar to an abstract base class but interfaces may not contain instance state
    - Typically, behavior is not to be implemented in an interface (only specified in terms of signature). However, in C# 8.0, default implementation for members is allowed. "Because we can" does not imply "We should". Interfaces may also define static members.
- One inheritance exception: ALL classes and structures automatically inherit from "object", which is a base class for all objects in the language. Has methods "Equals", "GetHashCode", "ToString" and a few others.

# Examples of interface, abstract class, concrete class

## IEquatable.cs

```
interface IEquatable<T>
{
   bool Equals(T obj);
}
```

- No state (i.e., data)
- Typically, it will **NOT** contain Implementation
- Name starts with "I"
- Cannot be instantiated

## Shape.cs

- Can have state
- Typically, it will contain methods with implementation
- Cannot be instantiated

```
abstract class Shape
{
   public abstract int GetArea();
}
```

## Car.cs

**// Implements the IEquatable<T> interface**

```
public class Car : IEquatable<Car>
{
   public string Make {get; set;}
   public string Model { get; set; }
   public string Year { get; set; }

   // Implementation of IEquatable<T> interface
   public bool Equals(Car car) =>
      (this.Make, this.Model, this.Year) ==
      (car.Make, car.Model, car.Year);
}
```

## Square.cs

**// Derives/inherits from the Shape abstract class**

```
class Square : Shape
{
   int side;

   public Square(int n) => side = n;

   // Implementation of GetArea method
   public override int GetArea() => side * side;
}
```

# *Overriding* versus *hiding*

## BaseClass.cs

```
public class BaseClass
{
   public virtual void Foo() =>
      WriteLine ("BaseClass.Foo");
}
```

## Overrider.cs

```
public class Overrider : BaseClass
{
   public override void Foo() =>
      WriteLine ("Overrider.Foo");
}
```

## Hider.cs

v1

```
public class Hider : BaseClass
{
   public void Foo() =>
      WriteLine ("Hider.Foo");
}
```

v2

```
public class Hider : BaseClass
{
   public new void Foo() =>
      WriteLine ("Hider.Foo");
}
```

## In the main method of Program.cs

```
Overrider over = new Overrider();
over.Foo();   // Overrider.Foo

Hider h = new Hider();
h.Foo();      // Hider.Foo

BaseClass b1 = over;
b1.Foo();    // Overrider.Foo

BaseClass b2 = h;
b2.Foo();   // BaseClass.Foo
```

In both v1 and v2 we observe ***hiding***. In v1 the compiler generates a warning, in v2 the warning is suppressed.

# Class and Structure Declarations and Implementations

- Declared and implemented in the same place. No more .h file to define and then have a separate .cpp to implement.

- C# code files (.cs files) can define multiple classes and structures and implement them in one code file (it does **NOT** mean you should!). 95% of the code you write will have files with a single type inside.

- Other files that want to use classes/structs declared in another .cs file don't have to include it (no more #include statements). If it's in the project, then it can be used by all other pieces of code in the project.

# .NET platforms

- Prewritten code at your disposal
- Organized into namespaces. Recall how cin, cout, vector and other things were in the std namespace in C++. Namespaces in C# function similarly.
- Predefined types in C# map to types in the System namespace.
    - string -> System.String
    - int -> System.Int32
    - short -> System.Int16
    - and many more…
- Most of the basic types are in the system namespace. Almost all .cs code files have “**using System;**” as one of the first statements in them. Note that in C++ it was “using namespace …” now it’s just “using …”.

# TODOs for this week

1. Read the guides on Git/GitLab or any other resource on the topic. Create a practice repository to help you remember how things work if you don’t remember.

2. Work on HW0 – posted; due in Week 2. This is a warmup HW to help you start setting up your environment.

3. Work on basic C# Tutorials

# Let’s touch base!

- Any questions?