Week 4 - C# NOTES

Abstract class

Abstract can be used with classes, methods, properties, indexers, events

Abstract indicate that a class is intended only to be a base class of other classes, not instantiated on its own.

Members marked as abstract must be implemented by non-abstract classes that derive from abstract class

Abstract class can contain both abstract members and full implemented members

Filed cannot be abstract

Abstract method is by default virtual

Abstract class cannot be instantiated

Abstract class cannot be sealed

It is an error to use static and virtual modifiers together

Abstract class can contain a constructor which will be called by derived class

```
public abstract class BaseClass
{
   protected BaseClass(){}

   public void M1() {}
   public abstact void M2();
}
```

Sealed class

A class declared as sealed cannot be inherited

It is error to use sealed and abstract together

```
sealed class A {}
```

If a method or property overriding base class's method or property declared as sealed cannot be further override

Interface

Interface is like a abstract class with all abstract methods

It provide a way to group related functionalities that a non-abstract class or a struct must implement

Interface can contain methods, properties, events, indexers

Interface can contain static constructor, fields, constants, operators

Interface can't contain instance fields, constructor

Interface members are public by default

A private member must have a default implementation

Implicit interface implementation

Normal use of interface
To use implicit interface implementation a member must be public, non-static, and have the same name and signature

```
interface I1
{
   void M1();
}
class C1:I1
```

```
{
   public void M1(){}
}
```

Class instance can access M1

Explicit interface implementation

Use to avoid name conflict

Only interface instance can access explicit methods

Must not use access specifier

```
interface I1
{
   void M1();
}
class C1:I1
{
   void I1.M1(){}
}
```

If property is set to read-only, write-only, read-write we change it while implementing

Default method implementation is available after c# 8 (.net core 3.0)

Generics

Generics let you tailor a method, class, structure, or interface to the precise data type it acts upon.

It provide type safety

It is more efficient than storing as object and perform boxing and unboxing

To make a generic class

```
class GClass<T>
{
   public GClass(T val){}
   public T Add(T val){}
}
```

To make a generic method

```
public void Display<T>(T val){}
```

To make a generic interface

```
interface IGen<T>
{
   void Show(T val);
}

class GenClass<T> : IGen<T>
{
   public void Show(T val)
   {
      Console.WriteLine(val);
   }
}
```

Static member is shared among all objects of the same generic type means if generic class have static member every generic type will have its own copy

Generic collections is available in System.Collection.Generic namespace

It provide generic class for data structure

It is recommend to not use non-generic class like ArrayList or HashMap

System.Collection.Generic provide many classes like List<T>, Dictionary<TKey, TValue>, Queue<T>, Stack<T>, HashSet<T>

Serialization

Serialization is the process of converting the state of an object into a form that can be persisted or transported.

The complement of serialization is deserialization, which converts a stream into an object.

Together, these processes allow data to be stored and transferred.

JSON : Javascript object notation, lightweight, human-readable

XML : eXtensible Markup Language, more verbose but very powerful

JSON Serialization

User-defined POCO, one-dimensional and jagged arrays and all collections of System.Collections, System.Collections.Generic can be serialized

To serialize

```
string json = JsonSerializer.Serialize(object);
```

To deserialize

```
Student student2 = JsonSerialize.Deserialize<Student>(jsonString);
```

To get pretty jsonstring

```
string jsonStudent3 = JsonSerializer.Serialize(student, new JsonSerializerO
ptions
{
   WriteIndented = true,
});
```

To give custom name

```
[JsonPropertyName("StudentId")]
```

To ignore

```
[JsonIgnore]
```

When we serialize enum it store number and when deserialize it it gets enum back

To store enum as string instead of integer

```
string jsonStudent4 = JsonSerializer.Serialize(student, new JsonSerializerO
ptions
{
   Converters =
   {
      new JsonStringEnumConverter()
   }
});
```

By default fields are ignored to use it

```
[JsonInclude]
```

By default, if the JSON payload you're deserializing contains properties that don't exist in the plain old CLR object (POCO) type, they're simply ignored.

Newtonsoft.Json

case-insensitive by default, camel-case, snake-case property names, allow comments, allow trailing commas, allow property names without quotes, allow single quotes around string values, etc.

```
string jsonStudent5 = Newtonsoft.Json.JsonConvert.SerializeObject(student, Newtonsoft.Json.Formatting.Indented, new Newtonsoft.Json.JsonSerializerSettings
{
    Converters =
    {
        new Newtonsoft.Json.Converters.StringEnumConverter()
    }
});
Console.WriteLine(jsonStudent5);

Student student3 = Newtonsoft.Json.JsonConvert.DeserializeObject<Student>(jsonStudent5);
Console.WriteLine(student3.Id + " " + student3.Name);
```

Attributes

To give custom name

```
[JsonProperty("StudentId")]
```

To ignore

```
[JsonIgnore]
```

XML

XML serialization does not include type information

```
XmlSerializer xmlSerializer = new XmlSerializer(typeof(Student));
using (StringWriter stringWriter = new StringWriter())
{
   xmlSerializer.Serialize(stringWriter, student1);
   string xmlStudent = stringWriter.ToString();
   Console.WriteLine(xmlStudent);
   using (StringReader stringReader = new StringReader(xmlStudent))
   {
      Student student2 = (Student)xmlSerializer.Deserialize(stringReader);
      Console.WriteLine($"Id: {student2.Id}, Name: {student2.Name}, Age:
{student2.Age}, Secret: {student2.Secret}, Grade: {student2.Grade}");
   }
}
```

```
<?xml version="1.0" encoding="utf-16"?>
<Student xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmln
s:xsd="http://www.w3.org/2001/XMLSchema">
  <Id>1</Id>
  <Name>John Doe</Name>
  <Age>20</Age>
  <Address>
    <City>New York</City>
    <State>NY</State>
    <Country>USA</Country>
  </Address>
  <Grade>A</Grade>
</Student>
```

Attributes to use in POCO

To give name to root element

```
[XmlRoot("studentDetails")]
```

To make attribute

```
[XmlAttribute("id")]
```

To give custom name

```
[XmlElement("FullName")]
```

To skip

```
[XmlIgnore]
```

To make array

```
[XmlArray("Skillset")]
[XmlArrayItem("Skil")]
```

Lambda

A lambda expression is a concise way to write an anonymous function

syntax:

```
(parameters) ⇒ expression_or_statement_block
```

parameters ⇒ input values like method parameter

Expression/Statement Block → The logic (single expression or multiple statements inside { }).

> In Func last one is always return type in below example last int is return type and first one is parameter

Lambda with single parameter and a return value

```
Func<int, int> Square = x ⇒ x * x;
```

Lambda with multiple parameters with no return value

```
Action<string> Greet = name ⇒ Console.WriteLine($"Hello, {name}!");
```

multiple statements in the lambda body

```
Action<string> Greet2 = (name) ⇒
{
    string greeting = $"Hello, {name}!";
    Console.WriteLine(greeting);
};
```

Lambda with no parameters

```
Action printMessage = () ⇒ Console.WriteLine("No parameters here!");
```

Lambda creates closure

```
class MyClass
{
    public Func<int> printIt;
    public void Fun()
    {
        int x = 5;

        printIt = () ⇒ x;
    }
}
```

```
MyClass myClass = new MyClass();
//Console.WriteLine(myClass.printIt()); // Error : printIt is not assigned
myClass.Fun();
Console.WriteLine(myClass.printIt());
```

Lambda in LINQ

```
List<int> l = new List<int> { 1, 2, 3, 4, 5 };
l.Where(x ⇒ x%2 == 0).ToList().ForEach(x ⇒ Console.WriteLine(x)); // Outp
ut: 2 4
```

File System

Path Class

Namespace: System.IO

Performs operations on string instances that contain file or directory path information. These operations are performed in a cross-platform manner.
It is recommended to perform file operation like getting extension, combining path using this class as it can operate on both / and \

Important fields

```
Console.WriteLine("AltDirectorySeparatorChar: " + Path.AltDirectorySepara
torChar); // Alternative separator
Console.WriteLine("DirectorySeparatorChar: " + Path.DirectorySeparatorCh
ar); // separator to seprate path
Console.WriteLine("PathSeparator: " + Path.PathSeparator); // separator of
environment variable
```

```
Console.WriteLine("VolumeSeparatorChar: " + Path.VolumeSeparatorChar);
// volume separator
```

Important methods

```
string ChangeExtension(path, extension)
```

Change extension of path if multiple extension is there last one is changed

If extension is null it will remove last extension

Both path(trailiIng) and extension(leading) can contain period or not result will have only one period

```
string Combine(string[]) // other: (string, string), (string, string, string)
```

If and string contain rooted path path before it is ignored

FileInfo

Namespace: System.IO

Provides properties and instance methods for the creation, copying, deletion, moving, and opening of files, and aids in the creation of FileStream objects. This class cannot be inherited.

Important property

Get Directory

```
DirectoryInfo directoryInfo = fileInfo.Directory;
```

Check if file exists

```
Console.WriteLine("File Exists: " + fileInfo.Exists);
```

Important methods

append text to file

```
using (StreamWriter writer = fileInfo.AppendText())
{
    writer.WriteLine("Appending a new line to the file.");
}
```

Copy file

```
FileInfo copiedFile = fileInfo.CopyTo(@".\example_copy.txt", true);
```

Write to file

```
using (StreamWriter writer = newFile.CreateText())
{
    writer.WriteLine("Creating a new text file and writing to it.");
}
```

Read file

```
// OpenText
using(StreamReader reader = fileInfo.OpenText())
{


}
```

Open File

```
using(FileStream fs = fileInfo.Open(FileMode.Open, FileAccess.Read))
{

}
```

Delete file

```
fileInfo.Delete();
```

File class

Provides static methods for the creation, copying, deletion, moving, and opening of a single file, and aids in the creation of FileStream objects.
Use this if need to use only one time for multiple operation on same file consider using FileInfo as it require only one time security check

Append

```
using(StreamWriter streamWriter = File.AppendText("example3.txt"))
{
    streamWriter.WriteLine("Appended text");
}
File.AppendAllLines("example.txt", new string[] { "First line", "Second line" });
File.AppendAllText("example2.txt", "Appended text", Encoding.Unicode); // UTF-16
```

Write

```
 byte[] data = Encoding.UTF8.GetBytes("Hello, World!");
 File.WriteAllBytes("example_bytes.txt", data);

 File.WriteAllLines("example_lines.txt", new string[] { "Line 1", "Line 2" });
 File.WriteAllText("example_text.txt", "Hello, World!", Encoding.UTF8);

 using(FileStream fs = File.OpenWrite("example.txt"))
 {
     // Write to the file
 }
 using(StreamWriter streamWriter = File.CreateText("example_createText.tx
t"))
{
   streamWriter.WriteLine("Created text");
}
```

Read

```
using(FileStream fs = File.OpenRead("example.txt"))
{
   // Read from the file
}
using(StreamReader streamReader = File.OpenText("example.txt"))
{
   string content1 = streamReader.ReadToEnd();
}
byte[] bytes = File.ReadAllBytes("example3.txt");
string[] lines = File.ReadAllLines("example3.txt");
IEnumerable<string> lines2 = File.ReadLines("example3.txt");
string content = File.ReadAllText("example3.txt");
```

Other way

```
using(FileStream fs = File.Open("example.txt", FileMode.OpenOrCreate, Fil
eAccess.ReadWrite))
```

```
{
    // Read or write to the file
}
```

Other methods

```
 File.Copy("example.txt", "example_copy.txt", true); // true to overwrite if ex
ists
 File.Create("example_create.txt").Close();  // By defaule it does not close t
he file
 File.Delete("example_create.txt");
 File.Exists("example.txt")
 File.Move("example_copy.txt", "example_moved.txt", true); // true to overw
rite if exists
```

Metadata manipulation

```
File.SetAttributes("hidden_example.txt", fileAttributes);
File.SetCreationTime("example.txt", DateTime.Now.AddDays(-5));
// SetCreationTimeUtc, SetLastAccessTime, SetLastAccessTimeUtc, SetLas
tWriteTime, SetLastWriteTimeUtc

FileAttributes attributes = File.GetAttributes("example.txt");
Console.WriteLine(attributes.ToString());
Console.WriteLine(File.GetCreationTime("example.txt"));
Console.WriteLine(File.GetCreationTimeUtc("example.txt"));
Console.WriteLine(File.GetLastAccessTime("example.txt"));
Console.WriteLine(File.GetLastAccessTimeUtc("example.txt"));
Console.WriteLine(File.GetLastWriteTime("example.txt"));
Console.WriteLine(File.GetLastWriteTimeUtc("example.txt"));
```

DirectoryInfo class

Exposes instance methods for creating, moving, and enumerating through directories and subdirectories. This class cannot be inherited.

Propertry

```
 Console.WriteLine("Directory Exists: " + directoryInfo.Exists);
 Console.WriteLine("Directory Name: " + directoryInfo.Name);
 Console.WriteLine("Directory Parent: " + directoryInfo.Parent); // Parent directory
 Console.WriteLine("Directory Root: " + directoryInfo.Root);  // Root directory
```

Method

Create directory or subdirectory

```
directoryInfo1.Create();
DirectoryInfo subDirectory = directoryInfo1.CreateSubdirectory("SubDirectory");
```

Delete directory

```
directoryInfo.Delete();
```

To get all subdirectory

```
IEnumerable<DirectoryInfo> directoryInfos = directoryInfo1.EnumerateDirectories("*txt", SearchOption.AllDirectories);
DirectoryInfo[] directoryInfos = directoryInfo1.GetDirectories();
```

To get all files

```
IEnumerable<FileInfo> fileInfos = directoryInfo1.EnumerateFiles("*", Search
Option.AllDirectories);
FileInfo[] fileInfos = directoryInfo1.GetFiles();
```

To move directory

```
directoryInfo1.MoveTo(@"D:\tempNew");
```

Directory class