Week 4 - C# NOTES

Created by | DG Dakshil Gorasiya

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

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

To make a generic method

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

To make a generic interface

```csharp
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>

SortedList

namespace: System.Collection.Generics and also avaliable in System.Collection

Represents a collection of key/value pairs that are sorted by the keys and are accessible by key and by index.

Important Properties

```
Console.WriteLine(sl4.Capacity); // get; set;

Console.WriteLine(sl3.Count); // get;

Console.WriteLine(sl3["1"]); // get; set; arg: TKey key

IList<string> keys = sl3.Keys; // get;

IList<string> values = sl3.Values; // get;
```

Important Methods

To add

```
sl.Add(TKey key, TValue value)
```

Contains

```
sl.ContainsKey(TKey key)
sl.ContainsValue(TValue value)
```

GetAtIndex

```
sl.GetKeyAtIndex(index);
sl.GetValueAtIndex(index);
```

IndexOf

```
sl.IndexOfKey(TKey key);
sl.IndexOfValue(TValue value);
```

Remove

```
sl.Remove(TKey key);
sl.RemoveAt(index)
```

TryGetValue

```
sl.TryGetValue(TKey key, out TValue? value);
```

BitArray

Namespace: System.Collection

Manages a compact array of bit values, which are represented as Booleans, where `true` indicates that the bit is on (1) and `false` indicates the bit is off (0).

Important property

```
Console.WriteLine("Count :" + br1.Count); // get;

Console.WriteLine("IsReadOnly :" + br1.IsReadOnly); // get;
```

```
Console.WriteLine("IsSynchronized :" + br1.IsSynchronized); // get;

br1[1] = true; // get; set; arg: int index



// If Length is set to a value that is less than Count, the BitArray is truncated
and the elements after the index value -1 are deleted.
// If Length is set to a value that is greater than Count, the new elements ar
e set to false.
br1.Length = 5; // get; set;
```

Important methods

```
br1.And(br2);
br1.Or(br2);
br1.Xor(br2);
bt1.Not();
```

To set a bit

```
br.Set(index, false);
```

To set all bits

```
br.SetAll(true);
```

Get

```
br.Get(index);
```

To check

```
br.HasAllSet();
br.HasAnySet();
```

LinkedList

Namespace: System.Collections.Generic

Represents a doubly linked list.

Important Property

```
Console.WriteLine(ll2.Count);

LinkedListNode<int> first = ll2.First;
Console.WriteLine(first.Value);
Console.WriteLine(first.Next.Value);

LinkedListNode<int> last = ll2.Last;
Console.WriteLine(last.Value);
Console.WriteLine(last.Previous.Value);
```

Important Method

To add

```
ll.AddAfter(LinkedListNode<T>, value); // reference, new node value
ll.AddAfter(LinkedListNode<T>, LinkedListNode<T>); // reference, new node
e
```

Also it have AddBefore, AddFirst, AddLast

Contains

```
ll.Contains(value);
```

Find

```
LinkedListNode<T> node = ll.Find(value);
```

Also it have FindLast

Remove

```
ll.Remove(value);
```

Also have RemoveFirst, RemoveLast

HashSet

Namespace: System.Collections.Generic

Represents a set of values.

Property

```
hs.Count
```

Important Methods

Add

```
hs.Add(value);
```

Contains

```
hs.Contains(value);
```

Set Operation

```
hs.ExceptWith(IEnumerable);
hs.IntersectWith(IEnumerable);
hs.UnionWith(IEnumerable);
hs.IsProperSubsetOf(IEnumerable);
hs.IsSubsetOf(IEnumerable);
hs.IsProperSuperSetOf(IEnumerable);
hs.IsSuperSetOf(IEnumerable);
hs.SetEquals(IEnumerable);
```

Remove

```
hs.Remove(T value);
hs.RemoveWhere(IPrediate);
```

ArrayList

Namespace: System.Collections

Implements the IList interface using an array whose size is dynamically increased as required.

Important Property

```
Console.WriteLine(al3.Count); // get; set;
Console.WriteLine(al3.Capacity); // get;
Console.WriteLine(al3.IsFixedSize); // get;
Console.WriteLine(al3.IsReadOnly); // get;
Console.WriteLine(al3.IsSynchronized); // get;
```

```
Console.WriteLine(al3[1]); // get; set; arg: int index
```

Important Methods

Add

```
al.Add(object);
al.AddRange(object[]);
```

Sort

```
al.Sort();
```

Search

```
al.BinarySearch(object);
al.Contains(object);
```

Insert

```
al.Insert(index, object);
al.InsertRange(index, object[]);
```

Remove

```
al.Remove(object);
al.RemoveAt(index);
al.RemoveRange(index, count);
```

Reverse

```
al.Reverse();
```

Copy

```
ArrayList.FixedSize(al);
al.GetRange(index, count);
ArrayList.ReadOnly(al);
```

HashTable

Namespace: System.Collection

Represents a collection of key/value pairs that are organized based on the hash code of the key.

Important property

```
Console.WriteLine(ht4.Count); // get;

ht4[1] = "One"; // get; set;
Console.WriteLine(ht4[1]);

foreach(object key in ht4.Keys) // get;
{
    Console.WriteLine(key);
}

foreach(object value in ht4.Values) // get;
{
    Console.WriteLine(value);
}
```

Important Methods

Add

```
ht.Add(object key, object value);
```

Contains

```
ht.Contains(key);
ht.ContainsKey(key);
ht.ContainsValue(val);
```

Remove

```
ht.Remoev(val);
```

Dictionary

Namespace: System.Collection.Generics

Represents a collection of keys and values.

Important property

```
IEqualityComparer<string> comparer = d3.Comparer; // get;
Console.WriteLine(comparer.Equals("hello", "HELLO")); // True

Console.WriteLine(d4.Count); // get;

d4[4] = "Four"; // get; set;
Console.WriteLine(d4[4]); // Four

foreach(int key in d4.Keys) // get;
{
```

```
        Console.WriteLine(key);
    }

    foreach(string value in d4.Values) // get;
    {
        Console.WriteLine(value);
    }
```

Important Methods

Add

```
    d.Add(TKey key, TValue value);
```

Contains

```
    d.ContainsKey(key);
    d.ContainsValue(value);
```

Remove

```
    d.Remove(TKey key);
    d.Remove(TKey key, out TValue? val);
```

TryAdd

```
    d.TryAdd(key, val);
```

TryGetValue

```
    d.TryGetValue(TKey key, out TValue? val);
```

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 JsonSerializerOptions
{
    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 JsonSerializerOptions
{
    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
<?xml version="1.0" encoding="utf-16"?>
<Student xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns: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)); // Output: 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.AltDirectorySeparatorChar); // Alternative separator
Console.WriteLine("DirectorySeparatorChar: " + Path.DirectorySeparatorChar); // 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(traililng) 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" });
```

```csharp
 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.txt"))
 {
    streamWriter.WriteLine("Created text");
 }
```

Read

```csharp
 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

```csharp
 using(FileStream fs = File.Open("example.txt", FileMode.OpenOrCreate, FileAccess.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 dir
ectory
 Console.WriteLine("Directory Root: " + directoryInfo.Root);  // Root director
y
```

Method

Create directory or subdirectory

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

Delete directory

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

To get all subdirectory

```
IEnumerable<DirectoryInfo> directoryInfos = directoryInfo1.EnumerateDire
ctories("*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

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

To create directory

```
Directory.CreateDirectory(@"D:\temp1");
```

To get FileSystemEntries

```
IEnumerable<string> entries = Directory.EnumerateFileSystemEntries(path,
"*", SearchOption.AllDirectories);
```

Attribute

GetCreationTime, GetLastAccessTime, GetLastWriteTime, GetCreationTimeUtc, GetLastAccessTimeUtc, GetLastWriteTimeUtc

```
SetCreationTime(path, DateTime);
```

SetCreationTimeUtc, SetLastAccessTime, SetLastAccessTimeUtc, SetLastWriteTime, SetLastWriteTimeUtc

Current directory

```
Directory.SetCurrentDirectory(@"D:\temp1");
Console.WriteLine(Directory.GetCurrentDirectory());
```

DriveInfo class

Provides access to information on a drive.

Property

```
// Avaliable to the user on the drive in bytes
// Available free space on the drive in bytes
driveInfo.AvailableFreeSpace

// Total free space available
// Total free space on the drive in bytes
driveInfo.TotalFreeSpace

// Total size of the drive in bytes
driveInfo.TotalSize

// Drive format (e.g., NTFS, FAT32)
driveInfo.DriveFormat

// Drive type (e.g., Fixed, Removable)
driveInfo.DriveType

// IsReady indicates if the drive is ready
driveInfo.IsReady

// Name
driveInfo.Name

// Root directory of the drive
DirectoryInfo rootDir = driveInfo.RootDirectory;
```

```
// Volume label of the drive
driveInfo.VolumeLabel
```

Method to get all drives

```
DriveInfo[] drives = DriveInfo.GetDrives();
```

StreamReader

Implements a TextReader that reads characters from a byte stream in a particular encoding.

Property

```
// Type → System.IO.Stream
Console.WriteLine("BaseStream: " + reader.BaseStream.ToString()); // Gets
the underlying stream of the StreamReader

// Type → System.Text.Encoding
Console.WriteLine("CurrentEncoding: " + reader.CurrentEncoding.ToString
()); // Gets the current character encoding used by the StreamReader

// Type → System.Boolean
Console.WriteLine("EndOfStream: " + reader.EndOfStream); // Indicates wh
ether the StreamReader has reached the end of the stream
```

Method

```
Close() // to close stream and StreamReader
```

```
DiscardBufferedData()
```

Use the DiscardBufferedData method to reset the internal buffer for the StreamReader object. You need to call this method only when the position of the internal buffer and the BaseStream do not match. These positions can become mismatched when you read data into the buffer and then seek a new position in the underlying stream. This method slows performance and should be used only when absolutely necessary, such as when you want to read a portion of the contents of a StreamReader object more than once.

```
Dispose(); // internally call close
```

```
Peek(); //return next character without consuming it or -1 if no more characters
```

Read

```
Console.WriteLine((char)reader.Read()); // Read next char and move to next position

Span<char> buffer = new Span<char>(new char[5]);
int charRead = reader.Read(buffer); // Read chars into buffer
Console.WriteLine(buffer.ToString());

char[] bufferArray = new char[5];
//bufferarray, buffer start index, number of chars to read
int charReadArray = reader.Read(bufferArray, 0, bufferArray.Length);
Console.WriteLine(bufferArray);
```

Other method are ReadLine() and ReadToEnd()

StreamWriter

Implements a TextWriter for writing characters to a stream in a particular encoding.

Property

```
Console.WriteLine(sw.AutoFlush); // get AutoFlush property
//sw.AutoFlush = true; // set AutoFlush property

Console.WriteLine(sw.Encoding); // get Encoding property

Console.WriteLine(sw.BaseStream); // get BaseStream property
```

```
// write buffer to stream and close it
Close();
Dispose();
```

```
Flush(); // Write data back to stream
```

Write

```
sw.Write("{0}", 123); // format, arg0(object)

char[] buffer = new char[] { 'a', 'b', 'c', 'd', 'e' };
sw.Write(buffer, 1, 2); // buffer, startIndex, count

sw.Write("HI"); // string

sw.Write(buffer); // char[]

sw.Write('Z'); // char

sw.WriteLine("HELLo");
```

FileStream class

Provides a Stream for a file, supporting both synchronous and asynchronous read and write operations.

Property

```
fs.CanRead // bool CanRead { get; }

fs.CanWrite // bool CanWrite { get; }

fs.CanSeek // bool CanSeek { get; }

fs.IsAsync // bool IsAsync { get; }

fs.Length // long Length { get; }

fs.Position // long Position { get; set; }

fs.Name // string Name { get; }

SafeFileHandle handle = fs.SafeFileHandle; // SafeFileHandle SafeFileHandle { get; }
```

Method

Copy from one stream to other

```
fs1.CopyTo(fs2);
```

Lock unlock file

```
fs1.Lock(startIndex, length);
fs2.Unlock(startIndex, length);
```

Read

```
fs.Read(buffer, startIndex, Length);
byte oneByte = (byte)fs2.ReadByte();
```

Write

```
fs.Write(buffer, startIndex, length);
fs.WriteByte(byte);
```

To set length of file

```
fs.SetLength(size);
```

Seek

```
fs.Seek(0, SeekOrigin.Begin);
```

Dispose

```
fs.Dispose();
fs.Close();
```

Extension Method

To create a extension method

1. Create a static class

2. Create a static method

3. The **first parameter** of the method must specify the type it extends, and it must be preceded by the `this` modifier.

To use extension method we can use it like instance method just make sure it is available by USING

We cannot override class method with extension method

If class have same method signature extension method will not be called as instance method have higher priority.

```
public static class StringExtension
{
    public static int WordCount(this string s)
    {
        return s.Split([' ', '.', '?'], StringSplitOptions.RemoveEmptyEntries).Length;
    }
}

internal class Program
{
    static void Main(string[] args)
    {
        string s = "Hello how are you?";
        Console.WriteLine(s.WordCount());
    }
}
```

Array

It is a special class
It is base class for all array in clrce

Only system and complier can derive from it

Property

```
Console.WriteLine("FixedSize : " + arr.IsFixedSize);
Console.WriteLine("IsReadOnly : " + arr.IsReadOnly);
Console.WriteLine("IsSynchronized : " + arr.IsSynchronized); // Thread Saf
e
Console.WriteLine("Length : " + arr.Length);
Console.WriteLine("LongLength : " + arr.LongLength);
Console.WriteLine("Rank : " + arr.Rank); // Number of dimensions
```

BinarySearch

```
Array.BinarySearch(arr, 5)
```

Copy

```
Copy(sourceArray, sourseIndex, destinationArray, destinationIndex, length)
CopyTo(destinationArray, destinationIndex)
```

Fill

```
Array.Fill(arr, 0); // Fill all elements with 0
```

Find

```
Array.Find(arr, element ⇒ element > 5); //FindLast, FindAll, FindIndex, Find
LastIndex
```

ForEach

```
Array.ForEach(arr, element ⇒ Console.Write(element + " "));
```

Get dimensionwise length

```
GetLength(dimension)
```

Give minimum and maximum index of the array at a particular dimension

```
Console.WriteLine(arr.GetLowerBound(0) + " " + arr.GetUpperBound(0)); //
GetLowerBound(dimension), GetUpperBound(dimension)
```

Resize array

```
Array.Resize(ref arr, 20);
```

Reverse

```
Array.Reverse(arr, 0, 10);
```

Sort

```
Array.Sort(arr, new ReverseComparer());
public class ReverseComparer : IComparer<int>
{
    public int Compare(int x, int y)
    {
        return y.CompareTo(x);
    }
}
```

Base Library

HttpClient

Provides a class for sending HTTP requests and receiving HTTP responses from a resource identified by a URI.

Namespace: System.Net.Http

To cancel all pending requests

```
httpClient.CancelPendingRequests();
```

To send get request

```
HttpResponseMessage response = await httpClient.GetAsync(url);
string responseBody = await response.Content.ReadAsStringAsync();
Console.WriteLine($"Status code {response.StatusCode}");
Console.WriteLine("IsSuccessStatusCode : " + response.IsSuccessStatusCode);
Console.WriteLine("Headers : " + response.Headers.ToString());
```

To send post request

```
string urlPost = "https://jsonplaceholder.typicode.com/posts";
var postData = new
{
    Title = "My Post Title",
    Body = "This is the body of my post.",
    UserId = 1
};
string json = System.Text.Json.JsonSerializer.Serialize(postData);
StringContent content = new StringContent(json, System.Text.Encoding.UTF8, "application/json");

response = await httpClient.PostAsync(urlPost, content);
```

To add default header

```
// To add headers which will be sent with every request
httpClient.DefaultRequestHeaders.Add("Custom-Header", "HeaderValue");
```

To set header for one request

```
HttpRequestMessage request = new HttpRequestMessage(HttpMethod.Get, url);

// To add headers just for this request
request.Headers.Add("Custom-Header", "HeaderValue");

response = await httpClient.SendAsync(request);
responseBody = await response.Content.ReadAsStringAsync();
Console.WriteLine(responseBody);
```

To add query parameter

```
UriBuilder builder = new UriBuilder("https://jsonplaceholder.typicode.com/posts");
var query = HttpUtility.ParseQueryString(builder.Query);
query["userId"] = "1";
builder.Query = query.ToString();
string urlWithParams = builder.ToString();
Console.WriteLine(urlWithParams);
```

To add file

```
MultipartFormDataContent multiPartContent = new ();
FileStream file = new FileStream("path/to/file.txt", FileMode.Open, FileAccess.Read);
multiPartContent.Add(new StreamContent(file), "fileField", "file.txt");
response = await httpClient.PostAsync(urlPost, multiPartContent);
```

RestHandler

A simple REST client that sends HTTP requests and receives responses.

To send get request

```
RestClientOptions options = new RestClientOptions("https://jsonplaceholde
r.typicode.com/")
{
    // Configure options as needed
    Timeout = TimeSpan.FromSeconds(10) // Set timeout to 10
};

RestClient client = new RestClient(options);

RestRequest request = new RestRequest("/posts/1", Method.Get);

RestResponse response = await client.GetAsync(request);

Console.WriteLine(response.Content);
Console.WriteLine(response.StatusCode);
```

To send post request

```
RestRequest postRequest = new RestRequest("posts", Method.Post);
var postData = new
{
    Title = "My Post Title",
    Body = "This is the body of my post.",
    UserId = 1
};
postRequest.AddJsonBody(postData);
```

```
RestResponse postResponse = await client.PostAsync(postRequest);
Console.WriteLine(postResponse.Content);
```

To add file

```
postRequest.AddFile("fileField", "path/to/file.txt");
```

To add header

```
request.AddHeader("Accept", "application/json");
```

To add query parameter

```
request.AddParameter("key", "value");
```

EPPLUS

EPPlus is a library that works directly with the Office Open XML (OOXML) file format, which is the foundation for `.xlsx` files.

Add

```
using OficeOpenXml;
```

To set license

```
ExcelPackage.License.SetNonCommercialPersonal("Dakshil");
```

To read excel file

```
using(ExcelPackage package = new ExcelPackage(fileInfo))
{
   ExcelWorksheet workSheet = package.Workbook.Worksheets["Sheet1"];

   int startRow = workSheet.Dimension.Start.Row;
   int endRow = workSheet.Dimension.End.Row;
   int startColumn = workSheet.Dimension.Start.Column;
   int endColumn = workSheet.Dimension.End.Column;

   for (int row = startRow; row <= endRow; row++)
   {
      for (int col = startColumn; col <= endColumn; col++)
      {
         Console.Write($"{workSheet.Cells[row, col].Value}\t");
      }
      Console.WriteLine();
   }
}
```

To write into excel file

```
using(ExcelPackage package = new ExcelPackage(fileInfo))
{
   ExcelWorksheet worksheet = package.Workbook.Worksheets.Add("She
et1");

   // Add some text to cell A1
   worksheet.Cells["A1"].Value = "Product";
   worksheet.Cells["B1"].Value = "Price";
   worksheet.Cells["C1"].Value = "Quantity";

   // Add some data
   worksheet.Cells["A2"].Value = "Laptop";
   worksheet.Cells["B2"].Value = 1200.50;
   worksheet.Cells["C2"].Value = 5;
```

```
worksheet.Cells["A3"].Value = "Mouse";
worksheet.Cells["B3"].Value = 25.00;
worksheet.Cells["C3"].Value = 20;

// Formating
worksheet.Cells["A1:C1"].Style.Font.Bold = true;
worksheet.Cells["A:C"].AutoFitColumns();

// Save the package
package.Save();
}
```

Quartz

A library for job scheduling and cron jobs

To create a job

```
public class SimpleJob : IJob
{
    public async Task Execute(IJobExecutionContext context)
    {
        await Console.Out.WriteLineAsync("Hello, Quartz.NET! " + DateTime.Now);
    }
}
```

To create trigger to call every second

```
ITrigger trigger = TriggerBuilder.Create()
    .WithIdentity("myTrigger", "group1")
    .StartNow()
    .WithSimpleSchedule(x ⇒ x
        .WithIntervalInSeconds(1)
```

```
    .RepeatForever())
  .Build();
```

To create trigger for cron jobs

```
ITrigger trigger1 = TriggerBuilder.Create()
    .WithIdentity("myTrigger1", "group1")
    .WithSchedule(CronScheduleBuilder
        .AtHourAndMinuteOnGivenDaysOfWeek(12, 0, DayOfWeek.Monday, DayOfWeek.Wednesday, DayOfWeek.Friday))
    .Build();
```

Complete code to start a job

```
ISchedulerFactory schedulerFactory = new StdSchedulerFactory();
IScheduler scheduler = await schedulerFactory.GetScheduler();
await scheduler.Start();

IJobDetail job = JobBuilder.Create<SimpleJob>()
    .WithIdentity("myJob", "group1")
    .Build();

ITrigger trigger = TriggerBuilder.Create()
    .WithIdentity("myTrigger", "group1")
    .StartNow()
    .WithSimpleSchedule(x ⇒ x
        .WithIntervalInSeconds(1)
        .RepeatForever())
    .Build();

await scheduler.ScheduleJob(job, trigger);

Console.WriteLine("Scheduler started. Press any key to exit...");
Console.ReadKey();
```

```
await scheduler.Shutdown();
```

Reflections

Reflection is a namespace which provide runtime dynamic information

To get constructor

```
ConstructorInfo[] constructorInfos = type.GetConstructors();
foreach (var constructor in constructorInfos)
{
    Console.WriteLine("Constructor: " + constructor.Name);
    ParameterInfo[] parameters = constructor.GetParameters();
    foreach (var param in parameters)
    {
        Console.WriteLine("  Parameter: " + param.Name + " Type: " + param.
ParameterType);
    }
}
```

To get fields

```
FieldInfo[] fieldInfos = type.GetFields();
foreach (var field in fieldInfos)
{
    Console.WriteLine("Field: " + field.Name + " Type: " + field.FieldType);
}
```

To get properties

```
PropertyInfo[] propertyInfos = type.GetProperties();
foreach (var property in propertyInfos)
```

```
{
    Console.WriteLine("Property: " + property.Name + " Type: " + property.P
ropertyType);
}
```

To get methods

```
MethodInfo[] methodInfos = type.GetMethods();
foreach (var method in methodInfos)
{
    Console.WriteLine("Method: " + method.Name + " Return Type: " + meth
od.ReturnType);
    ParameterInfo[] parameters = method.GetParameters();
    foreach (var param in parameters)
    {
        Console.WriteLine("  Parameter: " + param.Name + " Type: " + param.
ParameterType);
    }
}
```

To get information about current assembly

```
Assembly assembly = Assembly.GetExecutingAssembly();
Console.WriteLine("Assembly Full Name: " + assembly.FullName);
Type[] types = assembly.GetTypes();
foreach (var t in types)
{
    Console.WriteLine("Type: " + t.Name);
}
```

To creating instance using reflection

```
object obj = Activator.CreateInstance(type, new object[] { 10 });
```

To invoke method dynamically

```
MethodInfo sampleMethod = type.GetMethod("SampleMethod");
sampleMethod.Invoke(obj, null);
```

To invoke private method

```
MethodInfo privateMethod = type.GetMethod("PrivateMethod", BindingFlags.NonPublic | BindingFlags.Instance);
privateMethod.Invoke(obj, null);
```

To invoke generic method

```
MethodInfo genericMethod = type.GetMethod("GenericMethod");
MethodInfo genericMethodInt = genericMethod.MakeGenericMethod(typeof(int));
genericMethodInt.Invoke(obj, new object[] { 100 });
MethodInfo genericMethodString = genericMethod.MakeGenericMethod(typeof(string));
genericMethodString.Invoke(obj, new object[] { "Hello" });
```

To get value of private field

```
FieldInfo privateField = type.GetField("PrivateField", BindingFlags.NonPublic | BindingFlags.Instance);
Console.WriteLine("Private Field Value: " + privateField.GetValue(obj));
```

To create instance of generic class

```
object gc = Activator.CreateInstance(typeof(GenericClass<int>), null);
MethodInfo gcMethod = typeof(GenericClass<int>).GetMethod("fun");
gcMethod.Invoke(gc, new object[] { 500 });
```