CH : 04 EXCEPTION HADLING

#1 EXCEPTION

* Creating Custom Exception

- you can create your own Exception by creating class which inherits the ApplicationException class
- Syntax :

```
public class oddNumException : Exception
{
    public oddNumException();
    public oddNumException(string msg)        // Constructor
    public string message {                   // message property overriden
        get                                   // readonly property
        {
            return "your error msg";
        }
    }
}
```

- you can use try, catch & finally block for handling Exception

* Inner Exception

- The inner Exception is a property in Exception class. its a read only property having only get accessor
- public Exception InnerException {get;} : this inner Exception property gets the Exception instance that caused the current exception. It returns an object that describes the error that caused the current exception. it returns same value pass in the constructor or null
- is simple words its used for nested Exceptions loops. inner Exception returns the original Exception that caused the ~~current~~ current Exception.
- when their are nested try blocks, use innerException

eg:

```
try{
    try{                    }
    catch ()
    {
    }
}
catch(Exception obj)
{
    cwL("Exception msg & source {obj.Message} {obj.Source}");
    cwL("Exception stackTrack {obj.stackTrace}");
    cwL("inner Exception {obj.InnerException.Message}");
}
```

* Properties of Exception

1) Message : This property will store the reason why Exception has occured & display it . you can pass own Msg in constructor using new obj.Message

2) Source : This property will store the applications name from which the Exception has been raised. returns file or namespace name
obj.Source

3) HelpLink : This is used to provide a link to any file/URL to give help to the user to solve the Exception or why Exception is raised
obj.HelpLink

4) StackTrace : This is used to provide more information about the Exception, such as the reason for the Exception, what method and class the Exception occured in, what line number the Exception occured at, which helps ~~the~~ us to resolve the issue
obj.StackTrace

NOTE

- Exceptions are not Run time Errors, Exceptions are classes that are responsible for Abnormal termination of the program when runtime Error occurs
- Run Time Error : errors that occur during program Execution are called run time Error
- Compile Time Error : Error that occur at compilation, CLR detects it & object of Exception class is thrown by CLR to stop the Execution of Pgm

```
eg: catch (Exception generic)        // This is called the generic catch block
    {                                   because of Exception the parent class which
                                        handles all Exceptions
    }

eg: public class oddNumException : Exception    // Custom Exception
    {
        public override string Message
        {
            get {                   // override Message property
                return "Divisor Cannot be odd";
            }
        }
        public override string HelpLink  // override HelpLink
        {
            get {
                return "Get More info on: http://Link/learnda.com";
            }
        }
    }
```

* Nested try-catch

1) Inner catch block handles Exception first :
   if an Exception occurs in the inner try block & is handled there, the outer try-catch remains unaffected
2) Inner catch block may rethrow Exception
   if inner catch block cannot fully handle the Exception, it can rethrow it using just(throw), which allows outer catch block to handle it
3) outer catch will catch the inner Exception & handle it

* Exception Propogation

- Exception propogation refers to how Exceptions move up the call stack when they are not handled in the current method
- when Exception is thrown in Method :
  1) runtime looks for catch block in the current method
  2) if no catch block is found, the Exception is progated to the calling method
  3) This process continues until an appropriate catch block is found
  4) if the Exception is not catch (not handled) at any level it crashes the program

eg :

```
namespace ExceptionHandling
{
    public class divideByOdd : Exception
    {
        public divideByodd () { }
        public divideBydd (string msg = "Default msg") : base (msg)   // if you don't pass the parameter then it will take default value
        {
        }
        public overrid string Message // if you don't commit this then it will get always get printed
        {
            get {
                return "it will printed always
                        Even if you pass your own msg"
            }
        }
    }
```

```
class Program ()
{
    static void Main ()
    {
        try{
            cwL ("Main method started");
        }
        catch(Exception obj) // obj from method2 will come here
        {
            cwL ("Exception caught in Main : "+ obj.Message );
        }
        cwL ("Main method ended");
    }
    static void Method1 ()
    {
        try{
            cwL ("Method 1 streated");
            Method2 ();
        }
        catch (divideByOdd obj)
        {
            cwL ("Exception Caught Method1 :"+ obj.Message );
            throw ; // it will throw divideByodd obj to Main
            cwL ("not Execute"); // this line will not Execute
        }
        cwL ("Kela kele not Execute "); // this will also not Execute

    }
    Static void Method2 ()
    {
        try{
        cwL ("method2 started ");
        int res = 10/3 ; // causes divideByodd ;
        throw new divideByodd ("divide by 3"); // there is not try
                                               Catch here so
                                               it will prop propogate
                                               to Method 1
        cwL ("will not get Execute ");
    }
}
```

- Method2 throws an Exception
- it propogates to method1
- method1 throws the Exception and again it propogates to Main becoz there is not catch
- main handles the Exception becoz of which there is no Program crash

this msg will get printed if you print obj.Message

// you cannot use only Catch block without try Block

- catch must always follow a try
- you cannot have try without catch, but you need finally in that case
- finally is optional with try - catch
- you cannot have finally block without try block
- you cannot have try block alone
- you cannot have catch block without try block

(112) LOGGING EXCEPTION

* Logging Exception

- Logging Exception means recording error details (like the error msg, stack trace and timestamp in a structured way into a file / dB / Lg4Net
- We log Exception ~~because~~ for :
  1) Debugging : Helps developers understand what went wrong
  2) Tracking Issues : Allows monitoring of recurring errors
  3) User Experience : Instead of showing techical errors to user, log them silently
  4) Security : Prevent Exposing sensitive information

1) eg : Exception logging to file :

```
class program
{
    static void Main()
    {
        try{
            Cwl ("Program Started");
            int res = 10 / 3 ;
        }
        catch (odddiwsionException ex)
        {
            LogError(ex) // LogError(ex) is a fun
            Cwl ("error occured . see in log file");
        }
    }
    static void LogError (Exception ex)
    {
        string filePath = "error_log.txt";
        string logMessage = $"{DateTime.Now} : {ex.GetType()} -
                            {ex.Message} {Environment.NewLine}";

        File.AppendAllText (filePath , logMessage); // Appends (put) error details to file
    }
}
```

date and time

type of Error

// if Exception is occur , it is caught in catch ( ) block .
// then Catch ( ) block calls the logError ( ) fun which takes the Error obj and writes the error details to error_log.txt file
// then the program continues instead of crashing

* throw & throw ex

- Using throw & throw ex you can rethrow an exception inside a catch block
- only the difference is one preserve the stack trace & other doesn't

* **throw (Preserves Stack Trace)** : keeps the original stack strace as it is
  - Allows debugging to pinpoint where the original error occured
  - Recommended way to rethrow an Exception
  - Preserves stack trace means shows the origin of the Exception
  - Best for debugging, helps to identify the real issue

```
eg: catch (Exception ex)
    {
        cwl("Handling Exception in Method1");
        throw // now this Exception is throw to
              the function call
    }
```

* **throw obj (Resets Stack Trace)** : throw ex or even you can use any variable name in place of ex
  - throw obj will Reset the stack strace & throws a new Error from that line
  - it will loose information about where the Exception originally occured
  - Makes debugging harder becoz the Exception appears to have started from rethrowing method
  - eg:

```
catch(Exception ex)
{
    cwl("Handling Exception");
    throw ex; // new Exception is throw to the
              method calling this method
              & it will reset the stack strace
}
```

* Stack Trace

- Stack Trace is a report that includes of the sequence of method calls that leds to an Exception
- it helps to identinfy why & where Exactly on which line an error has occured
- Stack Trace pinpoints the Exact Error location & keeps a method call sequence
- Working: Records method calls in reverse order (most recent call to oldest call)
  - shows file name & line no on which Error occurs
- Stack Trace is an property which has only get access, & it shows all details about the Exception, including at which Line of code the Error has occur
- eg:

```
catch (Exception ex)
{
    Cwl ("Exception caught");
    Cwl ("Stack Trace is" + ex.StackTrace) // prints the Stack Trace
}

// output: Exception caught in file_name.
           at line no. & what Exception it is
```

* using when with catch

- Use "when" for applying filteers for Exception & if that condition matches then only the Exception will Execute

eg:

```
try{
    throw new InvalidOperationException ("specific error ocured");
}
catch (InvalidoperationException ex) when (ex.Message.Contains ("specific"))=
{
    Cwl ("Handling specific Error");
}
catch (Exception ex)
{
    Cwl ("Handling genral Error");
}
```

↳ if the exception obj contains "specific" in its msg, then it will be handled by first catch block

- Otherwise the second catch block will handle it

- you can give Condition in when & if that condition is true then only that Exception will go in that catch block

* Using log4net (logging library)

- log4net is a logging library for .Net, not a framework it is part of Apache log4j family
- Install log4net : Install Package log4net
- Configure log4net in app.config file
- dont use log4net use Serilog

* Using Serilog (logging library)

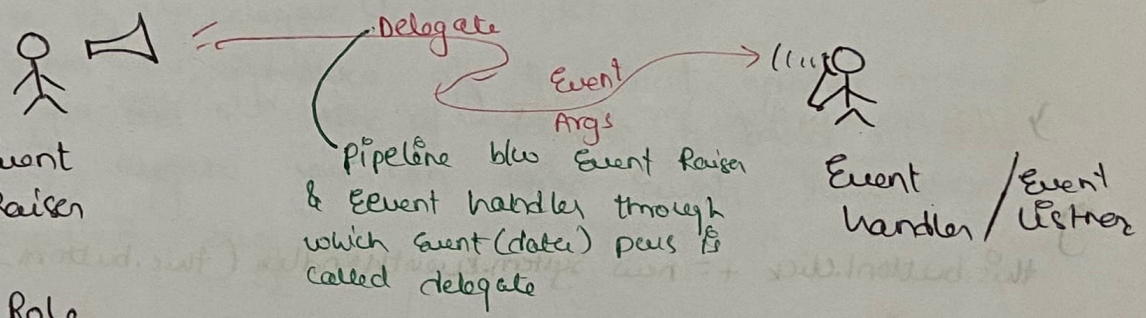
- Serilog is structured logging library
- you can use serilog with Console, Desktop, web Apps (MVC)
- install serilog using NuGet
- using serilog you can log to console as well as to files
- using serilog you can log (information about Exception) into cloud like CloudWatch of AWS & Azure

CH 06: EVENTS

#1 ROLES

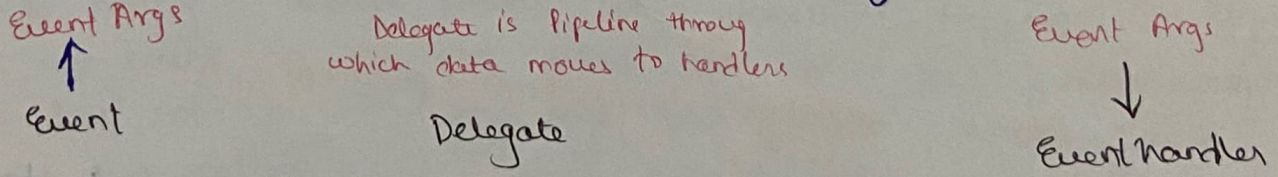
* Events Role

- any movement which happens on our page like, mouse hover, click, btn Press etc are all Events
- Events are notifications, & Events provide a way to trigger notification
- button element has many Events like: click event, mouse over, mouse out, & many more
- eg: in Win forms when you drag & drop a btn in UI & when you double click on the button element, behind the sence it generates a click event handler
- Events Args: This is the Event data which gets route from Event Raiser to Event handler
  - Event passes the Events Arguments (Events data)



* Delegates Role

- Delegate is a function pointer, when you want to pass function as parameter to other functions you can use delegates
- MultiCastDelegate: it is a class that tracks everyone who is listning to the Event
- Invocation list: it is list of the listners of the Events



- Eventhandler (): it is a function & delegate is pointing to the Event handler, when delegate is called, the Event handler fun is going to get Excute, so we called delegate as Event pointer or function pointer
- Event handler is attach to ~~deg~~ delegate

* Event handler Role

- when Event Raises (btn click) Raises data, delegates transfer data to some fun (Event handler) & then the Event handler process the data & does some work like Updating UI in Response to that Event
- Event handler is a Method that is Responsible for receiving & processing the data from delegeet
- Event handler has 2 parameters 1) Sender
  2) EventArgs object

1) Sender Parameter : it is the Sender who sent the data, it would be an object

2) EventArgs : object which includes the Event data.
   . it is a object which has properties, & these properties have info about the Event data

- eg:

```
private void button1_click (object sender, EventArgs e)
{

} // Event handler for a btn for click Event
```

eg:

```
this.button1.click += new System.EventHandler (this.button1_click);
```

btn click Event ↓ delegate ↓ callback fun

- button click Event is Wrapped by delegate & the delegate is pointing to the button1_click() call back fun

* Delegate

- Syntax

```
public delegate returntype delegateName (parameters);
```

- Event handler : it is a function which has logic to handle the Event & have the Exact same signature as of delegate by which Event is defined
- Event : ~~it is a type of delegate~~ Event has type as delegate & no body or logic
- delegate : it acts as pipeline blw Event & Event handler

#12 EVENTS

* Events

- Event is a Mechanism to Notify or that Enables a class or object to Notify a subscriber when something significant happens
- it is a type of delegate & follows the publisher-subscriber pattern
- Events Enable a class or object to notify other class or object when something happens (action occurs)
- class that sends the Event (Raiser) is called publisher & the class that Receives (or handles) the Event is called Subscriber
  ∴ publisher = Event Raiser
  subscriber = Event handler

  their can be multiple subscriber (handler) to a single Event (publisher)
- Event is a keyword used to define a Event
- Only the class that declare ~~Event~~ an Event can invoke it (ensuring encapsulation
- Other class can ~~sub~~ suscribe or unSubscribe to an event using += & -=
- Events can be typically created inside a class or struct. & Events are based on delegates, signature of Event has to match the signature of delegate
- A delegate defines the signature of the Event handler method
- first always you need to define a delegate & based on that singuneture of delegate you can define Events inside class
- type of Event is delegate
- syntax:

```
public delegate void WorkPerformHandler (int hrs, double sal);
                    //this is a delegate which will be type of the Event
public event WorkPerformHandler WorkEvent;
             ↓                  ↓
          delegate           EventName
```

- Event is not a function, Event is a mechanism that allows a class to notify another classes when something happens
  Events are based on delegates which are nothing but function pointer
- Events does not contain any logic like function, instead Events acts as a notifier and when event is triggered (or raised) it calls the function (called Event handler) that are subscribe to it
- Event is like a trigger, which calls function
- Event handlers are functions that run when Event occurs
- syntax of Event:

```
public event delegate-Name Event_Name;
```

- publisher : publisher (not a keyword) is a class that declares & raises an Event
  - it defines an Event using delegate
  - it triggers (raises) the Event when something happens

eg:
```
using system
namespace Events
{ public delegate void Notifydelegate (string msg);
  public class Publisher
  {                                   delegate        this is Event
                                        ↑               ↑
      public event Notifydelegate OnPublish ; //Event
      public void PublishNews (string news); //fun which invoke Event onPublish
      {
          Cwl ("Publishing News ....");
          OnPublish?.Invoke (news) ; //This "?" checks if Event is having any fun in invocation list or not
                                     //checking for null
          ← OnPublish.Invoke(news) or
          this will also work
          OnPublish (news) ; // invoke the Event, Raises Event & calls all subscriber attach to this Event
      }
  }// Think of News channel : it broadcasts all updates to all subscribed viewers
  // Publisher class declares the Event onPublish.
  // PublishNews raises the event & notifies subscriber. PublisNews is a function which invokes the Event
```

- Subscriber : Subscriber is a class that listens for an Event & handles it
  - it registers (subscribes) to event
  - When the Event happens or occurs (raises), its Event handlers (methods which are attach to that Event called subscribers) are Executed.

eg: Think of it as Viewers watching News : They receive updates when the news channel broadcasts news

```
public class Subscriber
{
  //Event handler (fun) that matches delegate signature
   public void ReceiveNews (string msg)
   {
      cwl ($" Subscriber raised news : {msg} ");
   }
}
```

//here Subscriber class have a method ReceiveNews that listens to onPublish Event

Connecting Publisher & Subscriber

- you need to hook up (Binding up) the Event in Main Method
- This means you need to bind subscriber to the publisher. This allows subscriber (Event handler function) to receive notification when the publisher (Event raise) raises the Event
- you need to bind up in main program or in any method you can bind by creating instance of the publisher & subscriber class wher publisher class includes the Event & Event raiser function & subscriber class includes the Event handler (Method) that matches the delegate signature

```
class program
{
    static void Main()
    {
        // Creating obj of Publisher & Subscriber for ~~ting~~ binding
        Publisher newsChanelObj = new Publisher();
        Subscriber obj2 = new Subscriber();

        ~~NewsChannelObj.PublishNews();~~

        // Binding Subscriber to Publisher
        newsChannelObj.OnPublish += obj2.ReceiveNews;
        // OR
        newsChannelObj.OnPublish += new NotifyDelegate(obj2.ReceiveNews);
        // This will also add the fun in invocation list of Event

        newsChannelObj.PublishNews("C# 13 Released");
        // calling of PublishNews fun which will invoke the Event
        // this is normal fun which invokes the Event from Publisher class

        newsChannel.OnPublish -= obj2.ReceiveNews
        // UnSubscribe (optional)
    }
}
// namespace closes
```

adding fun in invocation list of Event (Subscriber)

// onPublish is Event name

// Subscribing to Publisher

(Publisher)

Event of Publisher class

fun of Subscriber class

Output: Publishing News....
Subscriber Received News : C# 13 Release

Steps:
1) Publisher (newsChannel) declare an Event OnPublish
2) Subscriber (obj2) subscribes to Event using += (binding)
3) Publisher raises Event using OnPublish?.Invoke(), which
   - calls the RecieveNews method to Raise Event
4) Subscriber Executes its method ReceiveNews & prints the news

#14 Custom Event Creation

* Events Creation / EventHandler<T> & EventArgs

- An Event on its own does not really do anything. You must have a pipeline or delegate which routes the data from point A to point B that is from Event Raiser to Event handler. & delegate is going to do this
- Events are really wrappers around delegate
- Events are special kind of delegates. ~~Events~~ Events wrap a delegate means that an Event internally uses a delegate to store refrences of methods (subscribers)
- Events allow Encapsulation becoz by delegates you can directly invoke the methods by any class
- In Events, Only the class that declares the Event can raise (invoke) it.
- Other classes can can only subscribe & Unsubscribe it
- Events are special delegate that allow object to communicate
- EventHandler : it is a predefined delegate used for Event handling.
  - We also have EventHandler<T> Generic type delegate
  - EventHandler is a delegate type that represents a method that will handle an Event
  - EventHandler is defined in system namespace as

    public delegate void EventHandler (object sender, EventArgs e);
  - you can use EventHandler delegate to define & raise Events without creating a custom delegate

* EventArgs (class)

- EventArgs is a built in class that act as base class for all Event data class. it provides a way to pass Event data (Arguments)
- used as base class for custom Event argument classes
- Syntax:

```
public class EventArgs
{
    public static readonly EventArgs Empty;
    protected EventArgs() { }
}
```

↳ //protected Constructor meaning it cannot be instaintiated directly only can be instanciated in child class

- when you want to pass custom parameters to delegate then you need to create custom EventArgs class by inheriting it & then create fields to take parameters & pass this new custom class as <T> in the EventHandler<T> delegate

eg:
```
public class NewsEventArgs : EventArgs
{
    public string News {get;}
    public NewsEventArgs (string news)
    {
        News = news
    }
} // pass NewsEventArgs class as <T> in EventHandler<T> delegate
  while describing an Event
  // EventHandler<T> will be type for event directly without
     making custom delegate
public class ~~Subscriber~~ Publisher
{
  // Event of type EventHander<T> which is delegate that takes class as T
  public event EventHandler<NewsEventArgs> NewsPublished;
  public void PublishNews (string news) // fun to invoke Event
  {
      Cool ("Publishing ....");
      NewsPublished?.Invoked(this, new NewsEventArgs(news));
  }
}
```

Null condition Operator

// This ? operator checks if NewsPublished Event is not null before invoking it, prevents runtime Error

// if there are no subscriber attach to Publisher, then NewsPublished Event is Null, so calling Invoke() would normally cause a NullRefrenceException

// this (first parameter) refers to the publisher class object beoz NewsPublished event has delegate which has EventArgs type & this takes parameters as object sender, EventArgs e so send the sender obj & other event parameters beoz EventHandler delegate takes that parameters

// new NewsEventArgs (news) : creates event data to pass to the subscribers

NOTE

- Events uses delegates in backend (internally) so while calling or raising the Event you should pass the delegates parameters in the Event while raising it. The function handler will also have these parameters & you need to pass that also
- you can Raise Events in 3 ways
  1) Event_name (parameter)
  2) Event_name.Invoke (parameters)
  3) Event_name?.Invoke (parameters)

* EventHandler<T> (delegate)

- EventHandeler<T> is a delegate (Generic delegate), used to define Events with custom event data.
- Instead of defining a custom delegate, you can use this EventHandler<T> inbuit & pass <T> as a EventArgs class only
- no need to decalare delegate, directly you can define Event using EventHandler<T> as its type & pass inherited EventArgs class if custom parameters ~~you~~ you want to pass
- Syntax: this delegate takes (object sender, EventArgs e) as parameters

```
public event EventHandler<T> EventName;
```

- if you don't want to pass any parameters for the Event function (subscriber) then you can use "EventHandler" delegate

eg:

```
using system
namespace EventHandler
{
    //define Event Arguments to pass
    public class NewsEventArgs : EventArgs   //class to pass as parameter to EventHandler delegate & this delegate requires parameter as sender & Event Arguments
    {
        public string News {get;}
        public NewsEventArgs (string news)
        {
            News = news;
        }
    }

    public class NewsPublisher
    {
        //Creating an Event of EventHandler<T> delegate
        public event EventHandeler<NewsEventArgs> NewsPublishedEvent;

        public void Publish (string news) //fun to invoke Event
        {
            Cwl ("Publishing News...");
            NewsPublishedEvent?.Invoke (this, new NewsEventArgs (news));
            //Event Raise
        }
    } //when an Event is Raise, it should pass object sender & instance of EventArgs class if you are using EventHandler<T> delegate
```

"this" refers to the sender who raised the event

'this' refers to the current instance of NewsPublisher class that calls Publish method

//(object sender & EventArgs e) which the delegate (EventHandler) needs as parameters are pass through the fun which is invoking the event

//'this' parameters are passed in the Event as arguments so that the delegate will take them

//'this' means the object of NewsPublisher class

// Subscribes class (fun which handleds Event)

```
public class NewsSubscriber
{
    public void OnNewsReceived (object sender, NewsEventArgs e)
    {
        Cwl ($" News subscriber received {e.News}");
    }
}
```

→ fun should have same signature as of EventHandler delegate has

// this function should have same signature as that of the delegate EventHandler<T> has
// this fun will bind the delegate, Event & this fun itself which is noth but eventhandler function
// we will bind subscriber to publisher using their object

```
class Program
{
    static void Main ()
    {
        NewsPublisher publisher = new NewsPublisher();
        NewsSubscriber subscriber = new NewsSubscriber();

        // Binding (subscribing)
        publisher.NewsPublished += subscriber.OnNewsRecieved;

        publisher.Publish ("C# 13 released");

        publisher.NewsPublished -= subscriber.OnNewsReceived;
    }
}
```

//Subscriber = handling Event function
//publisher = Event
//we will bind subscriber with publisher

// bind Event handling function (subscriber) which will handel Event (onNewsRecieved) with the event which is present in publisher class
// Event += function which handles that Event
// the function has same signature as of delegate which handles the Event
// this binding proces is already (Predefined) defined

→ // Trigger Event by calling the fun which invokes the Event
// usually Event are trigger by btn pressing or mouse hover

publisher.NewsPublished → Event
subscriber.OnNewsReceived → Event handler function

// this process is called Unsubscribing or Unbinding

} → // closing namespace

NOTE

- EventHandler<T> delegate takes (object sender, & EventArgs e) as parameter
- So the fun, which is binding Event should also follow same signature as of EventHandler delegate

#13 Subscribing & Unsubscribing / Handling Event

* Subscribing

- it is the process of ~~bing~~ binding the Event handler (function) with the Event
- Syntax:

```
publisher.EventName += eventHandlerMethod ;
```

- this is also called "listner is added to invocational list" of delegate

* Unsubscribing

- UnBinding the Method from Event
- Syntax:

```
publisher.EventName -= eventHandlerMethod ;
```

* Custom Event with Add/Remove

- if you want more control on how listner (subscribers/Eventhandling function) is added to the invocation list (added to delegate), then you need to use add & remove accessors
- Although event cannot have body, but you can override how subscribtion (+=) & unsubscription works using custom accessors

eg:

```
public delegate void WorkPerformedDelegate (int hrs, WorkType wT);
public class publisher
{
    private WorkPerformedDelegate _WorkPerformed ; // private delegate field that stores event handlers manually
    public event WorkPerformedDelegate OnActionWork //actual event which uses degate to store fun address or event handlers (methods)
    {
        add
        {
            CWL ("subscriber added");
            _WorkPerformed += value;
        }
        remove
        {
            CWL ("subscriber removed");
            _WorkPerformed -= value;
        }
    }
    public void PerformAction()
    {
        CWL ("Performing action");
        _WorkPerformed?.Invoke (this, EventArgs.Empty);
    }
}
```

this field is just a delegate refrence not an event

this is noth but a delegate using this delegate we are going to add & remove listners (methods) from the invocation list, which is going to be called when the event is raised

// _WorkPerformed acts as delegate storage. // you manually store the handler methods

//actual event is "OnActionWork" which uses delegate "_WorkPerformed" internally to store functions. //internally "_WorkPerformed" delegate will store all function which we add in invocation list using Event. delegate are function pointers

Event will be invoke

Imp

1) delegate : it will be "type" for event. delegate stores the fun address
- Simple words delegate is function pointer & it should Match with the signature of event handling function
- EventHandler <T> is delegate & <T> takes EventArgs class as parameter

2) Event : it has delegate as its type. Event is not a function
- Events can be decalare only inside class or struct
- you should attach the functions to the Event by using the publisher (class which contains event) obj
- you need to bind all the methods (~~from~~ any which belog to any class) with the Event, so that when Event is invoked the functions get Execute
- The Event should pass the parameters of delegate, when Event is being invoke
- The function which are attach to Event should have same signature as of delegate

Invoking event is same as you invoke delegate eg: eventName (parameter) or eventName.Invoke (parameter)

3) Eventhandlers : These are the functions which handles the Event.
- you need to attach these functions to Events
- this functions should have same signature as of delegate. if EventHandler <T> is delegate then also fun should have same signature
- if you use EventHandler <T> delegate, then it takes (object sender, EventArgs e) as parameter, so while invoking the Event you should pass the sender as the class object (this) & object of class EventArgs.
- if you have custom data, which you want to send to EventHandler <T> delegate then you should make class of EventArgs (inheritance) & take the data using constructor & when you are invoking the fun which invokes Event handler that time in Event pass the data using object of EventArgs constructor
- The Eventhandling function should Match the delegate exact signature, Even the parameters must be same like (object sender, EventArgs e)
- Behind the Event we have delegate, so while calling Event we need to pass the delegate parameters along with event calling.
- Call event in the same way as we call function "eventName (parameter)".

Event has delegate behind the since, so when you bind event with fun you have bind delegate with fun to invocation list using "event += fun" functions. Adding

CH: 07 THREADS

#1) INTRO

* THREADS

- Threads are fundamental units of Execution within a process.
- process: any code or application, which is running is a process.
  eg: opening web browser; running hello world program.
- Thread: Light weight process is called thread
- threads allows multiple operations run concurrently within the same application. A process can contain multiple threads
- process: An instance of running application. it can contain multiple threads
- THREAD: it is a class defined under system.Threading namespace which allows us to create thread & do operations on it
- ThreadStart: its a delegate, the thread constructor takes in a delegate, which is threadstart, its void & does not take any parameters.

- Syntax:

```
Thread T1 = new Thread (function_name) // directly pass the fun without parameters. binding of delegate implicitly takes place
        OR
Thread T1 = new Thread (new ThreadStart (fun_name))
                                  ↳ // you can bind the function with Thread stard delegate
T1.Start()
Thread.Sleep()
    ↳ // static Method which makes the thread sleep for that many ms (millisecond).
```

- By default threads are Unnamed. you can give Names to the Threads & this will help you in debugging
- Managing threads effectively is essential to ensure Efficient use of resources & avoid deadlocks, race conditions etc
- Some Imp functions of thread are
  1) T1.Start() : for starting the Execution of thread
  2) Thread.Sleep(5000) : for pausing thread for 5 sec. it is static Method
  3) T1.Name : for giving Name to T1 thread
  4) T1.IsAlive :
  5) T1.Join()
  6) T1.Abort()

* Passing Parameters

- threads use delegate internally to bind with the function.
- when you create new thread then, in the constructor of that thread you need to pass the function name, which you want to execute when you do T1.start().
- the constructor of the thread takes delegate as parameter & when you create the thread.
- the delegate is ThreadStart or ParameterizedThreadStart, which you can bind to function & send the obj of this delegate into the constructor
- you can send the parameters of the functions, in the T1.start() function of thread
- ThreadStart delegate cannot accept parameters, to pass parameters to a thread use ParameterizedThreadStart delegate or lambda Expression. this ParameterizedThreadStart delegate accepts object as parameter, so create a function of same signature

- Syntax:

```
public delegate void ThreadStart();
public delegate void ParameterizedThreadStart(object obj);
```

eg: using parameterizedThreadStart()

```
using System.Threading
namespace Eg
{
    class Program
    {
        static void Main()
        {
            Thread T1 = new Thread(new ParameterizedThreadStart(Test));
            T1.Start(5);
            or
            ParameterizedThreadStart obj = new ParameterizedThreadStart();
            obj += Test; // Test() is a fun which prints no. from 1 to n &
                         you need to pass n which is int
            Thread T1 = new Thread(obj);
            T1.Start(5);
        }
        static void Test(int n)
        {
            for(int i=0; i<n; i++)
                Console.WriteLine(i);
        }
    }
}
```

Eg 2: using Lambda Expression

- using lambda expression you don't need to use Parameterized ThreadStart delegate & pass function parameters as object. you can directly call the function & send parameters
- you can bind the lambda Expression with the thread, & when the thread start, it will Execute the lambda Expression.
- In lambda Expression call the function & send parameters

```
class Program
{
    static void Test (int id, string name)
    {
        Cwl ("id is "+ id + " name "+ name);
    }
    static void Main()
    {
        Thread T1 = new Thread ( () =>
        {   // lambda Expression body
            Test (101, "vishu");
        });
        T1.Name = "Thread one";
        T1.Start ();
        T1.Join ();
    }
}
```

This is one function without name which we are passing in constructor of thread

// Join () fun will Ensure that the Main thread waits until ~~both~~ thread T1 finishes the Execution

Thread (fun1) → this fun is Lambda Expression in which we are calling other Test() fun

//so instead of lambda fun u can also write another function "fun1" & in that call the Test fun & pass this fun1 in constructor of thread.

// lambda function is parameter less & hence internally this lambda function is binded with the ThreadStart delgate // here we have bind lambda Expression with thread, so as you start the thread lambda Expression will start Executing

```
fun1()
{
    Test (1, "she")
}
```

OR

```
static void fun1 ()
{
    Test (101, "vishu"); //so calling fun1, will call test ()
}                        & send required parameters also
static void Main ()
{
    Thread T1 = new Thread (fun1) // both codes are same
    T1.Start ();
}
```

// fun1 is a function, in which it calls Test()

~~* Thread Life Cycle~~

21) Thread Life Cycle

* States of thread

State	Description
1) Unstarted	Thread is created, but not started.
2) Running	Thread is started & actively Executing Code
3) Waiting	Thread is blocked by another thread, sleeping, waiting for other resource which is in Lock()
4) Stopped	Thread has Completed Execution or has been terminated.
5) Aborted	Thread was forcefully stopped or terminated using Abort() method

* Lifecycle Method

1) T1.Start() : it is a Instance Method in thread class
   - it Begins the Execution of the thread
   - Moves the thread T1 from Unstarted to running state
2) T1.Join() : Join() method Makes the calling thread wait until the thread it is calling (T1) gets Complete
   - Main thread waits for the completion of T1 thread
3) Thread.Sleep(5000) : it is a Static Method, which pauses the Execution of current thread for 5 sec.
   - it takes time in milli seconds
4) T1.Abort() : This Method attempts to stop a ~~thread~~ thread immidiately by throwing threadAbortException
   - Abort() will forcefully teeminate the thread

* Thread Priority : Windows use preemptive, priority based shaduler

- "Priority" is a Instance property of thread class
- type of "Priority" is a Enum which is named as as "ThreadPriority" & priority is a property which sets the priority of the thread which you select from the Enum
- you can set priority as:

```
T1.Priority = ThreadPriority.Lowest
T1.Priority = ThreadPriority.Highest
```

- Normal is the default property set as priority for all threads

#13) Synchronization In Thread

- Synchronization is a proces in which we Ensure that multiple threads can acces shared resources without causing data inconsistency or Race Condition.

1) Lock ()

- lock keyword Ensures that only one thread accesses the lock block code at a time
- you have to pass (this) in the lock, so that it will acquire lock on when the current Executing thread will go inside lock Block

eg:

```
class Program
{
    static int counter=0;
    static void Increment ()
    {
        for (int i=0 ; i<50 ; i++)
        {
            Lock (this) // lock will Ensure that only one thread at
                        a time can access & increase counter
            {
                counter++;
            }
        }
    }
    static void Main()
    {
        Thread T1 = new Thread (Increment);
        T1.Start ();
        T1.Join ();
        Cwl ("final Counter value" + {counter}");
    }
}
```

- Lock keyword does not support inter proces synchronization, it only works for thread synchronization within the same proces
- lock does not throw Exception, but code inside lock can throw

2) Mutex (Mutual Exclusion)

- Mutex is a class in System.Threading namespace
- Syntax:

```
public sealed class Mutex : WaitHandle
{
    // inbuilt class
}
```

- using Mutex you can synchronize threads across processes.
- you need to create object of Mutex class & use the function to acquire lock on the code which is being share b/w diff threads
- mutexObj.WaitOne() : This functions Acquires lock when some thread goes in Critical Section (CS)
  , you can also send timeout, which allows you to specify a timeout period to wait for the lock
- mutexObj.ReleaseMutex() it will Release the Mutex lock that was acquired by the same thread using WaitOne() fun
- mutex support ~~Enter~~ inter proces • synchronization, meaning it can be shared b/w multiple proces
- mutex throws Exceptions, if thread acquire mutex & terminates without releasing it properly ∴ write Mutex code in try Catch & finally block
- eg:

```
class Program
{
    private static Mutex objMutex = new Mutex();
    static void AccesResource()
    {
        cwl ($"{Thread.CurrentThread.Name} waiting to Enter ");
        objmutex.WaitOne() //acquires lock
        try{
            cwl ($"{Thread.CurrentThreadName} has Enter");
            Thread.Sleep(2000);
            cwl ($"{ThreadCurrentThread.Name} is leaving");
        }
        catch (Exception e)
        {
            cwl ("Exception Caught");
        }
        finally
        {
            objmutex.RelaseMutex(); // Releases lock
        }
    }
```

```
static void Main ()
{
    Thread T1 = new Thread (AccessResource)
    T1.name = "T1 Thread";
    T1.Start();

    Thread T2 = new Thread (AccessResource)
    T2.Name = "T2 Thread";
    T2.Start();

    T1.Join() , T2.Join();

}
```

3) Semaphore

- Semaphore used for thread synchronization, it allows limited no. of threads to access a resource simultaneously
- Semaphore is a class, it allows limited no of threads to Enter Critical Section simultaneously.
- Unlike lock, & Mutex which allows only one thread in CS, But Semaphore allows multiple threads entry in CS.
- semaphoreObj.WaitOne() : this function same as Mutex functions, acquire lock on CS by only allowing max no. allowed to enter of thread in CS
- semaphoreObj.Release() : this will release the lock
- semaphore also throws Exception, so use try catch block for the code of Critical Section
- when you create object of semaphore, you need to pass parameter in its Constructor.

Syntax :

```
public static Semaphore obj = new Semaphore (int initialCount,
                                             int maxCount);
```

- initialCount : it is basically the no. threads that can Enter initially in Semaphor
  .it defines how many Threads are allowed to Enter CS when semaphor is created
- maxCount : Max no. of Threads that can allow Entry simultaneously

eg: using System.Threading

```
{
    class Program
    {
        public static Semaphore obj = new Semaphore(2, 2);
        Static void AccessResource()
        {
            Cwl ($"{Thread.Name} is waiting");
            obj.WaitOne // Acquire Lock
            try{
                Cwl ($"{Thread.Name} is inside");
                Thread.Sleep(5000);
                Cwl ($"Going out");
            }
            finally{
                obj.Release(); // Releasing Lock
            }
        }
        static void Main()
        {
            for(int i=1; i<=5; i++)
            {
                Thread T = new Thread(AccessResource);
                T.Name = $"Thread {i}";
                T.Start();
            }
        }
```

max 2 thread can enter

#4 Async & Await

* Synchronous Method

- Synchronous Method Executes tasks sequentially. it blocks calling thread until the tasks is completed
- eg: if a task is time consumming, it will freeze the UI or delay further operations until it finishes
- Even if you implement multi-threading in your application, it won't make methods (thread calling methods) asynchronous.
- multithreading will make our application capable of performing tasks concurrently, but it does not make the methods asynchronous
- Synchronous: Meaning line of codes will Execute one after the other and will wait if upper line waits or sleeps the program.
- Even if you use multiple threads, the methods still will run in synchronous manner, but multiple methods will run parallely if you use multiple threads

eg:

```
class program
{
    static void ProcessData()
    {
        for (int i=1; i<5; i++)
        {
            Col ($"processing item {i}");
            Thread.Sleep(2000); // will make main thread sleep
        }
    }
    static void Main()
    {
        Col("starting data processing");
        ProcessData();
        Col("Data processing Completed after 2 sec");
    }
}
```

- if a Methods call DB query, then it will wait until that query is resolved & then only next line of code within that method will Execute

* Asynchronous Methods

- Asynchronous methods executes tasks (Line of code) concurrently (or parallelly). It doesn't block the calling thread (the thread which is calling that method)
- instead it allows other tasks to run, while waiting for the operation to complete
- this method won't wait for the response of that line, it will execute the next line.
- for Making Asynchronous Method you need to use Async & Await keywords.
- if a method calls DB query & you have use await keyword, then the function will just execute that query & move to next line of code without waiting for that query to respond.
- Use Case: you can use this when you are working with
  1) File I/O
  2) DB connections
  3) Web Requests etc
- multi-threading is different & Asyn-Await is different. Using multi-threading you can run multiple functions concurrently (parallely) & using Asyn Await you make a function not to wait for other function call to resolve, basically that function will not run Synchronously
- for Creating Asynchronous Methods you need to use Async, Await & Task keywords, all 3 are Essential for Achieving Asynchronous Methods.

* Task

- Task is a class used to represent a Asynchronous operation It is defined under System.Threading.Tasks namespace
- Defination:

  public class Task : IAsyncResult, IDisposable
- Task represents work that is being done Asynchronously
- Tasks provides:
  1) State Management: you can check if task is completed or not
  2) Continuation: you can specify what to do when a Task complete using .ContinueWith()
  3) Result handling: for operation that proceed a result Task<TResult> is used.
- Tasks is like a promise to do some work & return a result when work is done or promise is resolved

eg: using System.Threading.Tasks

```
class Program
{
    static void PrintNum()
    {
        for (int i=1; i<=5; i++)
        {
            Col(i);
            Task.Delay(500).Wait() // it will pause the current thread for 500 sec because task is blocking delay
        }
    }
    Static void Main()
    {
        Task task = new Task()
        task.Run(()=>
        {
            PrintNum();
        });
        task.Wait(); // wait for task to Complete
        Col("task Completed");
    }
}
```

* Async

- Async keyword decalares a method as Asynchronous Method, means it will not Execute Code line by line
- Allows you to use the Await key word inside Async Method
- This method returns A Task or Task<TResult> (if returns a value) or void for Event handlers
- you can use async without await keyword, but not good practice
- Main method also can be Asynchronous & return Task

* Await

- Await keyword can only be used in Async function
- Await doesn't ~~to~~ pause the calling thread, instead it allows other operations to run

Use Case: We use asyn & Await for

1) non Blocking UI: keeps UI response in application
2) Improved Performance

- await will wait & Pauses the Execution of that method until the awaited tasks get Complete

eg: ~~Synchronous Method without Tasks~~

```
using System.Threading.Tasks;
class Program
{
    public async static void Test()
    {
        CWL("some Task started");
        await Task.Delay(Timespan.FromSecond(10));
                        or
        //Thread.Sleep(10000);
        CWL("task Ended after 10 sec");
    }
    static void Main()
    {
        CWL("Main Method Started");
        Test(); // The Main thread will not wait
                   for this Method Test to get resolve
        CWL("Main Method Ended");
    }
}
```

await will wait until the Task Delay fun gets resolved

Output: Main method started
Some Task started
Main method Ended

Task Ended after 10 sec

- If you will not use asyn & await & then call the Test Method then the output will be:

Main method started
Some Task started
Task Ended after 10 sec
Main method Ended

* Threads V/s Async-Await

THREADS

- threads are basic unit of execution within a proces
- multiple thread ren parallely or simutanously
- Each thread uses system resources independently & high resource consumption
- Use Case :
  1) performing CPU bound tasks. like Calculations.
  2) heavy procesing
- Context switching is very high
- low Scalability
- Use Case :
  1) image procesing
  2) CPU bound tasks
  3) Calculations

Async - Await

- Async Await is single threaded & this single thread is never blocked when you Use Async await
- you can also use multi-threading
- Resource consumption is low beoz same thread use it
- Use Case :
  1) performing DB connections
  2) file handling.
  3) Network request
- Context switching is low
- High Scalable
- Use Case :
  1) DB operation
  2) network connection

* Thread Safety

- you can use Concurrent Collections, locks, Monitor class, Interlocked class for Thread safety.
- Thread safety Encures that shared data is accesed & modified safely when multiple threads are involve
- for Achiving thread safety, make sure that only one thread can Enter CS, or you can also make use of Immutable Objects, Interlocked classes. Use threadLocal<T>

* Interlocked class

- Interlocked is class defined under Threading namespace.
- it provides automatic operations for variables shared b/w multiple threads
- it Ensures thread safety by Ensuring to prevent race condition
- When multiple threads attempts to R/W same variable simultaneously it can cause race condition. So to prevent this we use Interlocked class, which Ensures that operations are atomic, meaning they complete entirely or not at all, without interference from other threads.

eg:
```
public class Counter
{
    private int _value;
    public int Value
    {
        get { return _value; }
        set { _value = value; }
    }
    public void Increment() // Thread Safe increment operation allows only one thread to do increment
    {
        Interlocked.Increment (ref _value);
    }
    public void Add(int amt) // allows only one thread to do modification of _value
    {
        Interlocked.Add (ref _value, amt);
    }
}
public class Program
{
    var count = new Counter();
    var threads = new Thread[10];
    for (int i=0; i<5; i++)
    {
        threads[i] = new Thread (() =>
        {
            for (int j=0; j<100; j++) // only one thread will be able to increament &
            {
                count.Increment()
                count.Add(-1);        Add the _value
            }
        });
    }
    Cwl (count.Value);
}
```

→ Static methods of Interlocked class

115 TASK class, Async & Await

* TASK

- A Task represents an Asynchronous operation that run in the background
- It helps to execute method asynchronously without blocking the main thread
- Task is a single unit of work
- Task वो है जो काम करता है
- कोई भी छोटा सा Block of code है उसे हम task बना सकते है
- Task is Used to return in the Asynchronous Methods
- Task is a class which is mainly used for returning type in Asyn function
- if you want to return int value & that fun is Async & you consume that return value before that return value function was completed, then you will get Error
- So you need to return a task, beoz you can apply await keyword on that task when you use that return value from Async function

```
class Program
{
    public static async int Test()
    {
        cwL ("Calculation Started");
        await Task.Delay (5000);
        cwL ("Completed");
        return 0;
    }
    static async Task Main()
    {
        cwL("Main started");
        int res = Test();
        cwL ("Doing Something");
        cwL ("Reslut is' + res");
    }
}
```

↳ it will be null beoz for 5 sec the function Test takes to return value in res

- when you mark fun as Async it automatically returns Task or Task<T> so that you can use await on the object of task

```
class Program
{
    public static async Task<int> Test()
    {
        cwL ("Calculation Started
        await Task.Delay (5000);
        cwL ("Completed");
        return 50;
    }
    static void Main()
    {
        cwL ("Main Started");
        Task<int> obj = Test();
        cwL ("Doing Someting");
        int res = await obj;
        cwL ("Reslelt is"+res);
    }
}
```

↳ this will wait until result comes in res

// so to use await on return value async functions return Task

TASK

- Task is single unit of work
- Task is something to be done

THREAD

- Thread is a basic unit of CPU utilization
- Thread are responsible for doing tasks

- Syntax of Async fun

```
public async Task fun_name
{
    // await Code
    // Connection to DB Code
    await DB.Connect()
}
```

* Use Case of Task

1) Non Blocking Execution: improvers performance in UI & web application by not blocking main thread
2) Better Exception Handling: Can propogate Exception property
3) Task works with await for non Blocking Execution
4) By default Task (Async & await) method do not create new thread to run method on new thread
   - The method executes on same thread, but yields control (pauses) when encountering 'await', allowing the thread to do other work

CH:08 MEMORY MANAGEMENT

(#1) INTRO

* Memory Management

- memory Management is the process of controlling & coordinating computer Memory
- memory Management includes:
  - Allocating memory for Applications.
  - Tracking memory usage
  - freeing (deAllocating) memory
- Goal is to ensure Efficient use of memory resources & prevent issue like memory leaks, fragmentation, crashes, stackoverflow.
- Memory Management is Essential for smoothy running application
- Memory leaks: occurs when a program fails to release memory that is no longer needed, causing that memory to remain allocated & unsuable for future operations.
- OverTime memoryleaks can Consume all available memory, resulting in application slowdowns, crashes, stackoverflow
- Memory leaks occur when objects are created & never released even though they are no longer needed. GC fails to free these objects becoz the objects are still referenced (rooted) somewhere in the application
- Eg: of Memory leaks: when Eventhandlers (functions) are not unsubscribed.
  - When you subscribe to an Event and never unsubscribe (-=) that Eventhandler, the subscriber object remains alive & cannot be collected by GC
  - when working with filehandlers, DB connectivities, network sockets & you dont call Dispose() method or dont use using Block then memory leak can happen

* Advantage of MM

1) performance optimization: Efficient MM Ensure that application runs faster
2) No Memory leaks: Memory leaks can cause application to consum all available memory & crash application.
3) Prevents fragmentation: fragments (small gaps), MM minimizes fragments in memory (RAM)
4) Application Stability: prevents App Crashes, unexpected behaviour caused by invalid memory access.

* Managed & UnManaged Code

- **Managed Code** : Managed Code refers to the code that runs under the control of a Run time Environment (CLR). CLR is Responsible for many tasks including garbage collection, MM, type safety, security Enforcement etc.
- **Characteristics of Managed Code :**
  1) **Memory Management** : automatically Memory is managed for managed code beoz of CLR
  2) **Security** : Managed Code is subject to the security policies of the Runtime Environment
     - so managed code are bound to follow security policies set by CLR
  3) **Exception handling** ; Exception handling in Managed Code is uniform & controlled or integrated to CLR
     - this includes type safety . etc.
  4) **Interpolability** : Managed code can Easily interact with othe managed Code, & access high level services like DB connection, file system operation, provided by CLR

  eg : C#, F#, VB.net
- C# code is managed code as it is compiled into IL code, which is then executed by CLR
- **DisAdvantages of Managed Code**
  1) **Performance Overhead** : GC & runtime (CLR) checks can introduce performance overhead, which might not be suitable for real time application.
  2) **Less Control** : Since CLR manages memory, Execution & Enforces Rules on type safety & security, Developer Gets less control over low level operations.

- **~~Un-Managed Code~~**
- Managed Code does not Run directy on hardware, beoz the Runtime (CLR) is in between
- ∴ Managed Code gives you less control on hardwere, beoz it follows Rules of CLR

- **Un-Managed Code** : refers to code that runs directly on the hardware or the OS without any Run time Environment in b/w like CLR.
- Un managed code is typically written in C++ or Assembly
- **Characterstics**
  1) Memory Management : programmer is Responcible of MM by creating Destructors
  2) No RunTime Support : Unmanaged Code operates directly with system resources and does not rely on runtime environment for GC, type safety, or security.
  3) Performance : is better
  4) System level operation : Unmanaged Code can directly interact with hardware, OS.
- **DisAdvantage**
  1) Memory leaks
  2) less security
  3) platform Dependence : UnManaged Code is often platform specific, as it requires direct interaction with the OS or hardware
- UnManaged Code gives you more control on OS, or hardware & is faster then Managed Code becoz it contains Layer of Runtime environment

* COM Marshal

- Component Object Model marshaling is a process of packaging & un-packaging of data, so that it can be transfered b/w unmanged & managed Code
- COM marshal : with this communication b/w managed & unManaged Code happens.
- COM marshal is act of converting data type b/w managed & unmanaged Code & Ensures safe communication b/w them
- Eg : Suppose you want to call .exe file of C++ program from C# then data types of C# must be converted into C++ data type for communication b/w them, this is done by marshaling
- Marshaling process : Converts types from C# to C++ & vice versa copies data from managed heap to Un-managed memory & vice versa when needed.
- COM Marshal : using this communication b/w managed & Un Managed Code happens

#2 GARBAGE COLLECTOR

* Garbage Collector

- The GC is responsible for automatic memory mangeing like allocation & deallocation of memory
- Objects (variables) which are no longer used are periodically removed by GC to prevent memory leaks & optimize memory useage
- GC is present in CLR. CLR is Responsible for calling GC & Controlling GC
- GC can only clean Managed Code (Managed Resources) bcoz these Resources (Managed Code) is under control of CLR
- GC does not clean Un-Managed Resources bcoz that is not in control of CLR
- GC is an Automatic memory management process but it only works on Managed Code (Managed Resources)
- Managed Resources : Resources on which CLR has control
- GC automatically allocates & deAllocates memory for managed objects
- GC uses a Generational Garbage Collection Allogorithm to optimize performance, reduce cost of memory clean up, reduce memory leaks, Increase performance of program & stability of Application
- GC is non deterministic, you can't predict when GC will Exactly Run
- We can force GC to clean using GC.Collect()
- Use GC.GetGeneration() to find out which generation an object belongs to
- GC uses stop the world mechanism, meaning it pauses the application thread during Garbage collection (clean up)
- if object has a distructor/finalizer then it will be called before object memory is reclaimed

* Working : Managed HEAP

- The Managed heaped (memory) is a block of memory allocated by CLR to store Refrence Variables/objects
- Managed heap is divided into 3 generations (0, 1, 2)
- memory is automatically allocated when objects are created & freed when they are no longer in use by GC

* Generations

- Generations are partition within the managed heap, which is used to categorize object according to their lifespan
- Generations are logical partitions within managed heap.
- object memory Allocation & deAllocation is done by GC
- Object Allocation in Gen 0, Gen 1, Gen 2:
  1) • By default new objects are allocated in Gen 0 of heap
  2) • promoting to high Generation:
     - when Gen0 (heap partition) is full, Gen [0] collection occcur (GC comes and do clean up)
     - All live objects, which does not get collect by Gen [0] Collection are moved to Gen 1
     - dead objects are deleted, & memory is Reclamaid
  3) • when Gen 1 is full, Gen [1] collection by GC occcurs
     - objects which survive Gen [1] collection (clean up by GC) are promoted to Gen 2 heap
- GC decides which generation an object belongs to
- you can call GC.Collect (0) → for Gen 0 collection, similarly GC.Collect (1) & GC.Collect (2) Explicitly

Generation	Description	Eg
Gen [0]	• all newly created objects are into Gen 0 heap • frequency of collection : very frequent	• Temporary variables, short live objects are get cleaned by Gen [0] collection
Gen [1]	• objects which survicved Gen [0] collection are moved to Gen [1] heap • frequency of collection : occasionally	• mid life objects, objects surviving initial collection • mid live objects
Gen [2]	• all objects which survied Gen [1] Collection, are moved to Gen [2] • these are very long lived objects	• static variables data, global variables, long term cached data, classes objects

* Algorithm on GC Works

- GC works on Mark, Sweep & Compact algorithm

1) Marking Algorithm:
- in this GC traverse the object graph and identify all live (reachable) objects
- it marks object as live & Un Marked objects are considered garbage and will be cleaned up

2) Sweeping Algorithm
- After marking GC scans through the heap & collects all un-marked objects
- These objects are deleted & memory is reclaimed. for re-Use

3) Compacting Algo
- GC compacts Memory to Eliminate fragmentation
- heap memory is compacted by moving live objects together, so that their are no fregments in the memory (no Gap)
- So, live objects (surving object after sweep process) are moved to create contigious block of memory

* Methods

GC.Collect() : forces garbage Collection, GC[0], GC[1], GC[2] will come and Collect Garbage

GC.GetGeneration(object) : Returns the generation of a specific object you pass

GC.TotalMemory(false) : Returns the amount of memory currently allocated

GC.MaxGeneration : Returns the maximum supported generation (usually 2)

GC.SuppressFinalize() : GC will come and Collect garbage but it won't call its destructor/finalizer

* Garbage Collection Process

Step 1: Heap memory divided into Gen 0, Gen 1 & Gen 2

Step 2: Allocation of Memory :

- when object is created, memory is allocated by GC from Gen 0 segment of heap

Step 3: Mark Phase : GC collection phase

- The GC pause application thread & traverse to mark object as live

Step 4 : Sweep phase :

- The GC comes scans the heap & unmarked objects are cleaned

Step 5 : Compacting phase :

- To reduce fragmentation this happens

Step 6 : GC Trigger

- when Gen 0 heap is full, GC comes & collect the garbage
- then objects are moved to Gen [1] & Gen [2] when they survived GC collections
- GC Collection : works on 3 algo, Mark, Sweep & Compact

Step 7 : When GC gets trigger :

- GC is unpredictable, so it comes any time in generation to do clean up operation
- Gen 0 collection : when Gen 0 heap is full, GC comes & clean the heap & moves obj to gen [1] if survived
- Gen 1 Collection : when Gen 1 heap is full, GC [1] comes & clean the heap & moves obj to gen [2] if needed
- Similarly Gen [2] collection happens

* Garbage Collection Working

- GC is un-predicatable, it works on single process but can have multiple threads
- Gen [0] collection : for Generation 0 heap
- Gen [1] collection : GC [1] comes & cleans the Generation 1 heap
- similarly Gen [2] collection happens
- Concurrent Collection : Gen [0], Gen [1] & Gen [2] collection can also occur concurrently (same time or parallely)

#13 STACK & HEAP MEMORY

* Value Types

- these variables are stored in stack memory where they are declared as local variables within a method
- Method call (function call) is handled on stack
  - Method arguments/parameters: if they are value types, they are stored directly in stack, if they are refrence types then only the refrence (address) is stored in stack & value is stored in heap
- Value of Refrence type variables is storead in heap, but their address is storeed in stack.
- all value types data types are Stored in stack
- Value types: all Primitive data types (int, float, char)
  Struct Types (System.Int32)
  Enum (Console.Color)
  Nullable Types (eg = int? a = null;)
- So value types, directly the value is stored in stack & refrence type only address is stored in stack & their actual value is stored in heap
- when value type is assigned to a new variable, their value is copied (deep copy). changes to one variable does not affect the other
- when value type (int) is part of a refrence type (class), then value are storead inside heap & address of the value type exists as an offset within the heap object. Its (address of value type) is not stored separetly on the stack, if its part of Refrence type
- its not costly to retrive data from stack
- When we do Boxing, copy of value type is created on heap

eg:
```
struct B // Struct is a value type
{
    int a; // value type
    string s; // refrence type
}
```

B: since B is struct (it is value type) it will store in stack
a: is value type :. Stored directly within the stack
s: it is refrence type :. Actual data (value of s) will get store in Heap and address of that location will be in stack

- memory Management of Value types is not done by GC directly, if the value types are ~~location~~ Local variables, then automatically as the scope of method gets over clean up is done by Run time, not by GC

* Refrence type

- Refrence type variables store on heap memory
- Refrence types : class, string, array, interface, delegate
- When you Assign a Refrence type to a new variable, a refrence (address) pointer to that memory location is made. ∴ changes made to one variable will reflect in other, beoz both are pointing towards only one location. this is called shallow copy
- these Refrence types are null by default
- There is no specific Entry & Exit point for heap
- data Retrival is costly from Heap
- Refrence types variables data will get store on heap but their memory address will be in stack
- eg: class is refrence type, & if any value type is their like int or float, then even though these are value type, (Boxing will happen & converted to object) but beoz they are present in refrence type, they will act like Ref type & will get store on heap

- eg:

```
class A          // class is Refrence type
{
    int x = 10;  // value type
    string s = "Hello"; // refrence type
}
```

A : is a class ∴ memory allocation will be on Heap, but refrence (address) of object of A will be stored in stack

x : is value type but it will behave like ref type beoz its under Ref type ∴ x will get store on heap

s : is ref type ∴ data will store on heap along with address on heap only

- Memory Management for ~~heap~~ heap data is done by GC

* STACK & HEAP MM Difference

Aspect	STACK	HEAP
memory : Speed	Static, LIFO Fast	• Dynamic, from any wher Entry & Exit can happen • Slow becoz of dynamic allocation
memory management	Automatically cleare up stack, when scope gets End, by CLR	• GC is called & it will clean the memory of heap as Gen 0, 1 or 2
Data Stored	Value types, method calls, address to heap memory	• Refrence types, objects created with ~~new~~ new Boxed value types
Exception Risk	• Stack overflow occurs if too many fun call	• Out of memory (if heap is not cleared by GC)
fregmentation	not possible becoz of continous memory block	• possible
Allocation type	Continuous memory	• free memory :. fregments get created
Thread Safe	Each thread has its own stack :. thread safe	• not thread safe

aspect	Value Type	Ref types
memory allocation	• Stack	• heap
Storage	• Stores Actual data	• stores a Refrence (pointer) to the actual data on heap
Default value	• 0 or false	• null
Memory Management	• Stack runtime, Automatic	• managed by GC
Immutable	• Usually mutable	• can be Immutable (String) as well as mutable (array)
Assignment behaviour	• copy of value to different memory location	• copy of Refrence to same mory location
Passing to method	• value is passed	• Refrence is passed

#14) ASSEMBLIES (.dll/.exe)

* Assembly

- When .dotNet program is compiled with appropriate compiler, the output is an assembly.
- Assemblies can be either in .dll (Dynamic Link Library) or .exe format and contains IL Code files
- Assemply contains Code (IL) Intermidiate Lanaguage Code
- .dll : it is used as library whose Refrence is Given in other .cs files in Refrence property
- .exe : it is the Executable file, which contains the starting point of application (Main program)
- To Execute Assemblies (IL Code to Machine Code) we have CLR Runtime which has JIT compiler which converts IL Code (Assembly) to Machine Code
- Assembly is the Executable (exe or dll) Code along with meta data
- Assembly includes .DLL & .Exe files along with their meta data required by Runtime to Execute the program

• Exe :

- An .exe (Executable (IL Code) file) is a type of file that Contains compiled Code (IL Code) that can be Executed directly by OS
- When you run a desktop or Console App you run the .exe file present in Bin folder which is produce after Build
- .exe file includes a Entry Point (Main fun)

• DLL

- A dll file also contains compiled Code (IL Code), but unlike .exe file, you cannot run this .dll file directly, becoz it does'nt include Main fun()
- Dll are used to store reusable Code (Libraries, namespace, class) which can be used in Main program by adding the refrence of .dll file
- Dll contains functions, classes, & resources that can be used by other applications after adding Refrence

defination: Assembly is a Compiled (IL Code) unit of Code in .Net framework that consists of one or more file (.dll & .exe) along with metadata of those files, ~~Need~~ which is needed by the Run time to run Application

* Types of Assemblies

1) **Private Assemblies** : these assemblies are for a specific Application only and are stored in Applications directory/folder
   - These (.dll) files cannot be used by other Application, becoz they are out of scope

2) **Shared Assemblies** : These assemblies (.dll) files are designed to be shared among/across multiple application.
   - There scope is not Restricted to a specific Application
   - These Assemblies are stored inside GAC, so can be Accessed by different allplication
   - Even though shared Assemblies are inside GAC, they cannot be accessed without adding a refrence

3) **Dynamic Assemblies** : these Assemblies are Created at Runtime & typically used for Senarios where Code is generated dynamically (eg. Reflection.Emit)

(Q) System namespace is available without adding refrence

→ Becoz: .Net automatically refrences the Core Assemblies
   - As this System is Part of BCL :. .Net automatically refrence it
   - Common Namespaces are imported automatically via Global usings

#5 CLR (Common Language Runtime)

Assemblies

- The C# code, using C# compiler is converted to IL^Code & further the IL code using CLR (JIT) gets converted to ML code
- To Execute the assemblies (IL code) generated by C# compiler, System must have CLR

* CLR

- CLR provides a Managed Execution Environment for running the .Net application
- CLR handles : Memory Management (GC)
  Security
  Exception handling
  provides JIT
- CLR Includes following :

1) BCL : CLR Includes Base class library, which Contains multiple libraries & inBuilt functions, which provides features such as Collections, Data Types, Console class etc
   - All Basic class are defined in BCL, without it you can't Even print output

2) Thread Support : The CLR provides thread support for managing multiple threads under the class System.Threading

3) Com Marshal : CLR provides Com Marshal for communication b/w managed & unmanaged Code

4) CTS & CLS : Common Type System & Common Language Specification provides language interoperability (interoperability means the ability of code written in one .Net language C#, to interact with code written in other .Net language F# without compatibility issues.

CTS : Common type System defines a set of Rules and data types that all .Net languages must follow
- also checks the type safety of data types
- Ensures that int in C# and Integer in VB.Net are all mapped to the same type in IL Code
- Provides type Safety

CLS : Common Language Specification provides Rules for language interpolability
- focus on naming Conventions, method Signatures etc

~~* JIT :~~

#6 JIT (Just In Time Compiler)

* JIT

- JIT is part of CLR, which Converts the IL Code to Machine Code
- C# Compiler Converts the Code into IL Code & then JIT will convert the IL Code into Machine Code
- JIT Ensurece platform independence

* Types of JIT

1) Pre JIT : also called Ahead of time Compilation (AOT)
   - Converts all IL code into native code (all at once) before the Application starts
   - Compiles all IL Code into native code (at Once) before Execution starts
   - Useful in Senarios requiring faster Execution after initial Compilation happens
   - disadvantage : Requires longer startup time for application becoz all IL Code is converted to ML (native code) at once.
   - Used by Tools like : Ngen (Native Image Generator)
   - Entire IL Code is Converted to ML code before Application starts. The native Code (ML Code) is retained (stored) as a native image file on dick
   - when application runs, the CLR loads the precompiled Code directly without involving JIT becoz all code is already compiled.

2) Econo JIT : Converts IL Code to native code only when the method is called at the Runtime of application
   - But it does not store the native code, So the IL Code is recompiled on Every function call
   - it does not store the native Compiled Code, So suitable for Application where memory is short
   - Compiles methods ~~just~~ at Runtime, when the methods are called & does not retain Compiled Code
   - Advantage : Lower Memory Usage
   - disAdvantage : Slower Application performance, becoz of repitatly Compilation

3) Normal JIT: Compiles the IL Code ~~Just befor the~~ during Runtime, when the Methods are called, But once a method is Compiled into native code, it stores that native code

- Converts IL to native code only when the method is called for the first time, & the stores the native code
- Compiled code is et cached in memory for future calls
- Advantage: performance is high After first call
- disadvantage: performance is low during first call