**Assignment Day 1** –

1. **How Does GC work ?**

During garbage collection, the GC uses the "mark and sweep" technique, which marks all reachable objects as "live," then sweeps through the managed heap to reclaim memory from "dead" objects.

In the .NET framework, garbage collection is triggered automatically by specific events or criteria, including low memory, an exceeded threshold, and the GC.Collect() method being called.

The garbage collection process comprises three phases that work together to identify and collect unused objects:

* Marking phase
* Relocating phase
* Compacting phase

**Marking Phase -** In the marking phase, the GC finds and creates a list of all the objects in the managed heap that are in use and need to be retained. The GC starts at the root of the application with objects such as global variables, static objects, and references on the stack and moves up the object graph to mark reachable objects as "live." Objects not on the list are considered unreachable and become potential candidates for garbage collection.

**Relocating Phase** - The relocating phase occurs if the GC determines that there is significant fragmentation in the managed heap. It updates the references of all the live objects on the list to point to the new location where they will be moved during the compacting phase. The GC does this by maintaining a forwarding address for each relocated object. References to the object are redirected to the new location using the forwarding address.

**Compacting Phase** - The compacting phase reclaims the space occupied by the "dead" objects and compacts the surviving ("live") objects. It does this by moving live objects into contiguous address space and freeing up fragmented memory regions. Surviving objects are moved toward the older end of the heap memory in their original order. The compacting phase depends on memory allocation and fragmentation patterns. If there is sufficient memory available and the level of fragmentation is low, the compacting phase may not occur during a garbage collection cycle.

1. **What is the use of IDisposable interface ? Different ways to call dispose method.**

IDisposable is an interface defined in the System namespace. It is used to release managed and unmanaged resources. Implementing IDisposable interface compels us to implement 2 methods and 1 boolean variable –

1. Public Dispose() : This method will be called by the consumer of the object when resources are to be released. This method will invoke the Dispose(bool disposing) method. We also notify the Garbage collector(GC) about the current object’s resource cleaning so that GC avoids doing it again at the end.
2. Protected virtual Dispose(bool disposing): This method is the place where all the variables will be set to null. Also, the derived class can override this method and release its own resources. The actual resource cleaning code is written in this method.
3. Bool disposedValue: To ensure the dispose method is called once we use a boolean variable. The resource cleaning must not happen more than once.

The consumer of the object can call the first disposal method explicitly whenever the resources are no longer required. The garbage collector also cleans the object when it is out of scope. However, it is not capable of releasing a few objects such as files, streams, database connections, window handles, etc.

1. **Why should we suppress finalize call ?**

SuppressFinalize() system method is designed to prevent calling the finalizer on the specified object. If an object does not have a destructor, invoking SuppressFinalize on this object has no effect

1. **What is FCL & BCL ?**

**FCL** - FCL is a common class library for all applications which can be developed through .NET Framework. i.e., the way we access the Library Classes and Methods in VB.NET will be the same in C#, and it is common for all other languages in .NET. Class library is the collection of reusable types that are closely integrated with CLR.

**BCL** - is the base library which contains classes & methods for base classes such as System.String or System.IO etc.

**Assignment Day 2** –

1. **Upcasting & Downcasting –**

Upcasting is a concept in C# that allows us to treat a derived class as its base class.

In other words, upcasting is the process of converting an object of a derived class to an object of its base class. We achieve this through implicit type conversion, where the derived class object is assigned to the base class object.

After the implicit conversion, we can treat the object as if it is an instance of the base class. This means that the properties and methods of the derived class that are not present in the base class will not be accessible.

Downcasting is a technique that allows us to treat a base class object as an instance of its derived class.

In C#, Downcasting works by explicitly converting a base class reference to a derived class reference using the cast operator:

**[Ex](https://eb2.3lift.com/pass?tl_clickthrough=true&redir=https%3A%2F%2Fevs.jivox.com%2Ftrk%2F1%2F228305%2F8000%2F180654%2Fedcbd7ed316741c%2F80%2FjvxSId_1721188139.8176%2Fes_pId_97819f25%2Fes_encParams_%3D%3D&pr=0.31&bc=0.413&aid=21384991946824411708711&bmid=3658&biid=3658&sid=209208&did=65249&tid=20272893&clid=19875780&brid=66&adid=gdwrqu41&crid=246259001&ts=1721188137&bcud=413&ss=12&dcr=4&cb=45378" \t "_blank)** [- DerivedClass derivedObject = (DerivedClass) baseObject;](https://eb2.3lift.com/pass?tl_clickthrough=true&redir=https%3A%2F%2Fevs.jivox.com%2Ftrk%2F1%2F228305%2F8000%2F180654%2Fedcbd7ed316741c%2F80%2FjvxSId_1721188139.8176%2Fes_pId_97819f25%2Fes_encParams_%3D%3D&pr=0.31&bc=0.413&aid=21384991946824411708711&bmid=3658&biid=3658&sid=209208&did=65249&tid=20272893&clid=19875780&brid=66&adid=gdwrqu41&crid=246259001&ts=1721188137&bcud=413&ss=12&dcr=4&cb=45378" \t "_blank)

1. **Difference between abstract class & interfaces ?**

|  |  |
| --- | --- |
| **Abstract class** | **Interface** |
| 1) Abstract class can **have abstract and non-abstract** methods. | Interface can have **only abstract** methods. It can have **default and static methods** also. |
| 2) Abstract class **doesn't support multiple inheritance**. | Interface **supports multiple inheritance**. |
| 3) Abstract class **can have final, non-final, static and non-static variables**. | Interface has **only static and final variables**. |
| 4) Abstract class **can provide the implementation of interface**. | Interface **can't provide the implementation of abstract class**. |
| 5) An **abstract class** can extend another class and implement multiple interfaces. | An **interface** can extend another interface only. |
| 6) An **abstract class** can have class members like private, protected, etc. | Members of an interface are public by default. |

1. **Named & Optional arguments** –

*Named arguments* enable you to specify an argument for a parameter by matching the argument with its name rather than with its position in the parameter list. *Optional arguments* enable you to omit arguments for some parameters. Both techniques can be used with methods, indexers, constructors, and delegates.

When you use named and optional arguments, the arguments are evaluated in the order in which they appear in the argument list, not the parameter list.

Named and optional parameters enable you to supply arguments for selected parameters. This capability greatly eases calls to COM interfaces such as the Microsoft Office Automation APIs.

**Named arguments -** Named arguments free you from matching the order of arguments to the order of parameters in the parameter lists of called methods. The argument for each parameter can be specified by parameter name. For example, a function that prints order details (such as, seller name, order number & product name) can be called by sending arguments by position, in the order defined by the function.

**Ex** – PrintOrderDetails(orderNum: 31, productName: "Red Mug", sellerName: "Gift Shop");

**Optional arguments -**The definition of a method, constructor, indexer, or delegate can specify its parameters are required or optional. Any call must provide arguments for all required parameters but can omit arguments for optional parameters.

Each optional parameter has a default value as part of its definition. If no argument is sent for that parameter, the default value is used. A default value must be one of the following types of expressions:

* a constant expression;
* an expression of the form new ValType(), where ValType is a value type, such as an [enum](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/enum) or a [struct](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/struct);
* an expression of the form [default(ValType)](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/default), where ValType is a value type.

Optional parameters are defined at the end of the parameter list, after any required parameters. If the caller provides an argument for any one of a succession of optional parameters, it must provide arguments for all preceding optional parameters. Comma-separated gaps in the argument list aren't supported. For example, in the following code, instance method ExampleMethod is defined with one required and two optional parameters.

**Example** - public void ExampleMethod(int required, string optionalstr = "default string", int optionalint = 10)

**Assignment Day 3 -**

1. **Create a solution called Bosch calculator.**
2. Add a class library called Bosch.Calculator. Add two classes in the following library named BasicMath & AdvancedMath.
3. 4 functions in BasicMath – Add,Substarct,Multiply,Divide
4. 2 functions in AdvanceMath – Square & SquareRoot
5. Use in Bosch Calculator UI Console Application.

**Answer –** Done : Solution Name : Bosch.Calculator

1. **Create a solution called Bosch.PO.System.** 
   1. Create a class library called PO.Model
   2. Add 3 classes – Customer, Order & Product
   3. 5 properties in each class.
   4. Use all 3 classes in Console application using object initializer syntax.
   5. Add console with name Bosch.PO.UI.

**Answer –** Done : Solution Name : Bosch.PO.System

**Assignment Day 4** –

1. **Write the same join query using lambda expressions & extension methods for the purchase order solution.**
2. **What is the use of Mutex & Semaphore? When to use which one ?**

**Answer** - A mutex is a locking and unlocking mechanism, and a semaphore is a signaling mechanism. Both are used for critical section and mutual exclusion problems.

* Mutex uses a locking mechanism. When a process uses a resource, it locks the resource, uses it, then releases it. On the other hand, a semaphore uses a signaling mechanism using the wait() and signal() methods to indicate whether a process is releasing or acquiring a resource.
* A mutex is an object while a semaphore is an integer variable. Semaphores have wait() and signal() functions. However, Mutex has no such functionality.
* A Mutex object allows multiple process threads to access a shared resource, but only one at a time. On the other hand, Semaphores allow multiple process threads to access a finite instance of a resource until a finite instance of the resource becomes available.
* A mutex allows the same process to acquire and release the lock simultaneously. However, the value of a semaphore variable can be changed by any process that needs some resources, but only one process at a time can change the value.

1. **When to use lock and monitor ?**

The lock keyword is used to get a lock for a single thread. A lock prevents several threads from accessing a resource simultaneously. Typically, you want threads to run concurrently. Using the lock in C#, we can prevent one thread from changing our code while another does so.

A monitor is a mechanism for ensuring that only one thread at a time may be running a certain piece of code (critical section). A monitor has a lock, and only one thread at a time may acquire it. To run in certain blocks of code, a thread must have acquired the monitor. A monitor is always associated with a specific object and cannot be dissociated from or replaced within that object.

**Assignment Day 5 -**

1. Implement purchase order system using SOLID principles.

Entities – Employee, Customer, Supplier, Shipper, Category, Product, Order, OrderDetail, Invoice

* 1. Customer will place an order for a product by choosing a category
  2. Employee will process the order & generate the whole invoice
  3. Supplier will supply the processed order

**Assignment Day 6** –

**1.) Explain each method of CommonRepository class.**

**Answer –**

1. **GetAllAsync –** Asynchronous method to get list of all the details of a particular entity (all rows of a table in a database) without any filters.
2. **GetDetailsAsync -** Asynchronous method to get all the details of a particular entity (single row in a table in a database) with primary ID filter.
3. **InsertAsync -** Asynchronous method to insert list of all the details of a particular entity (multiple row insertion in a table in a database) with primary ID filter.
4. **UpdateAsync -** Asynchronous method to update list of all the details of a particular entity (multiple row update in a table in a database) .
5. **DeleteAsync -** Asynchronous method to delete all details of a particular entity (singe row deletion in a table in a database) with primary ID filter.
6. **Why should we use repository pattern.**

**Answer** -  To increase testability and have a loose coupling to underlying persistence technology.



**Assignment Day 9** –

1.) **Global Error Handling in ASP.NET MVC application.**

**Answer -** Between .NET, ASP.NET, and MVC there are several potential ways to handle application errors-

1.) Web.Config customErrors

2.) MVC HandleErrorAttribute

3.) Controller.OnException method

4.) HttpApplication Application\_Error event

**1.) Web.Config customErrors -** Global Error Page with Web.Config

**<customErrors>**

**<system.web>**

**<customErrors mode="On" defaultRedirect="~/ErrorHandler/Index">**

**<error statusCode="404" redirect="~/ErrorHandler/NotFound"/>**

**</customErrors>**

**<system.web/>**

**2.) MVC HandlerErrorAttribute -** Use MVC HandlerErrorAttribute to Customize Responses. The HandleErrorAttribute inherits from FilterAttribute and can be applied to an entire controller or individual controller action methods.

It can only handle 500-level errors that happen within an MVC action method. It does not track exceptions that help outside of the MVC pipeline. Exceptions may occur in other HTTP modules, MVC routing, etc.

**When to Use HandleErrorAttribute -**

Since it does not provide a way to collect all exceptions that could ever happen, it is a bad solution for a global unhandled error handler.

It works perfectly for tailoring specific error pages for a particular MVC controller or action method and specifying an error page in your Web.config <customErrors> works ideal for a universal error page. The HandleErrorAttribute gives you fine-grained control if you need it.

**Note**: HandleErrorAttribute requires customErrors to be enabled in your Web.Config.

For example, if you wanted to show a particular MVC view when a SqlException happens, you can do it with the code below:

**[HandleError(ExceptionType = typeof(SqlException), View = "SqlExceptionView")]**

**public string GetClientInfo(string username)**

**{**

**return "true";**

**}**

The problem with HandleErrorAttribute is it doesn’t provide a way to log the exception!

**3. ) Use MVC Controller OnException -**

OnException is like HandleErrorAttribute but provides more flexibility. It works with all HTTP status codes, and not just 500-level responses. It also gives you the ability to log the errors!

**public class UserMvcController : Controller**

**{**

**protected override void OnException(ExceptionContext filterContext)**

**{**

**filterContext.ExceptionHandled = true;**

**//Log the error!!**

**\_Logger.Error(filterContext.Exception);**

**//Redirect or return a view, but not both.**

**filterContext.Result = RedirectToAction("Index", "ErrorHandler");**

**// OR**

**filterContext.Result = new ViewResult**

**{**

**ViewName = "~/Views/ErrorHandler/Index.cshtml"**

**};**

**}**

**}**

**When to Use OnException for MVC Error Handling -**

If you want a way to present your users with custom MVC views or custom log exceptions, OnException is a good solution for you. It provides more flexibility than HandleErrorAttribute and does not require customErrors to be enabled in your Web.Config file.

**Note**: OnException gets called for all HTTP status codes. So be careful how you handle simple issues like a 404 caused by a bad URL.

**4. ) Use HttpApplication Application\_Error as Global Exception Handler –**

So far, we have covered three different ways to customize the response that your users see if an exception occurs. Only within OnException can you potentially log exceptions.

To log all unhandled exceptions that may occur within your application, you should implement a basic error logging code as shown below.

**public class MvcApplication : System.Web.HttpApplication**

**{**

**protected void Application\_Start()**

**{**

**AreaRegistration.RegisterAllAreas();**

**FilterConfig.RegisterGlobalFilters(GlobalFilters.Filters);**

**RouteConfig.RegisterRoutes(RouteTable.Routes);**

**BundleConfig.RegisterBundles(BundleTable.Bundles);**

**}**

**protected void Application\_Error()**

**{**

**var ex = Server.GetLastError();**

**//log the error!**

**\_Logger.Error(ex);**

**}**

**}**

If you want a way to present your users with custom MVC views or custom log exceptions, OnException is a good solution for you

**When to Use Application\_Error -**

Always! HttpApplication’s Error even provides the best mechanism to collect and log all unhandled application errors.

**2.) What is CORS ?**

**Answer - Cross-origin resource sharing** (**CORS**) is a mechanism that allows a web page to access restricted resources from a server on a domain different than the domain that served the web page.

A web page may freely embed cross-origin images, stylesheets, scripts, iframes, and videos. Certain "cross-domain" requests, notably Ajax requests, are forbidden by default by the same-origin security policy. CORS defines a way in which a browser and server can interact to determine whether it is safe to allow the cross-origin request. It allows for more freedom and functionality than purely same-origin requests, but is more secure than simply allowing all cross-origin requests.

**Assignment Day 10** –

**1.) List different types of constraints while formatting HTTP attributes** –

**1.) HTTP Methods -**Web API also selects actions based on the HTTP method of the request (GET, POST, etc). By default, Web API looks for a case-insensitive match with the start of the controller method name. For example, a controller method named PutCustomers matches an HTTP PUT request.

You can override this convention by decorating the method with any of the following attributes:

* **[HttpDelete]**
* **[HttpGet]**
* **[HttpHead]**
* **[HttpOptions]**
* **[HttpPatch]**
* **[HttpPost]**
* **[HttpPut]**

**2.) Route Prefixes** -Often, the routes in a controller all start with the same prefix. For example:

**[RoutePrefix("api/books")]**

public class BooksController : ApiController

{

// GET api/books

**[Route("")]**

public IEnumerable<Book> Get() { ... }

// GET api/books/5

**[Route("{id:int}")]**

public Book Get(int id) { ... }

// POST api/books

**[Route("")]**

public HttpResponseMessage Post(Book book) { ... }}

1. **Route Constraints** - Route constraints let you restrict how the parameters in the route template are matched. The general syntax is "{parameter:constraint}". For example:

[Route("users/{id:int}")]

public User GetUserById(int id) { ... }

[Route("users/{name}")]

public User GetUserByName(string name) { ... }

| **Constraint** | **Description** | **Example** |
| --- | --- | --- |
| alpha | Matches uppercase or lowercase Latin alphabet characters (a-z, A-Z) | {x:alpha} |
| bool | Matches a Boolean value. | {x:bool} |
| datetime | Matches a **DateTime** value. | {x:datetime} |
| decimal | Matches a decimal value. | {x:decimal} |
| double | Matches a 64-bit floating-point value. | {x:double} |
| float | Matches a 32-bit floating-point value. | {x:float} |
| guid | Matches a GUID value. | {x:guid} |
| int | Matches a 32-bit integer value. | {x:int} |
| length | Matches a string with the specified length or within a specified range of lengths. | {x:length(6)} {x:length(1,20)} |
| long | Matches a 64-bit integer value. | {x:long} |
| max | Matches an integer with a maximum value. | {x:max(10)} |
| maxlength | Matches a string with a maximum length. | {x:maxlength(10)} |
| min | Matches an integer with a minimum value. | {x:min(10)} |
| minlength | Matches a string with a minimum length. | {x:minlength(10)} |
| range | Matches an integer within a range of values. | {x:range(10,50)} |
| regex | Matches a regular expression. | {x:regex(^\d{3}-\d{3}-\d{4}$)} |