Unit 4

Exception Handling

What is an exception?

An exception is a **problem (error)** that arises during the execution of a program.

What is exception handling?

Exception Handling in C# is a **process to handle runtime errors**. We perform exception handling so that the normal flow of the application can be maintained even after runtime errors.

In C#, an exception is an event or object which is thrown at runtime. All exceptions the derived from System.Exception class.

It is a runtime error that can be handled. If we don't handle the exception, it prints an exception message and terminates the program.

Advantage

It *maintains the normal flow* of the application. In such a case, the rest of the code is executed even after the exception.

C# exception handling is built upon four keywords: **try**, **catch**, **finally**, and **throw**.

- **try** − A try block identifies a block of code for which particular exceptions is activated. It is followed by one or more catch blocks.
- **catch** − A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The catch keyword indicates the catching of an exception.
- **finally** − The finally block is used to execute a given set of statements, whether an exception is thrown or not thrown. For example, if you open a file, it must be closed whether an exception is raised or not.
- **throw** − A program throws an exception when a problem shows up. This is done using a throw keyword.

Handling Exception using try and catch:

In C# programming, exception handling is performed by try/catch statement. The **try block** in C# is used to place the code that may throw exception. The **catch block** is used to handle the exception. The catch block must be preceded by try block.

C# example without try/catch

```
using System;
public class ExExample
{
    public static void Main(string[] args)
    {
        int a = 10;
        int b = 0;
        int x = a/b;
        Console.WriteLine("Rest of the code");
    }
}
```

Output:

```
Unhandled Exception: System.DivideByZeroException: Attempted to divide by
zero.
```

C# with try/catch example

```
using System;
public class ExExample
{
    public static void Main(string[] args)
    {
        try
        {
            int a = 10;
            int b = 0;
            int x = a / b;
        }
        catch (Exception e)
            {
                    Console.WriteLine(e);
            }
        Console.WriteLine("Rest of the code");
    }
```

Raising exceptions using a throw

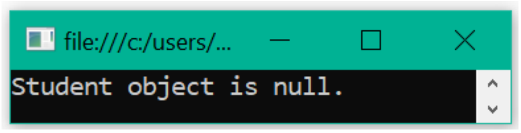
Throw statement is used for throwing exceptions in a program. The throwing exception is handled by a catch block.

An exception can be raised manually by using the throw keyword. Any type of exception which is derived from the Exception class can be raised using the throw keyword.

```
using System;

namespace throwdemo
{
    public class Student
    {
        public string StudentName { get; set; }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Student std = null;
            try
            {
                PrintStudentName(std);
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
            Console.ReadLine();
        }
        public static void PrintStudentName(Student std)
        {
            if (std == null)
            throw new NullReferenceException("Student object is null.");
            Console.WriteLine(std.StudentName);
        }
    }
}
```

Output:



```
Student object is null.
```

Pre-defined Exception classes

C# .NET includes built-in exception classes for every possible error.
The Exception class is the base class of all the exception classes.

The following table lists important built-in exception classes in .NET.

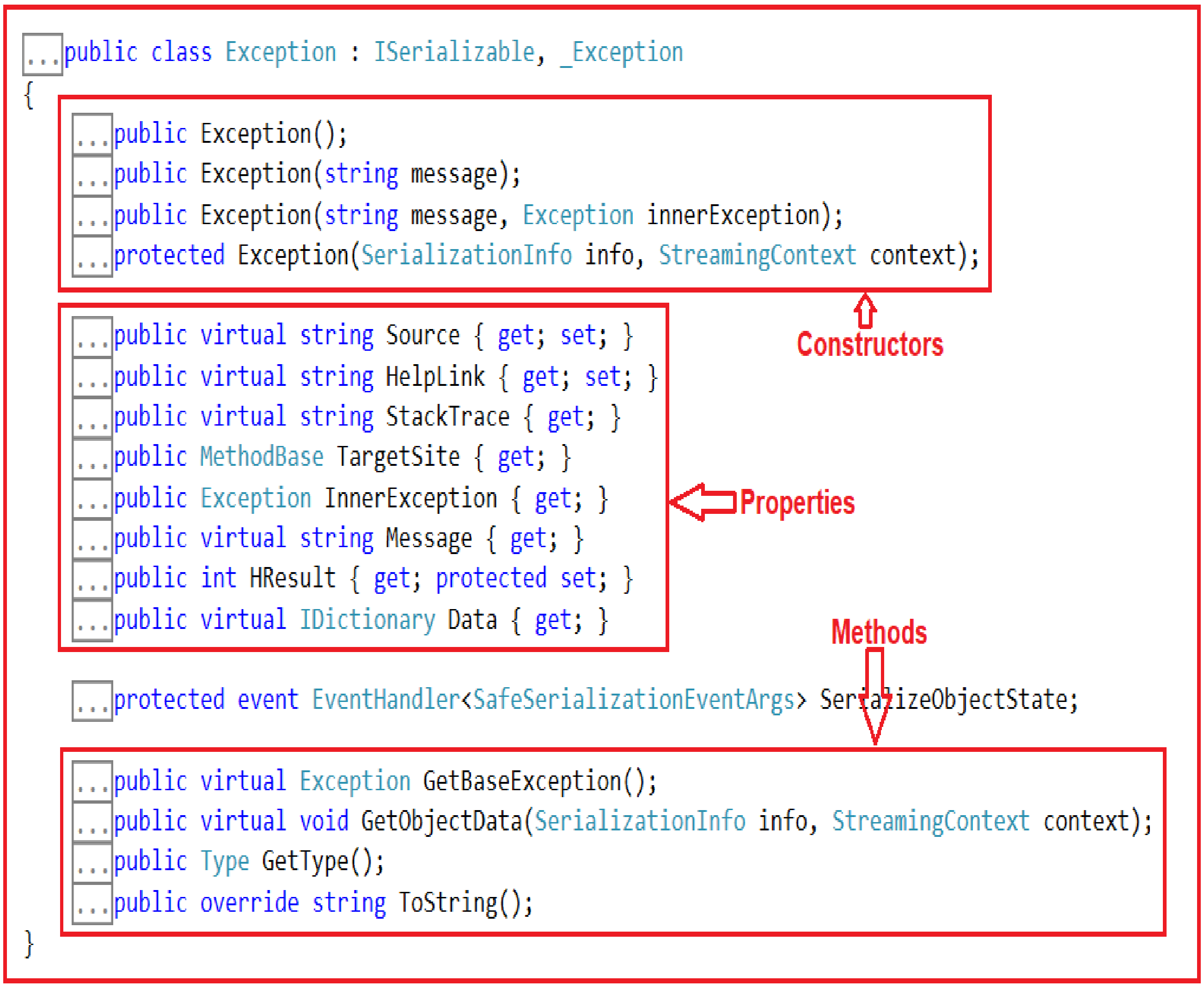
Exception Class	Description
ArgumentNullException	Raised when null argument is passed to a method.
ArgumentOutOfRangeException	Raised when the value of an argument is outside the range of valid values.
DivideByZeroException	Raised when an integer value is divide by zero.
FileNotFoundException	Raised when a physical file does not exist at the specified location.
FormatException	Raised when a value is not in an appropriate format to be converted from a string by a conversion method such as Parse.
IndexOutOfRangeException	Raised when an array index is outside the lower or upper bounds of an array or collection.
InvalidOperationException	Raised when a method call is invalid in an object's current state.
KeyNotFoundException	Raised when the specified key for accessing a member in a collection is not exists.
NotSupportedException	Raised when a method or operation is not supported.
NullReferenceException	Raised when program access members of null object.
OverflowException	Raised when an arithmetic, casting, or conversion operation results in an overflow.
OutOfMemoryException	Raised when a program does not get enough memory to execute the code.
StackOverflowException	Raised when a stack in memory overflows.
TimeoutException	The time interval allotted to an operation has expired.

When an error occurs, either the application code or the default handler handles the exception.

Custom Exception classes

If none of the already existing .NET exception classes serve our purpose then we need to go for custom exceptions in C#.

Before creating the Custom Exception class, we need to see the class definition of the Exception class as our Custom Exception class is going to be inherited from the parent Exception class. If you go to the definition of Exception class, then you will see the following.



```
public class Exception : ISerializable, _Exception
{
    public Exception();
    public Exception(string message);
    public Exception(string message, Exception innerException);
    protected Exception(SerializationInfo info, StreamingContext context);

    public virtual string Source { get; set; }
    public virtual string HelpLink { get; set; }
    public virtual string StackTrace { get; }
    public MethodBase TargetSite { get; }
    public Exception InnerException { get; }
    public virtual string Message { get; }
    public int HResult { get; protected set; }
    public virtual IDictionary Data { get; }

    protected event EventHandler<SafeSerializationEventArgs> SerializeObjectState;

    public virtual Exception GetBaseException();
    public virtual void GetObjectData(SerializationInfo info, StreamingContext context);
    public Type GetType();
    public override string ToString();
}
```

As you can see, the Exception class has some constructors, some virtual and non-virtual properties, and some virtual and non-virtual methods. The virtual members you can override in the child of this Exception class and you can directly consume the non-virtual members using the child class instance.
Now, to create a Custom Exception class in C#, we need to follow the below steps.
Step1: Define a new class inheriting from the predefined Exception class so that the new class also acts as an Exception class.

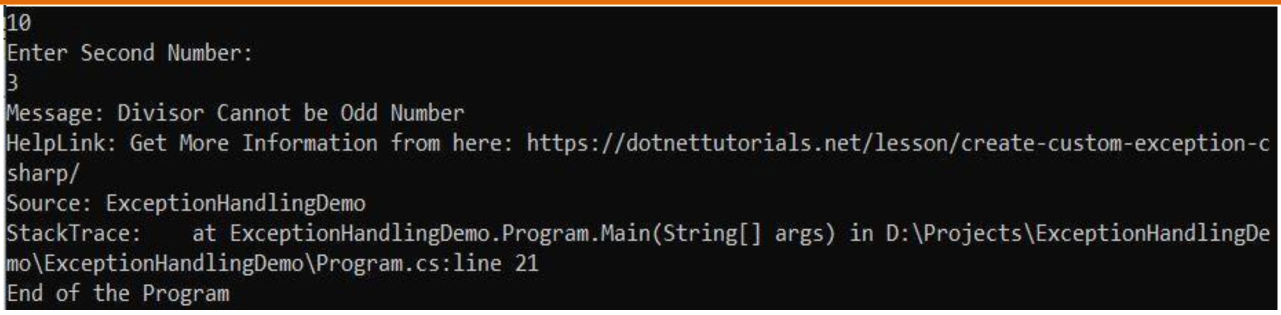
Step2: Then as per your requirement, override the virtual members that are defined inside the Exception class like Message, Source, StackTrace, etc with the required error message.
Let us understand how to create a custom exception in C# with an example. Create a class file with the name OddNumberException.cs and then copy and paste the following code into it. Here, you can see that the OddNumberException class is inherited from the built-in Exception class and here we are re-implementing two virtual properties i.e. Message and HelpLink. Now, we can create an instance of OddNumberException class and if we invoke Message and HelpLink properties, then these two properties are going to be executed from this class only. But if you invoke the Source and StackTrace properties, then those properties are going to be executed from the Exception class only as we have not re-implemented these properties. This is the concept of **Method Overriding in C#**.

```
using System;
namespace ExceptionHandlingDemo
{
        //Creating our own Exception Class by inheriting Exception class
        public class OddNumberException : Exception
        {
                //Overriding the Message property
                public override string Message
                {
                        get
                        {
                                return "Divisor Cannot be Odd Number";
                        }
                }
                //Overriding the HelpLink Property
                public override string HelpLink
                {
                        get
                        {
                                        return "Get More Information from here:
                                        https://dotnettutorials.net/lesson/create-custom-
                                        exception-csharp/";
                        }
                }
        }
}
```

Now, as per our business logic, we can explicitly create an instance of the OddNumberException class and we can explicitly throw that instance from our application code. For a better understanding, please have a look at the following code. Here, inside the Main method, we are taking two numbers from the user and then checking if the second number is odd or not. If the second number i.e. divisor is odd, then we are creating an instance of the OddNumberException class and throwing that instance. And in the Catch block, we are handling that exception and we simply printing Message, StackTrace, Source, and HellpLink properties. Here, if OddNumberException occurred, then Message and HelpLink properties are going to execute from OddNumberException class and Source and StackTrace properties are going to be executed from the pre-defined parent Exception class.

```
using System;
namespace ExceptionHandlingDemo
{
        class Program
        {
                static void Main(string[] args)
                {
                        int Number1, Number2, Result;
                        try
                        {
                                Console.WriteLine("Enter First Number:");
                                Number1 = int.Parse(Console.ReadLine());
                                Console.WriteLine("Enter Second Number:");
                                Number2 = int.Parse(Console.ReadLine());
                                if (Number2 % 2 > 0)
                                {
                                        throw new OddNumberException();
                                }
                                Result = Number1 / Number2;
                                Console.WriteLine(Result);
                        }
                        catch (OddNumberException one)
                        {
                                Console.WriteLine("Message:" +one.Message);
                                Console.WriteLine("HelpLink:" +one.HelpLink);
                                Console.WriteLine("Source:" +one.Source);
                                Console.WriteLine("StackTrace:" +one.StackTrace);
                        }
                        Console.WriteLine("End of the Program");
                        Console.ReadKey();
                }
}       }
```

Output:



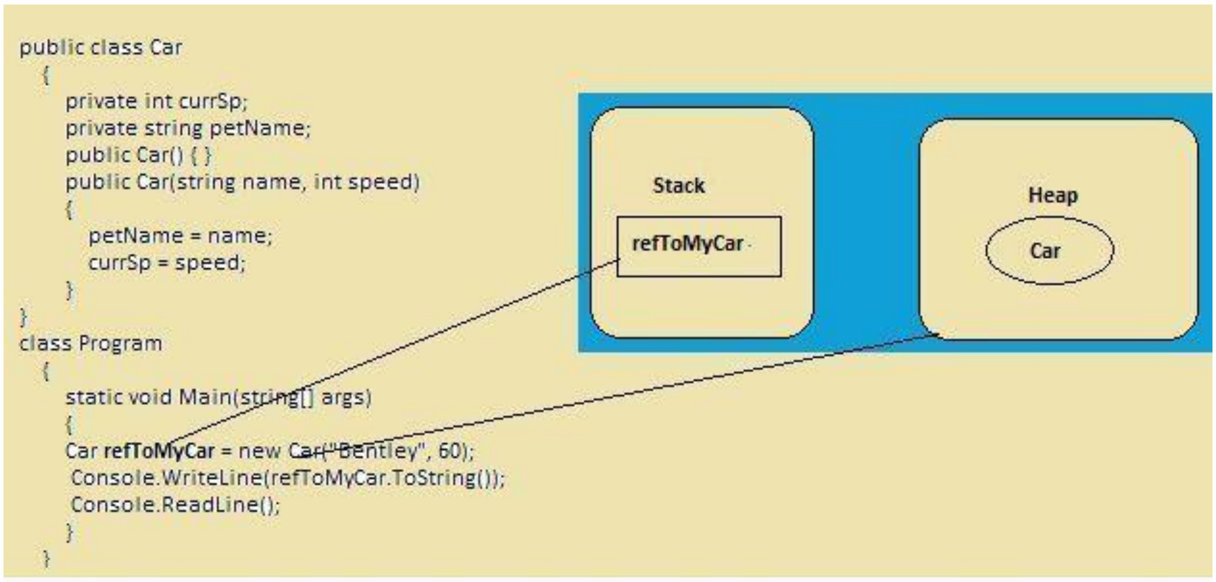
```
10
Enter Second Number:
3
Message: Divisor Cannot be Odd Number
HelpLink: Get More Information from here: https://dotnettutorials.net/lesson/create-custom-exception-c
sharp/
Source: ExceptionHandlingDemo
StackTrace:    at ExceptionHandlingDemo.Program.Main(String[] args) in D:\Projects\ExceptionHandlingDe
mo\ExceptionHandlingDemo\Program.cs:line 21
End of the Program
```

Understanding object lifetime classes, objects, and References.

Object lifetime is **the time when a block of memory is allocated to this object during some process of execution and that block of memory is released when the process ends.**

Once the object is allocated with memory, it is necessary to release that memory so that it is used for further processing, otherwise, it would result in memory leaks. We have a class in .Net that releases memory automatically for us when the object is no longer used. We will try to understand the entire scenario thoroughly of how objects are created and allocated memory and then deallocated when the object is out of scope.

The class is a blueprint that describes how an instance of this type will look and feel in memory. This instance is the object of that class type. A block of memory is allocated when the **new** keyword is used to instantiate the new object and the constructor is called. This block of memory is big enough to hold the object. When we declare a class variable it is allocated on the stack and the time it hits a new keyword and then it is allocated on the **heap**. In other words, when an object of a class is created it is allocated on the heap with the C# **new** keyword operator. However, a new keyword returns a reference to the object on the heap, not the actual object itself. This reference variable is stored on the stack for further use in applications.



```
public class Car
  {
      private int currSp;
      private string petName;
      public Car() { }
      public Car(string name, int speed)
      {
         petName = name;
         currSp = speed;
      }
}
class Program
  {
      static void Main(string[] args)
      {
      Car refToMyCar = new Car("Bentley", 60);
       Console.WriteLine(refToMyCar.ToString());
       Console.ReadLine();
      }
  }
```

Diagram - new keyword returns a reference to the object on the heap and the actual reference variable is stored on stack.

When the new operator is used to create an object, memory is taken from the managed heap for this object and the managed heap is more than just a random chunk of memory accessed by the CLR. When the object is no longer used then it is de-allocated from the memory so that this memory can be reused.

The garbage collector cleans up **managed resources** automatically since the managed code is directly targeted by the CLR. But when the object uses **unmanaged resources** like database connections or file manipulation, that needs to be released manually and this can be done by a finalized method.

We use the destructor method using the (~) sign in our code to destroy the objects and this destructor is converted into a finalize method (check in the compiled code). This is known as a finalization process. If we are implementing Finalize(), we do not have control since when this method should be called the garbage collector takes care of this on its own.

There is another method, Dispose(), that releases managed and unmanaged resources explicitly. This method is the single method in an IDisposable interface and can be used to release unmanaged resources manually.

The basics of object lifetime

What is the basic of object lifetime?

While the basic idea of object lifetime is simple – **an object is created, used, then destroyed** – details vary substantially between languages, and within implementations of a given language, and is intimately tied to how memory management is implemented.

```
namespace ObjectLifeTime;

class Foo
{
    public Foo()
    {
        // This is the implementation of
        // default constructor.
    }
    public Foo(int x)
    {
        // This is the implementation of
        // the one-argument constructor.
    }
     ~Foo()
    {
        // This is the implementation of the destructor.
    }
    public Foo(int x, int y)
    {
        // This is the implementation of
        // the two-argument constructor.
    }
    public static void Main(string[] args)
    {
        var defaultfoo = new Foo(); // Call default constructor
        var foo = new Foo(14); // Call first constructor
        var foo2 = new Foo(12, 16); // Call overloaded constructor
    }
}
```

System.GC Type

The garbage collector is a common language runtime component that controls the allocation and release of managed memory. The methods in this class influence when garbage collection is performed on an object and when resources allocated by an object are released.

The garbage collector will destroy the object when it is no longer needed.

The base class libraries provide a class type named System.GC allows you to programmatically interact with the garbage collector using a set of static members.

System.GC Member	**Meaning in Life**
AddMemoryPressure()	Informs the runtime of a large allocation of unmanaged memory that should be taken into account when scheduling garbage collection.
RemoveMemoryPressure()	Informs the runtime that unmanaged memory has been released and no longer needs to be taken into account when scheduling garbage collection.
Collect()	Forces the GC to perform a garbage collection.
CollectionCount()	Returns the number of times garbage collection has occurred for the specified generation of objects.
GetGeneration()	Returns the current generation number of the specified object.
GetTotalMemory()	Retrieves the number of bytes currently thought to be allocated. A parameterindicates whether this method can wait a short interval before returning, to allow the system to collect garbage and finalize objects.
MaxGeneration	Returns the maximum ofgenerations supported on the target system
SuppressFinalize()	Requests that the common language runtime not call the finalizer for the specified object.
WaitForPendingFinalizers()	Suspends the current thread until the thread that is processing the queue of finalizers has emptied that queue.

A program that demonstrates the number of heap generations in garbage collection using the **GC.MaxGeneration** property of the GC class is given as follows:

```
using System;
public class Demo
{
   public static void Main(string[] args)
   {
      Console.WriteLine("The number of generations are: " +  GC.MaxGeneration);
       Console.ReadLine();
    }
  }
```

In the above program, the ***GC.MaxGeneration*** property is used to find the maximum number of generations that are **supported by the system** i.e. 2.

- **GC.GetGeneration() Method :** This method returns the generation number of the target object. It requires a single parameter i.e. the target object for which the generation number is required.

  A program that demonstrates the **GC.GetGeneration()** method is given as follows:

```
using System;
public class Demo
{
public static void Main(string[] args)
    {
         Demo obj = new Demo();
         Console.WriteLine("The generation number of object obj is: "
        + GC.GetGeneration(obj));
         Console.ReadLine();
    }
}
```

Output:

```
The generation number of object obj is: 0
```

- **GC.GetTotalMemory() Method :** This method returns the number of bytes that are allocated in the system. It requires a single boolean parameter where true means that the method waits for the occurrence of garbage collection before returning and false means the opposite.
  A program that demonstrates the *GC.GetTotalMemory() method* is given as follows:

```
using System;
public class Demo
{
public static void Main(string[] args)
   {
          Console.WriteLine("Total Memory:" + GC.GetTotalMemory(false));
          Demo obj = new Demo();
          Console.WriteLine("The generation number of object obj is: "
          + GC.GetGeneration(obj));
          Console.WriteLine("Total Memory:" + GC.GetTotalMemory(false));
          Console.ReadLine();
    }
}
```

Output:

```
Total Memory:4197120
The generation number of object obj is: 0
Total Memory:4204024
```

- **GC.Collect() Method :** Garbage collection can be forced in the system using the *GC.Collect() method*. This method requires a single parameter i.e. number of the oldest generation for which garbage collection occurs.
  A program that demonstrates the *GC.Collect() Method* is given as follows:

```
using System;
public class Demo
{
public static void Main(string[] args)
    {
          GC.Collect(0);
          Console.WriteLine("Garbage Collection in Generation 0 is: "
          + GC.CollectionCount(0));
          Console.ReadLine();
     }
}
```

Output:

```
Garbage Collection in Generation 0 is: 1
```

ET Assemblies

An assembly is the compiled format of any .Net program which may be .dll or .exe. An Assembly is a basic building block of .Net Framework applications. It is basically a compiled code that can be executed by the CLR.

An assembly is a collection of types and resources that are built to work together and form a logical unit of functionality. An Assembly can be a DLL or exe depending upon the project that we choose.

Assemblies are the fundamental units of deployment, version control, reuse, activation scoping, and security permissions for .NET-based applications. An assembly is a collection of types and resources that are built to work together and form a logical unit of functionality. Assemblies take the form of executable (.exe) or dynamic link library (.dll) files, and are the building blocks of .NET applications. They provide the common language runtime with the information it needs to be aware of type implementations.
The .Net Framework keeps executable code or DLL in the form of assembly.

- The .Net Framework maintains multiple versions of the application in the system through assembly.

- The assemblies have MSIL code and manifest that contains metadata.

- The metadata contains version information of the assembly.

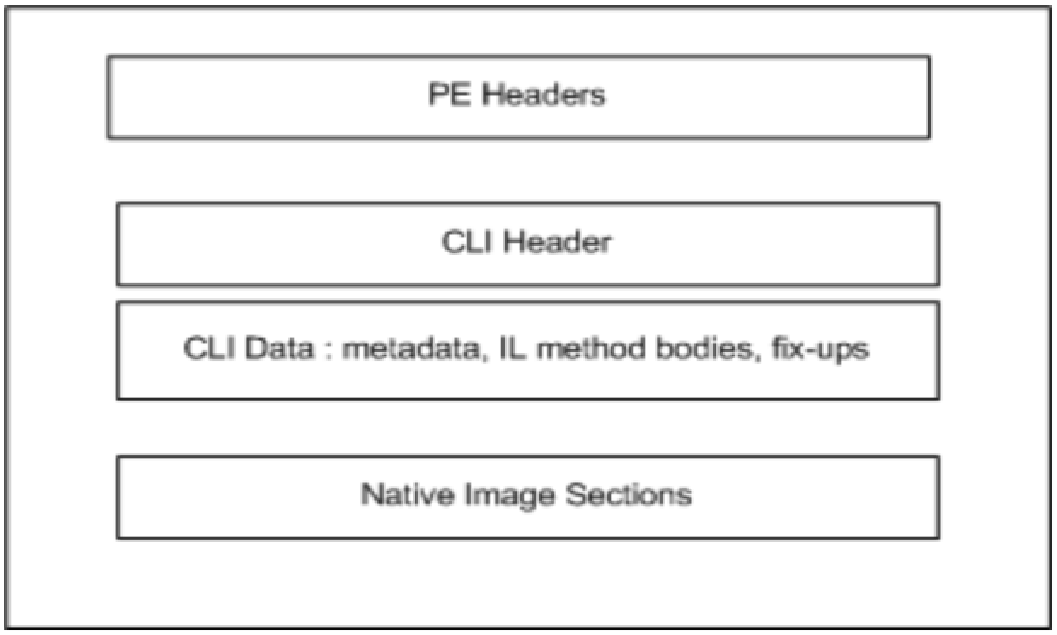
- Assemblies are the main building blocks.

- An assembly may be defined as a unit of deployment.

Format of .net assemblies

- NET defines a binary file format, *assembly*, that is used to fully describe and contain .NET programs.
- Assemblies are used for the programs themselves as well as any dependent libraries.
- .NET program can be executed as one or more assemblies, with no other required artifacts, beyond the appropriate .NET implementation.
- The format is fully specified and standardized as ECMA 335. All .NET compilers and runtimes use this format.

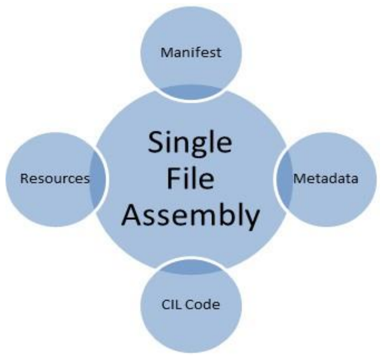
- The .NET binary format is based on the Windows PE file format. In fact, .NET class libraries are conformant Windows PEs, and appear on first glance to be Windows dynamic link libraries (DLLs) or application executables (EXEs).



- The format is fully specified and standardized as ECMA 335. All .NET compilers and runtimes use this format. The presence of a documented and infrequently updated binary format has been a major benefit (arguably a requirement) for interoperability. The format was last updated in a substantive way in 2005 (.NET Framework 2.0) to accommodate generics and processor architecture.
- The format is CPU- and OS-agnostic. It has been used as part of .NET implementations that target many chips and CPUs. While the format itself has Windows heritage, it is implementable on any operating system. Its arguably most significant choice for OS interoperability is that most values are stored in little-endian format. It doesn't have a specific affinity to machine pointer size (for example, 32-bit, 64-bit).
- The .NET assembly format is also very descriptive about the structure of a given program or library. It describes the internal components of an assembly, specifically assembly references and types defined and their internal structure. Tools or APIs can read and process this information for display or to make programmatic decisions.

Single File Assembly

- A single file assembly contains all the necessary elements such as CIL code, header files and manifests in a single *.exe or *. dll package.
- A project that has only one file is called single file assembly

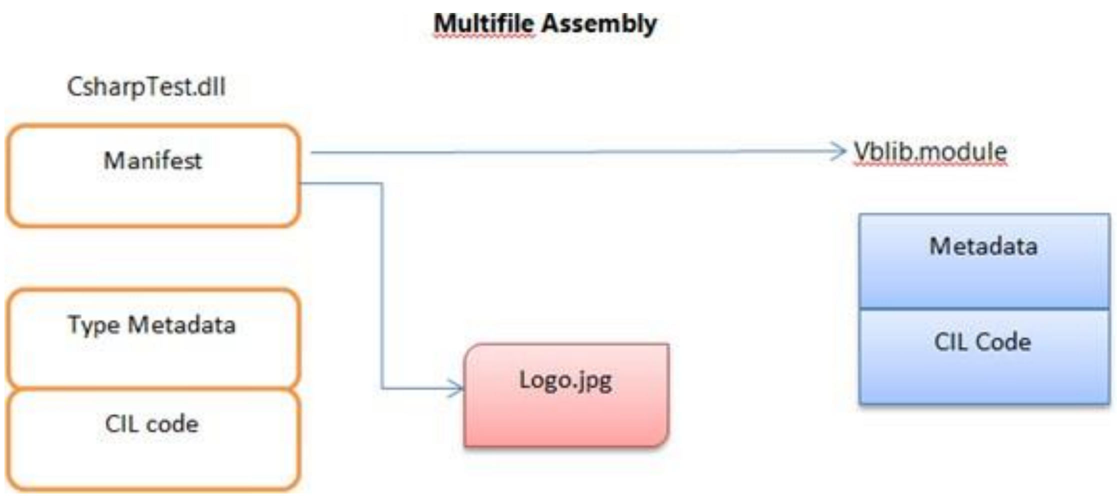


- Example:

```
Using System;
Namesapce SingleFileAssembly
{
  public class Employee
  {
      public static void main()
       {
         Console.Writeline("This is single file assembly");
         Console.ReadLine();
       }
    }
 }
```

MultiFile Assembly

A multi-file assembly, on the other hand, is a set of . NET modules that are deployed and versioned as a single unit.



Private and Shared Assembly

Assemblies are basically the following two types:

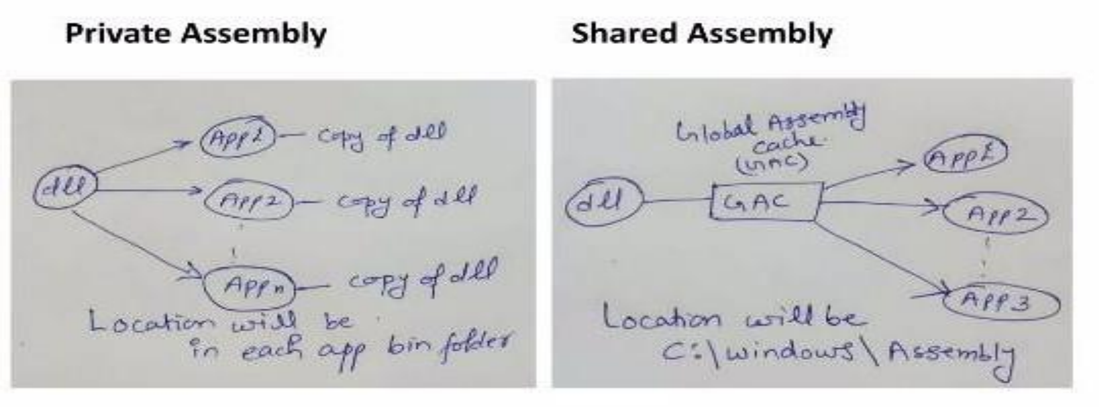
1. Private Assembly

Private Assemblies are designed to be used by one application and must reside in that application's directory or subdirectory.

It is an assembly that is being used by a single application only. Suppose we have a project in which we refer to a DLL so when we build that project that DLL will be copied to the bin folder of our project. That DLL becomes a private assembly within our project. Generally, the DLLs that are meant for a specific project are private assemblies.

2. Shared Assembly

Assemblies that can be used in more than one project are known to be shared assemblies. Shared assemblies are generally installed in the GAC. Assemblies that are installed in the GAC are made available to all the .Net applications on that machine.



Thank You

(Our Creative Info)