UNIT 4

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

In C#, exception is an event or object which is thrown at runtime. All exceptions the derived from *System.Exception* class. It is a runtime error which can be handled. If we don't handle the exception, it prints exception message and terminates the program. A C# exception is a responseto an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

Exception Handling using try catch

In C# Exception Handling is managed via four keywords:

1. try
2. catch
3. finally
4. throw

try − A try block identifies a block of code for which particular exceptions is activated. It isfollowed 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 israised or not.

throw − A program throws an exception when a problem shows up. This is done using a throwkeyword.

Syntax:

```
try
{
    // statements that may cause an exception
}
catch( ExceptionTypeobj)
{
    // statements for handling the exceptions
}
finally{
    // default code that is executed
}
```

```
using System;

namespace ex
{
    class Program
    {
       static void Main(string[] args)
       {
                int number1 = 3000;
                int number2 = 0;
          try
          {
                Console.WriteLine(number1 / number2);
          }
          catch (DivideByZeroException e)
          {
                Console.WriteLine(e.Message);

          }
          finally
          {
                Console.WriteLine("it always executes");
          }
       }
    }
```

Raising exceptions using throw

An exception can be raised manually by using the throw keyword. Any type of exceptions which is derived from Exception class can be raised using the throw keyword.Throw statement is used for throwing exception in a program. The throwing exception is handled by catch blockException objects that describe an error are created and then *thrown* with
the throw keyword. The runtime then searches for the most compatible exception handler.

```
catch(Exception e)
{
  ...
  throw e
}
```

```
using System; namespace exthrow
{
class Program
{
static void Main(string[] args)
{

        int num1, num2, result; Console.WriteLine("Enter First Number");
        num1 = Convert.ToInt32(Console.ReadLine());
        Console.WriteLine("Enter Second Number");
        num2 = Convert.ToInt32(Console.ReadLine());
try
{
if (num2 == 0)
{
        throw new Exception("Can't Divide by Zero Exception\n\n");
}
        result = num1 / num2;
        Console.WriteLine("{0} / {1} = {2}", num1, num2, result);
}
catch (Exception e)
{
        Console.WriteLine( "In catch block"+ e.ToString());
}
        Console.ReadLine();
}
}
}
```

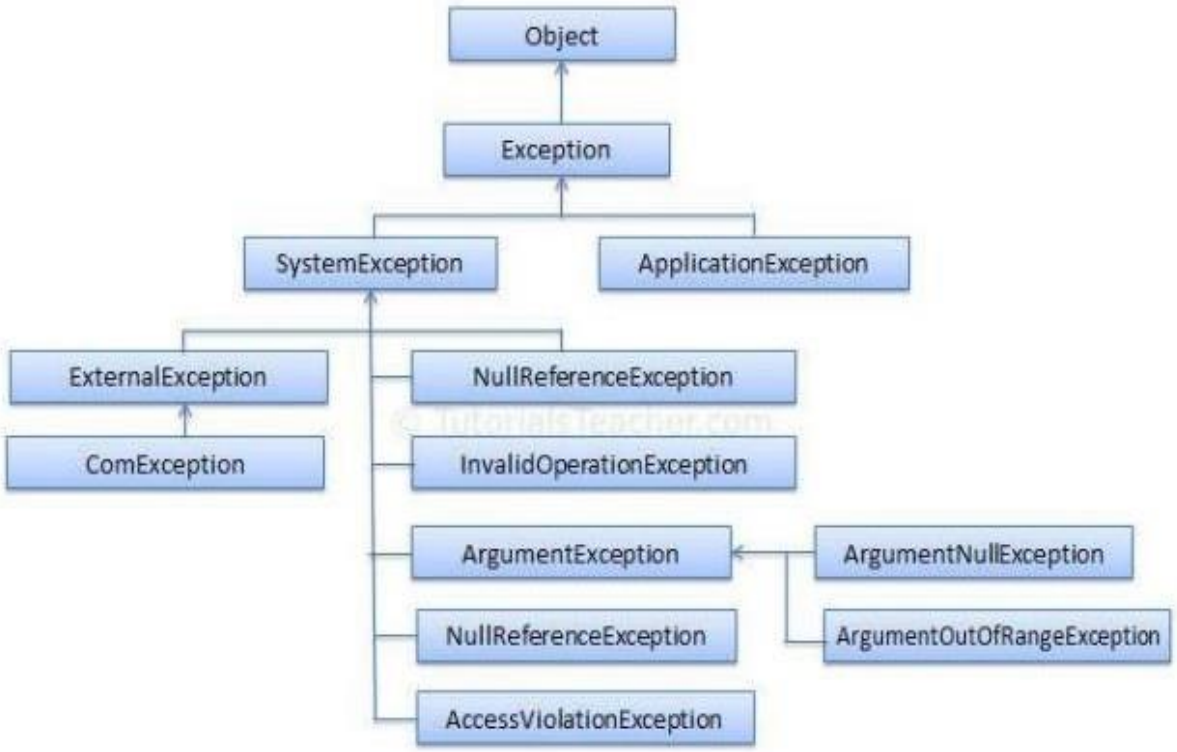
```
Enter     First
Number2
Enter     Second
Number0
In catch blockSystem.Exception: Can't Divide by Zero Exception
at exthrow.Program.Main(String[] args) in D:\C# Lab\exthrow\exthrow\Program.cs:line 21
```

Predefined Exception Classes in C#

C# exceptions are represented by classes. The exception classes in C# are mainly directly or indirectly derived from the **System.Exception** class. Some of the exception classes derived from the System.Exception class are the **System.ApplicationException** and **System.SystemException** classes.

The **System.ApplicationException** class supports exceptions generated by application programs. Hence the exceptions defined by the programmers should derive from this class.

The **System.SystemException** class is the base class for all predefined system exception.



Exception Class	Description
ArgumentException	Raised when a non-null argument that is passed to a method is invalid.
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.

Custom Exception Classes

An exception that is raised explicitly under a program based on our own condition (i.e. user-defined condition) is known as an custom exception.

For Implementing Custom Exception Handling, we need to derive the class CustomException from the system Base Class ApplicationException. Any Custom Exception you create needs to derive from the System.Exception class.

What are the Different Ways to Create Custom Exception in C#?

To create and throw an object of exception class by us, we have two different options.

1. Create the object of a predefined Exception class where we need to pass the error message as a parameter to its constructor and then throw that object so that whenever the exception occurs the given error message gets displayed.
2. Define a new class of type exception and throw that class object by creating it.

To define an exception class of our own we have to follow two steps

Step1: Define a new class inheriting from the predefined class Exception so that the new class also acts as an Exception class.
Step2: Now override the virtual property message with the required error message.

```
using System;
namespace ConsoleApplication9
{
   class TestTemperature
   {
      static void Main(string[] args)
      {
         Temperature temp = new
         Temperature();
          try
          {
               temp.showTemp();
          }
          catch (TempIsZeroException e)
          {
              Console.WriteLine("TempIsZeroException: {0}", e.Message);
           }
           Console.ReadKey();
       }
    }
}
```

```
public class TempIsZeroException : Exception
{
   public TempIsZeroException(string message):base(message)
   {
   }
}

public class Temperature
{
   int temperature = 0;

   public void showTemp()
   {
        if (temperature == 0)
        {
            throw (new TempIsZeroException("Zero Temperature found"));
        }
        else
        {
            Console.WriteLine("Temperature: {0}", temperature);
        }
   }
}
```

Output:

```
TempIsZeroException: Zero Temperature found
```

Understanding Object Lifetime classes:

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.

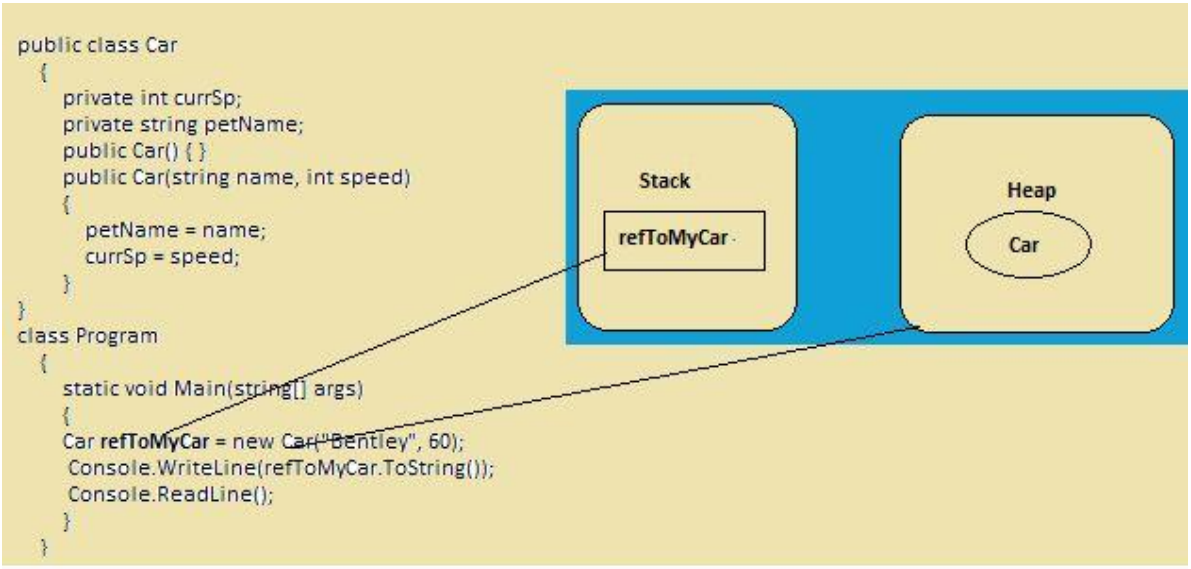


Diagram - new keyword returns a reference to the object on the heap and the actual reference variable is stored on stack.

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 key pillar of the .NET Framework is the automatic garbage collection that manages memory for all .NET applications. When an object is instantiated the garbage collector will destroy the object when it is no longer needed. There is no explicit memory deallocation since the garbage collector monitors unused objects and does a collection to free up memory that is an automatic process. The Garbage Collector removes objects from the heap when they are unreachable by any part of your codebase. The .Net garbage collector will compact empty blocks of memory for the purpose of optimization.

The heap is categorized into three generations so it can handle long-lived and short-lived objects. Garbage collection primarily occurs with the reclamation of short-lived objects that typically occupy only a small part of the heap.

Classes, Objects, and References

It is important to further clarify the distinction between classes, objects, and reference variables. Recall that a class is nothing more than a blueprint that describes how an instance of this type will look and feel in memory.

Classes, of course, are defined within a code file (which in C# takes a *.cs extension by convention). Consider the following simple Car class defined within a new C# application project named SimpleGC:

```
// Car.cs

 public class Car
{
   public int CurrentSpeed { get; set; }
   public string PetName { get; set; }
   public Car() { }
   public Car(string name, int speed)
   {
```

```
        PetName = name;
        CurrentSpeed = speed;
    }

    public override string ToString()
    {
        return string.Format("{0} is going {1} MPH",
        PetName, CurrentSpeed);
    }
}
```

After a class has been defined, you may allocate any number of objects using the C# "new" keyword. Understand, however, that the new keyword returns a reference to the object on the heap, not the actual object itself. If you declare the reference variable as a local variable in a method scope then it is stored on the stack for further use in your application. When you want to invoke members on the object, apply the C# dot operator to the stored reference, like so:

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** GC Basics *****");
        // Create a new Car object on
        // the managed heap. We are
        // returned a reference to this
        // object ("refToMyCar").
        Car refToMyCar = new Car("Zippy", 50);
        // The C# dot operator (.) is used
        // to invoke members on the object
        // using our reference variable.
        Console.WriteLine(refToMyCar.ToString());
        Console.ReadLine();
    }
}
```

The below Figure illustrates the class, object, and reference relationship.

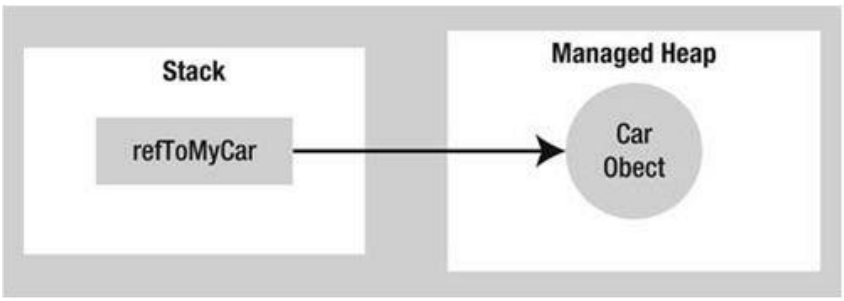


Figure: References to objects on the managed heap

Structures are value types that are always allocated directly on the stack and are never placed on the .NET managed heap. Heap allocation occurs only when you are creating instances of classes.

The Basics of Object Lifetime

When you are building your C# applications, you are correct to assume that the .NET runtime environment (a.k.a. the CLR) will take care of the managed heap without your direct intervention. In fact, the **golden rule** of .NET memory management is simple:

"*Allocate a class instance onto the managed heap using the new keyword and forget about it.*"

Once instantiated, the garbage collector will destroy an object when it is no longer needed. The next obvious question, of course, is, "How does the garbage collector determine when an object is no longer needed?" The short (i.e., incomplete) answer is that the garbage collector removes an object from the heap only if it is unreachable by any part of your code base. Assume you have a method in your Program class that allocates a local Car object as follows:

```
static void MakeACar()
{
        // If myCar is the only reference to the Car object,
        // it *may* be destroyed when this method returns.
        Car myCar = new Car();
}
```

Notice that this Car reference (myCar) has been created directly within the MakeACar() method and has not been ed outside of the defining scope (via a return value or ref/out parameters). Thus, once this method call completes, the myCar reference is no longer reachable, and the associated Car object is now a candidate for garbage collection. Understand, however, that you can't guarantee that this object will be reclaimed from memory immediately after MakeACar() has completed. All you can assume at this point is that when the CLR performs the next garbage collection, the myCar object could be safely destroyed.

As you will most certainly discover, programming in a garbage-collected environment greatly simplifies your application development. In stark contrast, C++ programmers are painfully aware that if they fail to manually delete heap-allocated objects, memory leaks are never far behind. In fact, tracking down memory leaks is one of the most time-consuming (and tedious) aspects of programming in unmanaged environments. By allowing the garbage collector to take charge of destroying objects, the burden of memory management has been lifted from your shoulders and placed onto those of the CLR.

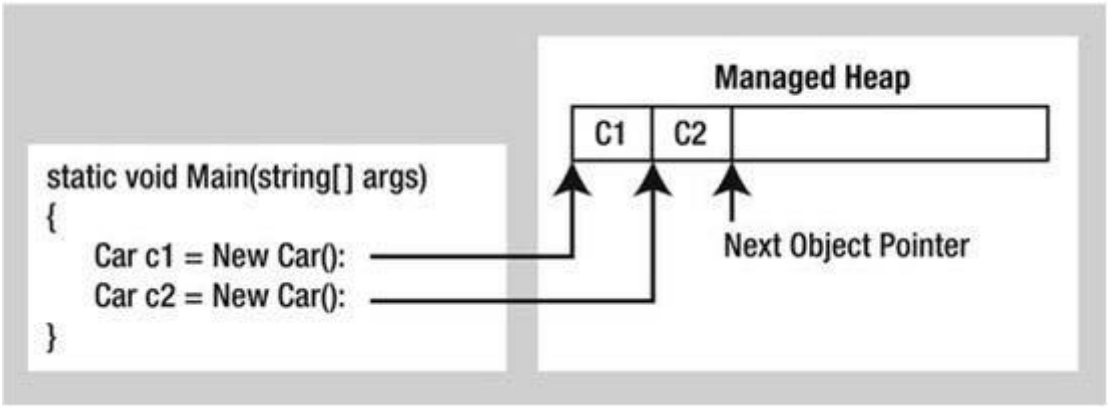
The CIL of new

When the C# compiler encounters the new keyword, it emits a CIL newobj instruction into the method implementation. If you compile the current example code and investigate the resulting assembly using ildasm.exe, you'd find the following CIL statements within the MakeACar() method:

```
 .method private hidebysig
static void MakeACar() cil managed
{
// Code size 8 (0x8)
.maxstack 1
.locals init ([0] class SimpleGC.Car myCar)
IL_0000: nop
IL_0001: newobj instance void SimpleGC.Car::.ctor()
IL_0006: stloc.0
IL_0007: ret
} // end of method Program::MakeACar
```

Before we examine the exact rules that determine when an object is removed from the managed heap, let's check out the role of the CIL newobj instruction in a bit more detail. First, understand that the managed heap is more than just a random chunk of memory accessed by the CLR. The .NET garbage collector is quite a tidy housekeeper of the heap, given that it will compact empty blocks of memory (when necessary) for purposes of optimization. To aid in this endeavor, the managed heap maintains a pointer (commonly referred to as the next object pointer or new object pointer) that identifies exactly where the next object will be located. That said, the newobj instruction tells the CLR to perform the following core operations:

- Calculate the total amount of memory required for the object to be allocated (including the memory required by the data members and the base classes).
- Examine the managed heap to ensure that there is indeed enough room to host the object to be allocated. If there is, the specified constructor is called and the caller is ultimately returned a reference to the new object in memory, whose address just happens to be identical to the last position of the next object pointer.
- Finally, before returning the reference to the caller, advance the next object pointer to point to the next available slot on the managed heap.



The details of allocating objects onto the managed heap

As your application is busy allocating objects, the space on the managed heap may eventually become full. When processing the newobj instruction, if the CLR determines that the managed heap does not have sufficient memory to allocate the requested type, it will perform a garbage collection in an attempt to free up memory. Thus, the **next rule** of garbage collection is also quite simple:
"*If the managed heap does not have sufficient memory to allocate a requested object, a garbage collection will occur*."

Exactly how this garbage collection occurs, however, depends on which version of the .NET platform your application is running under.

System.GC Class

A garbage collection consists of the following steps:

1. The garbage collector searches for managed objects that are referenced in managed code.
2. The garbage collector attempts to finalize unreferenced objects.
3. The garbage collector frees unreferenced objects and reclaims their memory.

Implementations of the garbage collector should track the following information:

- Memory allocated to objects that are still in use
- Memory allocated to objects that are no longer in use
- Objects that require finalization prior to being freed

GC.Collect Method

Requests that the garbage collector reclaim memory allocated to objects for which there are no valid references.

Assemblies and its Role:
An assembly is a file that is automatically generated by the compiler upon successful compilation of every .NET application. It can be either a Dynamic Link Library or an executable file. It is generated only once for an application and upon each subsequent compilation the assembly gets updated.

OR

An Assembly is a collection of logical units. Logical units refer to the types and resources which are required to build an application and deploy them using the .Net framework. Basically, Assembly is a collection of Exe and DLLs. It is portable and executable.

The terms DLL and EXE are used commonly in programming and software development. When a developer is done building an application or project, they can either export their software as an EXE or DLL.

EXE is a short form for the word ‘executable’. In any .NET application package, there will be at least one EXE file that may or may not complement one or more DLL files. Exe files have a part in the code from where the program execution starts or, put another way, has an entry point for the execution of a program. Launching an EXE file creates its own memory space by the operating system.

On the flip side, DLL is an acronym for Dynamic Link Library, which contains functions and procedures that can be used by EXE files or libraries. As DLL is a library, you cannot execute it directly. Doing so will result in an error. Instead, a programmer can call a DLL from another program if he knows the function name and its signature in the DLL file.

Because of this capability, DLLs are a perfect fit for distributing device driver software. Unlike an EXE, a DLL does not create its own memory space; rather, they simply share the memory space of the calling application.

Understanding the Format of a .NET Assembly
Structurally speaking, a .NET assembly (*.dll or *.exe) consists of the following elements:

- A Windows file header
- A CLR file header
- CIL code
- Type metadata
- An assembly manifest
- Optional embedded resources

The Win32 File Header

The Win32 file header establishes the fact that the assembly can be loaded and manipulated by the Windows family of operating systems. This header data also identifies the kind of application (consolebased, GUI-based, or *.dll code library) to be hosted by the Windows operating system.

The CLR File Header

The CLR header is a block of data that all .NET files must support in order to be hosted by the CLR. In a nutshell, this header defines numerous flags that enable the runtime to understand the layout of the managed file. For example, flags exist that identify the location of the metadata and resources within the file, the version of the runtime the assembly was built against, the value of the (optional) public key, and so forth.

CIL Code, Type Metadata, and the Assembly Manifest

At its core, an assembly contains CIL code, which as you recall is a platform- and CPU-agnostic intermediate language. At runtime, the internal CIL is compiled on the fly (using a just-in-time [JIT] compiler) to platform- and CPU-specific instructions.

An assembly also contains metadata that completely describes the format of the contained types as well as the format of external types referenced by this assembly. The .NET runtime uses this metadata to resolve the location of types (and their members) within the binary, lay out types in memory, and facilitate remote method invocations.

An assembly must also contain an associated manifest (also referred to as assembly metadata).The manifest documents each module within the assembly, establishes the version of the assembly, and also documents any external assemblies referenced by the current assembly.

Optional Assembly Resources

Finally, a .NET assembly may contain any number of embedded resources such as application icons, image files, sound clips, or string tables.

Single-File and Multifile Assemblies

Technically speaking, an assembly can be composed of multiple *modules*. A module is really nothing more than a generic term for a valid .NET binary file. In most situations, an assembly is in fact

composed of a single module. In this case, there is a one-to-one correspondence between the (logical) assembly and the underlying (physical) binary (hence the term *single-file assembly*).

Single-file assemblies contain all of the necessary elements (header information, CIL code, type metadata, manifest, and required resources) in a single *.exe or *.dll package. Figure 5-1 illustrates the composition of a single-file assembly.

Figure 5-1: A single-file assembly

A multifile assembly, on the other hand, is a set of .NET *.dlls that are deployed and versionedas a single logic unit. Formally speaking, one of these *.dlls is termed the primary module and contains the assembly-level manifest (as well as any necessary CIL code, metadata, header information, and optional resources). The manifest of the primary module records each of the related *.dll files it is dependent upon.

As a naming convention, the secondary modules in a multifile assembly take a *.netmodule file extension; however, this is not a requirement of the CLR. Secondary *.netmodules also contain CIL code and type metadata, as well as a module-level manifest, which simply records the externally required assemblies of that specific module.

The major benefit of constructing multifile assemblies is that they provide a very efficient way to download content. For example, assume you have a machine that is referencing a remote multifile assembly composed of three modules, where the primary module is installed on the client. If the client requires a type within a secondary remote *.netmodule, the CLR will download the binary to the local machine on demand to a specific location termed the download cache.

Another benefit of multifile assemblies is that they enable modules to be authored using multiple

.NET programming languages Once each of the individual modules has been compiled, the modules can be logically “connected” into a logical assembly using tools such as the assembly linker (al.exe). Figure 11-4 illustrates a multifile assembly composed of three modules, each authored using a unique .NET programming language.

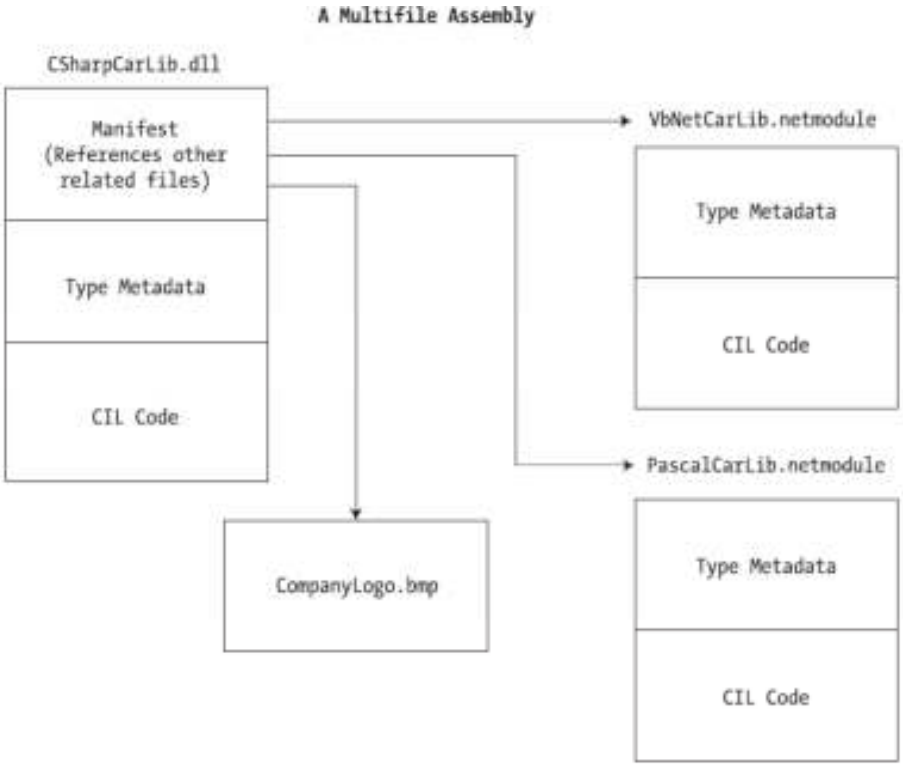


Figure 5-2. The primary module records secondary modules in the assembly manifest

Assembly is a compiled code and logical unit of code. There are two types of assemblies available in .Net.

- Process Assemblies: I'ts having an extension of .exe
- Library Assemblies: It's having an extension of .dll.

Library Assembly is further divided into the following types:

- Private Assembly
- Public or Shared Assembly
- Satellite Assembly

Private Assembly

Private assembly requires us to copy separately in all application folders where we want to use that assembly’s functionalities; without copying, we cannot access the private assembly features and power. Private assembly means every time we have one, we exclusively copy into the BIN folder of each application folder.

OR

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.

Public/Shared Assembly

Public assembly is not required to copy separately into all application folders. Public assembly is also called Shared Assembly. Only one copy is required at the system level, there is no need to copy the assembly into the application folder.

OR

Assemblies that can be used in more than one project are known to be a shared assembly. 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.

Public assembly should install in GAC.

GAC (Global Assembly Cache)

When the assembly is required for more than one project or application, we need to make the assembly with a strong name and keep it in GAC or in the Assembly folder by installing the assembly with the GACUtil command.

Satellite Assembly:

Satellite assemblies are used for deploying language and culture-specific resources for an application.

Difference between Private and Shared Assemblies

Private Assembly:

1. Private assembly can be used by only one application.
2. Private assembly will be stored in the specific application's directory or sub-directory.
3. There is no other name for private assembly.
4. Strong name is not required for private assembly.
5. Private assembly doesn't have any version constraint.

Shared Assembly:

1. Shared assembly can be used by multiple applications.
2. Shared assembly is stored in GAC (Global Assembly Cache).
3. Public assembly is also termed as shared assembly.
4. Strong name has to be created for public assembly.
5. Public assembly should strictly enforce version constraint.