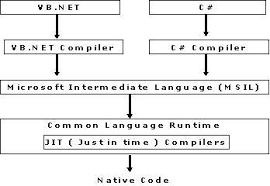
MSIL

Microsoft Intermediate Language (MSIL) is a language used as the output of a number of compilers (C#, VB, .NET, and so forth). **The ILDasm** (Intermediate Language Disassembler) program that ships with the .NET Framework SDK (FrameworkSDK\Bin\ildasm.exe) allows the user to see MSIL code in human-readable format. By using this utility, we can open any .NET executable file (EXE or DLL) and see MSIL code.

The **ILAsm** program (Intermediate Language Assembler) generates an executable file from the MSIL language. We can find this program in the WINNT\Microsoft.NET\Framework\vn.nn.nn directory.

Any Visual C++ programmer starting with .NET development is interested in what happens in the low level of the .NET Framework. Learning MSIL gives a user the chance to understand some things that are hidden from a programmer working with C# or VB.NET. Knowing MSIL gives more power to a .NET programmer. We never need to write programs in MSIL directly, but in some difficult cases it is very useful to open the MSIL code in ILDasm and see how things are done.



A MSIL reference in DOC format is available to a .NET developer and may be found in the Framework SDK directory:

* FrameworkSDK\Tool Developers Guide\docs\Partition II Metadata.doc **(Metadata Definition and Semantics)**. In this file, I found a description of all MSIL directives such as **.entrypoint**, **.locals**, and so on.
* FrameworkSDK\Tool Developers Guide\docs\Partition III CIL.doc **(CIL Instruction Set)** contains a full list of the MSIL commands.

I also used in my work in an ILDAsm tutorial from MSDN and an excellent article in the May 2001 issue of MSDN Magazine: "ILDASM is Your New Best Friend" by **John Robbins**.

I think the best way to learn the language is to write some programs in it. This is a reason I decided to make several small MSIL programs. Actually, I didn't write this code—the C# compiler generated it. I made some minor changes and added a lot of notes describing how MSIL is working.

Reading the sample projects attached to this article may help a .NET programmer understand Intermediate Language and easily read MSIL code when this is necessary.

**General Information**

All operations in MSIL are executed on the stack. When a function is called, its parameters and local variables are allocated on the stack. Function code starting from this stack state may push some values onto the stack, make operations with these values, and pop values from the stack.

Execution of both MSIL commands and functions is done in three steps:

1. Push command operands or function parameters onto the stack.
2. Execute the MSIL command or call function. The command or function pops their operands (parameters) from the stack and pushes onto the stack result (return value).
3. Read result from the stack.

Steps 1 and 3 are optional. For example, the **void** function doesn't push a return value to the stack.

The stack contains objects of value types and references to objects of reference type. Reference type objects are kept in the heap.

MSIL commands used to push values onto the stack are called **ld...** (load). Commands used to pop values from the stack are called **st...** (store), because values are stored in variables. Therefore, we will call the **push** operation **loading** and the **pop** operation **storing**.

JIT

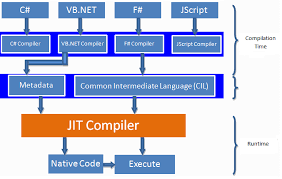
In the .NET Framework, all the Microsoft .NET languages use a Common Language Runtime, which solves the problem of installing separate runtimes for each of the programming languages. When the Microsoft .NET Common Language Runtime is installed on a computer then it can run any language that is Microsoft .NET compatible. Before the Microsoft Intermediate Language (MSIL) can be executed, it must be converted by a .NET Framework Just-In-Time (JIT) compiler to native code, which is CPU-specific code that runs on the same computer architecture as the JIT compiler.

## **JIT (JUST-IN-TIME) COMPILER**

A Web Service or Web Forms file must be compiled to run within the CLR. Compilation can be implicit or explicit. Although you could explicitly call the appropriate compiler to compile your Web Service or Web Forms files, it is easier to allow the file to be complied implicitly. Implicit compilation occurs when you request the .asmx via HTTP-SOAP, HTTP-GET, or HTTP-POST. The parser (xsp.exe) determines whether a current version of the assembly resides in memory or in the disk. If it cannot use an existing version, the parser makes the appropriate call to the respective compiler (as you designated in the Class property of the .asmx page).

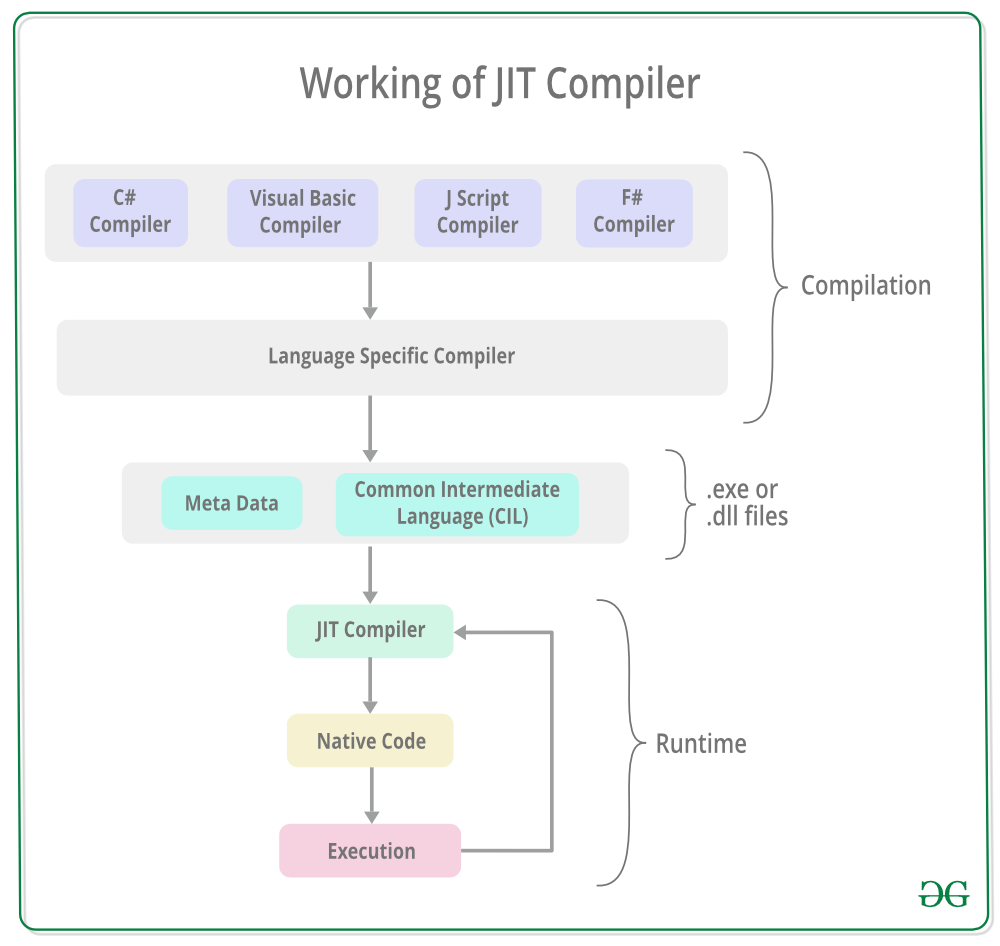
When the Web Service (or Web Forms page) is implicitly compiled, it is actually compiled twice. On the first pass, it is compiled into IL. On the second pass, the Web Service (now an assembly in IL) is compiled into machine language. This process is called Just-In-Time JIT compilation because it does not occurs until the assembly is on the target machine. The reason you do not compile it ahead of time is so that the specific JITter for your OS and processor type can be used. As a result, the assembly is compiled into the fastest possible machine language code, optimized and enhanced for your specific configuration. It also enables you to compile once and then run on any number of operating systems.

## **How JIT Works?**



Before MSIL(MS Intermediate Language) can be executed, it must converted by .net Framework Just in time (JIT) compiler to native code, which is CPU specific code that run on some computer architecture as the JIT compiler. Rather than using time and memory to convert all the MSIL in portable executable (PE) file to native code, it converts the MSIL as it is needed during execution and stored in resulting native code so it is accessible for subsequent calls.

The runtime supplies another mode of compilation called install-time code generation. The install-time code generation mode converts MSIL to native code just as the regular JIT compiler does, but it converts larger units of code at a time, storing the resulting native code for use when the assembly is subsequently loaded and executed. As part of compiling MSIL to native code, code must pass a verification process unless an administrator has established a security policy that allows code to bypass verification. Verification examines MSIL and metadata to find out whether the code can be determined to be type safe, which means that it is known to access only the memory locations it is authorized to access.



## **JIT Types**

In Microsoft .NET there are three types of JIT (Just-In-Time) compilers which are Explained as Under,

* Pre-JIT Compiler (Compiles entire code into native code completely)
* Econo JIT Compiler (Compiles code part by part freeing when required)
* Normal JIT Compiler (Compiles only that part of code when called and places in cache

## **Description**

* Pre-JIT COMPILER  
  Pre-JIT compiles complete source code into native code in a single compilation cycle. This is done at the time of deployment of the application.
* Econo-JIT COMPILER  
  Econo-JIT compiles only those methods that are called at runtime. However, these compiled methods are removed when they are not required.
* Normal-JIT COMPILER  
  Normal-JIT compiles only those methods that are called at runtime. These methods are compiled the first time they are called, and then they are stored in cache. When the same methods are called again, the compiled code from cache is used for execution.

These methods are compiled the first time they are called, and then they are stored in cache. When the same methods are called again, the compiled code from cache is used for execution.

Brought to you by:

# **Garbage Collection**

Automatic memory management is made possible by **Garbage Collection in .NET Framework**. When a class object is created at runtime, certain memory space is allocated to it in the heap memory. However, after all the actions related to the object are completed in the program, the memory space allocated to it is a waste as it cannot be used. In this case, garbage collection is very useful as it automatically releases the memory space after it is no longer required.

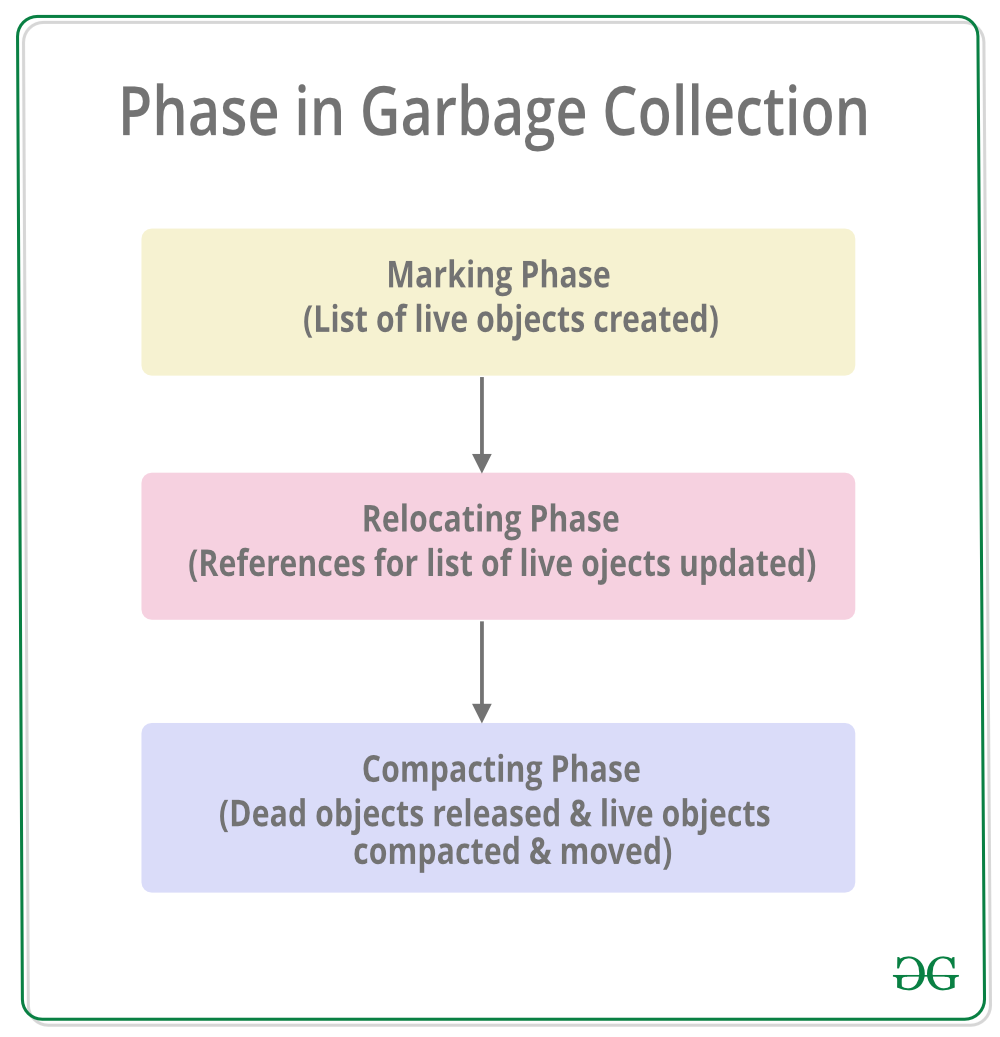
Garbage collection will always work on **Managed Heap** and internally it has an Engine which is known as the **Optimization Engine**.

Garbage Collection occurs if at least one of multiple conditions is satisfied. These conditions are given as follows:

* If the system has low physical memory, then garbage collection is necessary.
* If the memory allocated to various objects in the heap memory exceeds a pre-set threshold, then garbage collection occurs.
* If the GC.Collect method is called, then garbage collection occurs. However, this method is only called under unusual situations as normally garbage collector runs automatically.

#### Phases in Garbage Collection

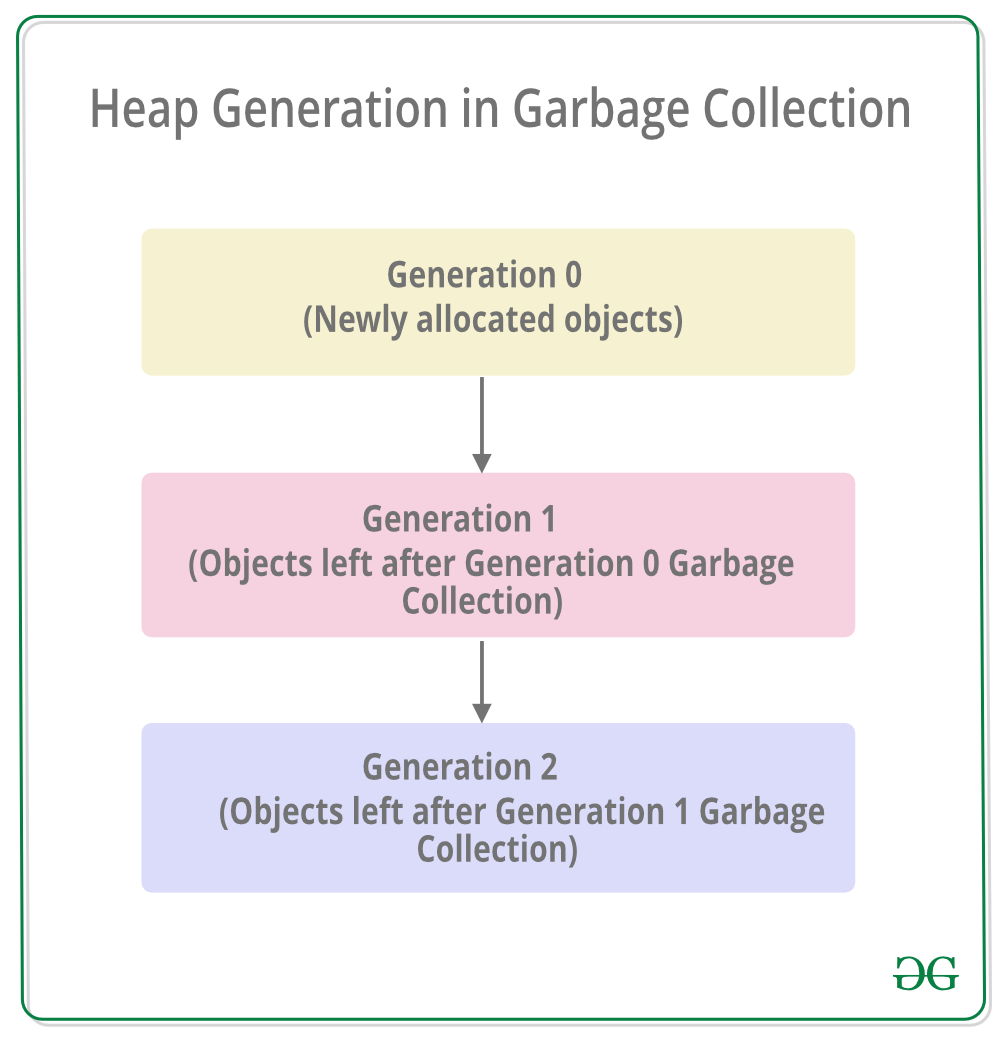
There are mainly **3** phases in garbage collection. Details about these are given as follows:

[](https://media.geeksforgeeks.org/wp-content/uploads/20190417112537/PhaseInGarbageCollection.png)

1. **Marking Phase:** A list of all the live objects is created during the marking phase. This is done by following the references from all the root objects. All of the objects that are not on the list of live objects are potentially deleted from the heap memory.
2. **Relocating Phase:** The references of all the objects that were on the list of all the live objects are updated in the relocating phase so that they point to the new location where the objects will be relocated to in the compacting phase.
3. **Compacting Phase:** The heap gets compacted in the compacting phase as the space occupied by the dead objects is released and the live objects remaining are moved. All the live objects that remain after the garbage collection are moved towards the older end of the heap memory in their original order.

#### Heap Generations in Garbage Collection

The heap memory is organized into 3 generations so that various objects with different lifetimes can be handled appropriately during garbage collection. The memory to each Generation will be given by the [**Common Language Runtime(CLR)**](https://www.geeksforgeeks.org/common-language-runtime-clr-in-c-sharp/) depending on the project size. Internally, Optimization Engine will call the Collection Means Method to select which objects will go into Gneration 1 or Generation 2.

[](https://media.geeksforgeeks.org/wp-content/uploads/20190417112732/HeapGenerationInGarbageCollection.png)

* **Generation 0 :**All the short-lived objects such as temporary variables are contained in the generation 0 of the heap memory. All the newly allocated objects are also generation 0 objects implicitly unless they are large objects. In general, the frequency of garbage collection is the highest in generation 0.
* **Generation 1 :**If space occupied by some generation 0 objects that are not released in a garbage collection run, then these objects get moved to generation 1. The objects in this generation are a sort of buffer between the short-lived objects in generation 0 and the long-lived objects in generation 2.
* **Generation 2 :**If space occupied by some generation 1 objects that are not released in the next garbage collection run, then these objects get moved to generation 2. The objects in generation 2 are long lived such as static objects as they remain in the heap memory for the whole process duration.

Assembly

The .NET assembly is the standard for components developed with the Microsoft.NET. Dot NET assemblies may or may not be executable, i.e., they might exist as the executable (.exe) file or dynamic link library (DLL) file. All the .NET assemblies contain the definition of types, versioning information for the type, meta-data, and manifest. The designers of .NET have worked a lot on the component (assembly) resolution.

An assembly can be a single file or it may consist of the multiple files. In the case of multi-file, there is one master module containing the manifest while other assemblies exist as non-manifest modules. A module in .NET is a subpart of a multi-file .NET assembly. Assembly is one of the most interesting and extremely useful areas of .NET architecture along with reflections and attributes.

.NET supports three kinds of assemblies:

1. private
2. shared
3. satellite

**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.

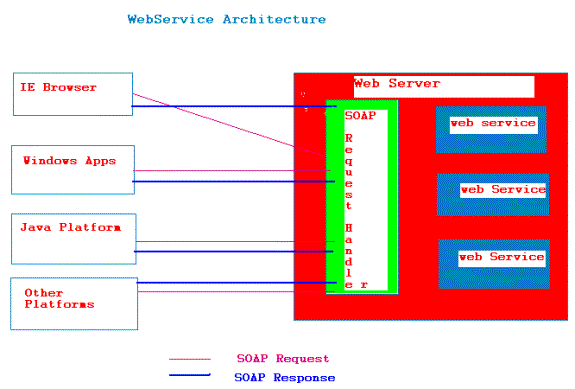
**Public Assembly**

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

Public assembly should install in GAC.

Shared assemblies (also called strong named assemblies) are copied to a single location (usually the Global assembly cache). For all calling assemblies within the same application, the same copy of the shared assembly is used from its original location. Hence, shared assemblies are not copied in the private folders of each calling assembly. Each shared assembly has a four-part name including its face name, version, public key token, and culture information. The public key token and version information makes it almost impossible for two different assemblies with the same name or for two similar assemblies with a different version to mix with each other.

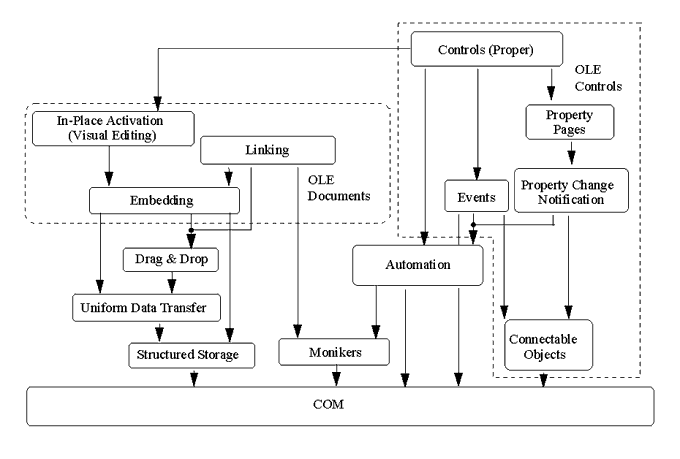
**Web Services**Web service is simply an application that exposes a Web-accessible API. That means you can invoke this application programmatically over the Web. Using SOAP Protocol .  
  
**XML+HTTP =SOAP (Simple Object Access protocol)**Web services allow applications to share data.    
  
Web services can be called across platforms and operating systems regardless of programming language.   
  
.NET is Microsoft's platform for XML Web services.   
  
Using SOAP you can invoke the WebSerices. (HTTP-GET, HTTP-POST, SOAP)   
  
SOAP -Simple Object Access protocol is message-based protocol. Based on Request and Response. Here I am not going to explain more details about SOAP.

**Web Service Application**Like Pay per view Channel we can make Software as pay per use, using Web service.  For example, let say Infovision Inc   developed Expensive Software for 3D Virtual Modeling. Lot of company does not want acquire the licensee because it is expensive and they need to pay for Support etc...   instead of this if Infovision makes this software as a WebService, most of the company will use as pay per use.    
  
Another example Credit card validation we can expose as a webservice. (Good example for webService).  
  
Using VisualStudio.NET we can easily create WebServices.  ASP.NET WebService this project type will be used to create WebService.  
  
**Web Services Architecture  
  
**

Simple Object Access Protocol

COM

The Component Object Model (COM) is a software architecture that allows applications to be built from binary software components. COM is the underlying architecture that forms the foundation for higher-level software services, like those provided by OLE. OLE services span various aspects of commonly needed system functionality, including compound documents, custom controls, interapplication scripting, data transfer, and other software interactions.



**Figure 1. OLE technologies build on one another, with COM as the foundation.**

These services provide distinctly different functionality to the user. However they share a fundamental requirement for a mechanism that allows binary software components, derived from any combination of pre-existing customers' components and components from different software vendors, to connect to and communicate with each other in a well-defined manner. This mechanism is supplied by COM, a software architecture that does the following:

* Defines a binary standard for component interoperability
* Is programming-language-independent
* Is provided on multiple platforms (Microsoft® Windows®, Windows 95, Windows NT™, Apple® Macintosh®, and many varieties of UNIX®)
* Provides for robust evolution of component-based applications and systems
* Is extensible by developers in a consistent manner
* Uses a single programming model for components to communicate within the same process, and also across process and network boundaries
* Allows for shared memory management between components
* Provides rich error and status reporting
* Allows dynamic loading and unloading of components

It is important to note that COM is a general architecture for component software. Although Microsoft is applying COM to address specific areas such as controls, compound documents, automation, data transfer, storage and naming, and others, any developer can take advantage of the structure and foundation that COM provides.

How does COM enable interoperability? What makes it such a useful and unifying model? To address these questions, it will be helpful to first define the basic COM design principles and architectural concepts. In doing so, we will examine the specific problems that COM is meant to solve, and how COM provides solutions for these problems.

**The Component Software Problem**

The most fundamental problem that COM solves is: How can a system be designed so that binary executables from different vendors, written in different parts of the world and at different times, are able to interoperate? To solve this problem, we must first find answers to these four questions:

* **Basic interoperability.** How can developers create their own unique binary components, yet be assured that these binary components will interoperate with other binary components built by different developers?
* **Versioning.** How can one system component be upgraded without requiring all the system components to be upgraded?
* **Language independence.** How can components written in different languages communicate?
* **Transparent cross-process interoperability.** How can we give developers the flexibility to write components to run in-process or cross-process, and even cross-network, using one simple programming model?

Additionally, high performance is a requirement for a component software architecture. Although cross-process and cross-network transparency is a laudable goal, it is critical for the commercial success of a binary component marketplace that components interacting within the same address space be able to use each other's services without any undue "system" overhead. Otherwise, the components will not realistically be scalable down to very small, lightweight pieces of software equivalent to C++ classes or graphical user interface (GUI) controls.

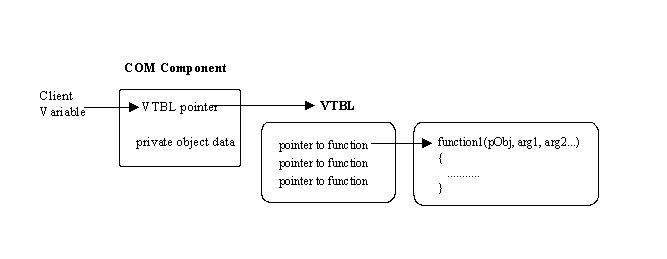
**COM Fundamentals**

The Component Object Model defines several fundamental concepts that provide the model's structural underpinnings. These include:

* A binary standard for function calling between components.
* A provision for strongly-typed groupings of functions into interfaces.
* A base interface providing:
* A way for components to dynamically discover the interfaces implemented by other components.
* Reference counting to allow components to track their own lifetime and delete themselves when appropriate.
* A mechanism to identify components and their interfaces uniquely, worldwide.
* A "component loader" to set up component interactions and, additionally (in the cross-process and cross-network cases), to help manage component interactions.

**Binary Standard**

For any given platform (hardware and operating system combination), COM defines a standard way to lay out virtual function tables (vtables) in memory, and a standard way to call functions through the vtables. Thus, any language that can call functions via pointers (C, C++, Smalltalk, Ada, and even BASIC) all can be used to write components that can interoperate with other components written to the same binary standard. Indirection (the client holds a pointer to a vtable) allows for vtable sharing among multiple instances of the same object class. On a system with hundreds of object instances, vtable sharing can reduce memory requirements considerably, because additional vtables pointing to the same component instance consume much less memory than multiple instances of the same component.



Localization

Developing a world-ready application, including an application that can be localized into one or more languages, involves three steps: globalization, localizability review, and localization.

Globalization

This step involves designing and coding an application that is culture-neutral and language-neutral, and that supports localized user interfaces and regional data for all users. It involves making design and programming decisions that are not based on culture-specific assumptions. While a globalized application is not localized, it nevertheless is designed and written so that it can be subsequently localized into one or more languages with relative ease.

Localizability review

This step involves reviewing an application's code and design to ensure that it can be localized easily and to identify potential roadblocks for localization, and verifying that the application's executable code is separated from its resources. If the globalization stage was effective, the localizability review will confirm the design and coding choices made during globalization. The localizability stage may also identify any remaining issues so that an application's source code doesn't have to be modified during the localization stage.

Localization

This step involves customizing an application for specific cultures or regions. If the globalization and localizability steps have been performed correctly, localization consists primarily of translating the user interface.

Following these three steps provides two advantages:

* It frees you from having to retrofit an application that is designed to support a single culture, such as U.S. English, to support additional cultures.
* It results in localized applications that are more stable and have fewer bugs.

.NET provides extensive support for the development of world-ready and localized applications. In particular, many type members in the .NET class library aid globalization by returning values that reflect the conventions of either the current user's culture or a specified culture. Also, .NET supports satellite assemblies, which facilitate the process of localizing an application.