CLR via C#

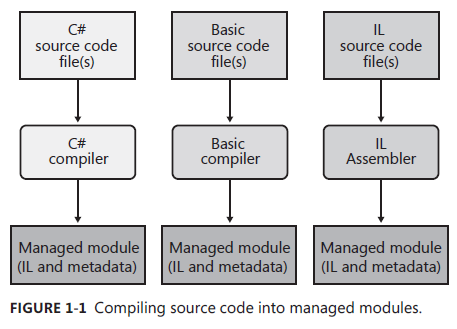
**CLR BASICS**

1. The CLR's Execution Model

CLR:

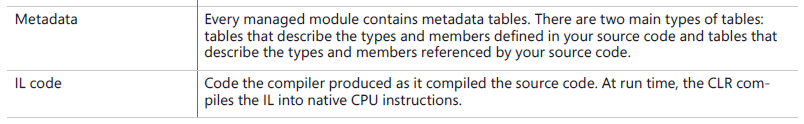
The common language runtime (CLR) is just what its name says it is: a runtime that is usable by different and varied programming languages. The core features of the CLR (such as memory management, assembly loading, security, exception handling, and thread synchronization) are available to any and all programming languages that target it

**Compiling Source Code into Managed Modules**



As we can see, all languages that are supported by CLR are compiled into Managed Module(IL(Intermediate Language) and metadata)

Parts of a Managed Module

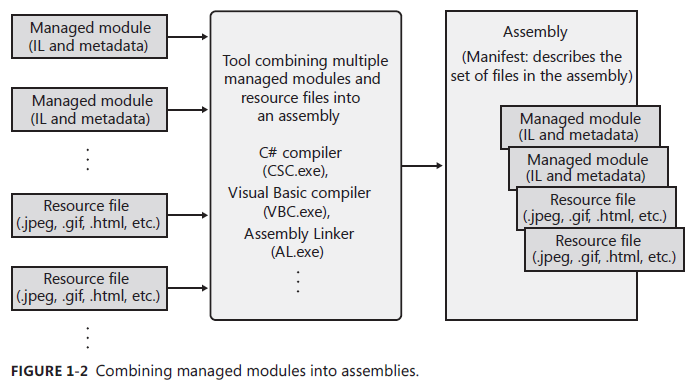


Note:

"C++ is unique in that it is the only compiler that allows the developer to write both managed and unmanaged code and have it emitted into a single module."

**Combining Managed Modules into Assemblies**

"The CLR doesn't actually work with modules, it works with assemblies."



**Loading the Common Language Runtime**

"The CLR is responsible for managing the execution of code contained within these assemblies. This means that the .NET Framework must be installed on the host machine."

The compiler setting(e.g. platform target) can decide the target platform of assemblies, CLR will use the right version of dll accordingly.

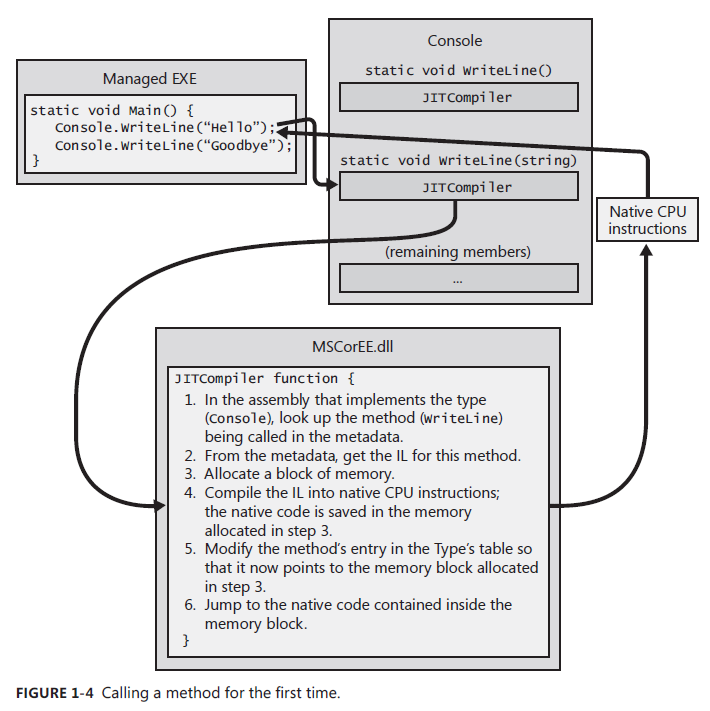
**Executing Your Assembly's Code**

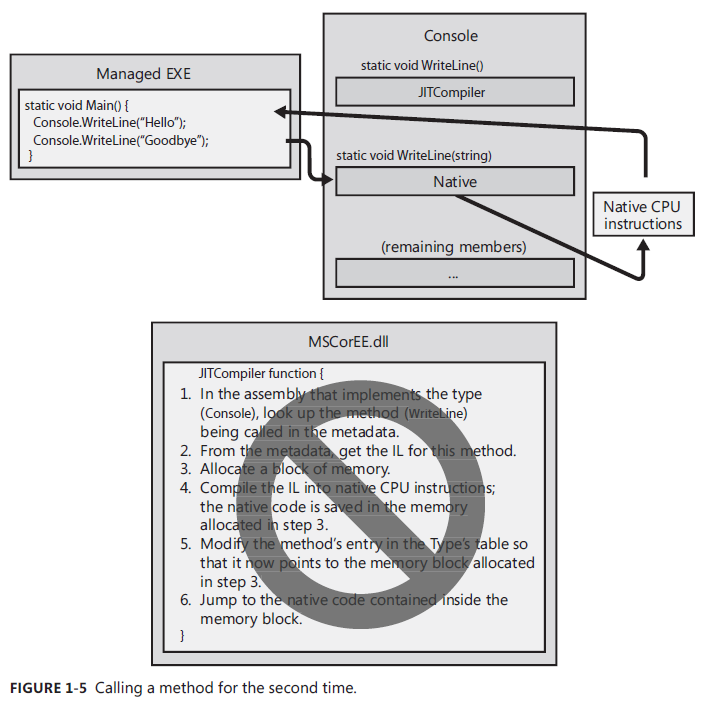
"Assemblies contain both metadata and IL."

"IL is a CPU-independent machine language created by Microsoft after consultation with several external commercial and academic language/compiler writers."

Note:

"Keep in mind that any high-level language will most likely expose only a subset of the facilities offered by the CLR. "





JIT(Just In Time) Compiler will verifies and compiles the IL code into native CPU instruction that are saved in a dynamically allocated block of memory.

Once the same method has been verified and compiled by JIT, the next time to call it will call the native CPU instruction directly without verified and compiled process.

In a word, JIT will compile IL at runtime but only once for each different type(e.g. method).

PDB(program Database) file helps the debugger find local variables and map the IL instructions to the source code.

Note:

**NGen.exe** tool can compiles all of an assembly's IL code into native code and saves the resulting native code to a file on disk.(avoid compilation at run time)

**IL and Verification**

IL features:

stack-based

typeless

Verification:

Verification examines the high-level IL code and ensures that everything the code does is safe. (This provides application robustness and security)

Compare Managed code with Unmanaged code:

Multiple applications can be runned in single process(Verification guarantee the security and robustness of managed code)

Unsafe Code:

Unsafe code is allowed to work directly with memory addresses and can manipulate bytes at these addresses.

/unsafe compiler switch to control whether allow to executing unsafe code

**PEVerify.exe** is used to examine all of an assembly's methods and notifies you of any methods that contain unsafe code.

**The Framework Class Library**

The .NET Framework includes the Framework Class Library(FCL).

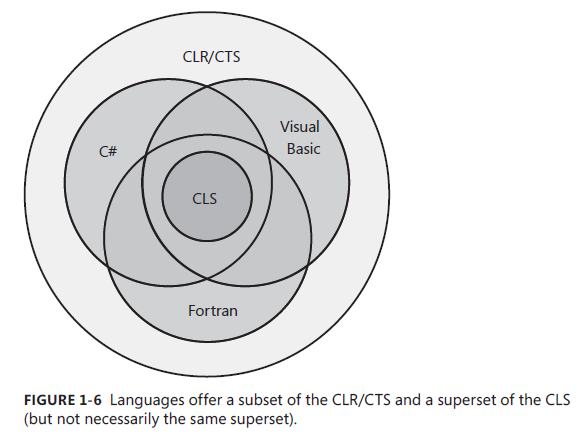
The FCL is a set of DLL assemblies that contain several thousand type definitions in which each type exposes some functionality.

**The Common Type System(CTS)**

Describes how types are defined and how they behave. Defines the rules governing type inheritance, virtual methods, object lifetime, and so on.

**The Common Language Specification**

Details for compiler vendors the minimum set of features their compilers must support if these compilers are to generate types compatible with other components written by other CLS-compliant languages on top of the CLR.



**Interoperability with Unmanaged Code**

Managed code can call an unmanaged function in a DLL

Managed code can use an existing COM component(server)

Unmanaged code can use a managed type(server)

.......

2. Building, Packaging, Deploying, and Administering Applications and Types

.Net Framework Deployment Goals

Reasons:

1. “When installing a new application, you discover that it has somehow corrupted an already-installed application -- **DLL hell**”

2. Windows installation complexities (the application isn’t isolated as a single entity)

3. Security (undetectable behavior for user)

Building Types into a Module

Response Files -- “A text file that contains a set of compiler command-line switches.“ \*\*\*.rsp

E.g.

Csc.exe @\*\*\*.rsp \*\*\*.cs \*\*\*.cs

Base response file -- CSC.rsp

A Brief Look at Metadata

A managed PE file parts:

1. PE32(+) header
2. The CLR header
3. The metadata -- is a block of binary data that consists of several tables. Three categorie of tables: **Definition tables**, **reference tables**, and **manifest tables**.
4. The IL

Combining Modules to Form an Assembly

Assembly -- is a collection of one or more files containing type definitions and resource files.

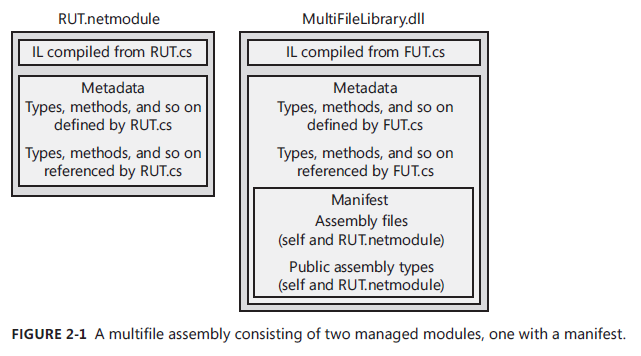
An assembly allows you to decouple the logical and physical notions of reusable types.

The reasons to use multifile aseemblies:

1. You can partition your types among separate files, allowing for files to be incrementally downloaded as described in the internet download scenario.

2. You can add resource or data files to your assembly.

3. You can create assemblies consisting of types implemented in different programming languages.



Using The Assembly Linker

Assembly Linker -- is useful if you want to create an assembly consisting of modules built from different compilers or perhaps if you just don’t know your assembly packaging requirements at build time.

......

3. Shared Assemblies and Strongly Named Assemblies

Two Kinds of Assemblies, Two Kinds of Deployment

Weakly named assemblies and Strongly named assemblies

The difference between weakly named and strongly named assemblies is that a strongly named assembly is signed with a publisher's public/private key pair that uniquely identifies the assembly's publisher.

Deploy way:

Privately -- deployed in the application's base directory or one of its subdirectories. (weakly named assemblies & strongly named assemblies)

Globally -- deployed into some well-know location that the CLR looks in when it's searching for the assembly.(strongly named assemblies)

Giving an Assembly a Strong Name

Strongly named assembly consists of four parts:

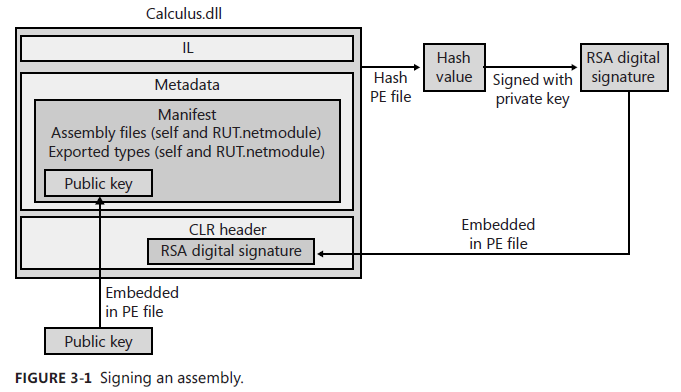
1. A file name

2. A version number

3. A culture identity

4. A public key

SN.exe -- Strong Name utility that generate a public/private key pair



The Global Assembly Cache(GAC)

GACUtil.exe -- %SystemRoot%\<Microsoft.NET\Assembly

You should never manually copy assemly files into the GAC instead use GACUtil.exe.

Building an Assembly That References a Strongly Named Assembly

csc.exe /r:\*\*\*

Note:

"The assemblies in the compiler/CLR directory are machine agnostic. That is, the assemblies contain only metadata in them. Because the IL code is not required at build time, the directory does not have to contain x86, x64, and ARM versions of an assembly. The assemblies in the GAC contain metadata and IL code because the code is need only at run time."

Strongly Named Assemblies Are Tamper-Resistant

The private key and public key and contents hash value will be used to check whether the file has been tampered.

Delayed Signing

Delayed signing allows you to build an assembly by using only your company's public key(lose all of the tampering protection afforded)

Cannot install assemble to GAC without finishing signing or avoid system verifying.

-delaysign delay singing process

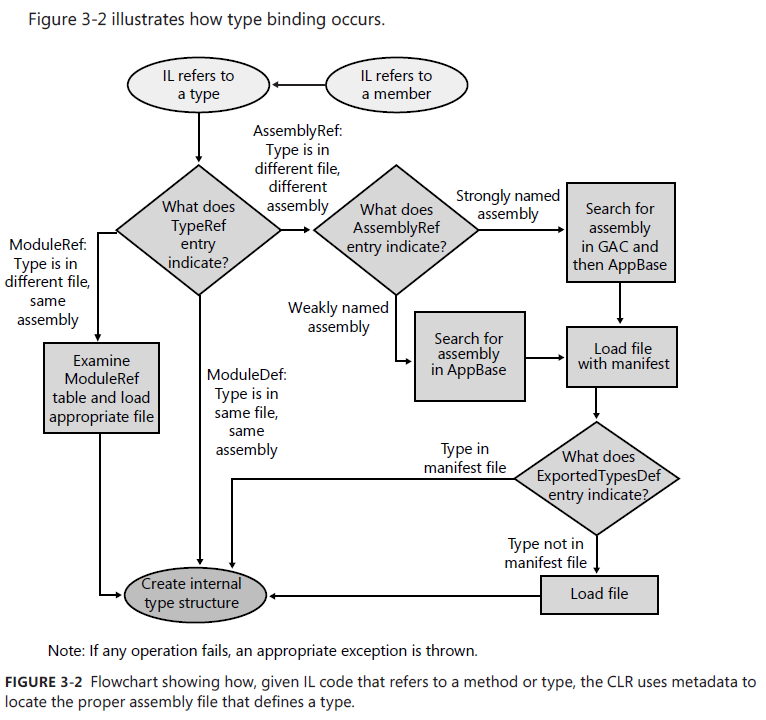
-Vr switch can prevent the system from verifying the integrity of the assembly's files

-Ra delay singing with private key

-Vu turn verification back on

Privately Deploying Strongly Named Assemblies

XNk - codeBase element indicate the path of the shared assembly.



Note:

Any assembly that references .NET Framework assemblies always binds to the version that matches the CLR's version. This is called unification.

GAC identifies assemblies by using name, version, culture, public key, and **CPU architecture**.(search for specific GPU architecture version firstly)

Note: The CLR supports the ability to move a type from one assembly to another. -- See System.Runtime.CompilerServices.TypeForwardedToAttribute

Advanced Administrative Control(Configuration)

Using Advanced Administrative Control, an administrator can really control what assembly the CLR decides to load.

Publisher Policy Control

/linkresource -- tells AL.exe that the XML configuration files it to be considered a separate file of the assembly.

**DESIGNING TYPES**

Type Fundamentals

All Types Are Derived from System.Object

Every object on the heap requires some additional members -- called the **type object pointer(System.Type, RTTI, reflection......)** and the **sync block index** -- used by the CLR to manage the object.

Object.GetType() -- returns an instance of a Type-derived object that identifies the type of the object used to call GetType -- return **type object ptr**. **The returned Type object can be used with the reflection classes to obtain metadata information about the object's type. (GetType ()-- NonVirtual method)**

Casting Between Types

CLR feature -- **type safety**

C# doesn't require any special syntax to cast an object to any of its base types(safe implicit conversion), but C# does require the developer to explicitly cast an object to any of its derived types because such a cast could fail at run time.

Casting with the C# is and as Operators

is - check whether an object is compatible with a given type, return true or false. **Never throw an exception**

as - check whether an object is compatible with a give type, and if it is, return a non-null reference to the same object, otherwise return null. **Never throw an exception**

Namespace and Assemblies

Type full name when the type exits in two different namespace

**using** -- allows you to create an alias for a single type or namespace

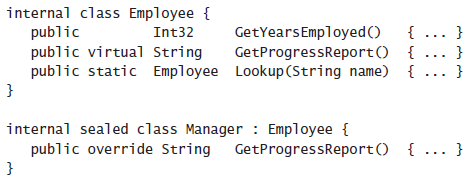
**extern** -- extern aliases also give you a way to access a single type from two different version of the same assembly

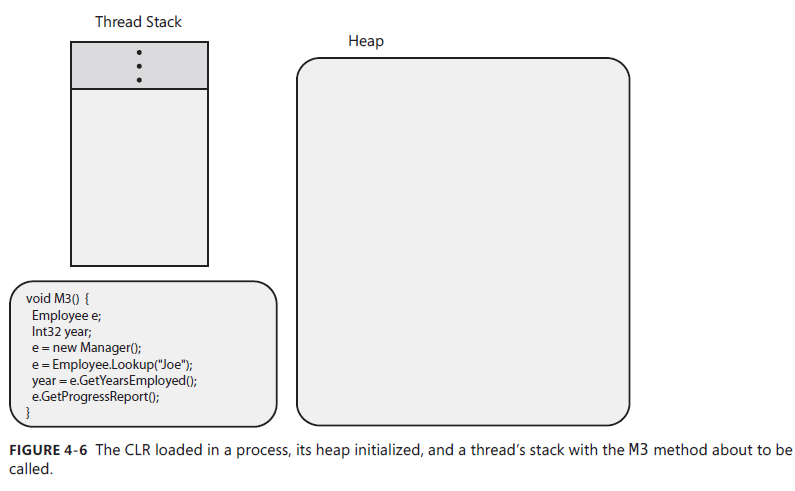
Note:

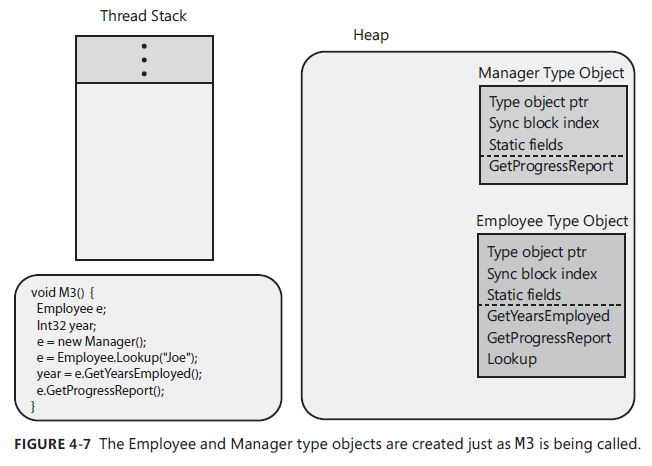
A namespace and an assembly are not necessarily related.

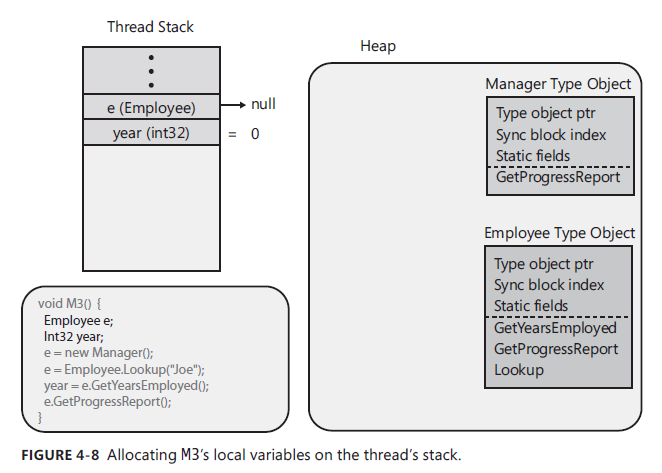
A single assembly can contain types in different namespaces.

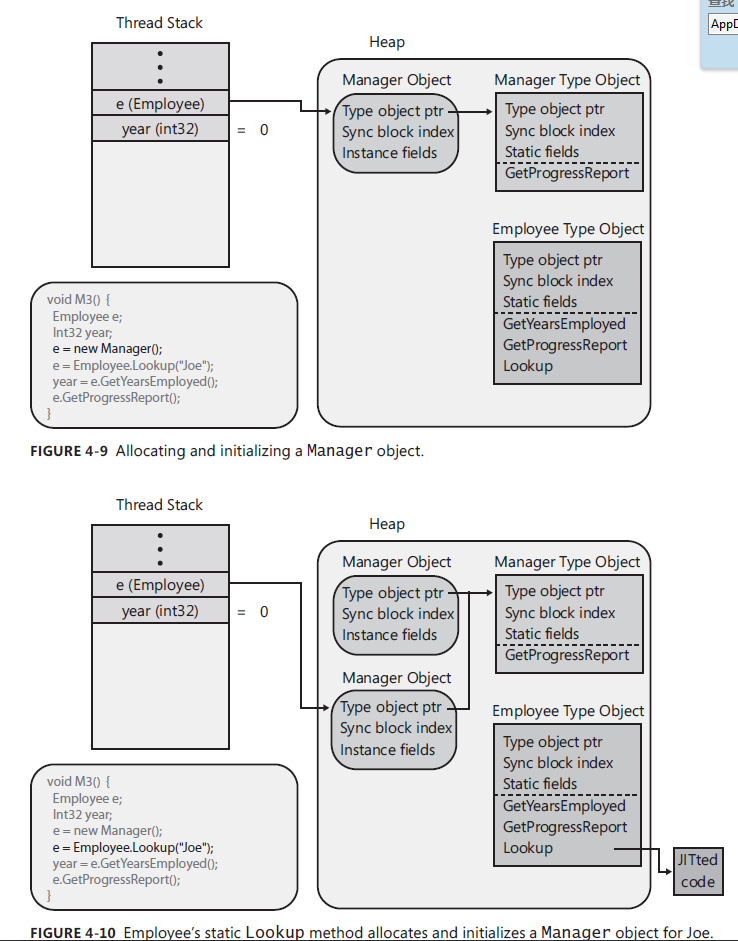
How Things Relate at Run Time

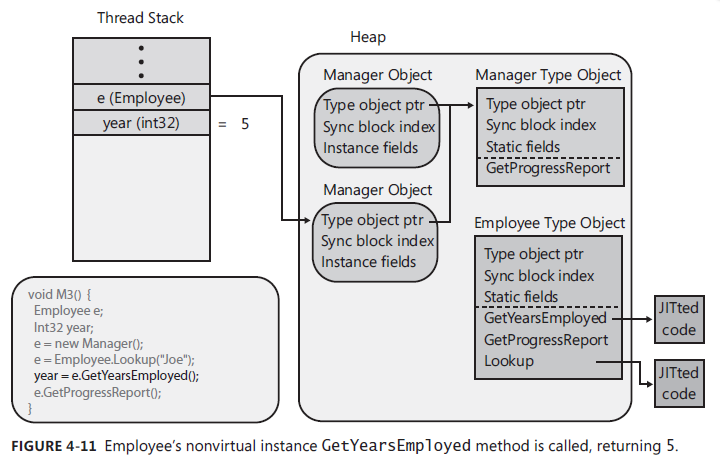


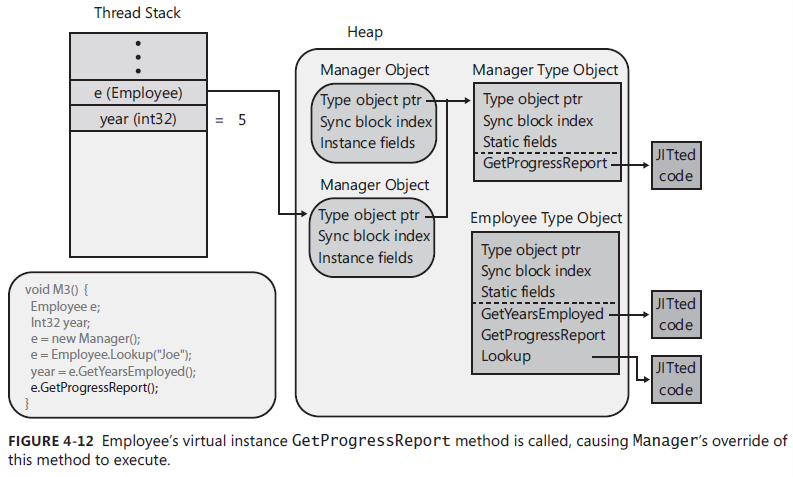


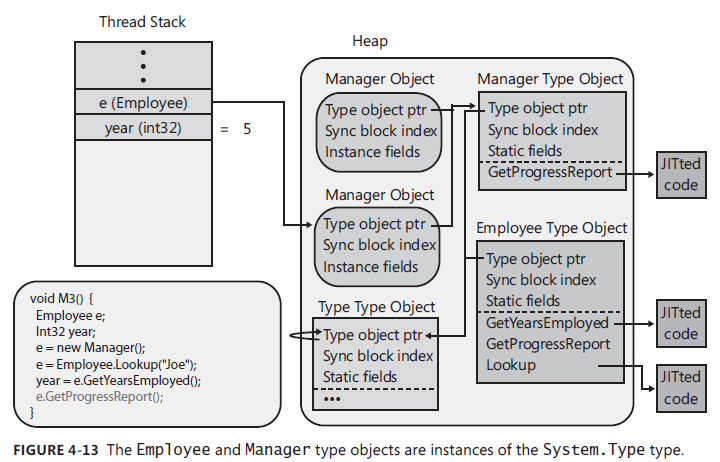










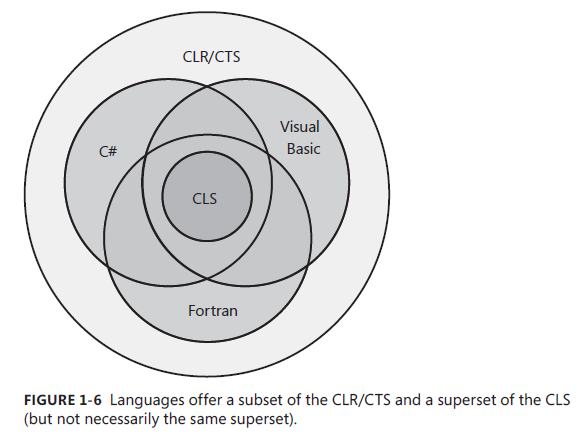


Primitive, Reference, and Value Types

Programming Language Primitive Types

Primitive Types -- Any data types the compiler directly supports

**Common Language Specification**



Checked and Unchecked Primitive Type Operations

/checked+ compiler switch

checked & unchecked keywords in code

Reference Types and Value Types

Reference types are always allocated from the managed heap.

e.g.

class

Value type instances are usually allocated on a thread's stack. -- System.ValueType

e.g.

primitive type, struct, enumeration

How the CLR Controls the Layout of a Type's Fields

System.Runtime.InteropServices

"Layout of a Type's Fields is typically used to simulate what would be a union in unmanaged C/C++."

Boxing and Unboxing Value Types

Boxing -- convert a value type to a reference type

Unboxing -- obtain a value type from reference type

Note:

"The biggest improvements is that the **generic collection classes** allow you to work with collections of value types **without requiring that items in the collection be boxed/unboxed**"

"**Unboxing is not the exact opposite of boxing.** The unboxing operation is much less costly than boxing. Unboxing is really just the operation of obtaining a pointer to the raw value type contained within an object. **Unboxing doesn't involve the coping of any bytes in memory.**"

Changing Fields in a Boxed Value Type by Using Interfaces(and Why You Shouldn't Do This)

Object Equality and Identity

System.Object.Equals -- purpose is to return true if two objects contain the same value

System.Object.ReferenceEquals -- check whether two references point to the same obejct

System.ValueType.Equals -- Value equality check

System.IEquatable<T> -- generic interface allows you to define a type-safe Equals method

System.IComparable<T> -- generic interface for type-safe CompareTo method(purpose of sorting)

Object Hash Codes

System.Object.GetHashCode -- obtain an Int32 hash code for any obejcts

The reason a type that defines Equals must also define GetHashCode is that the implementation of the System.Collections.Hashtable type, the System.Collections.Generic.Dictionary type, and some other collections require that any two objects that are equal must have the same hash code value.

The dynamic Primitive Type

Benefic of a type-safe programming language over a non-type-safe programming language -- many programmer errors are detected at compile time.

Mark an expression's type as dynamic -- to make it easier for developer using reflection or communicating with other components.

Microsoft.CSharp.RuntimeBinder -- provide the classes and interface that support operations between dynamic language runtime and C#

One of the **limitations** of **dynamic** is that you can only use it to access an object's instance members because the dynamic variable must refer to an object.

......

Type and Member Basics

The Different Kinds of Type Members

The **metadata** is the common information that all languages produce and consume, enabling code in one programming language to seamlessly access code written in a completely different programming language.

Type Visibility

public -- is visible to all code within the defining assembly as well as code written in other assemblies.

internal -- is visible to all code within the defining assembly, and the type is not visible to code written in other assemblies

**C#** compiler sets the **default type's visibility** to **internal**

Friend Assemblies

Friend Assemble -- is useful when you want to have one assembly containing code that performs unit tests against the internal types within another assembly.

e.g

[assembly:InternalsVisibleTo(assemblename,publickey)]

/out:file or /moduleassemblyname (decide friend assemble name)

Member Accessibility

C# -- default private

Interface -- must use public

Static Classes

No instance constructor

Partial Classes, Structures, and Interfaces

partial -- tells the C# compiler that the source code for a single class, structure, or interface definition may span one or more source code files.

Note:

"The partial types feature is completely implemented by the C# compiler; the CLR knows nothing about partial types at all. This is why all of the source code files for the type must use the same programming language, and they must all be compiled together as a single compilation unit."

Components, Polymorphism, and Versioning

......

How the CLR Calls Virtual Methods, Properties, and Events

The CLR allows a type to define multiple methods with the same name as long as each method has a different set of parameters or a different return type.

The CLR offers two IL instructions for calling a method:

1. The **call IL instruction** -- static, instance, and virtual methods (without check null)

2. The **callvirt IL instruction** -- instance and virtual methods (check null automatically)

Note:

When designing a type, you should try to minimize the number of virtual methods you define.

Using Type Visibility and Member Accessibility Intelligently

......

Dealing with Virtual Methods When versioning Types

The new keyword tells the compiler to emit metadata, making it clear to the CLR that child method is intended to be treated as a new function that is introduced by the child class. The CLR will know that there is no relationship between child's and base's methods.

Constants and Fields

Constants

A constant is a symbol that has a never-changing value. When defining a constant symbol, its value must be determinable at compile time. (a constant's value is embedded directly in code, constants don't require any memory to be allocated for them at run time. **cross-assembly versioning problem**)

Fields

A field is a data member that holds an instance of a value type or a reference to a reference type.

The CLR supports readonly fields and read/write fields.

Note:

Reflection can be used to modify a readonly field. (fields are stored in dynamic memory, their value can be obtained at run time only. **without versioning problem**)

When a field is of a reference type and the field is marked as readonly, it is the reference that is immutable, not the object that the field refers to.

Methods

Instance Constructors and Classes(Reference Types)

Constructor methods **.ctor**

Note:

Any fields that the constructor doesn't explicitly overwrite are guaranteed to have a value of 0 or null

Not call any virtual methods within a constructor that can affect the object being constructed. (fields may not been initialized yet before virtual method get called)

Instance Constructors and Structures(Value Types)

Note:

**C# doesn't allow a value type to define a parameter less constructors(but allow type constructors), the CLR does.**

Type Constructors (also known as static constructors, class constructors, or type initializers)

Type constructors are used to set the initial state of a type.

Note:

Because the CLR guarantees that a type constructor executes only once per AppDomain and is thread-safe, a type constructor is a great place to initialize any singleton objects required by the type.

Operator Overload Methods

Although the CLR doesn't know anything about operators, it does specify how languages should expose operator overloads so that they can be readily consumed by code written in a different programming language.

Note:

The CLR specification mandates that operator overload methods be public and static methods.

The CLR fully supports the ability for a type to define multiple methods that differ only by return type.

Extension Methods

this feature, example below:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Collections;

using System.Reflection;

namespace CSharpDeepStudy

{

#region Extention Method feature

public static class StringBuilderExtensions

{

public static Int32 Indexof(**this** StringBuilder sb, Char value)

{

for (Int32 index = 0; index < sb.Length; index++)

{

if (sb[index] == value)

{

return index;

}

}

return -1;

}

}

#endregion

class Program

{

static void Main(string[] args)

{

#region Extention Method feature

StringBuilder sb = new StringBuilder("Hello. My name is Tony.");

Int32 index = sb.Indexof('T');

Console.WriteLine("sb.Indexof('T') = " + index);

#endregion

Console.ReadKey();

}

#endregion

}

}

When the compiler sees code Int32 index = sb.Indexof('T');, the compiler first checks if the StringBuilder class or any of its base classes offers an instance method called IndexOf that takes single Char parameter. If an existing instance method exists, then the compiler produces IL code to call it. If no matching instance method exists, then the compiler will look at any static classes that define static methods called IndexOf that take as their first parameter a type matching the type of the expression being used to invoke the method. This type must also be marked with the **this** keyword.

Note:

Extension methods must be declared in non-generic, static classes.

The C# compiler looks only for extension methods defined in static classes that are themselves defined at the file cope.(not allowed nested static class)

There is potential versioning problem that exists with extension methods.(e.g. if StringBuilder define a IndexOf method in the future, it will call StringBuilder IndexOf method instead of StringBuilderExtension.IndexOf)

The Extension Attribute

In C#, when you mark a static method's first parameter with the **this** keyword, the compiler internally applies a custom attribute to the method and this attribute is persisted in the resulting file's metadata.

This attribute is used for fast searching static extension method.

Partial Methods

Partial methods advantages:

1. The type is no need to be a class that is sealed(normally we used override) or value types

e.g.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Collections;

using System.Reflection;

namespace CSharpDeepStudy

{

#region Partial Method

internal sealed partial class Base

{

public String Name

{

get

{

return mName;

}

set

{

OnNameChanging(value.ToUpper());

mName = value;

}

}

private String mName;

//This defining-partial-method-declaration is called before changing the mName field

partial void OnNameChanging(String value);

}

internal sealed partial class Base

{

partial void OnNameChanging(string value)

{

Console.WriteLine("Base::OnNameChanging({0})", value);

}

}

#endregion

static void Main(string[] args)

{

#region Partial Method

Base bs = new Base();

bs.Name = "Tony";

#endregion

Console.ReadKey();

}

}

}

output:

Base::OnNameChanging(Tony)

Note:

If there is no implementing partial method declaration, the compiler will not emit any metadata representing the partial method.

Only can be declared with a partial class or struct

Partial methods must always have a return type of void

Partial methods are always considered to be private methods

Parameters

Optional and Named Parameters

e.g.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Collections;

using System.Reflection;

namespace CSharpDeepStudy

{

#region Optional and Named Parameters

private static void OptionalAndNamedParameters(Int32 x = 9, String s = "A", Guid guid = new Guid())

{

Console.WriteLine("x = {0}, s = {1}, guid = {2}", x, s, guid);

}

#endregion

static void Main(string[] args)

{

#region Optional and Named Parameters

Int32 i = 11;

OptionalAndNamedParameters();

OptionalAndNamedParameters(1, "B");

OptionalAndNamedParameters(s: i++.ToString(),x: i++);

#endregion

Console.ReadKey();

}

}

}

output:

x = 9, s = A, guid = 00000000-0000-0000-0000-000000000000

x = 8, s = B, guid = 00000000-0000-0000-0000-000000000000

x = 12, s = 11, guid = 00000000-0000-0000-0000-000000000000

Note:

When we pass arguments by using named parameters, the compiler still evaluates the arguments from left to right.

Parameters with default values must come after any parameters that do not have default values.

Cannot set default values for parameters marked with either the ref or out keywords because there is no way to pass a meaning full default value for these parameters

Runtime.InteropServices.DefaultParameterValue and Runtime.InteropServices.OptionalAttributes are used to make default and name parameter works in other programming language

Implicitly Typed Local Variables

Var -- is a syntactical shortcut that has the compiler infer the specific data type from an expression. (have to explicitly initialize)

Dynamic -- enables the operations in which it occurs to bypass compile-time type checking. Instead, these operations are resolved at run time.(not have to initialize)

Passing Parameters by Reference to a Method

Reference type -- pass reference (**out & ref** keywords)

Value type -- pass copy of the instance

The difference between out and ref:

1. With which method is responsible for initializing the object being referred to.

out -- the caller isn't expected to have initialized the object prior to calling the method. The called method can't read from the value, and the called method must write to the value before returning.

ref -- the caller must initialize the parameter's value prior to calling the method. The called method can read from the value and/or write to the value.

Note:

From an IL or a CLR perspective, out and ref do exactly the same thing: they both cause a pointer to the instance to be passed. The difference is that the compiler helps ensure that your code is correct.

The CLR allows you to overload methods based on their use of out and ref parameters. It's not legal to overload methods that differ only by out and ref because the metadata representation of the method's signature for the methods would be identical.

When we pass by using ref or out, we must match the parameters expected by the method(for type safety), generic method can solve this problem.

Passing a Variable Number of Arguments to a Method

Format:

static Int32 Add(**params** Int32[] values)

{ ...... }

params -- System.ParamArrayAttribute

Note:

Only the last parameter to a method can be marked with the params keyword(ParamArrayAttribute). It's legal to pass null or a reference to an array of 0 entries as the last parameter to the method.

Properties

**Core Facilities**

**Exception and State Management**

**The Managed Heap and Garbage Collection**

Managed Heap Basics

“The CLR requires that all objects be allocated from the managed heap.”

C# new operator:

1. Calculate the number of bytes required for the type’s fields
2. Add the bytes required for an object’s overhead(contain a type object pointer and a sync block index)
3. Zero out the memory start at NextObjPtr(Indicates where the next object is to be allocated within the heap). Return reference. Move on NextObjPtr to next address that is available to be allocated..

The Garbage Collection(GC) Algorithm

1. Reference Counting Algorithm(COM use)

Problems:

Circular references

1. Referencing Tracking Algorithm(CLR use)

**Marking Phase:**

CLR first suspends all threads in the process(prevents threads from accessing objects and changing their state while the CLR examines them)

Marking All objects to 0(means all objects should be deleted)

Scan active roots to marking object(not mark the same object again to avoid circular references)

**Compacting Phase:**

Shifts the memory consumed by the marked objects down in the heap, compacting all the surviving objects together so that they are contiguous in memory.(reduce application’s working set size &access fast in future & no address space fragmentation issues)

CLR resumes all the application’s threads and they continue to access the objects as if the GC never happened at all

Note:

A static field keeps whatever object it refers to forever or until the AppDomain that the types are loaded into is unloaded

1. Garbage Collections and Debugging

4. Generations: Improving Performance

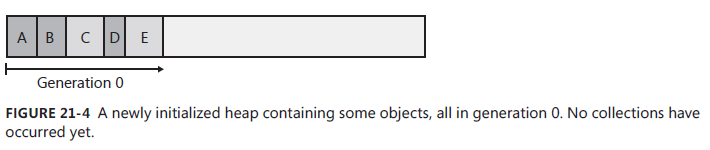
CLR’s GC assumptions:

The newer an object is, the shorter its lifetime will be

The older an object is, the longer its lifetime will be

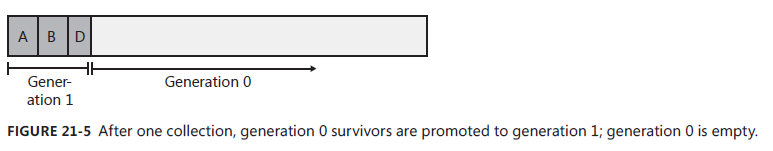
Collecting a portion of the heap is faster than collecting the whole heap

**Generation 0**

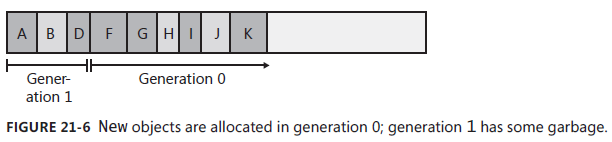


First new will all be initialized in generation 0

**Generation 1**

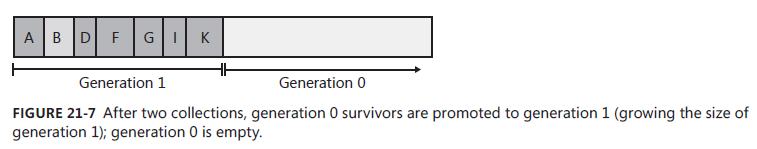


Once object survive from first time GC, it locates in generation 1

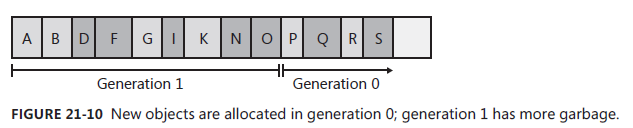


Next memory allocation will be allocated in generation 0

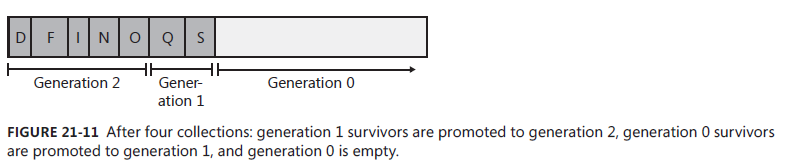
When GC happened again, CLR will GC the newly created objects(here is generation 0 -- suit first assumption)(in generation 1, only old objects that have had fields change need to be examined to see whether they refer to any new object in generation 0)



Generation 1 keep grown before generation 1 has more unreachable objects than generation 0, CLR will examine both generation 1 and 0



After GC generation 1 and 0, any objects that were in generation 1 that survived the collection are now in generation 2



Note:

**The Managed heap supports only three generations: generation 0,1,2**

**The garbage collector fine-tunes itself automatically based on the memory load required by your application.**

Garbage Collection Triggers

1. Code explicitly calls System.GC’s static Collect method
2. Windows is reporting low memory condition
3. The CLR is unloading an AppDomain
4. The CLR is shutting down

Large Objects

The CLR treats large objects slightly differently than how it treats small objects:

1. Large objects are not allocated within the same address space as small objects; They are allocated elsewhere within the process’ address space
2. GC doesn’t compact large objects because of the time it would require to move them in memory
3. Large objects are immediately considered to be part of generation 2

Garbage Collection Modes

1. Workstation
2. Server

Sub-Modes

Concurrent mode

Non-concurrent mode

Note:

**GCSettings class that specifies the garbage collection settings for the current process**

Forcing Garbage Collections

The **System.GC** type allows your application some direct control over the garbage collector.

Note:

Garbage Collection Notifications can help us to call GC.Collect at a more opportune time

Monitoring Your Application’s Memory Usage

.Net Framework tool -- PerfMon.exe

PrefView

Working with Types Requiring Special Cleanup

**Finalize** methods are called at the completion of a garbage collection on objects that the GC has determined to be garbage.

Finalize is not equal to destructor in C++

Monitoring and Controlling the Lifetime of Objects Manually

The CLR provides each AppDomain with a GC handle table. This table allows an application to monitor the lifetime of an object or manually control the lifetime of an object.

**System.Runtime.InteropServices.****GCHandle**

GCHandleType which is a flag indicating how you want to monitor/control the object

public enum GCHandleType{

Weak = 0,

WeakTrackResurrection = 1,

Normal = 2,

Pinned = 3

}

Note:

Use the GCHandle type explicitly when you need to pass the pointer to a managed object to native code.

**fixed** statement is more efficient that allocating a pinned GC handle.

Note:

Only in fixed statement block, the object that is pointed is pinned(that means will not collect or compact by GC)

Working with Types Requiring Special Cleanup

Monitoring and Controlling the Lifetime of Objects

Manually

Delegates

A First Look at Delegates

delegates -- type-safe

Unmanaged C/C++ callback functions are not type-safe

Using Delegates to Call Back Static Methods

Both C# and the CLR allow for **covariance** and **contra-variance** of **reference types** when binding a method to a delegate.

Covariance means that a method can return a type that is derived from the delegate's return type.

Contra-variance means that a method can take a parameter that is a base of the delegate's parameter type.

The reason why value types and void cannot be used for covariance and contra-variance is because the memory structure for these things varies, whereas the memory structure for reference type is always a pointer.

Using Delegates to Call Back Instance Methods

......

Demystifying Delegates

The compilers and the CLR do a lot of behind-the-scenes processing to hide the complexity

e.g.

internal delegate void Feedback(Int32 value);

After compile:

internal class Feedback : System.MulticastDelegate{

//Constructor

public Feedback(Object @object, IntPtr method);

//Method with same prototype as specified by the source code

public virtual void Invoke(Int32 value);

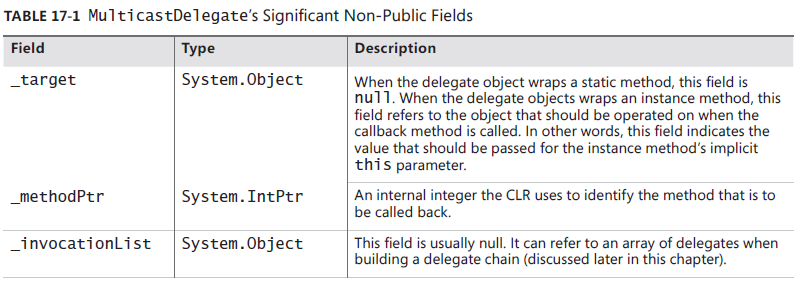
//Methods allowing the callback to be called asynchronously

public virtual IAsyncResult BeginInvoke(Int32 value, AsyncCallback callback, Object @object);

public virtual void EndInvoke(IAsyncResult result);

}

MulticastDelegate's most significant fields:



See Delegate example to see the story behind the Compiler and CLR

e.g.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Collections;

namespace CSharpDeepStudy

{

#region Delegate Study

internal delegate void DelegateStudy(int value);

#endregion

class Program

{

#region Delegate Study

private static void StaticDelegateDemo(int value)

{

Console.WriteLine("StaticDelegateDemo({0})", value);

}

private void InstanceDelegateDemo(int value)

{

Console.WriteLine("InstanceDelegateDemo({0})",value);

}

private static void ChainDelegateDemo(Program p)

{

DelegateStudy cdd = null;

cdd += Program.StaticDelegateDemo;

cdd += p.InstanceDelegateDemo;

cdd.Invoke(3);

}

#endregion

static void Main(string[] args)

{

#region dynamic type

/\*

dynamic value;

for (int demo = 0; demo < 2; demo++)

{

value = (demo == 0) ? (dynamic)5 : (dynamic)"A";

value = value + value;

M(value);

}

\*/

#endregion

#region Delegate Study

Program p = new Program();

DelegateStudy sdd = Program.StaticDelegateDemo;

DelegateStudy idd = p.InstanceDelegateDemo;

sdd.Invoke(1);

idd.Invoke(2);

Program.ChainDelegateDemo(p);

#endregion

Console.ReadKey();

}

#region dynamic type

/\*

private static void M(int n)

{

Console.WriteLine("M(int): " + n);

}

private static void M(String s)

{

Console.WriteLine("M(String): " + s);

}

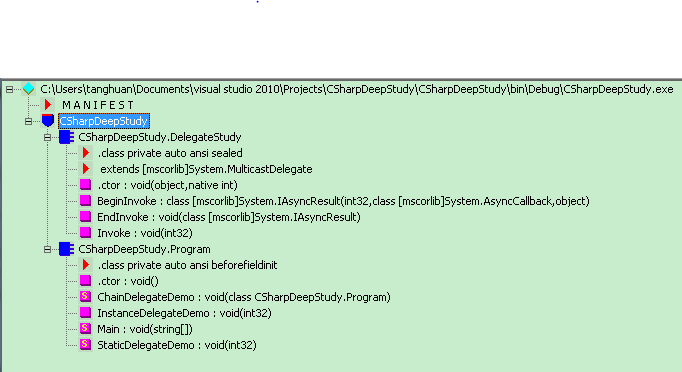
\*/

#endregion

}

}

After compile

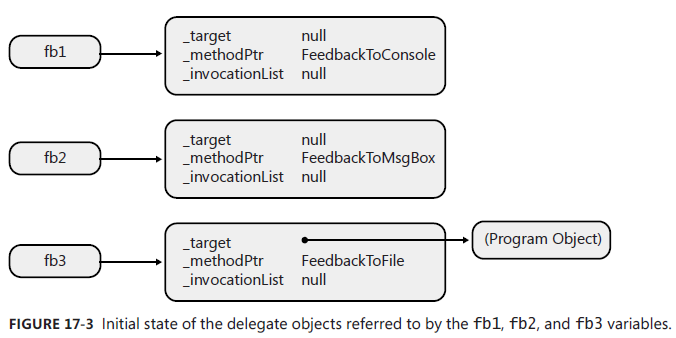


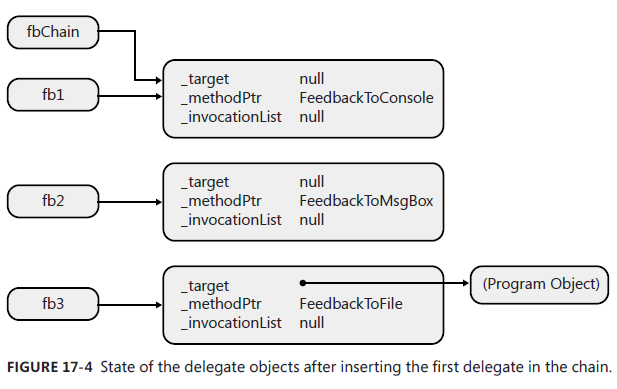
Using Delegates to Call Back Many Methods(Chaining)

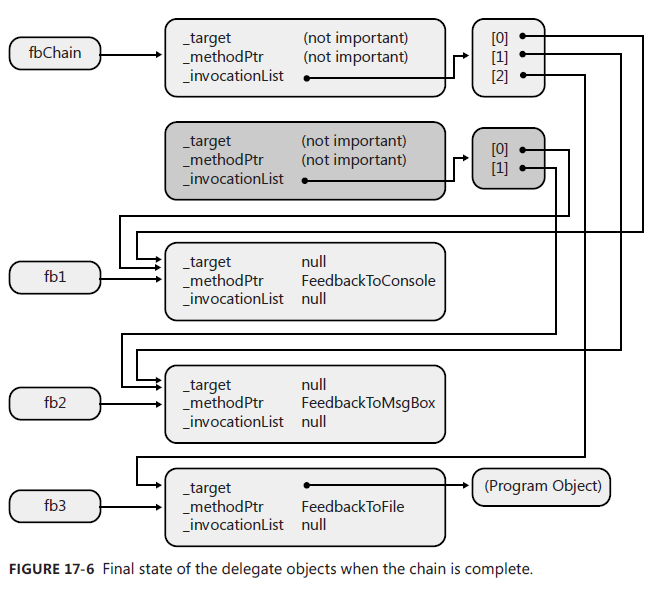
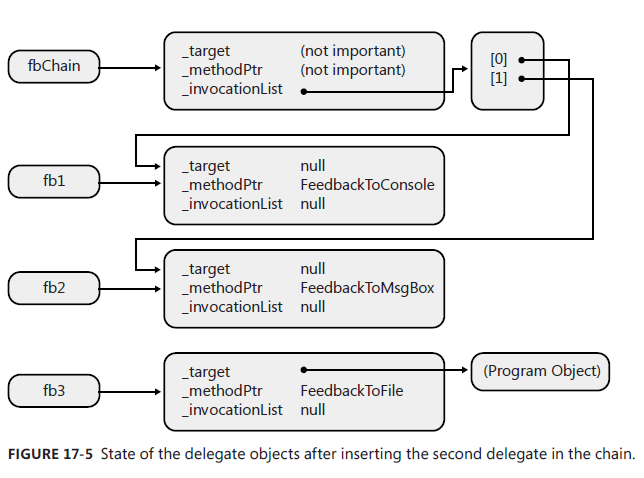
The Delegate class's public, static Combine method is used to add a delegate to the chain.

**\_****invocationList** field is initialized to refer to an array of delegate objects. (The key of delegate chain)

See below to know how delegate chain works







Delegate's public static Remove method that can remove delegate from \_ invocationList

Having More Control over Delegate Chain Invocation

The MulticastDelegate class offers an instance method, **GetInvocationList**, that you can use to call each delegate in a chain explicitly

Enough with the Delegate Definitions Already(**Generic Delegates**)

The .NET Framework supports generics, we really just need a few generic delegates that represent methods that take up to 16 arguments

C# Syntactical Sugar for Delegates

.....

Delegates and Reflection

**System.Reflection.MethodInfo** offers a **CreateDelegate** method that allows you to create a delegate when you just don't have all the necessary information about the delegate at compile time.

Delegate's **DynamicInvoke** method to call delegate

Simple Example:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Collections;

using System.Reflection;

namespace CSharpDeepStudy

{

#region Delegate Study

internal delegate void DelegateStudy(int value);

#endregion

class Program

{

#region Delegate Study

private static void StaticDelegateDemo(int value)

{

Console.WriteLine("StaticDelegateDemo({0})", value);

}

public void InstanceDelegateDemo(int value)

{

Console.WriteLine("InstanceDelegateDemo({0})",value);

}

private static void ChainDelegateDemo(Program p)

{

DelegateStudy cdd = null;

cdd += Program.StaticDelegateDemo;

cdd += p.InstanceDelegateDemo;

cdd.Invoke(3);

}

#endregion

static void Main(string[] args)

{

#region dynamic type

/\*

dynamic value;

for (int demo = 0; demo < 2; demo++)

{

value = (demo == 0) ? (dynamic)5 : (dynamic)"A";

value = value + value;

M(value);

}

\*/

#endregion

#region Delegate Study

Program p = new Program();

DelegateStudy sdd = Program.StaticDelegateDemo;

DelegateStudy idd = p.InstanceDelegateDemo;

sdd.Invoke(1);

idd.Invoke(2);

Program.ChainDelegateDemo(p);

#endregion

#region Dynamic Study

//Dynamic delegate part

MethodInfo mi = typeof(Program).GetMethod("InstanceDelegateDemo");

Delegate d = Delegate.CreateDelegate(typeof(DelegateStudy), p, mi);

d.DynamicInvoke(4);

#endregion

Console.ReadKey();

}

#region dynamic type

/\*

private static void M(int n)

{

Console.WriteLine("M(int): " + n);

}

private static void M(String s)

{

Console.WriteLine("M(String): " + s);

}

\*/

#endregion

}

}