**Topic 1: Research and explain what is Application Domain.**

Operating systems and runtime environments typically provide some form of isolation between applications. For example, Windows uses processes to isolate applications. This isolation is necessary to ensure that code running in one application cannot adversely affect other, unrelated applications.

Application domains provide an isolation boundary for security, reliability, and versioning, and for unloading assemblies. Application domains are typically created by runtime hosts, which are responsible for bootstrapping the common language runtime before an application is run.

**Topic 2: What are the different types of assembly? Explain each one of them.**

**Static Assemblies** are those Assemblies which are stored on the disk permanently. They may include .NET Framework classes, interfaces as well as resource file. These assemblies are not loaded directly from the memory instead they are directly loaded from the disk when CLR (Common Language RunTime) requests for them. These Assemblies used to store on the disk as a file or set of file. Whenever one compiles the C# code, one gets STATIC assemblies.

**Dynamic assemblies** are those assemblies which are not stored on the disk before execution in fact after execution they get stored on the disk. When .NET runtime calls them they are directly loaded from the memory not from the disk. Reflection emit provides many ways to create dynamic assemblies means These are created in the memory using System.Reflection.emit namespace.The System.Reflection.Emit namespace contains classes that allow a compiler or tool to emit metadata and Microsoft intermediate language (MSIL) and optionally generate a PE file on disk. When an application requires the types within these assemblies these dynamic assemblies are created dynamically at run time

There are several ways to create assemblies. You can use development tools, such as Visual Studio, that you have used in the past to create .dll or .exe files. You can use tools provided in the Windows Software Development Kit (SDK) to create assemblies with modules created in other development environments. You can also use common language runtime APIs, such as [Reflection.Emit](https://msdn.microsoft.com/en-us/library/system.reflection.emit(v=vs.110).aspx), to create dynamic assemblies.

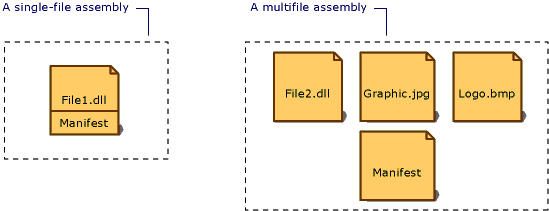
**Private Assembly** - an assembly that is deployed with an application and is available for the exclusive use of that application. That is, other applications do not share the private assembly. Private assemblies are one of the methods that can be used to create isolated applications.  
**Public/Shared Assembly** -A shared assembly is an assembly available for use by multiple applications on the computer. On Windows Vista and Windows XP, [side-by-side assemblies](https://msdn.microsoft.com/en-us/library/windows/desktop/ff951640(v=vs.85).aspx) can be installed as shared assemblies. Shared side-by-side assemblies are not registered globally on the system, but they are globally available to applications that specify a dependence on the assembly in [manifests](https://msdn.microsoft.com/en-us/library/windows/desktop/aa375365(v=vs.85).aspx). Multiple versions of side-by-side assemblies can be shared by different applications running at the same time. For more information, see [About Isolated Applications and Side-by-Side Assemblies](https://msdn.microsoft.com/en-us/library/windows/desktop/aa374029(v=vs.85).aspx). For example, the [supported Microsoft side-by-side assemblies](https://msdn.microsoft.com/en-us/library/windows/desktop/aa376609(v=vs.85).aspx) shipped with Windows XP are typically used as shared assemblies by multiple applications.

**Satelite assembly** - A satellite assembly is a compiled library (DLL) that contains “localizable” resources specific to a given culture such as strings, bitmaps, etc. A .NET Framework assembly containing resources specific to a given language. Using satellite assemblies, you can place the resources for different languages in different assemblies, and the correct assembly is loaded into memory only if the user elects to view the application in that language.

**Topic 3: What is a dynamic assembly?**

**Dynamic assemblies** are those assemblies which are not stored on the disk before execution in fact after execution they get stored on the disk. When .NET runtime calls them they are directly loaded from the memory not from the disk. Reflection emit provides many ways to create dynamic assemblies means These are created in the memory using System.Reflection.emit namespace.The System.Reflection.Emit namespace contains classes that allow a compiler or tool to emit metadata and Microsoft intermediate language (MSIL) and optionally generate a PE file on disk. When an application requires the types within these assemblies these dynamic assemblies are created dynamically at run time

**Topic 4: What is an assembly manifest?**

Every assembly, whether static or dynamic, contains a collection of data that describes how the elements in the assembly relate to each other. The assembly manifest contains this assembly metadata. An assembly manifest contains all the metadata needed to specify the assembly's version requirements and security identity, and all metadata needed to define the scope of the assembly and resolve references to resources and classes. The assembly manifest can be stored in either a PE file (an .exe or .dll) with Microsoft intermediate language (MSIL) code or in a standalone PE file that contains only assembly manifest information.   
  
The following illustration shows the different ways the manifest can be stored.   
  
  
Types of assemblies   
  
For an assembly with one associated file, the manifest is incorporated into the PE file to form a single-file assembly. You can create a multifile assembly with a standalone manifest file or with the manifest incorporated into one of the PE files in the assembly.   
  
Each assembly's manifest performs the following functions:   
  
 - Enumerates the files that make up the assembly.   
  
 - Governs how references to the assembly's types and resources map to the files that contain their declarations and implementations.   
  
 - Enumerates other assemblies on which the assembly depends.   
  
 - Provides a level of indirection between consumers of the assembly and the assembly's implementation details.   
  
 - Renders the assembly self-describing.   
  
Assembly Manifest Contents   
  
The following items show the information contained in the assembly manifest. The first four items—the assembly name, version number, culture, and strong name information—make up the assembly's identity.   
  
*Assembly name* - A text string specifying the assembly's name.   
  
*Version number* - A major and minor version number, and a revision and build number. The common language runtime uses these numbers to enforce version policy.   
  
*Culture* - Information on the culture or language the assembly supports. This information should be used only to designate an assembly as a satellite assembly containing culture- or language-specific information. (An assembly with culture information is automatically assumed to be a satellite assembly.)   
  
*Strong name information* - The public key from the publisher if the assembly has been given a strong name.   
  
*List of all files in the assembly* - A hash of each file contained in the assembly and a file name. Note that all files that make up the assembly must be in the same directory as the file containing the assembly manifest.   
  
*Type reference information* - Information used by the runtime to map a type reference to the file that contains its declaration and implementation. This is used for types that are exported from the assembly.   
  
*Information on referenced assemblies* - A list of other assemblies that are statically referenced by the assembly. Each reference includes the dependent assembly's name, assembly metadata (version, culture, operating system, and so on), and public key, if the assembly is strong named.

**Topic 5: How can you view assembly information?**

You can use the Ildasm.exe (IL Disassembler) to view Microsoft intermediate language (MSIL) information in a file. If the file being examined is an assembly, this information can include the assembly's attributes, as well as references to other modules and assemblies. This information can be helpful in determining whether a file is an assembly or part of an assembly, and whether the file has references to other modules or assemblies.

With assembly attributes it is possible to specify assembly meta information such as product title or version, either automatically by Visual Studio or manually by editing the file "AssemblyInfo.cs" in the project folder "Properties".

With the class provided with this tip these attributes can be easily read from any assembly.

To obtain information of an assembly the constructor of AssemblyInfo can be used by passing the specific assembly.

If the corresponding assembly attribute to a property is not found, null will be returned.

**Topic 6: What is a namespace?**

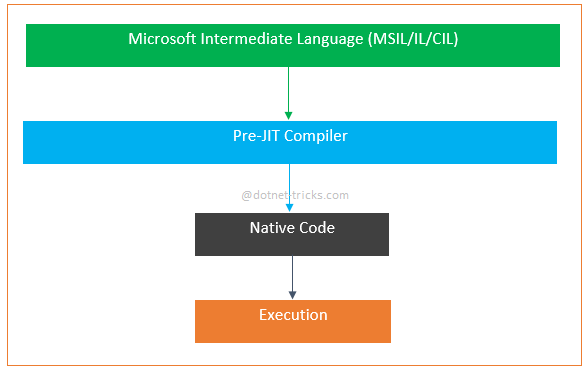
The namespace keyword is used to declare a scope that contains a set of related objects. You can use a namespace to organize code elements and to create globally unique types.Whether or not you explicitly declare a namespace in a C# source file, the compiler adds a default namespace. This unnamed namespace, sometimes referred to as the global namespace, is present in every file. Any identifier in the global namespace is available for use in a named namespace.   
  
Namespaces implicitly have public access and this is not modifiable. For a discussion of the access modifiers you can assign to elements in a namespace   
Namespaces have the following properties:   
  
 1. They organize large code projects.   
  
 2. They are delimited by using the . operator.   
  
 3. The using directive obviates the requirement to specify the name of the namespace for every class.   
  
 4. The global namespace is the "root" namespace: global::System will always refer to the .NET Framework namespace System.

**Topic 7: What is the difference between assembly and namespace?**

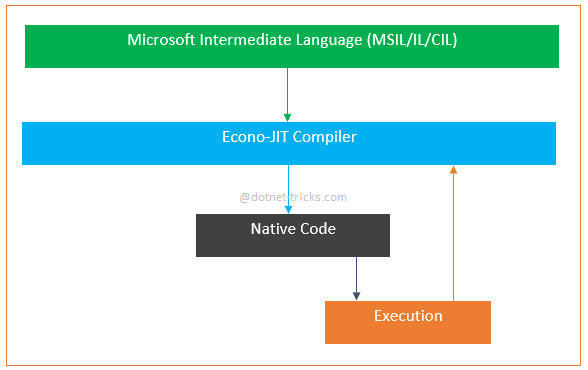
Namespace   
  
Namespaces is a logical group of related classes that can be used by any other language targeting the Microsoft .Net framework . It is more used for logical organization of your classes. Namespaces are a way of grouping type names and reducing the chance of name collisions.   
  
Hierarchy and Fully-Qualified Names   
  
The fully qualified name of a class is constructed by concatenating the names of all the namespaces that contain the type. For e.g. the fully qualified name of the TextBox class is System.Windows.Forms.TextBox . That means TextBox class is contained in the Forms namespace that is contained in the Windows namespace that is contained in the root System namespace.   
  
  
Assembly   
  
An assembly is a collection of types and resources that are built to work together and form a logical unit of functionality. It is an Output Unit, that is .exe or .dll file. It is a unit of Deployment and a unit of versioning and also it contain MSIL (Microsoft Intermediate Language) code. Assemblies are self describing, it contains all the metadata about the modules, types, and other elements in the form of a manifest.

**Topic 8: JIT Compiler – research further. Are there different types of JITs?**

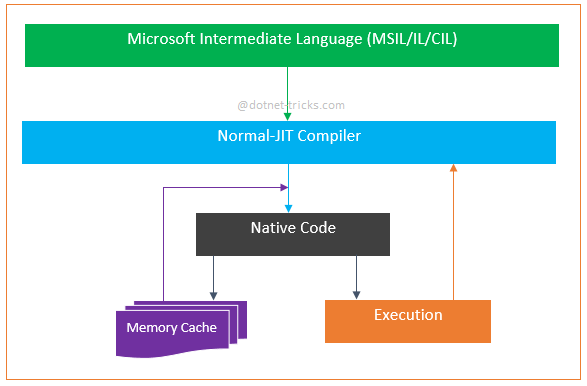
1. Pre JIT Compiler.
2. Econo JIT Compiler.
3. Normal JIT compiler.
4. Pre-JIT:- Pre-JIT complies complete source code into native code in a single compilation cycle. This is done at the time of deployment of the application.



2. Econo-JIT:- Econo-JIT complies only those methods that are called at runtime. However, these complied methods are removed when they are not required.



3. Normal-JIT:- Normal-JIT complies only those methods that are called at runtime. These methods are complied the first time they are called, and then they are stored in cache. When the same methods are called again, the complied code from cache is used for execution.



**Topic 9: Explain what decompile means. Give an example.**

Decompile is the process of converting computer readable code into human readable source code with original code functionality. Decompile is the reverse of compile.

It is impossible use decompilation for original source code reproduction because original code identifiers, such as variable and function names, are rarely stored in computer code. However, code containing metadata or debugging information may include identifiers.

Decompilation is often performed on computer code when source code is no longer available. Thus, it is a form of reverse engineering. Additionally, decompilation may be used for computer security, interoperability and error correction.  
  
Decompilation is applied for multiple reasons, including: 

* Understanding code functionality
* Correcting errors
* Enhancing computer security
* Removing restrictions, such as passwords, copy protection and time limit
* Studying computer viruses
* Recovering lost source code for archiving purposes

Successful decompilation depends on available code information and complexity.  
  
A decompiler (the tool used in decompilation) may be obtained separately, rather than bundled with vendor development tools.  
  
In most cases, copyrighted software decompilation is illegal. However, in the U.S. and Europe, copyright laws allow limited decompilation usage, such as interoperability.

**Topic 10: What is CAS? Explain why is it important.**

To help protect computer systems from malicious mobile code, to allow code from unknown origins to run with protection, and to help prevent trusted code from intentionally or accidentally compromising security, the .NET Framework provides a security mechanism called code access security. Code access security allows code to be trusted to varying degrees depending on where the code originates and on other aspects of the code's identity. Code access security also enforces the varying levels of trust on code, which minimizes the amount of code that must be fully trusted in order to run. Using code access security can reduce the likelihood that your code can be misused by malicious or error-filled code. It can reduce your liability because you can specify the set of operations your code should be allowed to perform as well as the operations your code should never be allowed to perform. Code access security can also help minimize the damage that can result from security vulnerabilities in your code.

All managed code that targets the common language runtime receives the benefits of code access security, even if that code does not make a single code access security call. However, all applications should make code access requests, as outlined in Code Access Security Basics.

Every application that targets the common language runtime must interact with the runtime's security system. When an application executes, it is automatically evaluated and given a set of permissions by the runtime. Depending on the permissions that the application receives, it either runs properly or generates a security exception. The local security settings on a particular computer ultimately decide which permissions code receives. Because these settings can change from computer to computer, you can never be sure that your code will receive sufficient permissions to run. This is in contrast to the world of unmanaged development, in which you do not have to worry about your code's permission to run.

Every developer must be familiar with the following code access security concepts in order to write effective applications targeting the common language runtime:

* Writing type-safe code: To enable code to benefit from code access security, you must use a compiler that generates verifiably type-safe code.
* Imperative and declarative syntax: Interaction with the runtime security system is performed using imperative and declarative security calls. Declarative calls are performed using attributes; imperative calls are performed using new instances of classes within your code. Some calls can be performed only imperatively, while others can be performed only declaratively. Some calls can be performed in either manner.
* Requesting permissions for your code: Requests are applied to the assembly scope, where your code informs the runtime about permissions that it either needs to run or specifically does not want. Security requests are evaluated by the runtime when your code is loaded into memory. Requests cannot influence the runtime to give your code more permissions than the runtime would have given your code had the request not been made. However, requests are what your code uses to inform the runtime about the permissions it requires in order to run.
* Using secure class libraries: Your class libraries use code access security to specify the permissions they require in order to be accessed. You should be aware of the permissions required to access any library that your code uses and make appropriate requests in your code.

**Topic 11\*: How can you prevent an assembly to be decompiled?**

When source code is complied it generates a Portable Executable (PE) file. Portable Executable (PE) is either a dll or an exe. PE file contains MSIL (Microsoft Intermediate Language) and Metadata. MSIL is ultimately converted by CLR into the native code which a processor can understand. Metadata contains assemble information like Assembly Name, Version, Culture and Public Key.

The easy way to reverse engineer and get the exact source code there decompliers available in the market for free such as [Telerik JustDecompile](http://www.telerik.com/products/decompiler.aspx) and [Jet Brains dotPeek](https://www.jetbrains.com/decompiler/) through which can convert the Intermediate Language into the original source code.

The process of protecting the exe and dll from getting decompiled into original source code is called Obfuscation. There are lot of paid and free software available to Obfuscate the .Net assemblies, [Dotfucator](http://www.preemptive.com/products/dotfuscator) from [PreEmptive Solutions](http://www.preemptive.com/) is one of the popular and their community edition is free and included with the visual studio. The Dofuscator community edition has limited features and the professional edition is very expensive. So instead of gaining profits by protecting them from reverse engineering we will end up spending a lot on Obfuscation.

The one of best alternate utility for obfuscating is [ConfuserEx](http://yck1509.github.io/ConfuserEx/) it is a completely free and opensource.