1. Introduction to .Net Framework

The .NET Framework is an integral Windows component that supports building and running the next generation of applications and XML Web services. The .NET Framework is designed to fulfill the following objectives:

- To provide a consistent object-oriented programming environment whether object code is stored and executed locally, executed locally but Internet-distributed, or executed remotely.
- To provide a code-execution environment that minimizes software deployment and versioning conflicts.
- To provide a code-execution environment that promotes safe execution of code, including code created by an unknown or semi-trusted third party.
- To provide a code-execution environment that eliminates the performance problems of scripted or interpreted environments.
- To make the developer experience consistent across widely varying types of applications, such as Windows-based applications and Web-based applications.
- To build all communication on industry standards to ensure that code based on the .NET Framework can integrate with any other code.

The .NET Framework has two main components: the common language runtime and the .NET Framework class library. The common language runtime is the foundation of the .NET Framework. You can think of the runtime as an agent that manages code at execution time, providing core services such as memory management, thread management, and remoting, while also enforcing strict type safety and other forms of code accuracy that promote security and robustness. In fact, the concept of code management is a fundamental principle of the runtime. Code that targets the runtime is known as managed code, while code that does not target the runtime is known as unmanaged code. The class library, the other main component of the .NET Framework, is a comprehensive, object-oriented collection of reusable types that you can use to develop applications ranging from traditional command-line or graphical user interface (GUI) applications to applications based on the latest innovations provided by ASP.NET, such as Web Forms and XML Web services.

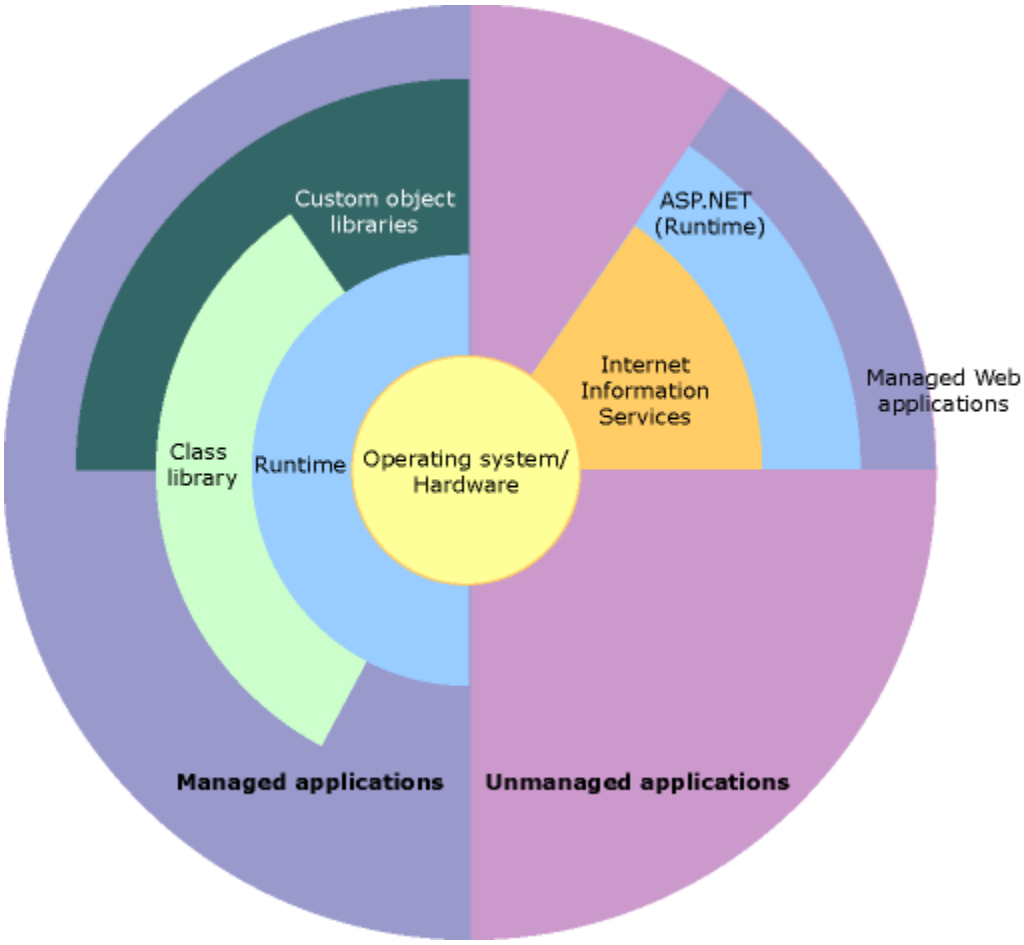
The .NET Framework can be hosted by unmanaged components that load the common language runtime into their processes and initiate the execution of managed code, thereby creating a software environment that can exploit both managed and unmanaged features. The .NET Framework not only provides several runtime hosts, but also supports the development of third-party runtime hosts.

For example, ASP.NET hosts the runtime to provide a scalable, server-side environment for managed code. ASP.NET works directly with the runtime to enable ASP.NET applications and XML Web services, both of which are discussed later in this topic.

Internet Explorer is an example of an unmanaged application that hosts the runtime (in the form of a MIME type extension). Using Internet Explorer to host the runtime enables you to embed managed components or Windows Forms controls in HTML documents. Hosting the runtime in this way makes managed mobile code (similar to Microsoft® ActiveX® controls) possible, but with significant improvements that only managed code can offer, such as semi-trusted execution and isolated file storage.

The following illustration shows the relationship of the common language runtime and the class library to your applications and to the overall system. The illustration also shows how managed code operates within a larger architecture.

.NET Framework in context



The following sections describe the main components and features of the .NET Framework in greater detail.

Features of the Common Language Runtime

The common language runtime manages memory, thread execution, code execution, code safety verification, compilation, and other system services. These features are intrinsic to the managed code that runs on the common language runtime.

With regards to security, managed components are awarded varying degrees of trust, depending on a number of factors that include their origin (such as the Internet, enterprise network, or local computer). This means that a managed component might or might not be able to perform file-access operations, registry-access operations, or other sensitive functions, even if it is being used in the same active application.

The runtime enforces code access security. For example, users can trust that an executable embedded in a Web page can play an animation on screen or sing a song, but cannot access their personal data, file system, or network. The security features of the runtime thus enable legitimate Internet-deployed software to be exceptionally feature rich.

The runtime also enforces code robustness by implementing a strict type-and-code-verification infrastructure called the common type system (CTS). The CTS ensures that all managed code is self-describing. The various Microsoft and third-party language compilers generate managed code that conforms to the CTS. This means that managed code can consume other managed types and instances, while strictly enforcing type fidelity and type safety.

In addition, the managed environment of the runtime eliminates many common software issues. For example, the runtime automatically handles object layout and manages references to objects, releasing them when they are no longer being used. This automatic memory management resolves the two most common application errors, memory leaks and invalid memory references.

The runtime also accelerates developer productivity. For example, programmers can write applications in their development language of choice, yet take full advantage of the runtime, the class library, and components written in other languages by other developers. Any compiler vendor who chooses to target the runtime can do so. Language compilers that target the .NET Framework make the features of the .NET Framework available to existing code written in that language, greatly easing the migration process for existing applications.

While the runtime is designed for the software of the future, it also supports software of today and yesterday. Interoperability between managed and unmanaged code enables developers to continue to use necessary COM components and DLLs.

The runtime is designed to enhance performance. Although the common language runtime provides many standard runtime services, managed code is never interpreted. A feature called just-in-time (JIT) compiling enables all managed code to run in the native machine language of the system on which it is executing. Meanwhile, the memory manager removes the possibilities of fragmented memory and increases memory locality-of-reference to further increase performance.

Finally, the runtime can be hosted by high-performance, server-side applications, such as Microsoft® SQL Server™ and Internet Information Services (IIS). This infrastructure enables you to use managed code to write your business logic, while still enjoying the superior performance of the industry's best enterprise servers that support runtime hosting.

.NET Framework Class Library

The .NET Framework class library is a collection of reusable types that tightly integrate with the common language runtime. The class library is object oriented, providing types from which your own managed code can derive functionality. This not only makes the .NET Framework types easy to use, but also reduces the time associated with learning new features of the .NET Framework. In addition, third-party components can integrate seamlessly with classes in the .NET Framework.

For example, the .NET Framework collection classes implement a set of interfaces that you can use to develop your own collection classes. Your collection classes will blend seamlessly with the classes in the .NET Framework.

As you would expect from an object-oriented class library, the .NET Framework types enable you to accomplish a range of common programming tasks, including tasks such as string management, data collection, database connectivity, and file access. In addition to these common tasks, the class library includes types that support a variety of specialized development scenarios. For example, you can use the .NET Framework to develop the following types of applications and services:

- Console applications.
- Windows GUI applications (Windows Forms).
- ASP.NET applications.
- XML Web services.

- Windows services
- Mobile Applications
- Mobile Web Applications

For example, the Windows Forms classes are a comprehensive set of reusable types that vastly simplify Windows GUI development. If you write an ASP.NET Web Form application, you can use the Web Forms classes.

Client Application Development

Client applications are the closest to a traditional style of application in Windows-based programming. These are the types of applications that display windows or forms on the desktop, enabling a user to perform a task. Client applications include applications such as word processors and spreadsheets, as well as custom business applications such as data-entry tools, reporting tools, and so on. Client applications usually employ windows, menus, buttons, and other GUI elements, and they likely access local resources such as the file system and peripherals such as printers.

Another kind of client application is the traditional ActiveX control (now replaced by the managed Windows Forms control) deployed over the Internet as a Web page. This application is much like other client applications: it is executed natively, has access to local resources, and includes graphical elements.

In the past, developers created such applications using C/C++ in conjunction with the Microsoft Foundation Classes (MFC) or with a rapid application development (RAD) environment such as Microsoft® Visual Basic®. The .NET Framework incorporates aspects of these existing products into a single, consistent development environment that drastically simplifies the development of client applications.

The Windows Forms classes contained in the .NET Framework are designed to be used for GUI development. You can easily create command windows, buttons, menus, toolbars, and other screen elements with the flexibility necessary to accommodate shifting business needs.

For example, the .NET Framework provides simple properties to adjust visual attributes associated with forms. In some cases the underlying operating system does not support changing these attributes directly, and in these cases the .NET Framework automatically recreates the forms. This is one of many ways in which the .NET Framework integrates the developer interface, making coding simpler and more consistent.

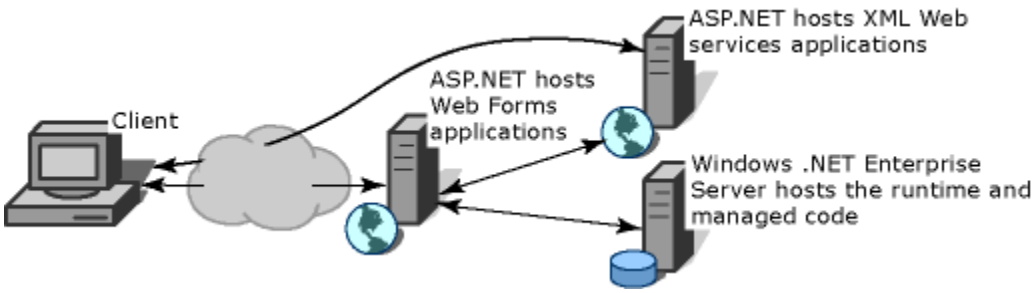
Unlike ActiveX controls, Windows Forms controls have semi-trusted access to a user's computer. This means that binary or natively executing code can access some of the resources on the user's system (such as GUI elements and limited file access) without being able to access or compromise other resources. Because of code access security, many applications that once needed to be installed on a user's system can now be deployed through the Web. Your applications can implement the features of a local application while being deployed like a Web page.

Server Application Development

Server-side applications in the managed world are implemented through runtime hosts. Unmanaged applications host the common language runtime, which allows your custom managed code to control the behavior of the server. This model provides you with all the features of the common language runtime and class library while gaining the performance and scalability of the host server.

The following illustration shows a basic network schema with managed code running in different server environments. Servers such as IIS and SQL Server can perform standard operations while your application logic executes through the managed code.

Server-side managed code



ASP.NET is the hosting environment that enables developers to use the .NET Framework to target Web-based applications. However, ASP.NET is more than just a runtime host; it is a complete architecture for developing Web sites and Internet-distributed objects using managed code. Both Web Forms and XML Web services use IIS and ASP.NET as the publishing mechanism for applications, and both have a collection of supporting classes in the .NET Framework.

XML Web services, an important evolution in Web-based technology, are distributed, server-side application components similar to common Web sites. However, unlike Web-based applications, XML Web services components have no UI and are not targeted for browsers such as Internet Explorer and Netscape Navigator. Instead, XML Web services consist of reusable software components designed to be consumed by other applications, such as traditional client applications, Web-based applications, or even other XML Web services. As a result, XML Web services

technology is rapidly moving application development and deployment into the highly distributed environment of the Internet.

If you have used earlier versions of ASP technology, you will immediately notice the improvements that ASP.NET and Web Forms offer. For example, you can develop Web Forms pages in any language that supports the .NET Framework. In addition, your code no longer needs to share the same file with your HTTP text (although it can continue to do so if you prefer). Web Forms pages execute in native machine language because, like any other managed application, they take full advantage of the runtime. In contrast, unmanaged ASP pages are always scripted and interpreted. ASP.NET pages are faster, more functional, and easier to develop than unmanaged ASP pages because they interact with the runtime like any managed application.

The .NET Framework also provides a collection of classes and tools to aid in development and consumption of XML Web services applications. XML Web services are built on standards such as SOAP (a remote procedure-call protocol), XML (an extensible data format), and WSDL (the Web Services Description Language). The .NET Framework is built on these standards to promote interoperability with non-Microsoft solutions.

For example, the Web Services Description Language tool included with the .NET Framework SDK can query an XML Web service published on the Web, parse its WSDL description, and produce C# or Visual Basic source code that your application can use to become a client of the XML Web service. The source code can create classes derived from classes in the class library that handle all the underlying communication using SOAP and XML parsing. Although you can use the class library to consume XML Web services directly, the Web Services Description Language tool and the other tools contained in the SDK facilitate your development efforts with the .NET Framework.

If you develop and publish your own XML Web service, the .NET Framework provides a set of classes that conform to all the underlying communication standards, such as SOAP, WSDL, and XML. Using those classes enables you to focus on the logic of your service, without concerning yourself with the communications infrastructure required by distributed software development.

Finally, like Web Forms pages in the managed environment, your XML Web service will run with the speed of native machine language using the scalable communication of IIS.

2. Common Language Runtime (CLR)

Compilers and tools expose the runtime's functionality and enable you to write code that benefits from this managed execution environment. Code that you develop with a language compiler that targets the runtime is called managed code; it benefits from features such as cross-language integration, cross-language exception handling, enhanced security, versioning and deployment support, a simplified model for component interaction, and debugging and profiling services.

To enable the runtime to provide services to managed code, language compilers must emit metadata that describes the types, members, and references in your code. Metadata is stored with the code; every loadable common language runtime portable executable (PE) file contains metadata. The runtime uses metadata to locate and load classes, lay out instances in memory, resolve method invocations, generate native code, enforce security, and set run-time context boundaries.

The runtime automatically handles object layout and manages references to objects, releasing them when they are no longer being used. Objects whose lifetimes are managed in this way are called managed data. Garbage collection eliminates memory leaks as well as some other common programming errors. If your code is managed, you can use managed data, unmanaged data, or both managed and unmanaged data in your .NET Framework application. Because language compilers supply their own types, such as primitive types, you might not always know (or need to know) whether your data is being managed.

The common language runtime makes it easy to design components and applications whose objects interact across languages. Objects written in different languages can communicate with each other, and their behaviors can be tightly integrated. For example, you can define a class and then use a different language to derive a class from your original class or call a method on the original class. You can also pass an instance of a class to a method of a class written in a different language. This cross-language integration is possible because language compilers and tools that target the runtime use a common type system defined by the runtime, and they follow the runtime's rules for defining new types, as well as for creating, using, persisting, and binding to types.

As part of their metadata, all managed components carry information about the components and resources they were built against. The runtime uses this information to ensure that your component or application has the specified versions of everything it needs, which makes your code less likely to break because of some unmet dependency. Registration information and state data are no longer stored in the registry where they can be difficult to establish and maintain. Rather, information about

the types you define (and their dependencies) is stored with the code as metadata, making the tasks of component replication and removal much less complicated.

Language compilers and tools expose the runtime's functionality in ways that are intended to be useful and intuitive to developers. This means that some features of the runtime might be more noticeable in one environment than in another. How you experience the runtime depends on which language compilers or tools you use. For example, if you are a Visual Basic developer, you might notice that with the common language runtime, the Visual Basic language has more object-oriented features than before. Following are some benefits of the runtime:

- Performance improvements.
- The ability to easily use components developed in other languages.
- Extensible types provided by a class library.
- New language features such as inheritance, interfaces, and overloading for object-oriented programming; support for explicit free threading that allows creation of multithreaded, scalable applications; support for structured exception handling and custom attributes.

If you use Microsoft® Visual C++® .NET, you can write managed code using the Managed Extensions for C++, which provide the benefits of a managed execution environment as well as access to powerful capabilities and expressive data types that you are familiar with. Additional runtime features include:

- Cross-language integration, especially cross-language inheritance.
- Garbage collection, which manages object lifetime so that reference counting is unnecessary.
- Self-describing objects, which make using Interface Definition Language (IDL) unnecessary.
- The ability to compile once and run on any CPU and operating system that supports the runtime.

You can also write managed code using the C# language, which provides the following benefits:

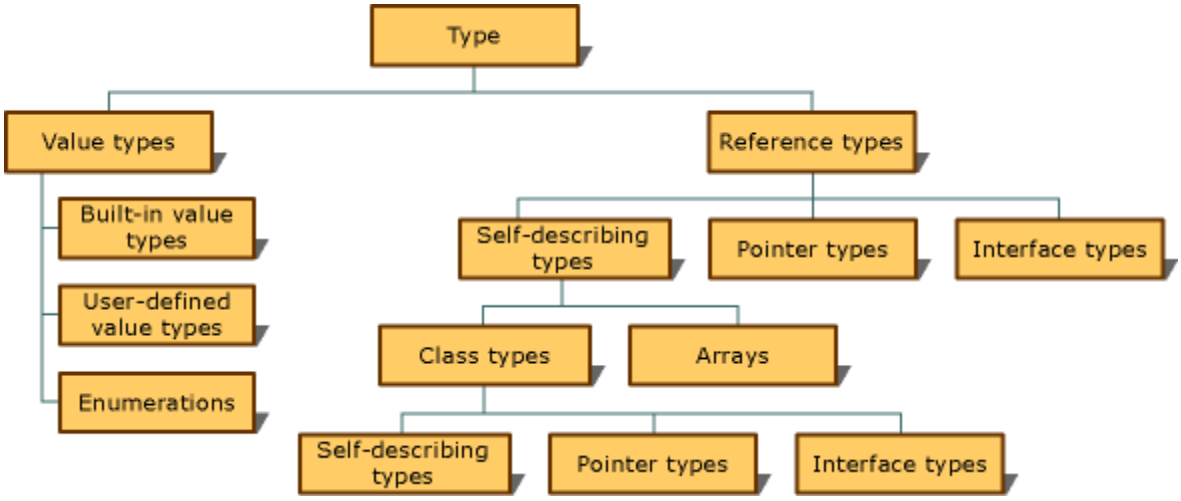
- Complete object-oriented design.
- Very strong type safety.
- A good blend of Visual Basic simplicity and C++ power.

Coalesce

- Garbage collection.
- Syntax and keywords similar to C and C++.
- Use of delegates rather than function pointers for increased type safety and security. Function pointers are available through the use of the **unsafe** C# keyword and the **/unsafe** option of the C# compiler (Csc.exe) for unmanaged code and data.

3. Common Type System (CTS)

Type classification



Values and Objects

Values are binary representations of data, and types provide a way of interpreting this data. A value type is stored directly as a binary representation of the type's data. The value of a reference type is the location of the sequence of bits that represent the type's data.

Every value has an exact type that completely defines the value's representation and the operations that are defined on the value. Values of self-describing types are called objects. While it is always possible to determine the exact type of an object by examining its value, you cannot do so with a value type or pointer type. A value can have more than one type. A value of a type that implements an interface is also a value of that interface type. Likewise, a value of a type that derives from a base type is also a value of that base type.

Types and Assemblies

The runtime uses assemblies to locate and load types. The assembly manifest contains the information that the runtime uses to resolve all type references made within the scope of the assembly.

A type name in the runtime has two logical parts: the assembly name and the name of the type within the assembly. Two types with the same name but in different assemblies are defined as two distinct types.

Coalesce

Assemblies provide consistency between the scope of names seen by the developer and the scope of names seen by the runtime system. Developers author types in the context of an assembly. The content of the assembly a developer is building establishes the scope of names that will be available at run time.

Types and Namespaces

From the viewpoint of the runtime, a namespace is just a collection of type names. Particular languages might have constructs and corresponding syntax that help developers form logical groups of types, but these constructs are not used by the runtime when binding types. Thus, both the **Object** and **String** classes are part of the **System** namespace, but the runtime only recognizes the full names of each type, which are System.Object and System.String, respectively.

You can build a single assembly that exposes types that look like they come from two different hierarchical namespaces, such as System.Collections and System.Windows.Forms. You can also build two assemblies that both export types whose names contain MyDll.MyClass.

If you create a tool to represent types in an assembly as belonging to a hierarchical namespace, the tool must enumerate the types in an assembly or group of assemblies and parse the type names to derive a hierarchical relationship.

4. COMMON LANGUAGE SPECIFICATION

To fully interact with other objects regardless of the language they were implemented in, objects must expose to callers only those features that are common to all the languages they must interoperate with. For this reason, the Common Language Specification (CLS), which is a set of basic language features needed by many applications, has been defined. The CLS rules define a subset of the common type system; that is, all the rules that apply to the common type system apply to the CLS, except where stricter rules are defined in the CLS. The CLS helps enhance and ensure language interoperability by defining a set of features that developers can rely on to be available in a wide variety of languages. The CLS also establishes requirements for CLS compliance; these help you determine whether your managed code conforms to the CLS and to what extent a given tool supports the development of managed code that uses CLS features.

If your component uses only CLS features in the API that it exposes to other code (including derived classes), the component is guaranteed to be accessible from any programming language that supports the CLS. Components that adhere to the CLS rules and use only the features included in the CLS are said to be CLS-compliant components.

Most of the members defined by types in the .NET Framework class library are CLS-compliant. However, some types in the class library have one or more members that are not CLS-compliant. These members enable support for language features that are not in the CLS.

The CLS was designed to be large enough to include the language constructs that are commonly needed by developers, yet small enough that most languages are able to support it. In addition, any language construct that makes it impossible to rapidly verify the type safety of code was excluded from the CLS so that all CLS-compliant languages can produce verifiable code if they choose to do so.

The following table summarizes the features that are in the CLS and indicates whether the feature applies to both developers and compilers (All) or only compilers.

Feature	Applies to	Description
General		
Visibility	All	CLS rules apply only to those parts of a type that are exposed outside the defining assembly.
Global members	All	Global **static** fields and methods are not CLS-compliant.
Naming		
Characters and casing	All	CLS-compliant language compilers must follow the rules of Annex 7 of Technical Report 15 of the Unicode Standard 3.0, which governs the set of characters that can start and

		be included in identifiers. This standard is available at www.unicode.org/unicode/reports/tr15/tr15-18.html. For two identifiers to be considered distinct, they must differ by more than just their case.
Keywords	Compilers	CLS-compliant language compilers supply a mechanism for referencing identifiers that coincide with keywords. CLS-compliant language compilers provide a mechanism for defining and overriding virtual methods with names that are keywords in the language.
Uniqueness	All	All names within a CLS-compliant scope must be distinct, even if the names are for two different kinds of members, except where the names are identical and resolved through overloading. For example, the CLS does not allow a single type to use the same name for a method and a field.
Signatures	All	All return and parameter types appearing in a type or member signature must be CLS-compliant.
Types		
Primitive types	All	The .NET Framework class library includes types that correspond to the primitive data types that compilers use. Of these types, the following are CLS-compliant: Byte, Int16, Int32, Int64, Single, Double, Boolean, Char, Decimal, IntPtr, and String. For more information about these types, see the table of types in .NET Framework Class Library.
Boxed types	All	Boxed value types (value types that have been converted to objects) are not part of the CLS. Instead, use System.Object, System.ValueType, or System.Enum, as appropriate.
Visibility	All	Type and member declarations must not contain types that are less visible or accessible than the type or member being declared.
Interface methods	Compilers	CLS-compliant language compilers must have syntax for the situation where a single type implements two interfaces and each of those interfaces requires the definition of a method with the same name and signature. Such methods must be considered distinct and need not have the same implementation.
Closure	All	The individual members of CLS-compliant interfaces and abstract classes must be defined to be CLS-compliant.
Constructor invocation	All	Before it accesses any inherited instance data, a constructor must call the base class's constructor.
Typed references	All	Typed references are not CLS-compliant. (A typed reference is a special construct that contains a reference to an object and a reference to a type. Typed references enable the common language runtime to provide C++-style support for methods that have a variable number of arguments.)

Coalesce

Type Members

Overloading	All	Indexed properties, methods, and constructors are allowed to be overloaded; fields and events must not be overloaded. Properties must not be overloaded by type (that is, by the return type of their getter method), but they are allowed to be overloaded with different numbers or types of indexes. Methods are allowed to be overloaded only based on the number and types of their parameters. Operator overloading is not in the CLS. However, the CLS provides guidelines about providing useful names (such as Add()) and setting a bit in metadata. Compilers that choose to support operator overloading should follow these guidelines but are not required to do so.
Uniqueness of overloaded members	All	Fields and nested types must be distinct by identifier comparison alone. Methods, properties, and events that have the same name (by identifier comparison) must differ by more than just the return type.
Conversion operators	All	If either **op_Implicit** or **op_Explicit** is overloaded on its return type, an alternate means of providing the conversion must be provided.

Methods

Accessibility of overridden methods	All	Accessibility must not be changed when overriding inherited methods, except when overriding a method inherited from a different assembly with **FamilyOrAssembly** accessibility. In this case, the override must have **Family** accessibility.
Argument lists	All	The only calling convention supported by the CLS is the standard managed calling convention; variable length argument lists are not allowed. (Use the **ParamArray** keyword in Microsoft Visual Basic and the **params** keyword in C# for variable number of arguments support.)

Properties

Accessor metadata	Compilers	The getter and setter methods that implement the methods of a property are marked with the **mdSpecialName** identifier in the metadata.
Accessor accessibility	All	The accessibility of the property and of its accessors must be identical.
Modifiers	All	The property and its accessors must all be **static**, all be **virtual**, or all be **instance**.
Accessor names	All	Properties must follow specific naming patterns. For a property called **Name**, the getter method, if defined, will be called **get_Name** and the setter method, if defined, will

		be called **set_Name**.
Return type and arguments	All	The type of the property is the return type of the getter and the type of the last argument of the setter. The types of the parameters of the property are the types of the parameters to the getter and the types of all but the final parameter of the setter. All these types must be CLS-compliant and cannot be managed pointers; they must not be passed by reference.
Events		
Event methods	All	The methods for adding and removing an event must both be present or absent.
Event method metadata	Compilers	The methods that implement an event must be marked with the **mdSpecialName** identifier in the metadata.
Accessor accessibility	All	The accessibility of the methods for adding, removing, and raising an event must be identical.
Modifiers	All	The methods for adding, removing, and raising an event must all be **static**, all be **virtual**, or all be **instance**.
Event method names	All	Events must follow specific naming patterns. For an event named **MyEvent**, the add method, if defined, will be named **add_MyEvent**, the remove method, if defined, will be named **remove_MyEvent**, and the raise method will be named **raise_MyEvent**.
Arguments	All	The methods for adding and removing an event must each take one parameter whose type defines the type of the event, and that type must be derived from System.Delegate.
Pointer Types		
Pointers	All	Pointer types and function pointer types are not CLS-compliant.
Interfaces		
Member signatures	All	CLS-compliant interfaces must not require the definition of non-CLS-compliant methods in order to implement them.
Member modifiers	All	CLS-compliant interfaces cannot define static methods, nor can they define fields. They are allowed to define properties, events, and virtual methods.
Reference Types		
Constructor invocation	All	For reference types, object constructors are only called as part of the creation of an object, and objects are only initialized once.
Class Types		
Inheritance	All	A CLS-compliant class must inherit from a CLS-compliant class (System.Object is CLS-compliant).
Arrays		
Element types	All	Array elements must be CLS-compliant types.
Dimensions	All	Arrays must have a fixed number of dimensions, greater than zero.
Bounds	All	All dimensions of an array must have a zero lower bound.
Enumerations		
Underlying type	All	The underlying type of an enumeration must be a built-in CLS integer type (Byte, Int16, Int32, or Int64).

Coalesce

FlagsAttribute	Compilers	The presence of the System.FlagsAttribute custom attribute on the definition of an enumeration indicates that the enumeration should be treated as a set of bit fields (flags), and the absence of this attribute indicates the type should be viewed as a group of enumerated constants. It is recommended that languages use either the **FlagsAttribute** or language-specific syntax for distinguishing between these two types of enumerations.
Field members	All	Literal **static** fields of an enumeration must be the same type as the type of the enumeration itself.
Exceptions		
Inheritance	All	Objects that are thrown must be of type System.Exception or inherit from **System.Exception**.
Custom Attributes		
Value encodings	Compilers	CLS-compliant compilers are required to deal with only a subset of the encodings of custom attributes (the representation of custom attributes in metadata). The only types that are permitted to appear in these encodings are: System.Type, System.String, System.Char, System.Boolean, System.Byte, System.Int16, System.Int32, System.Int64, System.Single, System.Double, and any enumeration type based on a CLS-compliant base integer type.
Metadata		
CLS compliance	All	Types whose CLS compliance differs from that of the assembly in which they are defined must be so marked with the System.CLSCompliantAttribute. Similarly, members whose CLS compliance differs from that of their type must also be marked. If a member or type is marked as not CLS-compliant, a CLS-compliant alternative must be provided.

5. Cross language Interoperability

Language interoperability is the ability of code to interact with code that is written using a different programming language. Language interoperability can help maximize code reuse and, therefore, improve the efficiency of the development process.

Because developers use a wide variety of tools and technologies, each of which might support different features and types, it has historically been difficult to ensure language interoperability. However, language compilers and tools that target the common language runtime benefit from the runtime's built-in support for language interoperability.

The common language runtime provides the necessary foundation for language interoperability by specifying and enforcing a common type system and by providing metadata. Because all languages targeting the runtime follow the common type system rules for defining and using types, the usage of types is consistent across languages. Metadata enables language interoperability by defining a uniform mechanism for storing and retrieving information about types. Compilers store type information as metadata, and the common language runtime uses this information to provide services during execution; the runtime can manage the execution of multilanguage applications because all type information is stored and retrieved in the same way, regardless of the language the code was written in.

Managed code benefits from the runtime's support for language interoperability in the following ways:

- Types can inherit implementation from other types, pass objects to another type's methods, and call methods defined on other types, regardless of the language the types are implemented in.
- Debuggers, profilers, or other tools are required to understand only one environment — the Microsoft intermediate language (MSIL) and metadata for the common language runtime — and they can support any programming language that targets the runtime.
- Exception handling is consistent across languages. Your code can throw an exception in one language and that exception can be caught and understood by an object written in another language.

Coalesce

Even though the runtime provides all managed code with support for executing in a multilanguage environment, there is no guarantee that the functionality of the types you create can be fully used by the programming languages that other developers use. This is primarily because each language compiler targeting the runtime uses the type system and metadata to support its own unique set of language features. In cases where you do not know what language the calling code will be written in, you are unlikely to know whether the features your component exposes are accessible to the caller. For example, if your language of choice provides support for unsigned integers, you might design a method with a parameter of type **UInt32**; but from a language that has no notion of unsigned integers, that method would be unusable.

To ensure that your managed code is accessible to developers using any programming language, the .NET Framework provides the Common Language Specification (CLS), which describes a fundamental set of language features and defines rules for how those features are used.

6. Framework Class Library

The .NET Framework class library is a library of classes, interfaces, and value types that are included in the Microsoft .NET Framework SDK. This library provides access to system functionality and is designed to be the foundation on which .NET Framework applications, components, and controls are built.

Usage

Exceptions

Thread Safety

Namespaces

The .NET Framework class library provides the following namespaces:

Microsoft.CSharp
: Contains classes that support compilation and code generation using the C# language.

Microsoft.JScript
: Contains classes that support compilation and code generation using the JScript language.

Microsoft.VisualBasic
: Contains classes that support compilation and code generation using the Visual Basic .NET language.

Microsoft.Vsa
: Contains interfaces that allow you to integrate script for the .NET Framework script engines into applications, and to compile and execute code at run time.

Microsoft.Win32
: Provides two types of classes: those that handle events raised by the operating system and those that manipulate the system registry.

System
: Contains fundamental classes and base classes that define commonly used value and reference data types, events and event handlers, interfaces, attributes, and processing exceptions.

  Other classes provide services supporting data type conversion, method parameter manipulation, mathematics, remote and local program invocation, application environment management, and supervision of managed and unmanaged applications.

System.CodeDom
: Contains classes that can be used to represent the elements and structure of a source code document. These elements can be used to model the structure of a source code document that can be output as source code in a supported language using the functionality provided by the System.CodeDom.Compiler namespace.

Coalesce

System.CodeDom.Compiler

Contains types for managing the generation and compilation of source code in supported programming languages. Code generators can each produce source code in a particular programming language based on the structure of Code Document Object Model (CodeDOM) source code models consisting of elements provided by the System.CodeDom namespace.

System.Collections

Contains interfaces and classes that define various collections of objects, such as lists, queues, bit arrays, hashtables and dictionaries.

System.Collections.Specialized

Contains specialized and strongly typed collections; for example, a linked list dictionary, a bit vector, and collections that contain only strings.

System.ComponentModel

Provides classes that are used to implement the run-time and design-time behavior of components and controls. This namespace includes the base classes and interfaces for implementing attributes and type converters, binding to data sources, and licensing components.

System.ComponentModel.Design

Contains classes that developers can use to build custom design-time behavior for components and user interfaces for configuring components at design time. The design time environment provides systems that enable developers to arrange components and configure their properties.

System.ComponentModel.Design.Serialization

Provides types that support customization and control of serialization at design time.

System.Configuration

Provides classes and interfaces that allow you to programmatically access .NET Framework configuration settings and handle errors in configuration files (.config files).

System.Configuration.Assemblies

Contains classes that are used to configure an assembly.

System.Configuration.Install

Provides classes that allow you to write custom installers for your own components. The Installer class is the base class for all custom installers in the .NET Framework.

System.Data

Consists mostly of the classes that constitute the ADO.NET architecture. The ADO.NET architecture enables you to build components that efficiently manage data from multiple data sources. In a disconnected scenario (such as the Internet), ADO.NET provides the tools to request, update, and reconcile data in multiple tier systems. The ADO.NET architecture is also implemented in client applications, such as Windows Forms, or HTML pages created by ASP.NET.

System.Data.Common

Contains classes shared by the .NET Framework data providers. A .NET Framework data provider describes a collection of classes used to access a data source, such as a database, in the managed space.

System.Data.Odbc

Encapsulates the .NET Framework Data Provider for ODBC. A .NET Framework data provider describes a collection of classes used to access a data source, such as a database, in the managed space. Using the OdbcDataAdapter class, you can fill a memory-resident DataSet, which you can use to query and update the data source.

For additional information about how to use this namespace, see the OdbcDataReader, the OdbcCommand, and the OdbcConnection classes.

Note: This namespace is supported only in version 1.1 of the .NET Framework.

System.Data.OleDb
: Encapsulates the .NET Framework Data Provider for OLE DB. The .NET Framework Data Provider for OLE DB describes a collection of classes used to access an OLE DB data source in the managed space.

System.Data.OracleClient
: Encapsulates the .NET Framework Data Provider for Oracle. The .NET Framework Data Provider for Oracle describes a collection of classes used to access an Oracle data source in the managed space.

Note: This namespace is supported only in version 1.1 of the .NET Framework.

System.Data.SqlClient
: Encapsulates the .NET Framework Data Provider for SQL Server. The .NET Framework Data Provider for SQL Server describes a collection of classes used to access a SQL Server database in the managed space.

System.Data.SqlServerCE
: Describes a collection of classes that can be used to access a database in SQL Server CE from Windows CE-based devices in the managed environment. With this namespace you can create SQL Server CE databases on a device and also establish connections to SQL Server databases that are on a device or on a remote server.

Note: This namespace is supported only in version 1.1 of the .NET Framework.

System.Data.SqlTypes
: Provides classes for native data types within SQL Server. These classes provide a safer, faster alternative to other data types. Using the classes in this namespace helps prevent type conversion errors caused in situations where loss of precision could occur. Because other data types are converted to and from SqlTypes behind the scenes, explicitly creating and using objects within this namespace results in faster code as well.

System.Diagnostics
: Provides classes that allow you to interact with system processes, event logs, and performance counters. This namespace also provides classes that allow you to debug your application and to trace the execution of your code. For more information, see the Trace and Debug classes.

System.Diagnostics.SymbolStore
: Provides classes that allow you to read and write debug symbol information, such as source line to Microsoft intermediate language (MSIL) maps. Compilers targeting the .NET Framework can store the debug symbol information into programmer's database (PDB) files. Debuggers and code profiler tools can read the debug symbol information at run time.

System.DirectoryServices
: Provides easy access to Active Directory from managed code. The namespace contains two component classes, DirectoryEntry and DirectorySearcher, which use the Active Directory Services Interfaces (ADSI) technology. ADSI is the set of interfaces that Microsoft provides as a flexible tool for working with a variety of network providers. ADSI gives the administrator the ability to locate and manage resources on a network with relative ease, regardless of the network's size.

System.Drawing
: Provides access to GDI+ basic graphics functionality. More advanced functionality is provided in the System.Drawing.Drawing2D, System.Drawing.Imaging, and System.Drawing.Text namespaces.

System.Drawing.Design

Contains classes that extend design-time user interface (UI) logic and drawing. You can further extend this design-time functionality to create custom toolbox items, type-specific value editors that can edit and graphically represent values of their supported types, or type converters that can convert values between certain types. This namespace provides the basic frameworks for developing extensions to the design-time UI.

System.Drawing.Drawing2D

Provides advanced 2-dimensional and vector graphics functionality. This namespace includes the gradient brushes, the Matrix class (used to define geometric transforms), and the GraphicsPath class.

System.Drawing.Imaging

Provides advanced GDI+ imaging functionality. Basic graphics functionality is provided by the System.Drawing namespace.

System.Drawing.Printing

Provides print-related services. Typically, you create a new instance of the PrintDocument class, set the properties that describe what to print, and call the Print method to actually print the document.

System.Drawing.Text

Provides advanced GDI+ typography functionality. Basic graphics functionality is provided by the System.Drawing namespace. The classes in this namespace allow users to create and use collections of fonts.

System.EnterpriseServices

Provides an important infrastructure for enterprise applications. COM+ provides a services architecture for component programming models deployed in an enterprise environment. This namespace provides .NET Framework objects with access to COM+ services, making the .NET Framework objects more practical for enterprise applications.

System.EnterpriseServices.CompensatingResourceManager

Provides classes that allow you to use a Compensating Resource Manager (CRM) in managed code. A CRM is a service provided by COM+ that enables you to include non-transactional objects in Microsoft Distributed Transaction Coordinator (DTC) transactions. Although CRMs do not provide the capabilities of a full resource manager, they do provide transactional atomicity (all-or-nothing behavior) and durability through the recovery log.

System.EnterpriseServices.Internal

Provides infrastructure support for COM+ services. The classes and interfaces in this namespace are specifically intended to support calls into System.EnterpriseServices from the unmanaged COM+ classes.

System.Globalization

Contains classes that define culture-related information, including the language, the country/region, the calendars in use, the format patterns for dates, currency, and numbers, and the sort order for strings. These classes are useful for writing globalized (internationalized) applications.

System.IO

Contains types that allow synchronous and asynchronous reading and writing on data streams and files.

System.IO.IsolatedStorage

Contains types that allow the creation and use of isolated stores. With these stores, you can read and write data that less trusted code cannot access and prevent the exposure of sensitive information that can be saved elsewhere on the file system. Data is stored in compartments that are isolated by the current user and by the assembly in which the code exists.

System.Management

Coalesce

Provides access to a rich set of management information and management events about the system, devices, and applications instrumented to the Windows Management Instrumentation (WMI) infrastructure.

System.Management.Instrumentation
: Provides the classes necessary for instrumenting applications for management and exposing their management information and events through WMI to potential consumers. Consumers such as Microsoft Application Center or Microsoft Operations Manager can then manage your application easily, and monitoring and configuring of your application is available for administrator scripts or other applications, both managed as well as unmanaged.

System.Messaging
: Provides classes that allow you to connect to, monitor, and administer message queues on the network and send, receive, or peek messages.

System.Net
: Provides a simple programming interface for many of the protocols used on networks today. The WebRequest and WebResponse classes form the basis of what are called pluggable protocols, an implementation of network services that enables you to develop applications that use Internet resources without worrying about the specific details of the individual protocols.

System.Net.Sockets
: Provides a managed implementation of the Windows Sockets (Winsock) interface for developers who need to tightly control access to the network.

System.Reflection
: Contains classes and interfaces that provide a managed view of loaded types, methods, and fields, with the ability to dynamically create and invoke types.

System.Reflection.Emit
: Contains classes that allow a compiler or tool to emit metadata and Microsoft intermediate language (MSIL) and optionally generate a PE file on disk. The primary clients of these classes are script engines and compilers.

System.Resources
: Provides classes and interfaces that allow developers to create, store, and manage various culture-specific resources used in an application.

System.Runtime.CompilerServices
: Provides functionality for compiler writers using managed code to specify attributes in metadata that affect the run-time behavior of the common language runtime. The classes in this namespace are for compiler writers use only.

System.Runtime.InteropServices
: Provides a wide variety of members that support COM interop and platform invoke services. If you are unfamiliar with these services, see Interoperating with Unmanaged Code .

System.Runtime.InteropServices.CustomMarshalers
: Supports the .NET infrastructure and is not intended to be used directly from your code.

System.Runtime.InteropServices.Expando
: Contains the IExpando interface which allows modification of an object by adding or removing its members.

System.Runtime.Remoting
: Provides classes and interfaces that allow developers to create and configure distributed applications.

System.Runtime.Remoting.Activation
: Provides classes and objects that support server and client activation of remote objects.

System.Runtime.Remoting.Channels

Contains classes that support and handle channels and channel sinks, which are used as the transport medium when a client calls a method on a remote object.

System.Runtime.Remoting.Channels.Http

Contains channels that use the HTTP protocol to transport messages and objects to and from remote locations. By default, the HTTP channels encode objects and method calls in SOAP format for transmission, but other encoding and decoding formatter sinks can be specified in the configuration properties of a channel.

System.Runtime.Remoting.Channels.Tcp

Contains channels that use the TCP protocol to transport messages and objects to and from remote locations. By default, the TCP channels encode objects and method calls in binary format for transmission, but other encoding and decoding formatter sinks can be specified in the configuration properties of a channel.

System.Runtime.Remoting.Contexts

Contains objects that define the contexts all objects reside within. A context is an ordered sequence of properties that defines an environment for the objects within it. Contexts are created during the activation process for objects that are configured to require certain automatic services such synchronization, transactions, just-in-time (JIT) activation, security, and so on. Multiple objects can live inside a context.

System.Runtime.Remoting.Lifetime

Contains classes that manage the lifetime of remote objects. Traditionally, distributed garbage collection uses reference counts and pinging for control over the lifetime of objects. This works well when there are a few clients per service, but doesn't scale well when there are thousands of clients per service. The remoting lifetime service associates a lease with each service, and deletes a service when its lease time expires. The lifetime service can take on the function of a traditional distributed garbage collector, and it also adjusts well when the numbers of clients per server increases.

System.Runtime.Remoting.Messaging

Contains classes used to create and remote messages. The remoting infrastructure uses messages to communicate with remote objects. Messages are used to transmit remote method calls, to activate remote objects, and to communicate information. A message object carries a set of named properties, including action identifiers, envoy information, and parameters.

System.Runtime.Remoting.Metadata

Contains classes and attributes that can be used to customize generation and processing of SOAP for objects and fields. The classes of this namespace can be used to indicate the SOAPAction, type output, XML element name, and the method XML namespace URI.

System.Runtime.Remoting.Metadata.W3cXsd2001

Contains the XML Schema Definition (XSD) defined by the World Wide Web Consortium (W3C) in 2001. The XML Schema Part2: Data types specification from W3C identifies format and behavior of various data types. This namespace contains wrapper classes for the data types that conform to the W3C specification. All date and time types conform to the ISO standards specification.

System.Runtime.Remoting.MetadataServices

Contains the classes used by the Soapsuds.exe command line tool and the user code to convert metadata to and from XML schema for the remoting infrastructure.

System.Runtime.Remoting.Proxies

Contains classes that control and provide functionality for proxies. A proxy is a local object that is an image of a remote object. Proxies enable clients to access objects across remoting boundaries.

System.Runtime.Remoting.Services

Contains service classes that provide functionality to the .NET Framework.

System.Runtime.Serialization

Contains classes that can be used for serializing and deserializing objects. Serialization is the process of converting an object or a graph of objects into a linear sequence of bytes for either storage or transmission to another location. Deserialization is the process of taking in stored information and recreating objects from it.

System.Runtime.Serialization.Formatters
: Provides common enumerations, interfaces, and classes that are used by serialization formatters.

System.Runtime.Serialization.Formatters.Binary
: Contains the BinaryFormatter class, which can be used to serialize and deserialize objects in binary format.

System.Runtime.Serialization.Formatters.Soap
: Contains the SoapFormatter class, which can be used to serialize and deserialize objects in the SOAP format.

System.Security
: Provides the underlying structure of the .NET Framework security system, including base classes for permissions.

System.Security.Cryptography
: Provides cryptographic services, including secure encoding and decoding of data, as well as many other operations, such as hashing, random number generation, and message authentication.

System.Security.Cryptography.X509Certificates
: Contains the common language runtime implementation of the Authenticode X.509 v.3 certificate. This certificate is signed with a private key that uniquely and positively identifies the holder of the certificate.

System.Security.Cryptography.Xml
: Contains classes to support the creation and validation of XML digital signatures. The classes in this namespace implement the World Wide Web Consortium Recommendation, "XML-Signature Syntax and Processing", described at http://www.w3.org/TR/xmldsig-core/.

System.Security.Permissions
: Defines classes that control access to operations and resources based on policy.

System.Security.Policy
: Contains code groups, membership conditions, and evidence. These three types of classes are used to create the rules applied by the .NET Framework security policy system. Evidence classes are the input to security policy and membership conditions are the switches; together these create policy statements and determine the granted permission set. Policy levels and code groups are the structure of the policy hierarchy. Code groups are the encapsulation of a rule and are arranged hierarchically in a policy level.

System.Security.Principal
: Defines a principal object that represents the security context under which code is running.

System.ServiceProcess
: Provides classes that allow you to implement, install, and control Windows service applications. Services are long-running executables that run without a user interface. Implementing a service involves inheriting from the ServiceBase class and defining specific behavior to process when start, stop, pause, and continue commands are passed in, as well as custom behavior and actions to take when the system shuts down.

System.Text
: Contains classes representing ASCII, Unicode, UTF-7, and UTF-8 character encodings; abstract base classes for converting blocks of characters to and from blocks of bytes; and a helper class that manipulates and formats String objects without creating intermediate instances of String.

Coalesce

System.Text.RegularExpressions

Contains classes that provide access to the .NET Framework regular expression engine. The namespace provides regular expression functionality that can be used from any platform or language that runs within the Microsoft .NET Framework.

System.Threading

Provides classes and interfaces that enable multithreaded programming. In addition to classes for synchronizing thread activities and access to data (Mutex, Monitor, Interlocked, AutoResetEvent, and so on), this namespace includes a ThreadPool class that allows you to use a pool of system-supplied threads, and a Timer class that executes callback methods on thread pool threads.

System.Timers

Provides the Timer component, which allows you to raise an event on a specified interval.

System.Web

Supplies classes and interfaces that enable browser-server communication. This namespace includes the HTTPRequest class that provides extensive information about the current HTTP request, the HTTPResponse class that manages HTTP output to the client, and the HTTPServerUtility object that provides access to server-side utilities and processes. System.Web also includes classes for cookie manipulation, file transfer, exception information, and output cache control.

System.Web.Caching

Provides classes for caching frequently used data on the server. This includes the Cache class, a dictionary that allows you to store arbitrary data objects, such as hash tables and data sets. It also provides expiration functionality for those objects, and methods that allow you to add and removed the objects. You can also add the objects with a dependency upon other files or cache entries, and perform a callback to notify your application when an object is removed from the Cache.

System.Web.Configuration

Contains classes that are used to set up ASP.NET configuration.

System.Web.Hosting

Provides the functionality for hosting ASP.NET applications from managed applications outside of Microsoft Internet Information Services (IIS).

System.Web.Mail

Contains classes that enable you to construct and send messages using the CDOSYS Message component. The mail message is delivered through either the SMTP mail service built into Microsoft Windows 2000 or through an arbitrary SMTP server. The classes in this namespace can be used either from ASP.NET or from any managed application.

System.Web.Mobile

Contains the core capabilities, including authentication and error-handling, required for building ASP.NET mobile Web applications.

System.Web.Security

Contains classes that are used to implement ASP.NET security in Web server applications.

System.Web.Services

Consists of the classes that enable you to create XML Web services using ASP.NET and XML Web service clients. XML Web services are applications that provide the ability to exchange messages in a loosely coupled environment using standard protocols such as HTTP, XML, XSD, SOAP, and WSDL. XML Web services enable the building of modular applications within and across companies in heterogeneous environments making them interoperable with a broad variety of implementations, platforms and devices. The SOAP-based XML messages of these applications can have well-defined (structured and typed), or loosely defined parts (using arbitrary XML). The ability of the messages to evolve over time

without breaking the protocol is fundamental to the flexibility and robustness of XML Web services as a building block for the future of the Web.

System.Web.Services.Configuration
: Consists of the classes that configure how XML Web services created using ASP.NET run.

System.Web.Services.Description
: Consists of the classes that enable you to publicly describe an XML Web service by using the Web Services Description Language (WSDL). Each class in this namespace corresponds to a specific element in the WSDL specification, and the class hierarchy corresponds to the XML structure of a valid WSDL document.

System.Web.Services.Discovery
: Consists of the classes that allows XML Web service clients to locate the available XML Web services on a Web server through a process called XML Web services Discovery.

System.Web.Services.Protocols
: Consists of the classes that define the protocols used to transmit data across the wire during the communication between XML Web service clients and XML Web services created using ASP.NET.

System.Web.SessionState
: Supplies classes and interfaces that enable storage of data specific to a single client within a Web application on the server. The session state data is used to give the client the appearance of a persistent connection with the application. State information can be stored within local process memory or, for Web farm configurations, out-of-process using either the ASP.NET State Service or a SQL Server database.

System.Web.UI
: Provides classes and interfaces that allow you to create controls and pages that will appear in your Web applications as user interface on a Web page. This namespace includes the Control class, which provides all controls, whether HTML, Web, or User controls, with a common set of functionality. It also includes the Page control, which is generated automatically whenever a request is made for a page in your Web application. Also provided are classes which provide the Web Forms Server Controls data binding functionality, the ability to save the view state of a given control or page, as well as parsing functionality for both programmable and literal controls.

System.Web.UI.Design
: Contains classes that can be used to extend design-time support for Web Forms.

System.Web.UI.Design.WebControls
: Contains classes that can be used to extend design-time support for Web server controls.

System.Web.UI.HtmlControls
: Consists of a collection of classes that allow you to create HTML server controls on a Web Forms page. HTML server controls run on the server and map directly to standard HTML tags supported by most browsers. This allows you to programmatically control the HTML elements on a Web Forms page.

System.Web.UI.MobileControls
: Contains a set of ASP.NET server controls that can intelligently render your application for different mobile devices.

System.Web.UI.MobileControls.Adapters
: Contains the core device adapter classes used by the ASP.NET mobile controls for device customization and extended device support.

System.Web.UI.WebControls
: Contains classes that allow you to create Web server controls on a Web page. Web server controls run on the server and include form controls such as buttons and text boxes. They also include special purpose controls such as a calendar. Because Web server controls run on

the server, you can programmatically control these elements. Web server controls are more abstract than HTML server controls. Their object model does not necessarily reflect HTML syntax.

System.Windows.Forms

Contains classes for creating Windows-based applications that take full advantage of the rich user interface features available in the Microsoft Windows operating system.

System.Windows.Forms.Design

Contains classes that support design-time configuration and behavior for Windows Forms components. These classes consist of: Designer classes that provide support for Windows Forms components, a set of design time services, UITypeEditor classes for configuring certain types of properties, and classes for importing ActiveX controls.

System.Xml

Provides standards-based support for processing XML.

System.Xml.Schema

Contains the XML classes that provide standards-based support for XML Schemas definition language (XSD) schemas.

System.Xml.Serialization

Contains classes that are used to serialize objects into XML format documents or streams.

System.Xml.XPath

Contains the XPath parser and evaluation engine. It supports the W3C XML Path Language (XPath) Version 1.0 Recommendation (www.w3.org/TR/xpath).

System.Xml.Xsl

Provides support for Extensible Stylesheet Transformation (XSLT) transforms. It supports the W3C XSL Transformations (XSLT) Version 1.0 Recommendation (www.w3.org/TR/xslt).

Exceptions

All instance methods in the class library throw an instance of NullReferenceException when an attempt is made to call the method and the underlying object holds a null reference. Because this exception can occur with any instance method, it is not explicitly listed here.

Thread Safety

All public static members (methods, properties, fields, and events) within the .NET Framework support concurrent access within a multithreaded environment. Therefore, any .NET Framework static member can be simultaneously invoked from two threads without encountering race conditions, deadlocks, or crashes.

If you want to use a class that is not thread-safe in a multithreaded environment, you must wrap an instance of the class with code that supplies the necessary synchronization constructs.