Important topics for Inteviews (dotnet, Asp.Net and Sql Server)

Contributor: Neeraj Kaushik

Email: neeraj.kaushik@live.in, neerajkaushi@gmail.com

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

Table of Contents

1. What CLR Does? 5
2. Explain CTS (Common Type System)? 6
3. Explain CLS (Common Language Specification). 7
4. Explain Boxing and unboxing? 9
5. Explain Variables in C#. 9
6. Explain Jump statements in c#. 11
7. What is nullable Type? 14
8. Why does string in .net is immutable? 16
9. Explain string Comparison? 16
10. Explain String Interning? 18
11. Explain String Pooling? 20
12. Explain string builder functioning behind the scene? 20
13. Explain Indexers? 20
14. Explain Iterators? 22
15. Explain secure strings? 24
16. Explain Enumerated types. 25
17. Explain interface. 26
18. Should I design a base type or an interface? 29
19. Explain App domain? 29
20. Explain Threading in dot net? 29
21. What is diffgram? 30
22. How assignment of value is different in value type and reference type? 30
23. Difference between shallow copy and deep copy of object. 32
24. What is use of using keyword? 32
25. What is cloning? 33
26. What is Assembly? Difference between Private and Shared Assembly? How can we make shared assembly? 33
27. Why value type cannot be inherited? 33
28. What is the difference between an event and a delegate? 34
29. What size is .net object? 34
30. When and How to Use Dispose and Finalize in C#? 34
31. What is difference between equivalent of objects and identical of objects? 40
32. What's the difference between the System.Array.CopyTo() and System.Array.Clone()? 40
33. How ado.net maintain transaction of data? 40
34. What is delay signing? 41
35. Can you declare a C++ type destructor in C# like ~MyClass ()? 42
36. What is difference between == and .Equals? 42
37. What is the difference between structures and enumeration? 42
38. Should I make my destructor virtual? 43
39. How do I declare a pure virtual function in C#? 43
40. Where would you use an iHTTPModule, and what are the limitations of any approach you might take in implementing one? 43
41. What is difference between code base security and role base security? Which one is better? 43
42. Is it possible to prevent a browser from caching an aspx page? 43
43. What is the difference between Debug. Write and Trace. Write? 43
44. What is difference between repeater over datalist and datagrid? 44
45. Describe Main Characteristics of static functions? 44
46. What is DataReader? Difference between datareader and dataset? 44
47. What is DLL hell? 44

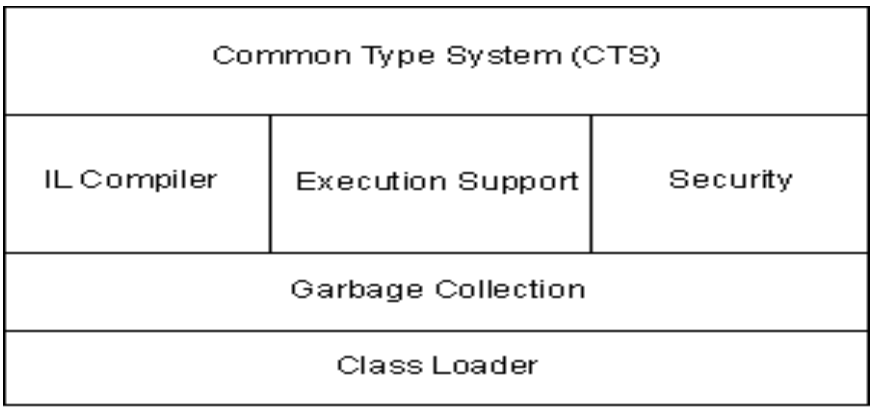
Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

48. What is Temporary Table? How can we create it? 45
49. What is strong key and what is the use of it? 46
50. What is Impersonation? 47
51. What is Partitioned Tables? 47
52. What types of data validation events are commonly seen in the client-side form validation? Web service support 48
54. What is PreProcessor in .NET and type , where it use? 49
55. Please brief not about XSD, XSLT & XML. 50
57. What is Polymorphism? 50
58. What is implicit operator overloading? 50
59. What is Operator Overloading? 52
60. What is difference between http handler and http module? 54
61. What is Caching? Where is it use? Why and when? 55
62. What is Cursor? Define Different type of cursor? 56
63. What is Views? 57
64. What is Triggers? What are the different types of triggers in Sql Server 2005? 61
65. What is use of extern keyword? 63
66. What is base assembly of Dot net? 65
67. What's difference between Shadowing and Overriding? 65
68. What's difference between Server.Transfer and response.Redirect? 65
69. Can you explain in brief how the ASP.NET authentication process works? 66
70. What are the various ways of authentication techniques in ASP.NET? 66
71. How does authorization work in ASP.NET? 67
72. What's difference between Datagrid, Datalist and repeater? 67
73. What exactly happens when ASPX page is requested from Browser? 68
74. What is the result of "select firstname, secondname from emp order by 2"? 70
75. How can we create proxy class without VS? 70
76. How can we create overloaded methods in Web Service? 70
77. How can we maintain State in WebService? 71
78. ASP.Net Page Life Cycle ? 72
79. What are the different data types in dot net? 74
80. What is Static class? 75
81. How can you increase SQL performance? 75
82. What is ACID fundamental and what are transactions in SQL SERVER? 75
83. If we have multiple AFTER Triggers on table how can we define the sequence of the triggers? 76
84. Give the output of following code ? 76
85. Give the output of following code? 76
86. Give output of following code? 77
87. What is Performance tips and tricks in .net application? 78
88. How Garbage Collector Works? 84
89. How objects are allocated on heap? 88
90. What causes finalize methods to be called? 89
91. What is Sync Block Index and object pointer? 90
92. What is JIT compiler? What is its role? 90
93. How Types are stored in dot net? 94
94. Explain structure of GAC in windows? 100
95. What is Advantage and disadvantage of GC? 100
96. What is Reflection? Explain about reflection performance. 101
97. What are members of a type? 102
98. Explain bridge between ISAPI and Application Domains? 102
99. How Securely Implement Request Processing, Filtering, and Content Redirection with HTTP Pipelines in ASP.NET? 107
100. Is there any difference in the way garbage collection works when it is called automatically by the Runtime environment, and when it is invoked intentionally by the programmer? 115
101. What is probing? 115
102. What is compilation and execution procedure for asp.net page? 117

103. If need an array of objects then which option when better ArrayList Class or List Generic Class ? Array[] / ArrayList / List<T> Difference i.e. string[] arraylist.add(string) / List<string> ? 118
104. Custom Paging in ASP.NET 2.0 with SQL Server 2005 123
106. What is the difference between a Struct and a Class? 128
108. What is Web Gardening? How would using it affect a design? 130
109. What is view state? Where it stored? Can we disable it? 130
110. Can you debug a Windows Service? How? 131
111. What are the different ways a method can be overloaded? 134
112. How do you handle data concurrency in .NET? 134
114. What are jagged array? 140
115. Who host CLR? How windows identify where running assembly is managed or not? 140
116. Conceptual view of DotNet Framework. 140
117. Explain Delegates? 145
118. Explaint about Events? 154
119. How to: Connect Event Handler Methods to Events 155
To add an event handler method for an event 156
120. How to: Consume Events in a Web Forms Application 159
To handle a button click event on a Web page 159
121. What's the difference between localization and globalization? 169
122. Difference between primitive types, ref types and value types? 170
123. Difference between gettype() and typeof. 170
124. What is Microsoft SharePoint Portal Server? 171
127. What is difference between static and singleton classes? 172
128. Explain Advantages of WCF. 172
129. What is dependency Injection? 173
130. What is difference between STA & MTA? 175
132. How does a database index work? 178
Database Index Tips 178
133. Give some threading best practices? 179
134. What is object pooling? 184
135. Static and Dynamic Assembly. 187
136. Why we have to consider object in lock statement as private? 188
137. How can we make class immutable? Give Example 188
138. How to override Equal, GetHashCode? 188
139. What is SSL and how can we implement it in our asp.net website?\ 188
140. What is difference between runtime polymorphism and compile time polymorphism? 188
141. What is difference between real proxy and transparent proxy? 188
142. What is prologue code? 188
143. Explain string class or what is behind the scene of string? 188
144. What is Encapsulation, polymorphism and abstraction? Give real time examples? 188
145. Why we use command object? 188
146. What is Satellite Assembly? How application knows which assembly should be loaded for particular culture? 188
147. Http request and response life cycle in ASP.Net? 189
148. If any body is editing data in webpage and at the time of editing row has been explicitly locked by user but if any one stop website processing then how will handle locking? ..... **Error! Bookmark not defined.**
149. Applied use of Inheritance (i.e aggregation, specialization etc) ..... **Error! Bookmark not defined.**
150. Explaing Connection pooling ..... **Error! Bookmark not defined.**
151. Difference between dbnull and null? ..... **Error! Bookmark not defined.**
152. What is snapshot in sql server? ..... **Error! Bookmark not defined.**

1. What CLR Does?

What is Microsoft's Common Language Runtime (CLR)? It is the life line of .NET applications. Before I describe the CLR - let's explain what is meant by runtime. A runtime is an environment in which programs are executed. The CLR is therefore an environment in which we can run our .NET applications that have been compiled to IL. Java programmers are familiar with the JRE (Java Runtime Environment). Consider the CLR as an equivalent to the JRE.



Common Language Runtime (CLR)

The above diagram shows various components of the CLR. Let's discuss each in details. [12] has an in-depth analysis.

- **The Common Type System (CTS)** is responsible for interpreting the data types into the common format - e.g. how many bytes is an integer.
- **The IL Compiler** takes in the IL code and converts it to the host machine language.
- **The execution support** is similar to the language runtime (e.g. in VB the runtime was VBRunxxx.dll; however with VB.NET we do not need individual language runtimes anymore).
- **Security component** in the CLR ensures that the assembly (the program being executed) has permissions to execute certain functions.
- **The garbage collector** is similar to the garbage collector found in Java. Its function is to reclaim the memory when the object is no longer in use; this avoids memory leaks and dangling pointers.
- **The class loader**: Its sole purpose is to load the classes needed by the executing application.

Here's the complete picture.

The programmer must first write the source code and then compile it. Windows programmers have always compiled their programs directly into machine code - but with .NET things have changed. The language compiler would compile the program into an intermediate language "MSIL" or simply "IL" (much like Java Byte code). The IL is fed to the CLR then CLR would use the IL compiler to convert the IL to the host machine code.

.NET introduces the concept of "managed code" and "unmanaged code". The CLR assumes the responsibility of allocating and de-allocating the memory. Any code that tries to bypass the CLR and attempts to handle these functions itself is considered "unsafe"; and the compiler would not compile the code. If the user insists on bypassing the CLR memory management functionality then he must specifically write such code in using the "unsafe" and "fixed" key words (see C# programmers guide for details). Such a code is called "unmanaged" code, as opposed to "managed code" that relies on CLR to do the memory allocation and de-allocation.

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

The IL code thus produced has two major issues with it. First it does not take advantage of platform specific aspects that could enhance the program execution. (for example if a platform has some complicated graphics rendering algorithm implemented in hardware then a game would run much faster if it exploit this feature; however, since IL cannot be platform specific it can not take advantage of such opportunities). Second issue is that IL can not be run directly on a machine since it is an intermediate code and not machine code. To address these issues the CLR uses an IL compiler. The CLR uses JIT compilers to compile the IL code into native code. In Java the byte code is interpreted by a Virtual Machine (JVM). This interpretation caused Java applications to run extremely slow. The introduction of JIT in JVM improved the execution speed. In the CLR Microsoft has eliminated the virtual machine step. The IL code is compiled to native machine and is not interpreted at all. For such a compilation the CLR uses the following **two JIT compilers:**

- **Econo-JIT:** This compiler has a very fast compilation time; but it produces un-optimized code - thus the program may start quickly but would run slow. This compiler is suitable for running scripts.
- **Standard-JIT:** This compiler has a slow compilation time; but it produces highly optimized code. Most of the times the CLR would use this compiler to run your IL code.

Install Time Compilation: This technique allows CLR to compile your application into native code at the time of installation. So the installation may take a few minutes more - but the code would run at speeds close to a native C/C++ application.

Once your program has been compiled into host machine code, it can begin execution. During execution the CLR provides security and memory management services to your code (unless you have specifically used unmanaged code).

2. Explain CTS (Common Type System)?

Types expose functionality to your applications and other types. Types are the mechanism by which code written in one Programming language can talk to code written in a different programming language. Because types are at the root of the CLR, Microsoft created a formal specification-the Common Type System (CTS)-that describes how types are defined and how they behave.

The CTS specification states that a type can contain zero or more members. Some of the members are as follows:

- Field
- Method
- Property
- Event

The CTS also specifies the rules for type visibility and access to the members of a type. For example, marking a type as *public* (called **public**) exports the type, making it visible and accessible to any assembly:

- Private
- Public
- Protected
- Internal
- Protected Internal

In addition, the CTS define the rules governing type inheritance, virtual methods, object lifetime, and so on. These rules have been designed to accommodate the semantics expressible in modern-day programming languages. In fact, you won't even need to learn the CTS rules perse because the language you choose will expose its own language syntax and type rules in the same way that you're familiar with today. And it will map the language-specific syntax into IL,
The "language" of the CLR, when it emits the assembly during compilation.

Using C++, you can define your own types with their own members. Of course, you could have used C# or Visual Basic to define the same type with the same members. Sure, the syntax you use for defining the type is different depending on the language you choose, but the behavior of the type will be absolutely identical regardless of the language because the

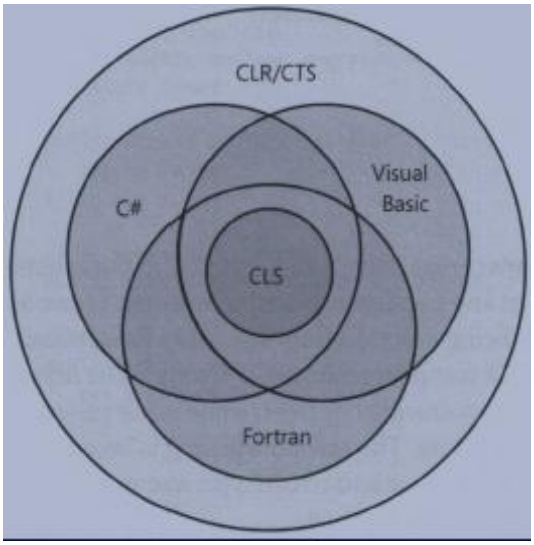
Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

CLR's CTS defines the behavior of the type. To help clarify this idea, let me give you an example. The CTS allows a type to derive from only one base class. So, while the C++ language supports types that can inherit from multiple base types, the CTS can't accept and operate on any such type. To help the developer, Microsoft's C++/CLI compiler reports an error if it detects that you're attempting to create managed code that includes a type deriving from multiple base types. Here's another CTS rule. All types must (ultimately) inherit from a predefined type: **System.Object.** As you can see, **Object** is the name of a type defined in the **System** amespace. This **Object** is the root of all other types and therefore guarantees that every type instance has a minimum set of behaviors. Specifically, the **System.Object** type allows you to do the following:

- Compare two instances for equality.
- Obtain a hash code for the instance.
- Query the true type of an instance.
- Perform a shallow (bitwise) copy of the instance.
- Obtain a string representation of the instance's object's current state.

3. Explain CLS (Common Language Specification).

COM allows objects created in different languages to communicate with one another. On the other hand, the CLR now integrates all languages and allows objects created in one language to be treated as equal citizens by code written in a completely different language. This integration is possible because of the CLR's standard set of types, metadata (self-describing type information), and common execution environment. While this language integration is a fantastic goal, the truth of the matter is that programming languages are very different from one another. For example, some languages don't treat symbols with case-sensitivity, and some don't offer unsigned integers, operator overloading, or methods to support a variable number of arguments. If you intend to create types that are easily accessible from other programming languages, you need to use only features of your programming language that are guaranteed to be available in all other languages. To help you with this, Microsoft has defined a Common Language Specification (CLS) that 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. The CLR/CTS supports a lot more features than the subset defined by the CLS, so if you don't care about interlanguage operability, you can develop very rich types limited only by the language's feature set. Specifically, the CLS defines rules that externally visible types and methods must adhere to if they are to be accessible from any CLS-compliant programming language. Note that the CLS rules don't apply to code that is accessible only within the defining assembly. Figure summarizes the ideas expressed in this paragraph.



As Figure 1-6 shows, the CLR/CTS offers a set of features. Some languages expose a large subset

of the CLR/CTS. A programmer willing to write in IL assembly language, for example, is able to use all of the features the CLR/CTS offers. Most other languages, such as C#, Visual Basic, and Fortran, expose a subset of the CLR/CTS features to the programmer. The CLS defines the minimum set of features that all languages must support. If you're designing a type in one language, and you expect that type to be used by another language, you shouldn't take advantage of any features that are outside of the CLS in its public and protected members. Doing so would mean that your type's members might not be accessible by programmers writing code in other programming languages. In the following code, a CLS-compliant type is being defined in C#. However, the type has a few non-CLS-compliant constructs causing the C# compiler to complain about the code.

```
using System;
// Tell compiler to check for CLS compliance
[assembly: CLSCompliant(true)]
namespace SomeLibrary {
// Warnings appear because the class is public
public sealed class SomeLibraryType {
// Warning: Return type of 'SomeLibrary.SomeLibraryType.Abc()'
// is not CLS-compliant
public UInt32 Abc() { return 0; }
```

In this code, the **[assembly:CLSCompliant(true)]** attribute is applied to the assembly. This attribute tells the compiler to ensure that any publicly exposed type doesn't have any construct that would prevent the type from being accessed from any other programming language. When this code is compiled, the C# compiler emits two warnings. The first warning is reported because the method **Abc** returns an unsigned integer; some other programming languages can't manipulate unsigned integer values. The second warning is because this type exposes two public methods that differ only by case and return type: **Abc** and **abc.** Visual Basic and some other languages can't call both of these methods. Interestingly, if you were to delete **public** from in front of **'sealed class SomeLibraryType'** and recompile, both warnings would go away. The reason is that the **SomeLibraryType** type would default to **internal** and would therefore no longer be exposed outside of the assembly. For a complete list of CLS rules, refer to the "Cross-Language Interoperability" section in the .NET Framework SDK documentation. Let me distill the CLS rules to something very simple. In the CLR, every member of a type is either a field (data) or a method (behavior). This means that every programming language must be able to access fields and call methods. Certain fields and certain methods are used in special and common ways. To ease programming, languages typically offer additional abstractions to make coding these common programming patterns easier. For example, languages expose concepts such as enums, arrays, properties, indexers, delegates, events, constructors, finalizers, operator overloads, conversion operators, and so on. When a compiler comes across any of these things in your source code, it must translate these constructs into fields and methods so that the CLR and any other programming language can access the construct.

Consider the following type definition, which contains a constructor, a finalizer, some overloaded operators, a property, an indexer, and an event. Note that the code shown is there just to make the code compile; it doesn't show the correct way to implement a type.

```
// Warning: Identifier 'someLibrary.SomeLibraryType.abc()'
// differing only in case is not CLS-compliant
public void abc() { }
// No error: this method is private
private UInt32 ABC() { return 0; }
}
}
using System;
internal sealed class Test {
// Constructor
public Test() {}
// Finalizer
~Test() {}

// Operator overload
public static Boolean operator == (Test t1, Test t2) {
return true;
}
```


Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

```
        public static Boolean operator != (Test t1, Test t2) {
        return false;
        }
        // An operator overload
        public static Test operator + (Test t1, Test t2) { return null; }
        // A property
        public String AProperty {
        get { return null; }
        set { }
        }
        // An indexer
        public String this[Int32 x] {
        get { return null; }
        set { }
        }
// An event
event EventHandler AnEvent;
}
```

4. Explain Boxing and unboxing?

A boxing conversion permits any *value-type* to be implicitly converted to the type object or to any *interfacetype* implemented by the *value-type*. Boxing a value of a *value-type* consists of allocating an object instance and copying the *value-type* value into that instance.

int i = 123; object box = i;

An unboxing conversion permits an explicit conversion from type object to any *value-type* or from any *interface-type* to any *value-type* that implements the *interface-type*. An unboxing operation consists of first checking that the object instance is a boxed value of the given *value-type*, and then copying the value out of the instance.

object box = 123; int i = (int)box;

5. Explain Variables in C#.

C# defines seven categories of variables: static variables, instance variables, array elements, value parameters, reference parameters, output parameters, and local variables. The sections that follow describe each of these categories. In the example

```
class A
{
public static int x;
int y;
void F(int[] v, int a, ref int b, out int c) {
int i = 1;
c = a + b++;
}
}
```

x is a static variable, y is an instance variable, v[0] is an array element, a is a value parameter, b is a reference parameter, c is an output parameter, and i is a local variable.

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

Static Variables

A field declared with the static modifier is called a ***static variable***. A static variable comes into existence when the type in which it is declared is loaded , and ceases to exist when the program terminates. The initial value of a static variable is the default value of the variable's type.
For the purpose of definite assignment checking, a static variable is considered initially assigned.

Instance Variables

A field declared without the static modifier is called an ***instance variable***.

- **Instance variables in classes:** An instance variable of a class comes into existence when a new instance of that class is created, and ceases to exist when there are no references to that instance and the instance's destructor (if any) has executed. The initial value of an instance variable of a class is the default value of the variable's type.
- **Instance Variables in struct:** An instance variable of a struct has exactly the same lifetime as the struct variable to which it belongs. In other words, when a variable of a struct type comes into existence or ceases to exist, so too do the instance variables of the struct.

Array Elements

The elements of an array come into existence when an array instance is created, and cease to exist when there are no references to that array instance.

Value Parameter

A parameter declared without a ref or out modifier is a ***value parameter***.

A value parameter comes into existence upon invocation of the function member (§7.4) to which the parameter belongs, and is initialized with the value of the argument given in the invocation. A value parameter ceases to exist upon return of the function member.

Reference Parameter

A parameter declared with a ref modifier is a ***reference parameter***.
A reference parameter does not create a new storage location. Instead, a reference parameter represents the same storage location as the variable given as the argument in the function member invocation. Thus, the value of a reference parameter is always the same as the underlying variable.
The following definite assignment rules apply to reference parameters. Note the different rules for output parameters.

- A variable must be definitely assigned before it can be passed as a reference parameter in a function member invocation.
- Within a function member, a reference parameter is considered initially assigned. Within an instance method or instance accessor of a struct type, the this keyword behaves exactly as a reference parameter of the struct type.

Output Parameter

A parameter declared with an out modifier is an ***output parameter***.
An output parameter does not create a new storage location. Instead, an output parameter represents the same storage location as the variable given as the argument in the function member invocation. Thus, the value of an output parameter is always the same as the underlying variable.

The following definite assignment rules apply to output parameters. Note the different rules for reference parameters.

- A variable need not be definitely assigned before it can be passed as an output parameter in a function member invocation.

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

- Following a function member invocation, each variable that was passed as an output parameter is considered assigned in that execution path.
- Within a function member, an output parameter is considered initially unassigned.
- Every output parameter of a function member must be definitely assigned before the function member returns.

Within an instance constructor of a struct type, “this” keyword behaves exactly as an output parameter of the struct type.

Local Variables

A ***local variable*** is declared by a *local-variable-declaration*, which may occur in a *block*, a *for-statement*, a *switch-statement*, or a *using-statement*. A local variable comes into existence when control enters the *block*, *forstatement*, *switch-statement*, or *using-statement* that immediately contains the local variable declaration. A local variable ceases to exist when control leaves its immediately containing *block*, *for-statement*, *switch-statement*, or *using-statement*.A local variable is not automatically initialized and thus has no default value. For the purpose of definite assignment checking, a local variable is considered initially unassigned. A *local-variable-declaration* may include a *variable-initializer*, in which case the variable is considered definitely assigned in its entire scope, except within the expression provided in the *variable-initializer*. Within the scope of a local variable, it is an error to refer to the local variable in a textual position that precedes its *variable-declarator*.

6. Explain Jump statements in c#.

The Break statement

The break statement exits the nearest enclosing switch, while, do, for, or foreach statement.

break-statement:
break ;

The target of a break statement is the end point of the nearest enclosing switch, while, do, for, or foreach statement. If a break statement is not enclosed by a switch, while, do, for, or foreach statement, a compile-time error occurs.
When multiple switch, while, do, for, or foreach statements are nested within each other, a break statement applies only to the innermost statement. To transfer control across multiple nesting levels, a goto statement must be used.
A break statement cannot exit a finally block . When a break statement occurs within a finally block, the target of the break statement must be within the same finally block. Otherwise, a compile-time error occurs.
A break statement is executed as follows:

- If the break statement exits one or more try blocks with associated finally blocks, control is initially transferred to the finally block of the innermost try statement. When and if control reaches the end point of a finally block, control is transferred to the finally block of the next enclosing try statement. This process is repeated until the finally blocks of all intervening try statements have been executed.
- Control is transferred to the target of the break statement.

Because a break statement unconditionally transfers control elsewhere, the end point of a break statement is never reachable.

The Continue Statement

The continue statement starts a new iteration of the nearest enclosing while, do, for, or foreach statement.

continue-statement:
continue ;

The target of a continue statement is the end point of the embedded statement of the nearest enclosing while, do, for, or foreach statement. If a continue statement is not enclosed by a while, do, for, or foreach statement, a compile-time error occurs.
When multiple while, do, for, or foreach statements are nested within each other, a continue statement applies only to the innermost statement. To transfer control across multiple nesting levels, a goto statement must be used. A continue statement cannot exit a finally block (§8.10). When a continue statement occurs within a finally block, the target of the continue statement must be within the same finally block. Otherwise a compile-time error occurs.

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

A continue statement is executed as follows:

- If the continue statement exits one or more try blocks with associated finally blocks, control is initially transferred to the finally block of the innermost try statement. When and if control reaches the end point of a finally block, control is transferred to the finally block of the next enclosing try statement. This process is repeated until the finally blocks of all intervening try statements have been executed.
- Control is transferred to the target of the continue statement.

Because a continue statement unconditionally transfers control elsewhere, the end point of a continue statement is never reachable.

The Goto statement

The goto statement transfers control to a statement that is marked by a label.
goto-statement:
goto *identifier* ;
goto case *constant-expression ;*
goto default ;
The target of a goto *identifier* statement is the labeled statement with the given label. If a label with the given name does not exist in the current function member, or if the goto statement is not within the scope of the label, a compile-time error occurs. This rule permits the use of a goto statement to transfer control *out of* a nested scope, but not *into* a nested scope. In the example

```
class Test
{
static void Main(string[] args) {
string[,] table = { {"red", "blue", "green"},
{"Monday", "Wednesday", "Friday"} };
foreach (string str in args) {
int row, colm;
for (row = 0; row <= 1; ++row) {
for (colm = 0; colm <= 2; ++colm) {
if (str == table[row,colm]) {
goto done;
}
}
}
Console.WriteLine("{0} not found", str);
continue;
done:
Console.WriteLine("Found {0} at [{1}][{2}]", str, row, colm);
}
}
```

}a goto statement is used to transfer control out of a nested scope.
The target of a goto case statement is the statement list in the immediately enclosing switch statement which contains a case label with the given constant value. If the goto case statement is not enclosed by a switch statement, if the *constant-expression* is not implicitly convertible (§6.1) to the governing type of the nearest enclosing switch statement, or if the nearest enclosing switch statement does not contain a case label with the given constant value, a compile-time error occurs.
The target of a goto default statement is the statement list in the immediately enclosing switch statement which contains a default label. If the goto default statement is not enclosed by a switch statement, or if the nearest enclosing switch statement does not contain a default label, a compile-time error occurs.
A goto statement cannot exit a finally block. When a goto statement occurs within a finally block, the target of the goto statement must be within the same finally block, or otherwise a compile-time error occurs.
A goto statement is executed as follows:

- If the goto statement exits one or more try blocks with associated finally blocks, control is initially transferred to the finally block of the innermost try statement. When and if control reaches the end point of a finally block, control

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

is transferred to the finally block of the next enclosing try statement. This process is repeated until the finally blocks of all intervening try statements have been executed.

- Control is transferred to the target of the goto statement.

Because a goto statement unconditionally transfers control elsewhere, the end point of a goto statement is never reachable.

The Return statement

The return statement returns control to the caller of the function member in which the return statement appears.
return-statement:
return *expressionopt* ;
A return statement with no expression can be used only in a function member that does not compute a value, that is, a method with the return type void, the set accessor of a property or indexer, the add and remove accessors of an event, an instance constructor, a static constructor, or a destructor. A return statement with an expression can be used only in a function member that computes a value, that is, a method with a non-void return type, the get accessor of a property or indexer, or a user-defined operator. An implicit conversion must exist from the type of the expression to the return type of the containing function member.
It is an error for a return statement to appear in a finally block.
A return statement is executed as follows:

- If the return statement specifies an expression, the expression is evaluated and the resulting value is converted to the return type of the containing function member by an implicit conversion. The result of the conversion becomes the value returned to the caller.
- If the return statement is enclosed by one or more try blocks with associated finally blocks, control is initially transferred to the finally block of the innermost try statement. When and if control reaches the end point of a finally block, control is transferred to the finally block of the next enclosing try statement. This process is repeated until the finally blocks of all enclosing try statements have been executed.
- Control is returned to the caller of the containing function member.

Because a return statement unconditionally transfers control elsewhere, the end point of a return statement is never reachable

The throw statement

The throw statement throws an exception.

throw-statement:
throw *expressionopt* ;

A throw statement with an expression throws the value produced by evaluating the expression. The expression must denote a value of the class type System.Exception or of a class type that derives from System.Exception. If evaluation of the expression produces null, a System.NullReferenceException is thrown instead.
A throw statement with no expression can be used only in a catch block, in which case it re-throws the exception that is currently being handled by the catch block. Because a throw statement unconditionally transfers control elsewhere, the end point of a throw statement is never reachable.
When an exception is thrown, control is transferred to the first catch clause in an enclosing try statement that can handle the exception. The process that takes place from the point of the exception being thrown to the point of transferring control to a suitable exception handler is known as ***exception propagation***. Propagation of an exception consists of repeatedly evaluating the following steps until a catch clause that matches the exception is found. In this description, the ***throw point*** is initially the location at which the exception is thrown.

- In the current function member, each try statement that encloses the throw point is examined. For each statement S, starting with the innermost try statement and ending with the outermost try statement, the following steps are evaluated:
    - if the try block of S encloses the throw point and if S has one or more catch clauses, the catch clauses are examined in order of appearance to locate a suitable handler for the exception. The first catch clause that specifies the exception type or a base type of the exception type is considered a match. A general

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

catch clause is considered a match for any exception type. If a matching catch clause is located, the exception propagation is completed by transferring control to the block of that catch clause.
    - Otherwise, if the try block or a catch block of S encloses the throw point and if S has a finally block, control is transferred to the finally block. If the finally block throws another exception, processing of the current exception is terminated. Otherwise, when control reaches the end point of the finally block, processing of the current exception is continued.
- If an exception handler was not located in the current function member invocation, the function member invocation is terminated. The steps above are then repeated for the caller of the function member with a throw point corresponding to the statement from which the function member was invoked.
- If the exception processing terminates all function member invocations in the current thread, indicating that the thread has no handler for the exception, then the thread is itself terminated. The impact of such termination is implementation-defined.

7. What is nullable Type?

Nullable types are instances of the System..::.Nullable<(Of <(T>)>) struct. A nullable type can represent the correct range of values for its underlying value type, plus an additional **null** value. For example, a Nullable<Int32>, pronounced "Nullable of Int32," can be assigned any value from -2147483648 to 2147483647, or it can be assigned the **null** value. A Nullable<bool> can be assigned the values truefalse, or null. The ability to assign **null** to numeric and Boolean types is especially useful when you are dealing with databases and other data types that contain elements that may not be assigned a value. For example, a Boolean field in a database can store the values **true** or **false**, or it may be undefined. This small change could be a huge help for those who deal with databases containing fields that are optional. Nullable data types can also be helpful in other situations as well.

To make myInt be able to store a null value, you would declare it as such:

int? myNullableInt = 1;

Using a Nullable Type

A nullable type can be used in the same way that a regular value type can be used. In fact, implicit conversions are built in for converting between a nullable and non-nullable variable of the same type. This means you can assign a standard integer to a nullable integer and vice-versa:

int? nFirst = null;

int Second = 2; nFirst = Second;   // Valid

nFirst = 123;   // Valid

Second = nFirst;   // Also valid

nFirst = null;   // Valid

Second = nFirst;   // Exception, Second is nonnullable.

In looking at the above statements, you can see that a nullable and nonnullable variable can exchange values as long as the nullable variable does not contain a null. If it contains a null, an exception is thrown. To help avoid throwing an exception, you can use the nullable's HasValue property:

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

if (nFirst.HasValue) Second = nFirst;

As you can see, if nFirst has a value, the assignment will happen; otherwise, the assignment is skipped.

Using Operators with Nullable Values: Lifted Operators

In addition to the automatic conversions between a nullable and non-nullable variable of the same value type, there are also changes with the operators to allow them to work with nullable and non-nullable values. These operators are called *lifted operators*.

Consider the following code:

int ValA = 10;

int? ValB = 3;

int? ValC = ValA * ValB;

What is stored in Val C? The value of 30 would be stored into ValC. The standard operators have been modified so that they "lift" the non-nullable values to being nullable and thus allow the standard operations to work. Now, consider the following change:

int ValA = 10;

int? ValB = null;

int? ValC = ValA * ValB;

What would ValC contain this time? ValC would contain null. In the case where either operand is null, the result of a lifted operation will also be null. Even if you were doing addition or subtraction, it would still be null. So, ValA + ValB using the above values would result in null, not 10.

What if ValC were not a nullable type? What does the following do then?

int ValA = 10;

int? ValB = null;

int ValC = ValA * ValB; // ValC not nullable

This code would actually throw an exception. The result of ValA * ValB is null and a null can't be assigned to a non-nullable type. As such, an exception is thrown.

Removing Nullability

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

C# also gets an additional operator in its newest version. This is the ?? operator used for null coalescing. The null coalescing operator takes the following format:

returnValue = first ?? second;

In this case, if first is not null, its value will be returned to returnValue. If first is null, then the value of second will be returned. You should note that returnValue can be either a nullable or non-nullable variable.

If you wanted to move a nullable varaible's value to a non-nullable version, you could do the following:

int? ValA= 123;

int? ValB = null;

int NewVarA = ValA ?? -1;

int NewVarB = ValB ?? -1;

When the above is completed, NewVarA will contain the value of 123 because ValA was not null. NewVarB will contain the value of -1 because ValB was null. As you can see, this allows you to change variables with a null value to a defaulted value. In this case, the defaulted value is -1.

8. Why does string in .net is immutable?

The most important thing to know about a **String** object is that it is immutable. That is, once created, a string can never get longer, get shorter, or have any of its characters changed. Having immutable strings offers several benefits. First, it allows you to perform operations on a string without actually changing the string:

if (s.ToUpperInvariant().SubString(10, 21).EndsWith("EXE")) {...}
Here, **ToUpperInvariant** returns a new string; it doesn't modify the characters of the string **s. SubString** operates on the string returned by **ToUpperInvariant** and also returns a new string, which is then examined by **EndsWith.** The two temporary strings created by **ToUpper**- **Invariant** and **SubString** are not referenced for long by the application code, and the garbage collector will reclaim their memory at the next collection. If you perform a lot of string manipulations, you end up creating a lot of **String** objects on the heap, which causes more frequent garbage collections, thus hurting your application's performance. To perform a lot of string manipulations efficiently, use the **StringBuilder** class.
Having immutable strings also means that there are no thread synchronization issues when manipulating or accessing a string. In addition, it's possible for the CLR to share multiple identical **String** contents through a single **String** object. This can reduce the number of strings in the system—thereby conserving memory usage-and it is what string interning

For performance reasons, the **String** type is tightly integrated with the CLR. Specifically, the CLR knows the exact layout of the fields defined within the **String** type, and the CLR accesses these fields directly. This performance and direct access come at a small development cost: the **String** class is sealed. If you were able to define your own type, using **String** as a base type, you could add your own fields, which would break the CLR's assumptions. In addition, you could break some assumptions that the CLR team has made about **String** objects being immutable.

9. Explain string Comparison?

Comparing is probably the most common operation performed on strings. There are two reasons to compare two strings with each other. We compare two strings to determine equality or to sort them (usually for presentation to a user). In determining string equality or when comparing strings for sorting, I highly recommend

that you call one of these methods (defined by the **String** class):

Boolean Equals(String value, StringComparison comparisonType)
static Boolean Equals(String a, String b,
StringComparison comparisonType)
static Int32 Compare(String strA, String strB,
StringComparison comparisonType)
static Int32 Compare(String strA, String strB,
Boolean ignoreCase, CultureInfo culture)
static Int32 Compare(String strA, Int32 indexA,
String strB, Int32 indexB, Int32 length, StringComparison comparisonType)
static Int32 Compare(String strA, Int32 indexA, String strB,
Int32 indexB, Int32 length, Boolean ignoreCase, CultureInfo culture)
Boolean StartsWith(String value, StringComparison comparisonType)
Boolean StartsWith(String value,
Boolean ignoreCase, CultureInfo culture)
Boolean EndsWith(String value, StringComparison comparisonType)
Boolean EndsWith(String value, Boolean ignoreCase, CultureInfo culture)

When sorting, you should always perform case-sensitive comparisons. The reason is that if two strings differing only by case are considered to be equal, they could be ordered differently each time you sort them; this would confuse the user. The **comparisonType** argument (in most of the methods shown above) is one of the values defined by the **StringComparison** enumerated type, which is defined as follows:

```
public enum StringComparison {
CurrentCulture = 0,
CurrentCultureIgnoreCase = 1,
InvariantCulture = 2,
InvariantCultureIgnoreCase = 3,
Ordinal = 4,
OrdinalIgnoreCase = 5
}
```

Many programs use strings for internal programmatic purposes such as path names, file names, URLs, registry keys and values, environment variables, reflection, XML tags, XML attributes, and so on. Often, these strings are not shown to a user and are used only within the program. When comparing programmatic strings, you should always use **StringComparison .Ordinal** or **StringComparison.OrdinalIgnoreCase.** This is the fastest way to perform a comparison that is not to be affected in any linguistic way because culture information is not taken into account when performing the comparison. On the other hand, when you want to compare strings in a linguistically correct manner (usuallyfor display to an end user), you should use **StringComparison.CurrentCulture** or **StringComparison.CurrentCultureIgnoreCase.**

Important For the most part, **StringComparison.InvariantCulture** and **StringComparison.InvariantCultureIgnoreCase** should not be used. Although these values cause the comparison to be linguistically correct, using them to compare programmatic strings takes longer than performing an ordinal comparison. Furthermore, the invariant culture is culture agnostic, which makes it an incorrect choice when working with strings that you want to show to an end user.

Important If you want to change the case of a string's characters before performing an ordinal comparison, you should use **String**'s **ToUpperInvariant** or **ToLowerInvariant** method. When normalizing strings, it is highly recommended that you use **ToUpperInvariant** instead of **ToLowerInvariant** because Microsoft has optimized the code for performing uppercase comparisons. In fact, the FCL normalizes strings to uppercase prior to performing case-insensitive comparisons.

Sometimes, when you compare strings in a linguistically correct manner, you want to specify a specific culture rather than use a culture that is associated with the calling thread. In this case, you can use the overloads of the **StartsWith, EndsWith,** and **Compare** methods shown earlier, all of which take **Boolean** and **CultureInfo** arguments.

Important The **String** type defines several overloads of the **Equals, StartsWith, EndsWith,** and **Compare** methods in addition to the versions shown earlier. Microsoft recommends that these other versions (not shown in this book) be

avoided. Furthermore, **String**'s other comparison methods—**CompareTo** (required by the **IComparable** interface), **CompareOrdinal,** and the == and ! = operators—should also be avoided. The reason for avoiding these methods and operators is because the caller does not explicitly indicate how the string comparison should be performed, and you cannot determine from the name of the method what the default comparison will be. For example, by default, **CompareTo** performs a culture-sensitive comparison, whereas **Equals** performs an ordinal comparison. Your code will be easier to read and maintain if you always indicate explicitly how you want to perform your string comparisons.
The following code demonstrates the difference between performing an ordinal comparison
and a culturally aware string comparison:

```
using System;
using System.Globalization;
public static class Program {
public static void Main() {
String s1 = "Strasse";
String s2 = "Straße";
Boolean eq;
// CompareOrdinal returns nonzero.
eq = String.Compare(s1, s2, StringComparison.Ordinal) == 0;
Console.WriteLine("Ordinal comparison: '{0}' {2} '{1}'", s1, s2,
eq ? "==" : "!=");
// Compare Strings appropriately for people
// who speak German (de) in Germany (DE)
CultureInfo ci = new CultureInfo("de-DE");
// Compare returns zero.
eq = String.Compare(s1, s2, true, ci) == 0;
Console.WriteLine("Cultural comparison: '{0}' {2} '{1}'", s1, s2,
eq ? "==" : "!=");
}
}
```

CurrentCulture This property is used for everything that CurrentUICulture isn't used for, including number and date formatting, string casing, and string comparing. When formatting, both the language and country parts of the CultureInfo object are used. By default, when you create a thread, this thread property is set to a CultureInfo object, whose value is determined by calling the Win32 GetUserDefaultLCID method, whose value is also set in the Regional Options tab of the Regional and Language Options application in Windows Control Panel.
Building and running this code produces the following output:
Ordinal comparison: 'Strasse' != 'Straße'
Cultural comparison: 'Strasse' == 'Straße'

10. Explain String Interning?

As I said in the preceding section, checking strings for equality is a common operation for many applications—this task can hurt performance significantly. When performing an ordinal equality check, the CLR quickly tests to see if both strings have the same number of characters. If they don't, the strings are definitely not equal; if they do, the strings might be equal, and the CLR must then compare each individual character to determine for sure. When performing
a culturally aware comparison, the CLR must always compare all of the individual characters because strings of different lengths might be considered equal.
In addition, if you have several instances of the same string duplicated in memory, you're wasting memory because strings are immutable. You'll use memory much more efficiently if there is just one instance of the string in memory and all variables needing to refer to the string can just point to the single string object.
If your application frequently compares strings for equality using case-sensitive, ordinal comparisons, or if you expect to have many string objects with the same value, you can enhance performance substantially if you take advantage of the *string interning* mechanism in the CLR. When the CLR initializes, it creates an internal hash table in which the keys are strings and the values are references to **String** objects in the managed heap. Initially, the table is empty (of course). The **String** class offers two methods that allow you to access this internal hash table:

```
public static String Intern(String str);
public static String IsInterned(String str);
```

The first method, **Intern,** takes a **String,** obtains a hash code for it, and checks the internal hash table for a match. If an identical string already exists, a reference to the already existing **String** object is returned. If an identical string doesn't exist, a copy of the string is made, the copy is added to the internal hash table, and a reference to this copy is returned. If the application no longer holds a reference to the original **String** object, the garbage collector is able to free the memory of that string. Note that the garbage collector can't free the strings that the internal hash table refers to because the hash table holds the reference to those **String** objects. **String** objects referred to by the internal hash table can't be freed until the
AppDomain is unloaded or the process terminates. As does the **Intern** method, the **IsInterned** method takes a **String** and looks it up in the internal hash table. If a matching string is in the hash table, **IsInterned** returns a reference to the interned string object. If a matching string isn't in the hash table, however, **IsInterned** returns **null;** it doesn't add the string to the hash table.
By default, when an assembly is loaded, the CLR interns all of the literal strings described in the assembly's metadata. Microsoft learned that this hurts performance significantly due to the additional hash table lookups, so it is now possible to turn this "feature" off. If an assembly is marked with a **System.Runtime.CompilerServices.CompilationRelaxationsAttribute** specifying the **System.Runtime.CompilerServices.CompilationRelaxations.NoString-Interning** flag value, the CLR *may,* according to the ECMA specification, choose not to intern all of the strings defined in that assembly's metadata. Note that, in an attempt to improve your application's performance, the C# compiler always specifies this attribute/flag whenever you compile an assembly.
Even if an assembly has this attribute/flag specified, the CLR may chose to intern the strings, but you should not count on this. In fact, you really should never write code that relies on strings being interned unless you have written code that explicitly calls the **String**'s **Intern** method yourself. The following code demonstrates string interning:

```
String s1 = "Hello";
String s2 = "Hello";
Console.WriteLine(Object.ReferenceEquals(s1, s2)); // Should be 'False'
s1 = String.Intern(s1);
s2 = String.Intern(s2);
Console.WriteLine(Object.ReferenceEquals(s1, s2)); // 'True'
```

In the first call to the **ReferenceEquals** method, **s1** refers to a **"Hello"** string object in the heap, and **s2** refers to a different **"Hello"** string object in the heap. Since the references are different, **False** should be displayed. However, if you run this on version 2.0 of the CLR, you'll see that **True** is displayed. The reason is because this version of the CLR chooses to ignore the attribute/flag emitted by the C# compiler, and the CLR interns the literal **"Hello"** string when the assembly is loaded into the AppDomain. This means that **s1** and **s2** refer to the single **"Hello"** string in the heap. However, as mentioned previously, you should never write code that relies on this behavior because a future version of the CLR might honor the attribute/flag and not intern the **"Hello"** string. In fact, version 2.0 of the CLR does honor the attribute/flag when this assembly's code has been compiled using the NGen.exe utility. Before the second call to the **ReferenceEquals** method, the **"Hello"** string has been explicitly interned, and **s1** now refers to an interned **"Hello".** Then by calling **Intern** again, **s2** is set to refer to the same **"Hello"** string as **s1.** Now, when **ReferenceEquals** is called the second time, we are guaranteed to get a result of **True** regardless of whether the assembly was compiled with the attribute/flag.

So now, let's look at an example to see how you can use string interning to improve performance and reduce memory usage. The **NumTimesWordAppearsEquals** method below takes two arguments: a word and an array of strings in which each array element refers to a single word. This method then determines how many times the specified word appears in the word list and returns this count:

```
private static Int32 NumTimesWordAppearsEquals(String word, String[] wordlist) {
Int32 count = 0;
for (Int32 wordnum = 0; wordnum < wordlist.Length; wordnum++) {
```


Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

```
if (word.Equals(wordlist[wordnum], StringComparison.Ordinal))
count++;
}
return count;
}
```

11. Explain String Pooling?

When compiling source code, your compiler must process each literal string and emit the string into the managed module's metadata. If the same literal string appears several times in your source code, emitting all of these strings into the metadata will bloat the size of the resulting file.To remove this bloat, many compilers (include the C# compiler) write the literal string into the module's metadata only once. All code that references the string will be modified to refer to the one string in the metadata. This ability of a compiler to merge multiple occurrences of a single string into a single instance can reduce the size of a module substantially. This process is nothing new—C/C++ compilers have been doing it for years. (Microsoft's C/C++ compiler calls this string pooling.) Even so, string pooling is another way to improve the performance of strings and just one more piece of knowledge that you should have in your repertoire.

12. Explain string builder functioning behind the scene?

Because the String type represents an immutable string, the FCL provides another type, System.Text.StringBuilder, which allows you to perform dynamic operations efficiently with strings and characters to create a String. Think of StringBuilder as a fancy constructor to create a String that can be used with the rest of the framework. In general, you should design methods that take String parameters, not StringBuilder parameters, unless you define a method that "returns" a string dynamically constructed by the method itself.
Logically, a StringBuilder object contains a field that refers to an array of Char structures. StringBuilder's members allow you to manipulate this character array effectively, shrinking the string or changing the characters in the string. If you grow the string past the allocated array of characters, the StringBuilder automatically allocates a new, larger array, copies the characters, and starts using the new array. The previous array is garbage collected.
When finished using the StringBuilder object to construct your string, "convert" the String-Builder's character array into a String simply by calling the StringBuilder's ToString method. Internally, this method just returns a reference to the string field maintained inside the StringBuilder. This makes the StringBuilder's ToString method very fast because the array of characters isn't copied.
The String returned from StringBuilder's ToString method must not be changed. So if you ever call a method that attempts to modify the string field maintained by the StringBuilder, the StringBuilder's methods will have the information that ToString was called on the string field and they will internally create and use a new character array, allowing you to perform manipulations without affecting the string returned by the previous call to ToString.

13. Explain Indexers?

C# introduces a new concept known as Indexers which are used for treating an object as an array. The indexers are usually known as smart arrays in C# community. Defining a C# indexer is much like defining properties. We can say that an indexer is a member that enables an object to be indexed in the same way as an array.

```
<modifier> <return type> this [argument list]
{
 get
```


Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

```
 {
  // Get codes goes here
 }
 set
 {
  // Set codes goes here
 }
}
```

Where the modifier can be private, public, protected or internal. The return type can be any valid C# types. The 'this' is a special keyword in C# to indicate the object of the current class. The formal-argument-list specifies the parameters of the indexer. The formal parameter list of an indexer corresponds to that of a method, except that at least one parameter must be specified, and that the ref and out parameter modifiers are not permitted. Remember that indexers in C# must have at least one parameter. Other wise the compiler will generate a compilation error.

The following program shows a C# indexer in action

```
using System;
using System.Collections;

class MyClass
{
 private string []data = new string[5];
 public string this [int index]
 {
  get
  {
   return data[index];
  }
  set
  {
   data[index] = value;
  }
 }
}


class MyClient
{
 public static void Main()
 {
  MyClass mc = new MyClass();
  mc[0] = "Rajesh";
  mc[1] = "A3-126";
  mc[2] = "Snehadara";
  mc[3] = "Irla";
  mc[4] = "Mumbai";
 Console.WriteLine("{0},{1},{2},{3},{4}",mc[0],mc[1],mc[2],mc[3],mc[4]);
 }
}
```

The indexers in C# can be overloaded just like member functions. The formal parameter list of an indexer defines the signature of the indexer. Specifically, the signature of an indexer consists of the number and types of its formal parameters. The element type is not part of an indexer's signature, nor is the names of the formal parameters. The

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

signature of an indexer must differ from the signatures of all other indexers declared in the same class. C# do not have the concept of static indexers. If we declare an indexer static, the compiler will show a compilation time error.

Indexers & Inheritance

Just like any other class members, indexers can also participate in inheritance. A base class indexer is inherited to the derived class.

```
//C#: Indexer : Inheritance
//Author: rajeshvs@msn.com
using System;
class Base
{
 public int this[int indxer]
 {
get
  {
   Console.Write("Base GET");
   return 10;
  }
  set
  {
   Console.Write("Base SET");
  }
 }
}

class Derived : Base
{

}
class MyClient
{
 public static void Main()
 {
  Derived d1 = new Derived();
  d1[0] = 10;
  Console.WriteLine(d1[0]);//Displays 'Base SET Base GET 10'
 }
}
```

14. Explain Iterators?

An iterator is a method, get accessor, or operator that performs a custom iteration over an array or collection class by using the yield keyword. The **yield** return statement causes an element in the source sequence to be returned immediately to the caller before the next element in the source sequence is accessed. Although you write an iterator as a method, the compiler translates it into a nested class that is, in effect, a state machine. This class keeps track of the position of the iterator as long the **foreach** loop on the client code continues.

An iterator is invoked from client code by using a foreach statement. For example, you can create an iterator for a class that returns the elements in reverse order, or that performs an operation on each element before the iterator returns it. When you create an iterator for your class or struct, you do not have to implement the whole IEnumerator interface. When

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

the compiler detects your iterator, it will automatically generate the Current, MoveNext and Dispose methods of the IEnumerator or IEnumerator<(Of <(T>)>) interface.

Iterators Overview

An iterator is a section of code that returns an ordered sequence of values of the same type.

An iterator can be used as the body of a method, an operator, or a **get** accessor.

The iterator code uses the **yield return** statement to return each element in turn. **yield break** ends the iteration.

Multiple iterators can be implemented on a class. Each iterator must have a unique name just like any class member, and can be invoked by client code in a **foreach** statement as follows: foreach(int x in SampleClass.Iterator2){}.

The return type of an iterator must be IEnumerable, IEnumerator, IEnumerable<(Of <(T>)>), or IEnumerator<(Of <(T>)>).

Iterators are the basis for the deferred execution behavior in LINQ queries.

```
public class DaysOfTheWeek : System.Collections.IEnumerable

{

string[] m_Days = { "Sun", "Mon", "Tue", "Wed", "Thr", "Fri", "Sat" };

public System.Collections.IEnumerator GetEnumerator()

{

      for (int i = 0; i < m_Days.Length; i++)

      {

          yield return m_Days[i];

      }

   }

}

class TestDaysOfTheWeek

{

static void Main()

{

// Create an instance of the collection class
```


Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

```
DaysOfTheWeek week = new DaysOfTheWeek();

// Iterate with foreach

foreach (string day in week)

{

System.Console.Write(day + " ");

}

}

}
```

15. Explain secure strings?

Often, **String** objects are used to contain sensitive data such as a user's password or creditcard information. Unfortunately, **String** objects contain an array of characters in memory, and if some unsafe or unmanaged code is allowed to execute, the unsafe/unmanaged code could snoop around the process's address space, locate the string containing the sensitive information, and use this data in an unauthorized way. Even if the **String** object is used for just a short time and then garbage collected, the CLR might not immediately reuse the **String** object's memory (especially if the **String** object was in an older generation), leaving the **String**'s characters in the process' memory, where the information could be compromised. In addition, since strings are immutable, as you manipulate them, the old copies linger in memory and you end up with different versions of the string scattered all over memory.

Some governmental departments have stringent security requirements that require very specific security guarantees. To meet these requirements, Microsoft added a more secure string class to the FCL: **System.Security.SecureString.** When you construct a **SecureString** object, it internally allocates a block of unmanaged memory that contains an array of characters. Unmanaged memory is used so that the garbage collector isn't aware of it.

These string's characters are encrypted, protecting the sensitive information from any malicious unsafe/unmanaged code. You can append, insert, remove, or set a character in the secure string by using any of these methods: **AppendChar, InsertAt, RemoveAt,** and **SetAt.** Whenever you call any of these methods, internally, the method decrypts the characters, performs the operation, and then re-encrypts the characters. This means that the characters are in an unencrypted state for a very short period of time. This also means that these operations mutate the string's characters in place and that their performance is less than stellar, so you should perform as few of these operations as possible.

The **SecureString** class implements the **IDisposabie** interface to provide an easy way to deterministically destroy the string's secured contents. When your application no longer needs the sensitive string information, you simply call **SecureString**'s **Dispose** method. Internally, **Dispose** zeroes out the contents of the memory buffer to make sure that the sensitive information is not accessible to malicious code, and then the buffer is freed. You'll also notice that the **SecureString** class is derived from **CriticalFinalizerObject.** "Automatic Memory Management (Garbage Collection)," which ensures that a garbagecollected **SecureString** object has its **Finalize** method called, guaranteeing that the string's characters are zeroed out and that its buffer is freed. Unlike a **String** object, when a **Secure-String** object is collected, the encrypted string's characters will no longer be in memory.

```
using System;
using System.Security;
using System.Runtime.InteropServices;
public static class Program {
```


Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

```
public static void Main() {
using (SecureString ss = new SecureString()) {
Console.Write("Please enter password: ");
while (true) {
ConsoleKeyInfo cki = Console.ReadKey(true);
if (cki.Key == ConsoleKey.Enter) break;
// Append password characters into the SecureString
ss.AppendChar(cki.KeyChar);
Console.Write("*");
}
Console.WriteLine();
// Password entered, display it for demonstration purposes
DisplaySecureString(ss);
}
// After 'using', the SecureString is Disposed; no sensitive data in memory
}
// This method is unsafe because it accesses unmanaged memory
private unsafe static void DisplaySecureString(SecureString ss) {
Char* pc = null;
try {
// Decrypt the SecureString into an unmanaged memory buffer
pc = (Char*) Marshal.SecureStringToCoTaskMemUnicode(ss);
// Access the unmanaged memory buffer that
// contains the decrypted SecureString
for (Int32 index = 0; pc[index] != 0; index++)
Console.Write(pc[index]);
}
finally {
// Make sure we zero and free the unmanaged memory buffer that contains
// the decrypted SecureString characters
if (pc != null)
Marshal.ZeroFreeCoTaskMemUnicode((IntPtr) pc);
}
}
}
```

16. Explain Enumerated types.

An *enumerated type* is a type that defines a set of symbolic name and value pairs. For example, the **Color** type shown here defines a set of symbols, with each symbol identifying a single color:

```
internal enum Color {
White, // Assigned a value of 0
Red, // Assigned a value of 1
Green, // Assigned a value of 2
Blue, // Assigned a value of 3
Orange // Assigned a value of 4
}
```

Of course, programmers can always write a program using 0 to represent white, 1 to represent red, and so on. However, programmers shouldn't hard-code numbers into their code and should use an enumerated type instead, for at least two reasons:

- Enumerated types make the program much easier to write, read, and maintain. With enumerated types, the symbolic name is used throughout the code, and the programmer doesn't have to mentally map the meaning of

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

each hard-coded value (for example, white is 0 or vice versa). Also, should a symbol's numeric value change, the code can simply be recompiled without requiring any changes to the source code. In addition, documentation tools and other utilities, such as a debugger, can show meaningful symbolic names to the programmer.

- Enumerated types are strongly typed. For example, the compiler will report an error if I attempt to pass **Color.Orange** as a value to a method requiring a **Fruit** enumerated type as a parameter.

In the Microsoft .NET Framework, enumerated types are more than just symbols that the compiler cares about. Enumerated types are treated as first-class citizens in the type system, which allows for very powerful operations that simply can't be done with enumerated types in other environments (such as in unmanaged C++, for example). Every enumerated type is derived directly from **System.Enum,** which is derived from **System .ValueType,** which in turn is derived from **System.Object.** So enumerated types are value types (described in Chapter 5, "Primitive, Reference, and Value Types") and can be represented in unboxed and boxed forms. However, unlike other value types, an enumerated type can't define any methods, properties, or events. When an enumerated type is compiled, the C# compiler turns each symbol into a constant field of the type. For example, the compiler treats the **Color** enumeration shown earlier as if you had written code similar to the following:

```
internal struct Color : System.Enum {
// Below are public constants defining Color's symbols and values
public const Color White = (Color) 0;
public const Color Red = (Color) 1;
public const Color Green = (Color) 2;
public const Color Blue = (Color) 3;
public const Color Orange = (Color) 4;
// Below is a public instance field containing a Color variable's value
// You cannot write code that references this instance field directly
public Int32 value__;
}
```

The C# compiler won't actually compile this code because it forbids you from defining a type derived from the special **System.Enum** type. However, this pseudo-type definition shows you what's happening internally. Basically, an enumerated type is just a structure with a bunch of constant fields defined in it and one instance field. The constant fields are emitted to the assembly's metadata and can be accessed via reflection. This means that you can get all of the symbols and their values associated with an enumerated type at run time. It also means that you can convert a string symbol into its equivalent numeric value. These operations are made available to you by the **System.Enum** base type, which offers several static and instance methods that can be performed on an instance of an enumerated type, saving you the trouble of having to use reflection. I'll discuss some of these operations next.

17. Explain interface.

Interface is reference type object that is like blue print of method, events, properties, indexers, or any combination of those four member types. Following are the characterstics of interface.

- An interface is like an abstract base class: any non-abstract type inheriting the interface must implement all its members.
- An interface cannot be instantiated directly.
- Interfaces can contain events, indexers, methods and properties.
- Interfaces contain no implementation of methods.
- Classes and structs can inherit from more than one interface.
- An interface can itself inherit from multiple interfaces.

The C# compiler requires that a method that implements an interface be marked as **public.** The CLR requires that interface methods be marked as **virtual.** If you do not explicitly mark the method as **virtual** in your source code, the compiler marks the method as **virtual** and **sealed;** this prevents a derived class from overriding the interface method. If you explicitly mark the method as **virtual**, the compiler marks the method as **virtual** (and leaves it unsealed); this allows a

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

derived class to override the interface method. If an interface method is **sealed,** a derived class cannot override the method. However, a derived class can re-inherit the same interface and can provide its own implementation for the interface's methods. When calling an interface's method on an object, the implementation associated with the object's type is called. Here is an example that demonstrates this:

```
using System;
public static class Program {
public static void Main() {
/************************* First Example *************************/
Base b = new Base();
// Calls Dispose by using b's type: "Base's Dispose"
b.Dispose();
// Calls Dispose by using b's object's type: "Base's Dispose"
((IDisposable)b).Dispose();
/************************** Second Example ************************/
Derived d = new Derived();
// Calls Dispose by using d's type: "Derived's Dispose"
d.Dispose();
// Calls Dispose by using d's object's type: "Derived's Dispose"
((IDisposable)d).Dispose();
/************************* Third Example *************************/
b = new Derived();
// Calls Dispose by using b's type: "Base's Dispose"
b.Dispose();
// Calls Dispose by using b's object's type: "Derived's Dispose"
((IDisposable)b).Dispose();
}
}
// This class is derived from Object and it implements IDisposable
internal class Base : IDisposable {
// This method is implicitly sealed and cannot be overridden
public void Dispose() {
Console.WriteLine("Base's Dispose");
}
}
// This class is derived from Base and it re-implements IDisposable
internal class Derived : Base, IDisposable {
// This method cannot override Base's Dispose. 'new' is used to indicate
// that this method re-implements IDisposable's Dispose method
new public void Dispose() {
Console.WriteLine("Derived's Dispose");
// NOTE: The next line shows how to call a base class's implementation (if desired)
// base.Dispose();
}
}
```

Generic Interface

C#'s and the CLR's support of generic interfaces offers many great features for developers. In this section, I'd like to discuss the benefits offered when using generic interfaces. First, generic interfaces offer great compile-time type safety. Some interfaces (such as the nongeneric **IComparable** interface) define methods that have **Object** parameters or return types. When code calls these interface methods, a reference to an instance of any type can be passed.
But this is usually not desired. The following code demonstrates:

```
private void SomeMethod1() {
Int32 x = 1, y = 2;
IComparable c = x;
```


Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

```
// CompareTo expects an Object; passing y (an Int32) is OK
c.CompareTo(y); // Boxing occurs here
// CompareTo expects an Object; passing "2" (a String) compiles
// but an ArgumentException is thrown at runtime
c.CompareTo("2");
}
```

Obviously, it is preferable to have the interface method strongly typed, and this is why the FCL includes a generic **IComparable<T>** interface. Here is the new version of the code revised by using the generic interface:

```
private void SomeMethod2() {
Int32 x = 1, y = 2;
IComparable<Int32> c = x;
// CompareTo expects an Int32; passing y (an Int32) is OK
c.CompareTo(y); // Boxing occurs here
// CompareTo expects an Int32; passing "2" (a String) results
// in a compiler error indicating that String cannot be cast to an Int32
c.CompareTo("2");
}
```

The second benefit of generic interfaces is that much less boxing will occur when working with value types. Notice in **SomeMethod1** that the non-generic **IComparable** interface's **CompareTo** method expects an **Object;** passing **y** (an **Int32** value type) causes the value in **y** to be boxed. However, in **SomeMethod2**, the generic **IComparable<T>** interface's **CompareTo** method expects an **Int32;** passing **y** causes it to be passed by value, and no boxing is necessary

Implementing Multiple Interfaces That Have the Same Method Name and Signature

Occasionally, you might find yourself defining a type that implements multiple interfaces that define methods with the same name and signature. For example, imagine that there are two interfaces defined as follows:

```
public interface IWindow {
Object GetMenu();
}
public interface IRestaurant {
Object GetMenu();
}
```

Let's say that you want to define a type that implements both of these interfaces. You'd have to implement the type's members by using explicit interface method implementations as follows:

```
// This type is derived from System.Object and
// implements the IWindow and IRestaurant interfaces.
public sealed class MarioPizzeria : IWindow, IRestaurant {
...
// This is the implementation for IWindow's GetMenu method.
Object IWindow.GetMenu() { ... }
// This is the implementation for IRestaurant's GetMenu method.
Object IRestaurant.GetMenu() { ... }
// This (optional method) is a GetMenu method that has nothing
// to do with an interface,
public Object GetMenu() { ... }
}
```

Because this type must implement multiple and separate **GetMenu** methods, you need to tell the C# compiler which **GetMenu** method contains the implementation for a particular interface. Code that uses a **MarioPizzeria** object must cast to the specific interface to call the desired method. The following code demonstrates:

```
MarioPizzeria mp = new MarioPizzeria();
// This line calls MarioPizzeria's public GetMenu method
mp.GetMenu();
// These lines call MarioPizzeria's IWindow.GetMenu method
IWindow window = mp;
window.GetMenu();
// These lines call MarioPizzeria's IRestaurant.GetMenu method
```


Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

```
IRestaurant restaurant = mp;
restaurant.GetMenu()
```

18. Should I design a base type or an interface?

Here is some guideline which tells us where to use base class or interface.

- **IS-A vs. CAN-DO relationship** A type can inherit only one implementation. If the derived type can't claim an IS-A relationship with the base type, don't use a base type; use an interface. Interfaces imply a CAN-DO relationship. If the CAN-DO functionality appears to belong with various object types, use an interface. For example, a type can convert instances of itself to another type (**IConvertible**), a type can serialize an instance of itself (**ISerializable**) , etc. Note that value types must be derived from **System.ValueType**, and therefore, they cannot be derived from an arbitrary base class. In this case, you must use a CAN-DO relationship and define an interface.
- **Ease of use** It's generally easier for you as a developer to define a new type derived from a base type than to implement all of the methods of an interface. The base type can provide a lot of functionality, so the derived type probably needs only relatively small modifications to its behavior. If you supply an interface, the new type must implement all of the members.
- **Consistent implementation** No matter how well an interface contract is documented, it's very unlikely that everyone will implement the contract 100 percent correctly. In fact, COM suffers from this very problem, which is why some COM objects work correctly only with Microsoft Office Word or with Microsoft Internet Explorer. By providing a base type with a good default implementation, you start off using a type that works and is well tested; you can then modify parts that need modification.
- **Versioning** If you add a method to the base type, the derived type inherits the new method's default implementation for free. In fact, the user's source code doesn't even have to be recompiled. Adding a new member to an interface forces the inheritor of the interface to change its source code and recompile.

19. Explain App domain?

This application domain is not a physical isolation as a process is; it is a further logical isolation within the process. Since more than one application domain can exist within a single process, we receive some major advantages. In general, it is impossible for standard processes to access each other's data without using a proxy. Using a proxy incurs major overheads and coding can be complex. However, with the introduction of the application domain concept, we can now launch several applications within the same process. The same isolation provided by a process is also available with the application domain. Threads can execute across application domains without the overhead associated with inter-process communication. Another benefit of these additional in-process boundaries is that they provide type checking of the data they contain.

20. Explain Threading in dot net?

Process

When an application is launched, memory and any other resource for that application are allocated. The physical separation of this memory and resources is called a **process.** Of course, the application may launch more than one process. It's important to note that the words "application" and "process" are not synonymous. The memory allocated to the process is isolated from that of other processes and only that process is allowed to access it.

Threads

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

You will also notice that the Task Manager of windows has summary information about process CPU utilization. This is because the process also has an execution sequence that is used by the computer's processor. This execution sequence is known as a **thread.** This thread is defined by the registers in use on the CPU, the stack used by the thread, and a container that keeps track of the thread's current state. The container mentioned in the last sentence is known as **Thread Local Storage.** The concepts of registers and stacks should be familiar to any of you used to dealing with low-level issues like memory allocation; however, all you need to know here is that a stack in the .NET Framework is an area of memory that can be used for fast access and either stores value types, or pointers to objects, method arguments, and other data that is local to each method call.

Time Slices

When we discussed multitasking, we stated that the operating system grants each application a period to execute before interrupting that application and allowing another one to execute. This is not entirely accurate. The processor actually grants time to the process. The period that the process can execute is known as a **time slice** or a **quantum.** The period of this time slice is unknown to the programmer and unpredictable to anything besides the operating system. Programmers should not consider this time slice as a constant in their applications. Each operating system and each processor may have a different time allocated.

21. What is diffgram?

A DiffGram is an XML format that is used to identify current and original versions of data elements. The **DataSet** uses the DiffGram format to load and persist its contents, and to serialize its contents for transport across a network connection. When a **DataSet** is written as a DiffGram, it populates the DiffGram with all the necessary information to accurately recreate the contents, though not the schema, of the **DataSet**, including column values from both the **Original** and **Current** row versions, row error information, and row order.

When sending and retrieving a **DataSet** from an XML Web service, the DiffGram format is implicitly used. Additionally, when loading the contents of a **DataSet** from XML using the **ReadXml** method, or when writing the contents of a **DataSet** in XML using the **WriteXml** method, you can select that the contents be read or written as a DiffGram. For more information, see Loading a DataSet from XML and Writing a DataSet as XML Data.

While the DiffGram format is primarily used by the .NET Framework as a serialization format for the contents of a DataSet.

22. How assignment of value is different in value type and reference type?

Ans: Assignment works differently for value and reference types. For reference types, assignment simply duplicates the reference to the original instance, resulting in two variables that refer to the same instance in memory. For value types, assignment overwrites one instance with the contents of another, with the two instances remaining completely unrelated after the assignment is done.

Eg*: using System;*

public class clssize

{

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

```
public int height;
}

public class testassignment
{
    public static void Main()
    {
    clssize objsize1=new clssize();
    objsize1.height=100;
    clssize objsize2=objsize1;
    objsize1.height=150;
    Console.WriteLine("objsize1.height {0}\n",objsize1.height);
    Console.WriteLine("objsize2.height {0}",objsize2.height);

    int a=150;
    int b=a;
    a=250;
    Console.WriteLine("a {0}\n",a);
    Console.WriteLine("b {0}",b);

    }
}
Output : objsize1.height 150
      Objsize2.height 150
      a 250
```


Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

b 150

23. Difference between shallow copy and deep copy of object.

Ans: shallow copy, which means that it simply copies each field's value from the source object to the clone. If the field is an object reference, only the reference, and not the referenced object, is copied. The following class implements ICloneable using shallow copies:

```
public sealed class Marriage : System.ICloneable {

  internal Person g;

  internal Person b;



  public Object Clone() {

    return this.MemberwiseClone();

  }

}
```

A deep copy is one that recursively copies all objects that its fields refer to. Deep copying is often what people expect; however, it is not the default behavior, nor is it a good idea to implement in the general case. In addition to causing additional memory movement and resource consumption, deep copies can be problematic when a graph of objects has cycles because a naive recursion would wind up in an infinite loop. However, for simple object graphs, it is at least implementable.

Eg. System.Array.CopyTo() does deep copy and System.Array.Clone() does shallow copy.

24. What is use of using keyword?

1. it is also used for referring namespace at the top of class.

```
2. using (ResourceGobbler theInstance = new ResourceGobbler())

{

// do your processing

}

3. using alias= system.Data;
```


Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

The using statement, followed in brackets by a reference variable declaration and instantiation, will causem that variable to be scoped to the accompanying statement block. In addition, when that variable goes out of scope, its Dispose() method will be called automatically, even if exceptions occur. If you are already using try blocks to catch other exceptions, it is cleaner and avoids additional code indentation if you avoid the using statement and simply call Dispose() in the Finally clause of the existing try block.

25. What is cloning?

Ans: Assigning one reference variable to another simply creates a second reference to the same object. To make a second copy of an object, one needs some mechanism to create a new instance of the same class and initialize it based on the state of the original object. The Object.MemberwiseClone method does exactly that; however, it is not a public method. Rather, objects that wish to support cloning typically implement the System.ICloneable interface, which has one method, Clone:

```
namespace System {

  public interface ICloneable {

    Object Clone();

  }

}
```

26. What is Assembly? Difference between Private and Shared Assembly? How can we make shared assembly?

The terms 'private' and 'shared' refer to how an assembly is deployed, not any intrinsic attributes of the assembly.

A private assembly is normally used by a single application, and is stored in the application's directory, or a sub-directory beneath. A shared assembly is intended to be used by multiple applications, and is normally stored in the global assembly cache (GAC), which is a central repository for assemblies. (A shared assembly can also be stored outside the GAC, in which case each application must be pointed to its location via a codebase entry in the application's configuration file.) The main advantage of deploying assemblies to the GAC is that the GAC can support multiple versions of the same assembly side-by-side.

Assemblies deployed to the GAC must be strong-named. Outside the GAC, strong-naming is optional.

27. Why value type cannot be inherited?

All value types are marked as sealed in the type's metadata and cannot declare abstract methods. Additionally, because instances of value types are not allocated as distinct entities on the heap, value types cannot have finalizers. These are restrictions imposed by the CLR. The C# programming language imposes one additional restriction, which is that value types cannot have default constructors. In the absence of a default constructor, the CLR simply sets all of the fields of the value type to their default values when constructing an instance of a value type. Finally, because instances of value types do not have an object header, method invocation against a value type does not use virtual method dispatch. This helps performance but loses some flexibility.

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

28. What is the difference between an event and a delegate?

An event is just a wrapper for a multicast delegate. Adding a public event to a class is almost the same as adding a public multicast delegate field. In both cases, subscriber objects can register for notifications, and in both cases the publisher object can send notifications to the subscribers. However, **a public multicast delegate has the undesirable property that external objects can *invoke* the delegate,** something we'd normally want to restrict to the publisher. Hence events - an event adds public methods to the containing class to add and remove receivers, but does not make the invocation mechanism public.

29. What size is .net object?

Each instance of a reference type has two fields maintained by the runtime - a method table pointer and a sync block index. These are 4 bytes each on a 32-bit system, making a total of 8 bytes per object overhead. Obviously the instance data for the type must be added to this to get the overall size of the object. So, for example, instances of the following class are 12 bytes each:

```
class MyInt

{       private int x;

}
```

However, note that with the current implementation of the CLR there seems to be a minimum object size of 12 bytes, even for classes with no data (e.g. System.Object).

Values types have no equivalent overhead

30. When and How to Use Dispose and Finalize in C#?

Ans: When the .NET framework instantiates an object, it allocates memory for that object on the managed heap. The object remains on the heap until it's no longer referenced by any active code, at which point the memory it's using is "garbage," ready for memory deallocation by the .NET Garbage Collector (GC). Before the GC deallocates the memory, the *framework* calls the object's *Finalize()* method, but *developers* are responsible for calling the *Dispose()* method.

The two methods are not equivalent. Even though both methods perform object cleanup, there are distinct differences between them. To design efficient .NET applications, it's essential that you have a proper understanding of both how the .NET framework cleans up objects and when and how to use the *Finalize* and *Dispose* methods. This article discusses both methods and provides code examples and tips on how and when to use them.

An Insight into the Dispose and Finalize Methods
the .NET garbage collector manages the memory of managed objects (native .NET objects) but it does not manage, nor is it directly able to clean up unmanaged resources. Managed resources are those that are cleaned up implicitly by the garbage collector. You do not have to write code to release such resources explicitly. In contrast, you must clean up unmanaged resources (file handles, database collections, etc.) explicitly in your code.

There are situations when you might need to allocate memory for unmanaged resources from managed code. As an example, suppose you have to open a database connection from within a class. The database connection instance is an unmanaged resource encapsulated within this class and should be released as soon as you are done with it. In such cases, you'll need to free the memory occupied by the unmanaged resources explicitly, because the GC doesn't free them implicitly.

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

Briefly, the GC works as shown below:

It searches for managed objects that are referenced in managed code.

It then attempts to finalize those objects that are not referenced in the code.

Lastly, it frees the unreferenced objects and reclaims the memory occupied by them.

The GC maintains lists of managed objects arranged in "generations." A generation is a measure of the relative lifetime of the objects in memory. The generation number indicates to which generation an object belongs. Recently created objects are stored in lower generations compared to those created earlier in the application's life cycle. Longer-lived objects get promoted to higher generations. Because applications tend to create many short-lived objects compared to relatively few long-lived objects, the GC runs much more frequently to clean up objects in the lower generations than in the higher ones.

Keep this information about the .NET garbage collector in mind as you read the remainder of the article. I'll walk you through the *Finalize* method first, and then discuss the *Dispose* method of the relative lifetime of the objects in memory. The generation number indicates to which generation an object belongs. Recently created objects are stored in lower generations compared to those created earlier in the application's life cycle. Longer-lived objects get promoted to higher generations. Because applications tend to create many short-lived objects compared to relatively few long-lived objects, the GC runs much more frequently to clean up objects in the lower generations than in the higher ones. Keep this information about the .NET garbage collector in mind as you read the remainder of the article. I'll walk you through the *Finalize* method first, and then discuss the *Dispose* method.

Finalizers—Implicit Resource Cleanup

Finalization is the process by which the GC allows objects to clean up any unmanaged resources that they're holding, before the actually destroying the instance. An implementation of the *Finalize* method is called a "finalizer." Finalizers should free only external resources held directly by the object itself. The GC attempts to call finalizers on objects when it finds that the object is no longer in use—when no other object is holding a valid reference to it. In other words, finalizers are methods that the GC calls on **"seemingly dead objects"** before it reclaims memory for that object.

The GC calls an object's finalizer automatically, typically once per instance—although that's not always the case (see the Author's Note below for more information). The framework calls finalizers on a secondary thread handled by the GC. You should never rely on finalizers to clean up managed resources. A class that has no finalizer implemented but is holding references to unmanaged objects can cause memory leaks, because the resources might become orphaned if a class instance is destroyed before releasing the unmanaged objects.

You must implement finalizers very carefully; it's a complex operation that can carry considerable performance overhead. The performance overhead stems from the fact that finalizable objects are enlisted and removed from the *finalization queues*, which are internal data structures containing pointers to instances of classes that implement a finalizer method. When pointers to these objects are placed in this data structure, the object is said to be enlisted in the Finalization Queue. Note that the GC periodically scans this data structure to locate these pointers. When it finds one, it removes the pointer from the queue and appends the pointer at the end of another queue called the f*reachable queue*.

Further, finalizable objects tend to get promoted to the higher generations and hence stay in memory for a relatively longer period of time. Note that the GC works more frequently in the lower generations than in the higher ones.

The time and order of execution of finalizers cannot be predicted or pre-determined. This is why you'll hear that the nature of finalization is "non-deterministic." Further, due to the non-deterministic nature of finalization the framework does not and cannot guarantee that the *Finalize* method will ever be called on an instance. Hence, you cannot rely upon this method to free up any un-managed resources (such as a file handle or a database connection instance) that would otherwise not be garbage collected by the GC.

Note that you cannot call or override the *Finalize* method. It is generated implicitly if you have a destructor for the class. This is shown in the following piece of C# code:—

```
class Test
{

 // Some Code

 ~Test
 {
   //Necessary cleanup code
 }
}
```

In the preceding code, the *~Test* syntax declares an explicit destructor in C#, letting you write explicit cleanup code that will run during the finalize operation.

The framework implicitly translates the explicit destructor to create a call to *Finalize*:

```
protected override void Finalize()
{
   try
   {
      //Necessary cleanup code
   }
   finally
   {
      base.Finalize();
   }
}
```

Note that the generated code above calls the base.Finalize method. You should note the following points should when implementing finalizers:
Finalizers should always be protected, not public or private so that the method cannot be called from the application's code directly and at the same time, it can make a call to the *base.Finalize* method

- Finalizers should release unmanaged resources only.
- The framework does not guarantee that a finalizer will execute at all on any given instance.
- Never allocate memory in finalizers or call virtual methods from finalizers.
- Avoid synchronization and raising unhandled exceptions in the finalizers.
- The execution order of finalizers is non-deterministic—in other words, you can't rely on another object still being available within your finalizer.
- Do not define finalizers on value types.

Don't create empty destructors. In other words, you should never explicitly define a destructor unless your class needs to clean up unmanaged resources—and if you do define one, it should do some work. If, later, you no longer need to clean up unmanaged resources in the destructor, remove it altogether.

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

To close out this section, *Finalize()* is a non-explicit way to clean up resources. Because you can't control when (or even if) the GC calls *Finalize*, you should treat destructors only as a fallback mechanism for releasing unmanaged resources. Instead, the approved way to release unmanaged resources is to make your class inherit from the IDisposable interface and implement the *Dispose()* method.

The Dispose Method—Explicit Resource Cleanup

Unlike *Finalize*, developers *should* call *Dispose* explicitly to free unmanaged resources. In fact, you should call the *Dispose* method explicitly on *any object that implements it* to free any unmanaged resources for which the object may be holding references. The *Dispose* method generally doesn't free managed memory—typically, it's used for early reclamation of only the *unmanaged* resources to which a class is holding references. In other words, this method can release the unmanaged resources in a deterministic fashion.

However, *Dispose* doesn't remove the object itself from memory. The object will be removed when the garbage collector finds it convenient. It should be noted that the developer implementing the *Dispose* method must call *GC.SuppressFinalize(this)* to prevent the finalizer from running.

Note that an object should implement IDisposable and the *Dispose* method not only when it must explicitly free unmanaged resources, but also when it instantiates managed classes which in turn use such unmanaged resources. Implementing IDisposable is a good choice when you want your code, not the GC, to decide when to clean up resources. Further, note that the *Dispose* method should not be called concurrently from two or more different threads as it might lead to unpredictable results if other threads still have access to unmanaged resources belonging to the instance.

The IDisposable interface consists of only one *Dispose* method with no arguments.

```
public interface IDisposable
{
  void Dispose();
}
```

The following code illustrates how to implement the Dispose method on a class that implements the IDisposable interface:

```
class Test : IDisposable
{
  private bool isDisposed = false;

  ~Test()
  {
    Dispose(false);
  }

  protected void Dispose(bool disposing)
  {
    if (disposing)
    {
      // Code to dispose the managed resources of the class
    }
    // Code to dispose the un-managed resources of the class

    isDisposed = true;
  }

  public void Dispose()
  {
    Dispose(true);
    GC.SuppressFinalize(this);
```


Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

```
    }
  }
```

In the preceding code, when the Boolean variable *disposed* equals *true*, the object can free both managed and unmanaged resources; but if the value equals *false*, the call has been initiated from within the finalizer (*~Test*) in which case the object should release only the unmanaged resources that the instance has reference to.

The Dispose/Finalize Pattern

Microsoft recommends that you implement both *Dispose* and *Finalize* when working with unmanaged resources. The correct sequence then would be for a developer to call *Dispose*. The *Finalize* implementation would run and the resources would still be released when the object is garbage collected even if a developer neglected to call the *Dispose* method explicitly. the *Dispose/Finalize* pattern should be used only when your type invokes unmanaged code that allocates unmanaged resources (including unmanaged memory) and returns a handle that you must use eventually to release the resource. Both dispose and finalize must chain up to their parent objects by calling their parent's respective methods after they have disposed or finalized their own members".

Simply put, cleanup the unmanaged resources in the *Finalize* method and the managed ones in the *Dispose* method, when the *Dispose/Finalize* pattern has been used in your code.

As an example, consider a class that holds a database connection instance. A developer can call *Dispose* on an instance of this class to release the memory resource held by the database connection object. After it is freed, the *Finalize* method can be called when the class instance needs to be released from the memory.

Suppressing Finalization

After the *Dispose* method has been called on an object, you should suppress calls to the *Finalize* method by invoking the *GC.SuppressFinalize* method as a measure of performance optimization. Note that you should never change the order of calls in the finalization context (first *Dispose(true)* and then *GC.SupressFinalize*) to ensure that the latter gets called if and only if the *Dispose* method has completed its operation successfully.

The following code illustrates how to implement both the *Dispose* and *Finalize* pattern for a class.

```
public class Base: IDisposable
{
  private bool isDisposed = false;

   public void Dispose()
   {
      Dispose(true);
      GC.SuppressFinalize(this);
   }
   protected virtual void Dispose(bool disposing)
   {
      if(!isDisposed)
      {
       if (disposing)
       {
          // Code to dispose the managed resources
          // held by the class
       }
      }
     // Code to dispose the unmanaged resources
     // held by the class
    isDisposed = true;
    base.Dispose(disposing);
   }
```


Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

```
    ~Base()
    {
      Dispose (false);
    }
  }
```

You should not reimplement IDisposable for a class that inherits from a base class in which IDispose has already been implemented. The following code snippet may help you understand this concept:

```
  public class Base: IDisposable
  {
    private bool isDisposed = false;

    public void Dispose()
    {
      Dispose(true);
      GC.SuppressFinalize(this);
    }
    protected virtual void Dispose(bool disposing)
    {
      if(!isDisposed)
      {
       if (disposing)
       {
          // Code to dispose managed resources
          // held by the class
       }
      }
      // Code to dispose unmanaged resources
      // held by the class
     isDisposed = true;
     base.Dispose(disposing);
    }
    ~Base()
    {
      Dispose (false);
    }
  }

  public class Derived: Base
  {
    protected override void Dispose(bool disposing)
    {
      if (disposing)
      {
        // Code to cleanup managed resources held by the class.
      }

      // Code to cleanup unmanaged resources held by the class.

      base.Dispose(disposing);
    }
  // Note that the derived class does not // re-implement IDisposable
  }
```

In the preceding code, what if the Dispose method were to throw an exception? In that case, the Finalize method would exit prematurely, and the memory would never be reclaimed. Hence, in such situations, it is advisable to wrap the Dispose method in a try-catch block. This will prevent finalization exceptions from orphaning the object.

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

Note the following points when implementing disposable types:

- Implement IDisposable on every type that has a finalizer
- Ensure that an object is made unusable after making a call to the Dispose method. In other words, avoid using an object after the Dispose method has been called on it.
- Call Dispose on all IDisposable types once you are done with them
- Allow Dispose to be called multiple times without raising errors.
- Suppress later calls to the finalizer from within the Dispose method using the GC.SuppressFinalize method
- Avoid creating disposable value types
- Avoid throwing exceptions from within Dispose methods

In a managed environment, the GC takes care of freeing unused objects. In contrast, in unmanaged languages such as C, developers had to release unused objects explicitly that were created dynamically in the heap. However, a proper understanding of both the Dispose and Finalize methods goes a long way toward designing efficient applications.

31. What is difference between equivalent of objects and identical of objects?

The CLR (like many other technologies) distinguishes between object equivalence and identity. This is especially important for reference types such as classes. In general, two objects are equivalent if they are instances of the same type and if each of the fields in one object matches the values of the fields in the other object. That does not mean that they are "the same object" but only that the two objects have the same values. In contrast, two objects are identical if they share an address in memory. Practically speaking, two references are identical if they refer to the same object.
Comparing object references for identity is trivial, requiring only a comparison of memory addresses, independent of type. One can perform this test via the System.Object.ReferenceEquals static method. This method simply compares the addresses contained in two object references independent of the types of objects involved.
Unlike identity comparison, comparing for equivalence is type-specific, and for that reason, System.Object provides an Equals virtual method to compare any two objects for equivalence, the Equals method returns true provided that the two objects have equivalent values. System.Object.ReferenceEquals returns true only when the references refer to the same object.

32. What's the difference between the System.Array.CopyTo() and System.Array.Clone()?

The first one performs a deep copy of the array, the second one is shallow. A shallow copy of an Array copies only the elements of the Array, whether they are reference types or value types, but it does not copy the objects that the references refer to. The references in the new Array point to the same objects that the references in the original Array point to. In contrast, a deep copy of an Array copies the elements and everything directly or indirectly referenced by the elements.

33. How ado.net maintain transaction of data?

Often when there is more than one update to be made to the database, these updates must be performed within the scope of a transaction. A transaction in ADO.NET is initiated by calling one of the BeginTransaction() methods on the database connection object. These methods return an object that implements the IDbTransaction interface, defined within System.Data.

The following sequence of code initiates a transaction on a SQL Server connection:

string source = “server=(local)\\NetSDK;” +

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

```
"integrated security=SSPI;" +

"database=Northwind";

SqlConnection conn = new SqlConnection(source);

conn.Open();

SqlTransaction tx = conn.BeginTransaction();

// Execute some commands, then commit the transaction

tx.Commit();

conn.Close();
```

When you begin a transaction, you can choose the isolation level for commands executed within that transaction. The level determines how changes made in one database session are viewed by another.Not all database engines support all of the four levels presented in the following table.

Isolation Level

ReadCommitted The default for SQL Server. This level ensures that data written by one transaction will only be accessible in a second transaction after the first transaction commits.

ReadUncommitted This permits your transaction to read data within the database, even data that has not yet been committed by another transaction. For example, if two users were accessing the same database, and the first inserted some data without concluding their transaction (by means of a Commit or Rollback), then the second user with their isolation level set to ReadUncommitted could read the data.

RepeatableRead This level, which extends the ReadCommitted level, ensures that if the same statement is issued within the transaction, regardless of other potential updates made to the database, the same data will always be returned.This level does require extra locks to be held on the data, which could adversely affect performance. This level guarantees that, for each row in the initial query, no changes can be made to that data. It does, however, permit "phantom" rows to show up—these are completely new rows that another transaction might have inserted while your transaction is running.

Serializable This is the most "exclusive" transaction level, which in effect serializes access to data within the database. With this isolation level, phantom rows can never show up, so a SQL statement issued within a serializable transaction will always retrieve the same data. The negative performance impact of a Serializable transaction should not be underestimated—if you don't absolutely need to use this level of isolation, stay away from it.

The SQL Server default isolation level, ReadCommitted, is a good compromise between data coherence and data availability, because fewer locks are required on data than in RepeatableRead or Serializable modes. However, there are situations where the isolation level should be increased, and so within .NET you can simply begin a transaction with a different level from the default. There are no hard-and-fast rules as to which levels to pick—that comes with experience.

34. What is delay signing?

Ans: When we talk about the Assembly then the first thing comes into our mind is the security for high level development. Delayed signing is the terminology when we are certifying the assembly which will prevent hi-jacking of that assembly.

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

Delayed signing refers to a technique of partially signing assemblies while they are in development phase. So, signing an assembly basically certifies that assembly by the manufacturer and prevents tampering and hi-jacking of that assembly. This is achievable by using public key/private key encoding of parts of the assembly. The public key is embedded in the assembly and will be used by third-parties who want to reference the assembly. There are many more benefits to signing an assembly, but the main purpose of delayed signing is to allow a company to protect and control its private key and only use it during the packaging process. A delayed signed assembly can still be used like a signed assembly; you just can't package and ship it.

Steps to certify the Assembly:-

Delays sign a .NET app:

sn -k keypair.snk

sn -p keypair.snk public.snk

Build assembly with:
[assembly: AssemblyDelaySign("false")]
[assembly: AssemblyKeyFile("..\\..\\keypair.snk")]

sn -Vr AssemblyName.dll

This step is critical and is not mentioned anywhere. Exit and restart every instance of VisualStudio running. Until you do this Visual Studio will not know of the sn -Vr from step 4 and you will get
"COM Interop registration failed. The check of the signature failed for assembly AssemblyName.dll"

35. Can you declare a C++ type destructor in C# like ~MyClass ()?

Yes, but what's the point, since it will call Finalize (), and Finalize () has no guarantees when the memory will be cleaned up, plus, it introduces additional load on the garbage collector. The only time the finalizer should be implemented, is when you're dealing with unmanaged code.

36. What is difference between == and .Equals?

== Operator compares two objects by reference whereas .equals method compares objects by comparing each value of object.

37. What is the difference between structures and enumeration?

Unlike classes, structs are value types and do not require heap allocation. A variable of a struct type directly contains the data of the struct, whereas a variable of a class type contains a reference to the data. They are derived from System.ValueType class.

Enum->An enum type is a distinct type that declares a set of named constants. They are strongly typed constants. They are unique types that allow declaring symbolic names to integral values. Enums are value types, which mean they contain their own value, can't inherit or be inherited from and assignment copies the value of one enum to another.

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

```
public enum Grade
{
A,
B,
C
}
```

38. Should I make my destructor virtual?

A C# destructor is really just an override of the System.Object Finalize method, and so is virtual by definition.

39. How do I declare a pure virtual function in C#?

Use the abstract modifier on the method. The class must also be marked as abstract (naturally). Note that abstract methods cannot have an implementation (unlike pure virtual C++ methods).

40. Where would you use an iHTTPModule, and what are the limitations of any approach you might take in implementing one?

One of ASP.NET's most useful features is the extensibility of the HTTP pipeline, the path that data takes between client and server. You can use them to extend your ASP.NET applications by adding pre- and post-processing to each HTTP request coming into your application. For example, if you wanted custom authentication facilities for your application, the best technique would be to intercept the request when it comes in and process the request in a custom HTTP module.

41. What is difference between code base security and role base security? Which one is better?

Code security is the approach of using permissions and permission sets for a given code to run. The admin, for example, can disable running executables off the Internet or restrict access to corporate database to only few applications. Role-based security most of the time involves the code running with the privileges of the current user. This way the code cannot supposedly do more harm than mess up a single user account. There's no better, or 100% thumbs-up approach, depending on the nature of deployment, both code-based and role-based security could be implemented to an extent.

42. Is it possible to prevent a browser from caching an aspx page?

Just call SetNoStore on the HttpCachePolicy object exposed through the Response object's Cache property, as demonstrated here:SetNoStore works by returning a Cache-Control: private, no-store header in the HTTP response. In this example, it prevents caching of a Web page that shows the current time.

43. What is the difference between Debug. Write and Trace. Write?

The Debug.Write call won't be compiled when the Debug symbol is not defined (when doing a release build). Trace. Write calls will be compiled. Debug.Write is for information you want only in debug builds, Trace. Write is for when you want it in release build as well

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

44. What is difference between repeater over datalist and datagrid?

The Repeater class is not derived from the WebControl class, like the DataGrid and DataList. Therefore, the Repeater lacks the stylistic properties common to both the DataGrid and DataList. What this boils down to is that if you want to format the data displayed in the Repeater, you must do so in the HTML markup. The Repeater control provides the maximum amount of flexibility over the HTML produced. Whereas the DataGrid wraps the DataSource contents in an HTML < table >, and the DataList wraps the contents in either an HTML < table > or < span > tags (depending on the DataList's RepeatLayout property), the Repeater adds absolutely no HTML content other than what you explicitly specify in the templates. While using Repeater control, If we wanted to display the employee names in a bold font we'd have to alter the "ItemTemplate" to include an HTML bold tag, Whereas with the DataGrid or DataList, we could have made the text appear in a bold font by setting the control's ItemStyle-Font-Bold property to True. The Repeater's lack of stylistic properties can drastically add to the development time metric. For example, imagine that you decide to use the Repeater to display data that needs to be bold, centered, and displayed in a particular font-face with a particular background color. While all this can be specified using a few HTML tags, these tags will quickly clutter the Repeater's templates. Such clutter makes it much harder to change the look at a later date. Along with its increased development time, the Repeater also lacks any built-in functionality to assist in supporting paging, editing, or editing of data. Due to this lack of feature-support, the Repeater scores poorly on the usability scale.
However, The Repeater's performance is slightly better than that of the DataList's, and is more noticeably better than that of the DataGrid's. Following figure shows the number of requests per second the Repeater could handle versus the Data Grid and Data List

45. Describe Main Characteristics of static functions?

The main characteristics of static functions include:

- It is without the a this pointer
- It can't directly access the non-static members of its class
- It can't be declared const, volatile or virtual.
- It doesn't need to be invoked through an object of its class, although for convenience, it may.

46. What is DataReader? Difference between datareader and dataset?

An DataReader is a forward-only "connected" cursor. In other words, you can only traverse through the records returned in one direction, and the database connection used is kept open until the data reader has been closed. The DataReader class cannot be instantiated directly—it is always returned by a call to the ExecuteReader() method of the Command class. DataSet is an object which can contain tables, relationship among tables. It can also do batch update.

47. What is DLL hell?

Previously, before .NET, this used to be a major issue. "DLL Hell" refers to the set of problems caused when multiple applications attempt to share a common component like a dynamic link library (DLL) or a Component Object Model (COM) class. In the most typical case, one application will install a new version of the shared component that is not backward compatible with the version already on the machine. Although the application that has just been installed works well, existing applications that depended on a previous version of the shared component might no longer work. In some cases, the cause of the problem is even more subtle. In many cases there is a significant delay before a user discovers that an application has stopped working. As a result, it is often difficult to remember when a change was made to the machine that could have affected the application. A user may remember installing something a week ago, but there is no obvious correlation between that installation and the behavior they are now seeing. The reason for these issues is that version information about the different components of an application aren't recorded or enforced by the system. Also, changes made to the system on behalf of one application will typically affect all applications on the machine

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

48. What is Temporary Table? How can we create it?

You can create local and global temporary tables. Local temporary tables are visible only in the current session, and global temporary tables are visible to all sessions. Temporary tables cannot be partitioned.
Prefix local temporary table names with single number sign (#table_name), and prefix global temporary table names with a double number sign (##table_name).
SQL statements reference the temporary table by using the value specified for table_name in the CREATE TABLE statement, for example:

Copy Code

```
CREATE TABLE #MyTempTable (cola INT PRIMARY KEY)

INSERT INTO #MyTempTable VALUES (1)
```

If more than one temporary table is created inside a single stored procedure or batch, they must have different names. If a local temporary table is created in a stored procedure or application that can be executed at the same time by several users, the Database Engine must be able to distinguish the tables created by the different users. The Database Engine does this by internally appending a numeric suffix to each local temporary table name. The full name of a temporary table as stored in the sysobjects table in tempdb is made up of the table name specified in the CREATE TABLE statement and the system-generated numeric suffix. To allow for the suffix, table_name specified for a local temporary name cannot exceed 116 characters.

Temporary tables are automatically dropped when they go out of scope, unless explicitly dropped by using DROP TABLE:

A local temporary table created in a stored procedure is dropped automatically when the stored procedure is finished. The table can be referenced by any nested stored procedures executed by the stored procedure that created the table. The table cannot be referenced by the process that called the stored procedure that created the table. All other local temporary tables are dropped automatically at the end of the current session. Global temporary tables are automatically dropped when the session that created the table ends and all other tasks have stopped referencing them. The association between a task and a table is maintained only for the life of a single Transact-SQL statement. This means that a global temporary table is dropped at the completion of the last Transact-SQL statement that was actively referencing the table when the creating session ended.

A local temporary table created within a stored procedure or trigger can have the same name as a temporary table that was created before the stored procedure or trigger is called. However, if a query references a temporary table and two temporary tables with the same name exist at that time, it is not defined which table the query is resolved against. Nested stored procedures can also create temporary tables with the same name as a temporary table that was created by the stored procedure that called it. However, for modifications to resolve to the table that was created in the nested procedure, the table must have the same structure, with the same column names, as the table created in the calling procedure. This is shown in the following example.

Copy Code

```
CREATE PROCEDURE Test2
AS
CREATE TABLE #t(x INT PRIMARY KEY)
INSERT INTO #t VALUES (2)
SELECT Test2Col = x FROM #t;
GO
CREATE PROCEDURE Test1
AS
CREATE TABLE #t(x INT PRIMARY KEY)
INSERT INTO #t VALUES (1)
SELECT Test1Col = x FROM #t;
```


Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

```
EXEC Test2;
GO
CREATE TABLE #t(x INT PRIMARY KEY)
INSERT INTO #t VALUES (99);
GO
EXEC Test1;
GO
```

Here is the result set.

Copy Code

```
(1 row(s) affected)

Test1Col
-----------
1

(1 row(s) affected)

Test2Col
-----------

2
```

When you create local or global temporary tables, the CREATE TABLE syntax supports constraint definitions except for FOREIGN KEY constraints. If a FOREIGN KEY constraint is specified in a temporary table, the statement returns a warning message that states the constraint was skipped. The table is still created without the FOREIGN KEY constraints. Temporary tables cannot be referenced in FOREIGN KEY constraints.
We recommend using table variables instead of temporary tables. Temporary tables are useful when indexes must be created explicitly on them, or when the table values must be visible across multiple stored procedures or functions. Generally, table variables contribute to more efficient query processing. For more information, see table (Transact-SQL).

49. What is strong key and what is the use of it?

A strong name consists of the assembly's identity—its simple text name, version number, and culture information (if provided)—plus a public key and a digital signature. It is generated from an assembly file (the file that contains the assembly manifest, which in turn contains the names and hashes of all the files that make up the assembly), using the corresponding private key. Microsoft® Visual Studio® .NET and other development tools provided in the .NET Framework SDK can assign strong names to an assembly. Assemblies with the same strong name are expected to be identical.
You can ensure that a name is globally unique by signing an assembly with a strong name. In particular, strong names satisfy the following requirements:

- Strong names guarantee name uniqueness by relying on unique key pairs. No one can generate the same assembly name that you can, because an assembly generated with one private key has a different name than an assembly generated with another private key.
- Strong names protect the version lineage of an assembly. A strong name can ensure that no one can produce a subsequent version of your assembly. Users can be sure that a version of the assembly they are loading comes from the same publisher that created the version the application was built with.
- Strong names provide a strong integrity check. Passing the .NET Framework security checks guarantees that the contents of the assembly have not been changed since it was built. Note, however, that strong names in and of themselves do not imply a level of trust like that provided, for example, by a digital signature and supporting certificate.

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

50. What is Impersonation?

When using impersonation, ASP.NET applications can execute with the Windows identity (user account) of the user making the request. Impersonation is commonly used in applications that rely on Microsoft Internet Information Services (IIS) to authenticate the user.
ASP.NET impersonation is disabled by default. If impersonation is enabled for an ASP.NET application, that application runs in the context of the identity whose access token IIS passes to ASP.NET. That token can be either an authenticated user token, such as a token for a logged-in Windows user, or the token that IIS provides for anonymous users (typically, the IUSR_MACHINENAME identity).
When impersonation is enabled, only your application code runs under the context of the impersonated user. Applications are compiled and configuration information is loaded using the identity of the ASP.NET process. For more information, see Configuring ASP.NET Process Identity. The compiled application is put in the Temporary ASP.NET files directory. The application identity that is being impersonated needs to have read/write access to this directory. The impersonated application identity also requires at least read access to the files in your application directory and subdirectories. For more information.
You control impersonation using the identity configuration element. As with other configuration directives, this directive applies hierarchically. A minimal configuration file to enable impersonation for an application might look like the following example:
Copy Code
<configuration> <system.web> <identity impersonate="true"/> </system.web> </configuration>
You can also add support for specific names to run an application as a configurable identity, as shown in the following example:
Copy Code
<identity impersonate="true" userName="contoso\Jane" password="E@1bp4!T2" />
Note

In the preceding example, the user name and password are stored in clear text in the configuration file. To improve the security of your application, it is recommended that you restrict the access to your Web.config file using an Access Control List (ACL) and that you encrypt the identityconfiguration element in your Web.config file using protected configuration. For more information, see Encrypting Configuration Information Using Protected Configuration.

The configuration illustrated in the example enables the entire application to run using the contoso\Jane identity, regardless of the identity of the request. This type of impersonation can be delegated to another computer. That is, if you specify the user name and password for the impersonated user, you can connect to another computer on the network and request resources, such as files or access to SQL Server, using integrated security. If you enable impersonation and do not specify a domain account as the identity, you will not be able to connect to another computer on the network unless your IIS application is configured to use Basic authentication.
Note

On Windows 2000, you cannot impersonate using specific user credentials for the identity of the ASP.NET worker process. But you can enable impersonation without specific user credentials so that your application impersonates the identity determined by IIS. For more information, see article 810204, "PRB: Per Request Impersonation Does Not Work on Windows 2000 with ASP.NET," in the Microsoft Knowledge Base at http://support.microsoft.com.

Reading the Impersonated Identity
The following code example shows how to programmatically read the identity of the impersonated user:
C#
Copy Code
String username = System.Security.Principal.WindowsIdentity.GetCurrent().Name;

51. What is Partitioned Tables?

Partitioning makes large tables or indexes more manageable, because it lets you manage and access and subsets of data quickly and efficiently, while maintaining the integrity of a data collection. By using partitioning, operations such as loading data from an OLTP to an OLAP system, take only seconds instead of the minutes and hours it took in

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

previous releases. Maintenance operations that are performed on subsets of data are also performed more efficiently, because they target only the data that is required, instead of the whole table.
Note Partitioned tables and indexes are supported in the Microsoft SQL Server 2005 Enterprise and Developer editions only.
The data of partitioned tables and indexes is divided into units that can be spread across more than one filegroup in a database. The data is partitioned horizontally, so that groups of rows are mapped into individual partitions. The table or index is treated as a single logical entity when queries or updates are performed on the data. All partitions of a single index or table must reside in the same database.
Partitioned tables and indexes support all the properties and features associated with designing and querying standard tables and indexes, including constraints, defaults, identity and timestamp values, and triggers. Therefore, if you want to implement a partitioned view that is local to one server, you may want to implement a partitioned table instead.
Deciding whether to implement partitioning depends primarily on how large your table is or how large it will become, how it is being used, and how well it is performing against user queries and maintenance operations.
Generally, a large table might be appropriate for partitioning if both of the following are true:
The table contains, or is expected to contain, lots of data that are used in different ways.
Queries or updates against the table are not performing as intended, or maintenance costs exceed predefined maintenance periods.

52. What types of data validation events are commonly seen in the client-side form validation? Web service support

a) Data set b) dataReader c) both of above d) none of above

Web service support Data Set and not support data reader.

53. How to create dynamic Gridview?

Many times we have the requirement where we have to create columns dynamically.
This article describes you about the dynamic loading of data using the DataTable as the datasource.
Details of the Grid
Let's have a look at the code to understand better.
Create a gridview in the page,
Drag and drop the GridView on to the page Or Manually type GridView definition in the page.

```
public partial class _Default : System.Web.UI.Page
{

#region constants

const string NAME = "NAME";

const string ID = "ID";

#endregion
protected void Page_Load(object sender, EventArgs e)
{
loadDynamicGrid();
}
private void loadDynamicGrid()
{
#region Code for preparing the DataTable
//Create an instance of DataTable
```


Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

```
DataTable dt = new DataTable();
//Create an ID column for adding to the Datatable
DataColumn dcol = new DataColumn(ID ,typeof(System.Int32));
dcol.AutoIncrement = true;
dt.Columns.Add(dcol);
//Create an ID column for adding to the Datatable
dcol = new DataColumn(NAME, typeof(System.String));
dt.Columns.Add(dcol);
//Now add data for dynamic columns
//As the first column is auto-increment, we do not have to add any thing.
//Let's add some data to the second column.
for (int nIndex = 0; nIndex < 10; nIndex++)
{
//Create a new row
DataRow drow = dt.NewRow();
//Initialize the row data.
drow[NAME] = "Row-" + Convert.ToString((nIndex + 1));
//Add the row to the datatable.
dt.Rows.Add(drow);
}
#endregion
//Iterate through the columns of the datatable to set the data bound field dynamically.
foreach (DataColumn col in dt.Columns)
{
//Declare the bound field and allocate memory for the bound field.
BoundField bfield = new BoundField();
//Initalize the DataField value.
bfield.DataField = col.ColumnName;
//Initialize the HeaderText field value.
bfield.HeaderText = col.ColumnName;
//Add the newly created bound field to the GridView.
GrdDynamic.Columns.Add(bfield);
}
//Initialize the DataSource
GrdDynamic.DataSource = dt;
//Bind the datatable with the GridView.
GrdDynamic.DataBind();
}
}
```

54. What is PreProcessor in .NET and type , where it use?

The pre-processing directives provide the ability to conditionally skip sections of source files, to report error and warning conditions, and to delineate distinct regions of source code. The term "pre-processing directives" is used only for consistency with the C and C++ programming languages. In C#, there is no separate pre-processing step; pre-processing directives are processed as part of the lexical analysis phase.
A preprocessor directive must be the only instruction on a line. Preprocessing directives are lines in your program that start with `#'. Whitespace is allowed before and after the `#'. The `#' is followed by an identifier that is the directive name. For example, `#define' is the directive
Types are:
#if, #else, #elif, #endif, #define, #undef, #warning, #error, #line, #region, #endregion

They are used for:

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

Conditional compilation
Line control
Error and Warning reporting

55. Please brief not about XSD, XSLT & XML.

XSD stands for XML Schema Definition. It defines the structure of the XML file and the elements and attributes it contains.The datatype of the elements. So that when u populate XML data into dataset, the dataset can treat elements differently based on their type.If XSD is not present dataset treats all elements as string type. XSLT stands for XML style sheet lang tranformation.It is lang used for transforming XML data in one format to another format. Example XML data into HTML format. XSLT uses XPath to identify the elements in XML doc and transform those to desired format.

56. Which dll handles the request of .aspx page?

When the Internet Information Service process (inetinfo.exe) receives an HTTP request, it uses the filename extension of the requested resource to determine which Internet Server Application Programming Interface (ISAPI) program to run to process the request. When the request is for an ASP.NET page (.aspx file), IIS passes the request to the ISAPI DLL capable of handling the request for ASP.NET pages, which is aspnet_isapi.dll

57. What is Polymorphism?

One of the fundamental concepts of object oriented software development is polymorphism. The term polymorphism (from the Greek meaning "having multiple forms") in OO is the characteristic of being able to assign a different meaning or usage to something in different contexts - specifically, to allow a variable to refer to more than one type of object.

When a derived class inherits from a base class, it gains all the methods, fields, properties and events of the base class. To change the data and behavior of a base class, you have two choices: you can replace the base member with a new derived member, or you can override a virtual base member.

Generally, the ability to appear in many forms. In object-oriented programming, polymorphism refers to a programming language's ability to process objects differently depending on their data type or class. More specifically, it is the ability to redefine methods for derived classes.

For example, given a base class shape, polymorphism enables the programmer to define different area methods for any number of derived classes, such as circles, rectangles and triangles. No matter what shape an object is, applying the area method to it will return the correct results. Polymorphism is considered to be a requirement of any true object-oriented programming language.

58. What is implicit operator overloading?

An implicit keyword is used to declare an implicit user-defined type conversion operator. In other word, this gives a power to your C# class, which can accepts any reasonably convertible data type without type casting. And such kind of class can also be assigned to any convertible object or variable. If you want to create implicit operator function, here is a signature of creating them in C#.
«access specifier» static implicit operator «converting type» («convertible type» rhs)

Above signature states that operator accepts «convertible type» and converts into «converting type».

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

Following code shows you how to create them.

```
/// <summary>
/// Creates Currency object from string supplied as currency sign.
/// </summary>
/// <param name="rhs">The currency sign like $,£,¥,€,Rs etc. </param>
/// <returns>Returns new Currency object.</returns>
public static implicit operator Currency(string rhs)
{
   Currency c = new Currency(0, rhs); //Internally call Currency constructor
return c;
}
/// <summary>
/// Creates a currency object from decimal value.
/// </summary>
/// <param name="rhs">The currency value in decimal.</param>
/// <returns>Returns new Currency object.</returns>
public static implicit operator Currency(decimal rhs)
{
Currency c = new Currency(rhs, NumberFormatInfo.CurrentInfo.CurrencySymbol);
return c;
}
/// <summary>
/// Creates a decimal value from Currency object,
/// used to assign currency to decimal.
/// </summary>
/// <param name="rhs">The Currency object.</param>
/// <returns>Returns decimal value of the currency</returns>
public static implicit operator decimal(Currency rhs)
{    return rhs.Value;
}
/// <summary>
/// Creates a long value from Currency object, used to assign currency to long.
/// </summary>
/// <param name="rhs">The Currency object.</param>
/// <returns>Returns long value of the currency</returns>
public static implicit operator long(Currency rhs)
{    return (long)rhs.Value;
}
```

Behind the scene

Such kind of implicit operator overloading is not supported by all languages, then how does csharp incorporate such a nice feature. The answer lies in a assembly code generated by the csharp compiler. Following table shows the C# syntax with corresponding IL syntax.

C# declaration	IL Declaration
public static implicit operator Currency(decimal rhs)	.method public hidebysig specialname static class Currency op_Implicit(valuetype [mscorlib]System.Decimal rhs) cil managed
public static implicit operator decimal(Currency rhs)	.method public hidebysig specialname static valuetype [mscorlib]System.Decimal op_Implicit(class Currency rhs) cil managed

IL syntax in above table makes it clear that C# compiler generates **op_Implicit** function which returns «converting type» and accepts «converting type».

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

59. What is Operator Overloading?

All unary and binary operators have pre-defined implementations that are automatically available in any expressions. In addition to this pre-defined implementations, user defined implementations can also be introduced in C#. The mechanism of giving a special meaning to a standard C# operator with respect to a user defined data type such as classes or structures is known as operator overloading. Remember that it is not possible to overload all operators in C#. The following table shows the operators and their overloadability in C#.

Operators	Overloadability
+, -, *, /, %, &, |, <<, >>	All C# binary operators can be overloaded.
+, -, !, ~, ++, --, true, false	All C# unary operators can be overloaded.
==, !=, <, >, <= , >=	All relational operators can be overloaded, but only as pairs.
&&, ||	They can't be overloaded
() (Conversion operator)	They can't be overloaded
+=, -=, *=, /=, %=	These compound assignment operators can be overloaded. But in C#, these operators are automatically overloaded when the respective binary operator is overloaded.
=, . , ?:, ->, new, is, as, sizeof	These operators can't be overloaded

In C#, a special function called operator function is used for overloading purpose. These special function or method must be public and static. They can take only value arguments. The ref and out parameters are not allowed as arguments to operator functions. The general form of an operator function is as follows.
public static return_type operator op (argument list)
Where the op is the operator to be overloaded and operator is the required keyword. For overloading the unary operators, there is only one argument and for overloading a binary operator there are two arguments. Remember that at least one of the arguments must be a user-defined type such as class or struct type.
Overloading Unary Operators The general form of operator function for unary operators is as follows. public static return_type operator op (Type t) { // Statements } Where Type must be a class or struct. The return type can be any type except void for unary operators like +, ~, ! and dot (.). but the return type must be the type of 'Type' for ++ and o remember that the true and false operators can be overloaded only as pairs. The compilation error occurs if a class declares one of these operators without declaring the other.
The following program overloads the unary - operator inside the class Complex

```
// Unary operator overloading
// Author: rajeshvs@msn.com
using System;
class Complex
{
 private int x;
 private int y;
 public Complex()
 {
 }
 public Complex(int i, int j)
 {
  x = i;
  y = j;
```


Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

```
 }
 public void ShowXY()
 {
  Console.WriteLine(\"{0} {1}\",x,y);
 }
 public static Complex operator -(Complex c)
 {
  Complex temp = new Complex();
  temp.x = -c.x;
  temp.y = -c.y;
  return temp;
 }
}
class MyClient
{
 public static void Main()
 {
  Complex c1 = new Complex(10,20);
  c1.ShowXY(); // displays 10 & 20
  Complex c2 = new Complex();
  c2.ShowXY(); // displays 0 & 0
  c2 = -c1;
  c2.ShowXY(); // diapls -10 & -20
 }
}
```

Overloading Binary Operators

An overloaded binary operator must take two arguments, at least one of them must be of the type class or struct, in which the operation is defined. But overloaded binary operators can return any value except the type void. The general form of a overloaded binary operator is as follows.

```
public static return_type operator op (Type1 t1, Type2 t2)
{
 //Statements
}
```

A concrete example is given below

```
// binary operator overloading
// Author: rajeshvs@msn.com

using System;

class Complex
{
 private int x;
 private int y;
 public Complex()
 {
 }
 public Complex(int i, int j)
 {
  x = i;
  y = j;
 }
```


Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

```
 public void ShowXY()
 {
  Console.WriteLine(\"{0} {1}\",x,y);
 }

 public static Complex operator +(Complex c1,Complex c2)
 {
  Complex temp = new Complex();
  temp.x = c1.x+c2.x;
  temp.y = c1.y+c2.y;
  return temp;
 }
}


class MyClient
{
 public static void Main()
 {
  Complex c1 = new Complex(10,20);
  c1.ShowXY(); // displays 10 & 20
  Complex c2 = new Complex(20,30);
  c2.ShowXY(); // displays 20 & 30
  Complex c3 = new Complex();
  c3 = c1 + c2;
  c3.ShowXY(); // dislplays 30 & 50
 }
}
```

The binary operators such as = =, ! =, <, >, < =, > = can be overloaded only as pairs. Remember that when a binary arithmetic operator is overloaded, corresponding assignment operators also get overloaded automatically. For example if we overload + operator, it implicitly overloads the + = operator also.

Summary

1.The user defined operator declarations can't modify the syntax, precedence or associativity of an operator. For example, a + operator is always a binary operator having a predefined precedence and an associativity of left to right.
2.User defined operator implementations are given preference over predefined implementations.
3.Operator overload methods can't return void.
4.The operator overload methods can be overloaded just like any other methods in C#. The overloaded methods should differ in their type of arguments and/or number of arguments and/or order of arguments. Remember that in this case also the return type is not considered as part of the method signature.

60. What is difference between http handler and http module?

An ASP.NET HTTP handler is the process (frequently referred to as the "endpoint") that runs in response to a request made to an ASP.NET Web application. The most common handler is an ASP.NET page handler that processes .aspx files. When users request an .aspx file, the request is processed by the page through the page handler. You can create your own HTTP handlers that render custom output to the browser.

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

An HTTP module is an assembly that is called on every request that is made to your application. HTTP modules are called as part of the ASP.NET request pipeline and have access to life-cycle events throughout the request. HTTP modules let you examine incoming and outgoing requests and take action based on the request.

Typical uses for custom HTTP handlers include the following:

- **RSS feeds** To create an RSS feed for a Web site; you can create a handler that emits RSS-formatted XML. You can then bind a file name extension such as .rss to the custom handler. When users send a request to your site that ends in .rss, ASP.NET calls your handler to process the request.

- **Image server** If you want a Web application to serve images in a variety of sizes, you can write a custom handler to resize images and then send them to the user as the handler's response.

Typical uses for HTTP modules include the following:

- **Security** Because you can examine incoming requests, an HTTP module can perform custom authentication or other security checks before the requested page, XML Web service, or handler is called. In Internet Information Services (IIS) 7.0 running in Integrated mode, you can extend forms authentication to all content types in an application.

- **Statistics and logging** Because HTTP modules are called on every request, you can gather request statistics and log information in a centralized module, instead of in individual pages.

- **Custom headers or footers** Because you can modify the outgoing response, you can insert content such as custom header information into every page or XML Web service response.

61. What is Caching? Where is it use? Why and when?

Caching is the process of storing frequently used data on the server to fulfill subsequent requests. You will discover that grabbing objects from memory is much faster than re-creating the Web pages or items contained in them from scratch each time they are requested. Caching increases your application's performance, scalability, and availability. The more you fine-tune your application's caching approach, the better it performs.

Output Caching
Output caching is a way to keep the dynamically generated page content in the server's memory for later retrieval. After a cache is saved to memory, it can be used when any subsequent requests are made to the server. You apply output caching by inserting an OutputCache page directive at the top of an .aspx page, as follows:
<%@ OutputCache Duration="60" VaryByParam="None" %>
Partial Page (UserControl) Caching
Similar to output caching, partial page caching enables you to cache only specific blocks of a Web page. You can, for example, cache only the center of the page. Partial page caching is achieved with the caching of user controls. You can build your ASP.NET pages utilizing numerous user controls and then apply output caching to the user controls you select. This, in essence, caches only the parts of the page that you want, leaving other parts of the page outside the reach of caching. This is a nice feature and, if done correctly, it can lead to pages that perform better.
Typically, UserControls are placed on multiple pages to maximize reuse. However, when these UserControls (ASCX files) are cached with the @OutputCache directive's default attributes, they are cached on a per-page basis. That means that even if a UserControl outputs the identical HTML when placed on pagea.aspx as it does when placed on pageb.aspx, its output is cached twice. By enabling the Shared="true" attribute, the UserControl's output can be shared among multiple pages and on sites that make heavy use of shared UserControls:
<%@ OutputCache Duration="300" VaryByParam="*" Shared="true" %>

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

Data Caching

62. What is Cursor? Define Different type of cursor?

In order to process a SQL statement, Oracle/Ms-Sql will allocate am area of memory known as the context area. A cursor is a handle, or pointer, to the context area. Through the cursor, a PL/SQL/T-Sql program can control the context area and what happens to it as the statement is processed.
A cursor is a database object that applications use to manipulate data by rows instead of recordsets. You can use cursors to perform multiple operations in a row-by-row manner, against the resultset. You can do this with or without returning to the original table. In other words, cursors conceptually return a resultset based on tables within a database.
DECLARE <Cursor_Name> CURSOR
[LOCAL | GLOBAL]
[FORWARD ONLY | SCROLL]
[STATIC | KEYSET | DYNAMIC | FAST_FORWARD]
[READ_ONLY | SCROLL_LOCKS | OPTIMISTIC]
[TYPE_WARNING]
FOR <Selecting Statements>
[FOR UPDATE [OF Column_name[,....N]]]
LOCAL: Specifies that the scope of the cursor is local to the batch, stored procedure, or trigger in which the cursor was created. The cursor name is valid only within this scope. The cursor can be referenced by local cursor variables in the batch, stored procedure, or trigger, or a stored procedure OUTPUT parameter. An OUTPUT parameter is used to pass the local cursor back to the calling batch, stored procedure, or trigger, which can assign the parameter to a cursor variable to reference the cursor after the stored procedure terminates.
The cursor is implicitly deallocated when the batch, stored procedure, or trigger terminates, unless the cursor was passed back in an OUTPUT parameter. If it is passed back in an OUTPUT parameter, the cursor is deallocated when the last variable referencing it is deallocated or goes out of scope.
GLOBAL: Specifies that the scope of the cursor is global to the connection. The cursor name can be referenced in any stored procedure or batch executed by the connection. The cursor is only implicitly deallocated at disconnect.
FORWARD_ONLY: Specifies that the cursor can only be scrolled from the first to the last row. FETCH NEXT is the only supported fetch option. If FORWARD_ONLY is specified without the STATIC, KEYSET, or DYNAMIC keywords, the cursor operates as a DYNAMIC cursor.
When neither FORWARD_ONLY nor SCROLL is specified, FORWARD_ONLY is the default, unless the keywords STATIC, KEYSET, or DYNAMIC are specified. STATIC, KEYSET, and DYNAMIC cursors default to SCROLL. Unlike database APIs such as ODBC and ADO, FORWARD_ONLY is supported with STATIC, KEYSET, and DYNAMIC Transact-SQL cursors. FAST_FORWARD and FORWARD_ONLY are mutually exclusive; if one is specified, the other cannot be specified.
STATIC: Defines a cursor that makes a temporary copy of the data to be used by the cursor. All requests to the cursor are answered from this temporary table in tempdb; therefore, modifications made to base tables are not reflected in the data returned by fetches made to this cursor, and this cursor does not allow modifications.
KEYSET: Specifies that the membership and order of rows in the cursor are fixed when the cursor is opened. The set of keys that uniquely identify the rows is built into a table in tempdb known as the keyset. Changes to nonkey values in the base tables, either made by the cursor owner or committed by other users, are visible as the owner scrolls around the cursor. Inserts made by other users are not visible (inserts cannot be made through a Transact-SQL server cursor).
If a row is deleted, an attempt to fetch the row returns an @@FETCH_STATUS of -2. Updates of key values from outside the cursor resemble a delete of the old row followed by an insert of the new row. The row with the new values is not visible, and attempts to fetch the row with the old values return an @@FETCH_STATUS of -2. The new values are visible if the update is done through the cursor by specifying the WHERE CURRENT OF clause.
DYNAMIC: Defines a cursor that reflects all data changes made to the rows in its result set as you scroll around the cursor. The data values, order, and membership of the rows can change on each fetch. The ABSOLUTE fetch option is not supported with dynamic cursors.
FAST_FORWARD: Specifies a FORWARD_ONLY, READ_ONLY cursor with performance optimizations enabled. FAST_FORWARD cannot be specified if SCROLL or FOR_UPDATE is also specified. FAST_FORWARD and FORWARD_ONLY are mutually exclusive; if one is specified, the other cannot be specified.

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

READ_ONLY: Prevents updates from being made through this cursor. The cursor cannot be referenced in a WHERE CURRENT OF clause in an UPDATE or DELETE statement. This option overrides the default capability of a cursor to be updated.
SCROLL_LOCKS: Specifies that positioned updates or deletes made through the cursor are guaranteed to succeed. Microsoft SQL Server locks the rows as they are read into the cursor to ensure their availability for later modifications. SCROLL_LOCKS cannot be specified if FAST_FORWARD is also specified.
OPTIMISTIC: Specifies that positioned updates or deletes made through the cursor do not succeed if the row has been updated since it was read into the cursor. SQL Server does not lock rows as they are read into the cursor. It instead uses comparisons of timestamp column values, or a checksum value if the table has no timestamp column, to determine whether the row was modified after it was read into the cursor. If the row was modified, the attempted positioned update or delete fails. OPTIMISTIC cannot be specified if FAST_FORWARD is also specified.
TYPE_WARNING: Specifies that a warning message is sent to the client if the cursor is implicitly converted from the requested type to another.
select_statement is a standard SELECT statement that defines the result set of the cursor. The keywords COMPUTE, COMPUTE BY, FOR BROWSE, and INTO are not allowed within a select_statement of a cursor declaration. SQL Server implicitly converts the cursor to another type if clauses in select_statement conflict with the functionality of the requested cursor type. For more information, see Implicit Cursor Conversions.
UPDATE [OF column_name [,...n]] defines updatable columns within the cursor. If OF column_name [,...n] is supplied, only the columns listed allow modifications. If UPDATE is specified without a column list, all columns can be updated, unless the READ_ONLY concurrency option was specified.

63. What is Views?

A view is a virtual table whose contents are defined by a query. Like a real table, a view consists of a set of named columns and rows of data. Unless indexed, a view does not exist as a stored set of data values in a database. The rows and columns of data come from tables referenced in the query defining the view and are produced dynamically when the view is referenced.
A view acts as a filter on the underlying tables referenced in the view. The query that defines the view can be from one or more tables or from other views in the current or other databases. Distributed queries can also be used to define views that use data from multiple heterogeneous sources. This is useful, for example, if you want to combine similarly structured data from different servers, each of which stores data for a different region of your organization.
There are no restrictions on querying through views and few restrictions on modifying data through them.
Types of Views:
Standard Views
Combining data from one or more tables through a standard view lets you satisfy most of the benefits of using views. These include focusing on specific data and simplifying data manipulation. These benefits are described in more detail in Scenarios for Using Views.
Indexed Views
An indexed view is a view that has been materialized. This means it has been computed and stored. You index a view by creating a unique clustered index on it. Indexed views dramatically improve the performance of some types of queries. Indexed views work best for queries that aggregate many rows. They are not well-suited for underlying data sets that are frequently updated. For more information, see Designing Indexed Views.
Partitioned Views
A partitioned view joins horizontally partitioned data from a set of member tables across one or more servers. This makes the data appear as if from one table. A view that joins member tables on the same instance of SQL Server is a local partitioned view.
Note:
Local partitioned views are included in SQL Server 2005 for backward compatibility purposes only, and are in the process of being deprecated. The preferred method for partitioning data locally is through partitioned tables. For more information, see Partitioned Tables and Indexes.
When a view joins data from tables across servers, it is a distributed partitioned view. Distributed partitioned views are used to implement a federation of database servers. A federation is a group of servers administered independently, but which cooperate to share the processing load of a system. Forming a federation of database servers by partitioning data

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

is the mechanism that lets you scale out a set of servers to support the processing requirements of large, multitiered Web sites.

Scenarios of Using Views

Views are generally used to focus, simplify, and customize the perception each user has of the database. Views can be used as security mechanisms by letting users access data through the view, without granting the users permissions to directly access the underlying base tables of the view. Views can be used to provide a backward compatible interface to emulate a table that used to exist but whose schema has changed. Views can also be used when you copy data to and from Microsoft SQL Server 2005 to improve performance and to partition data.

To Focus on Specific Data

Views let users focus on specific data that interests them and on the specific tasks for which they are responsible. Unnecessary or sensitive data can be left out of the view.

For example, a view vBikes in the AdventureWorks sample database would let a user see the names of all bicycles that are currently in stock. The view filters out all fields from the Product table except Name, and returns only names of finished bicycles instead of bicycle components.

Copy Code

```
CREATE VIEW vBikes AS
SELECT DISTINCT p.[Name] FROM Production.Product p
   JOIN Production.ProductInventory i ON p.ProductID = i.ProductID
   JOIN Production.ProductSubCategory ps
      ON p.ProductSubcategoryID = ps.ProductSubCategoryID
   JOIN Production.ProductCategory pc
      ON (ps.ProductCategoryID = pc.ProductCategoryID
         AND pc.Name = N'Bikes')
      AND i.Quantity > 0
```

To Simplify Data Manipulation

Views can simplify how users work with data. You can define frequently used joins, projections, UNION queries, and SELECT queries as views so that users do not have to specify all the conditions and qualifications every time an additional operation is performed on that data. For example, a complex query that is used for reporting purposes and performs subqueries, outer joins, and aggregation to retrieve data from a group of tables can be created as a view. The view simplifies access to the data because the underlying query does not have to be written or submitted every time the report is generated; the view is queried instead. For more information about manipulating data, see Query Fundamentals.

Although not a complex query, view vBikes in the AdventureWorks sample database lets users focus on specific data without having to construct the JOIN clauses that are required to produce the view.

You can also create inline user-defined functions that logically operate as parameterized views, or views that have parameters in WHERE-clause search conditions or other parts of the query. For more information, see Inline User-defined Functions**.**

To Provide Backward Compatibility

Views enable you to create a backward compatible interface for a table when its schema changes. For example, an application may have referenced a nonnormalized table that has the following schema:

Employee(Name, BirthDate, Salary, Department, BuildingName)

To avoid redundantly storing data in the database, you could decide to normalize the table by splitting it into the following two tables:

Employee2(Name, BirthDate, Salary, DeptId)

Department(DeptId, BuildingName)

To provide a backward-compatible interface that still references data from Employee, you can drop the old Employee table and replace it by the following view:

Copy Code

```
CREATE VIEW Employee AS
SELECT Name, BirthDate, Salary, BuildingName
FROM Employee2 e, Department d
WHERE e.DeptId = d.DeptId
```

Applications that used to query the Employee table can now to obtain their data from the Employee view. The application does not have to be changed if it only reads from Employee. Applications that update Employee can sometimes also be

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

supported by adding INSTEAD OF triggers to the new view to map INSERT, DELETE, and UPDATE operations on the view to the underlying tables. For more information, see Designing INSTEAD OF Triggers.
Designing INSTEAD OF Triggers**.**
The primary advantage of INSTEAD OF triggers is that they enable views that would not be updatable to support updates. A view based on multiple base tables must use an INSTEAD OF trigger to support inserts, updates, and deletes that reference data in more than one table. Another advantage of INSTEAD OF triggers is that they enable you to code logic that can reject parts of a batch while letting other parts of a batch to succeed.
An INSTEAD OF trigger can take actions such as:
Ignoring parts of a batch.
Not processing a part of a batch and logging the problem rows.
Taking an alternative action when an error condition is encountered.

Note:
INSTEAD OF DELETE and INSTEAD OF UPDATE triggers cannot be defined on a table that has a foreign key that is defined by using a DELETE or UPDATE cascading action.

Coding this logic as part of an INSTEAD OF trigger prevents all applications that access the data from having to reimplement the logic.
Example
In the following sequence of Transact-SQL statements, an INSTEAD OF trigger updates two base tables from a view. Additionally, the following approaches to handling errors are shown:
Duplicate inserts to the Person table are ignored, and the information from the insert is logged in the PersonDuplicates table.
Inserts of duplicates to the EmployeeTable are turned into an UPDATE statement that retrieves the current information into the EmployeeTable without generating a duplicate key violation.
The Transact-SQL statements create two base tables, a view, a table to record errors, and the INSTEAD OF trigger on the view. The following tables separate personal and business data and are the base tables for the view.
Copy Code

```
CREATE TABLE Person
  (
   SSN         char(11) PRIMARY KEY,
   Name        nvarchar(100),
   Address     nvarchar(100),
   Birthdate   datetime
  )

CREATE TABLE EmployeeTable
  (
   EmployeeID       int PRIMARY KEY,
   SSN              char(11) UNIQUE,
   Department       nvarchar(10),
   Salary           money,
   CONSTRAINT FKEmpPer FOREIGN KEY (SSN)
   REFERENCES Person (SSN)
  )
```

The following view reports all relevant data from the two tables for a person.
Copy Code

```
CREATE VIEW Employee AS
SELECT P.SSN as SSN, Name, Address,
      Birthdate, EmployeeID, Department, Salary
FROM Person P, EmployeeTable E
WHERE P.SSN = E.SSN
```

You can record attempts to insert rows with duplicate social security numbers. The PersonDuplicates table logs the inserted values, the name of the user who tried the insert, and the time of the insert.
Copy Code

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

```
CREATE TABLE PersonDuplicates
  (
   SSN          char(11),
   Name         nvarchar(100),
   Address      nvarchar(100),
   Birthdate    datetime,
   InsertSNAME  nchar(100),
   WhenInserted datetime
  )
```

The INSTEAD OF trigger inserts rows into multiple base tables from a single view. Attempts to insert rows with duplicate social security numbers are recorded in the PersonDuplicates table. Duplicate rows in the EmployeeTable are changed to update statements.

Copy Code

```
CREATE TRIGGER IO_Trig_INS_Employee ON Employee
INSTEAD OF INSERT
AS
BEGIN
SET NOCOUNT ON
-- Check for duplicate Person. If there is no duplicate, do an insert.
IF (NOT EXISTS (SELECT P.SSN
     FROM Person P, inserted I
     WHERE P.SSN = I.SSN))
  INSERT INTO Person
     SELECT SSN,Name,Address,Birthdate
     FROM inserted
ELSE
-- Log an attempt to insert duplicate Person row in PersonDuplicates table.
  INSERT INTO PersonDuplicates
     SELECT SSN,Name,Address,Birthdate,SUSER_SNAME(),GETDATE()
     FROM inserted
-- Check for duplicate Employee. If no there is duplicate, do an INSERT.
IF (NOT EXISTS (SELECT E.SSN
     FROM EmployeeTable E, inserted
     WHERE E.SSN = inserted.SSN))
  INSERT INTO EmployeeTable
     SELECT EmployeeID,SSN, Department, Salary
     FROM inserted
ELSE
--If there is a duplicate, change to UPDATE so that there will not
--be a duplicate key violation error.
  UPDATE EmployeeTable
     SET EmployeeID = I.EmployeeID,
        Department = I.Department,
        Salary = I.Salary
  FROM EmployeeTable E, inserted I
  WHERE E.SSN = I.SSN
END
```

Designing Views

Before you create a view, consider these guidelines:

You can create views only in the current database. However, the tables and views referenced by the new view can exist in other databases or even other servers if the view is defined using distributed queries.

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

View names must follow the rules for identifiers and must be unique for each schema. Additionally, the name must not be the same as any tables contained by that schema.

- You can build views on other views. Microsoft SQL Server 2005 allows views to be nested. Nesting may not exceed 32 levels. The actual limit on nesting of views may be less depending on the complexity of the view and the available memory.
- You cannot associate rules or DEFAULT definitions with views.
- You cannot associate AFTER triggers with views, only INSTEAD OF triggers.
- The query defining the view cannot include the COMPUTE or COMPUTE BY clauses, or the INTO keyword.
- The query defining the view cannot include the ORDER BY clause, unless there is also a TOP clause in the select list of the SELECT statement.
- The query defining the view cannot contain the OPTION clause specifying a query hint.
- The query defining the view cannot contain the TABLESAMPLE clause.
- You cannot define full-text index definitions on views.
- You cannot create temporary views, and you cannot create views on temporary tables.
- Views, tables, or functions participating in a view created with the SCHEMABINDING clause cannot be dropped, unless the view is dropped or changed so that it no longer has schema binding. In addition, ALTER TABLE statements on tables that participate in views having schema binding will fail if these statements affect the view definition.
- You cannot issue full-text queries against a view, although a view definition can include a full-text query if the query references a table that has been configured for full-text indexing.
- You must specify the name of every column in the view if: Any of the columns in the view are derived from an arithmetic expression, a built-in function, or a constant. Two or more of the columns in the view would otherwise have the same name (usually because the view definition includes a join and the columns from two or more different tables have the same name).
- You want to give any column in the view a name different from the column from which it is derived. (You can also rename columns in the view.) A view column inherits the data type of the column from which it is derived, whether or not you rename it.

Note:
This rule does not apply when a view is based on a query containing an outer join, because columns may change from not allowing null values to allowing them.

Otherwise, you do not need to specify column names when creating the view. SQL Server gives the columns of the view the same names and data types as the columns to which the query defining the view refers. The select list can be a full or partial list of the column names in the base tables.
To create a view you must be granted permission to do so by the database owner and, if the view is created with schema binding, you must have appropriate permissions on any tables or views referenced in the view definition.
By default, as rows are added or updated through a view, they disappear from the scope of the view when they no longer fall into the criteria of the query defining the view. For example, a query can be created, defining a view that retrieves all rows from a table where the employee's salary is less than $30,000. If the employee's salary is increased to $32,000, then querying the view no longer displays that particular employee because his or her salary does not conform to the criteria set by the view. However, the WITH CHECK OPTION clause forces all data modification statements executed against the view to adhere to the criteria set within the SELECT statement defining the view. If you use this clause, rows cannot be modified in a way that causes them to disappear from the view. Any modification that would cause this to happen is canceled and an error is displayed.
The definition of a sensitive view can be encrypted to ensure that its definition cannot be obtained by anyone, including the owner of the view.

64. What is Triggers? What are the different types of triggers in Sql Server 2005?

A trigger is a database object similar to a stored procedure that executes in response to certain actions that occur in your database environment. SQL Server 2005 is packaged with three flavors of trigger objects: AFTER, data definition language (DDL), and INSTEAD-OF.

AFTER triggers are stored procedures that occur after a data manipulation statement has occurred in the database, such as a delete statement. DDL triggers are new to SQL Server 2005, and allow you to respond to object definition level events that occur in the database engine, such as a DROP TABLE statement. INSTEAD-OF triggers are objects that will execute instead of data manipulation statements in the database engine. For example, attaching an INSTEAD-OF INSERT trigger to a table will tell the database engine to execute that trigger instead of executing the statement that would insert values into that table.
Why use an INSTEAD-OF trigger?
INSTEAD-OF triggers are very powerful objects in SQL Server. They allow the developer to divert the database engine to do something different than what the user is trying to do. An example of this would be to add an INSTEAD-OF trigger to any table in your database that rolls back transactions on tables that you do not want modified. You must be careful when using this method because the INSTEAD-OF trigger will need to be disabled before any specified modifications can occur to this table.
Perhaps a more functional reason to use an INSTEAD-OF trigger would be to add the trigger to a view. Adding an INSTEAD-OF trigger to a view essentially allows you to create updateable views. Updateable views allow you to totally abstract your database schema so you can potentially design a system in such a way that your database developers do not have to worry about the OLTP database schema and instead rely upon a standard set of views for data modifications.

Types of DML Triggers

AFTER Triggers
AFTER triggers are executed after the action of the INSERT, UPDATE, or DELETE statement is performed. Specifying AFTER is the same as specifying FOR, which is the only option available in earlier versions of Microsoft SQL Server. AFTER triggers can be specified only on tables.
INSTEAD OF Triggers
INSTEAD OF triggers are executed in place of the usual triggering action. INSTEAD OF triggers can also be defined on views with one or more base tables, where they can extend the types of updates a view can support.
For more information about AFTER and INSTEAD OF triggers, see DML Trigger Planning Guidelines.
CLR Triggers
A CLR Trigger can be either an AFTER or INSTEAD OF trigger. A CLR trigger can also be a DDL trigger. Instead of executing a Transact-SQL stored procedure, a CLR trigger executes one or more methods written in managed code that are members of an assembly created in the .NET Framework and uploaded in SQL Server. For more information, see Programming CLR Triggers.
Examples
A. Using the INSTEAD OF trigger to replace the standard triggering action

Copy Code

```
CREATE TRIGGER TableAInsertTrig ON TableA
INSTEAD OF INSERT
AS ...
```

B. Using the AFTER trigger to augment the standard triggering action

Copy Code

```
CREATE TRIGGER TableBDeleteTrig ON TableB
AFTER DELETE
AS ...
```

C. Using the FOR trigger to augment the standard triggering action

Copy Code

```
-- This statement uses the FOR keyword to generate an AFTER trigger.
CREATE TRIGGER TableCUpdateTrig ON TableC
FOR UPDATE
AS ...
```

DDL Triggers
DDL triggers, like regular triggers, fire stored procedures in response to an event. However, unlike DML triggers, they do not fire in response to UPDATE, INSERT, or DELETE statements on a table or view. Instead, they fire in response to a variety of Data Definition Language (DDL) statements. These statements are primarily statements that start with

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

CREATE, ALTER, and DROP. DDL triggers can be used for administrative tasks such as auditing and regulating database operations.
Use DDL triggers when you want to do the following:

- You want to prevent certain changes to your database schema.
- You want something to occur in the database in response to a change in your database schema.
- You want to record changes or events in the database schema.

DDL triggers fire only after the DDL statements that trigger them are run. DDL triggers cannot be used as **INSTEAD OF triggers.**
The following example illustrates how a DDL trigger can be used to prevent any table in a database from being modified or dropped:

Copy Code

```
CREATE TRIGGER safety
ON DATABASE
FOR DROP_TABLE, ALTER_TABLE
AS
   PRINT 'You must disable Trigger "safety" to drop or alter tables!'
   ROLLBACK ;
```

DDL triggers fire only in response to DDL events specified by Transact-SQL DDL syntax. System stored procedures that perform DDL-like operations are not supported.
DDL triggers can fire in response to a Transact-SQL event processed in the current database, or on the current server. The scope of the trigger depends on the event. For more information about DDL trigger scope, see Designing DDL Triggers**.**
To obtain a DDL trigger example in the AdventureWorks sample database, open the Database Triggers folder in the SQL Server Management Studio Object Explorer, located in the Programmability folder of the AdventureWorks database. Right-click ddlDatabseTriggerLog and select "Script Database Trigger as". DDL trigger ddlDatabseTriggerLog is disabled by default.

65. What is use of extern keyword?

Ans : The extern modifier is used to declare a method that is implemented externally. A common use of the extern modifier is with the DllImport attribute when using Interop services to call into unmanaged code; in this case, the method must also be declared as static, as shown in the following example:

```
[DllImport("avifil32.dll")] private static extern void AVIFileInit();
```

The extern keyword also can define an external assembly alias, making it possible to reference different versions of the same component from within a single assembly.
extern alias
It can sometimes be necessary to reference two versions of assemblies that have the same fully-qualified type names, for example when you need to use two or more versions of an assembly in the same application. By using an external assembly alias, the namespaces from each assembly can be wrapped inside root-level namespaces named by the alias, allowing them to be used in the same file.
To reference two assemblies with the same fully-qualified type names, an alias must be specified on the command line, as follows:

```
/r:GridV1=grid.dll
/r:GridV2=grid20.dll
```

This creates the external aliases GridV1 and GridV2. To use these aliases from within a program, reference them using the extern keyword. For example:

```
extern alias GridV1;
extern alias GridV2;
```

Each extern alias declaration introduces an additional root-level namespace that parallels (but does not lie within) the global namespace. Thus types from each assembly can be referred to without ambiguity using their fully qualified name, rooted in the appropriate namespace-alias

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

In the above example, GridV1::Grid would be the grid control from grid.dll, and GridV2::Grid would be the grid control from grid20.dll.

Until now, C# has supported only a single namespace hierarchy into which types from referenced assemblies and the current program are placed. Because of this design, it has not been possible to reference types with the same fully qualified name from different assemblies, a situation that arises when types are independently given the same name, or when a program needs to reference several versions of the same assembly. Extern aliases make it possible to create and reference separate namespace hierarchies in such situations.
Consider the following two assemblies:
Assembly a1.dll:

```
namespace N
{
        public class A {}
        public class B {}
}
```

Assembly a2.dll:

```
namespace N
{
        public class B {}
        public class C {}
}
```

and the following program:

```
class Test
{
        N.A a;          // Ok
        N.B b;          // Error
        N.C c;          // Ok
}
```

When the program is compiled with the command-line
csc /r:a1.dll /r:a2.dll test.cs
the types contained in a1.dll and a2.dll are all placed in the global namespace hierarchy, and an error occurs because the type N.B exists in both assemblies. With extern aliases, it becomes possible to place the types contained in a1.dll and a2.dll into separate namespace hierarchies.
The following program declares and uses two extern aliases, X and Y, each of which represent the root of a distinct namespace hierarchy created from the types contained in one or more assemblies.

```
extern alias X;
extern alias Y;
class Test
{
        X::N.A a;
        X::N.B b1;
        Y::N.B b2;
        Y::N.C c;
}
```

The program declares the existence of the extern aliases X and Y, but the actual definitions of the aliases are external to the program. With the command line compiler, the definition takes place when assemblies are referenced using the /r option. In Visual Studio, the definition takes place by including the alias in the Aliases property for one or more of the assemblies referenced by the project. The command line
csc /r:X=a1.dll /r:Y=a2.dll test.cs
defines the extern alias X to be the root of a namespace hierarchy formed by the types in a1.dll and Y to be the root of a namespace hierarchy formed by the types in a2.dll. The identically named N.B classes can now be referenced as X.N.B and Y.N.B, or, using the namespace alias qualifier, X::N.B and Y::N.B. An error occurs if a program declares an extern alias for which no external definition is provided.
An extern alias can include multiple assemblies, and a particular assembly can be included in multiple extern aliases. For example, given the assembly

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

Assembly a3.dll:

```
namespace N
{
        public class D {}
        public class E {}
}
```

the command line

```
csc /r:X=a1.dll /r:X=a3.dll /r:Y=a2.dll /r:Y=a3.dll test.cs
```

defines the extern alias X to be the root of a namespace hierarchy formed by the types in a1.dll and a3.dll and Y to be the root of a namespace hierarchy formed by the types in a2.dll and a3.dll. Because of this definition, it is possible to refer to the class N.D in a3.dll as both X::N.D and Y::N.D.

An assembly can be placed in the global namespace hierarchy even if it is also included in one or more extern aliases. For example, the command line

```
csc /r:a1.dll /r:X=a1.dll /r:Y=a2.dll test.cs
```

places the assembly a1.dll in both the global namespace hierarchy and the namespace hierarchy rooted by the extern alias X. Consequently, the class N.A can be referred to as N.A or X::N.A.

It is possible to ensure that a lookup always starts at the root of the global namespace hierarchy by using the identifier global with the namespace alias qualifier, such as global::System.IO.Stream.

A using directive may reference an extern alias that was defined in the same immediately enclosing namespace declaration or compilation unit. For example:

```
extern alias X;
using X::N;
class Test
{
        A a;                    // X::N.A
        B b;                    // X::N.B
}
```

66. What is base assembly of Dot net?

Ans : MsCorLib.dll is basic assembly which contain int32,string,object etc classes.

67. What's difference between Shadowing and Overriding?

Ans: Following are the differences between shadowing and overriding:-

- Overriding redefines only the implementation while shadowing redefines the whole element.
- In overriding derived classes can refer the parent class element by using "this" keyword , but in shadowing you can access it by "base"

68. What's difference between Server.Transfer and response.Redirect?

Following are the major differences between them:-

- Response.Redirect sends message to the browser saying it to move to some different page.While server.transfer does not send any message to the browser but rather redirects the user directly from the server itself. So in server.transfer there is no round trip while response.redirect has a round trip and hence puts a load on server.
- Using Server.Transfer you can not redirect to a different from the server itself. Example If your server is www.yahoo.com you can't use server.transfer to move to www.microsoft.com but yes you can move to www.yahoo.com/travels , i.e. within websites. This cross server redirect is possible only using Response.redirect.

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

- With server.transfer you can preserve your information. It has a parameter called as “preserveForm”. So the existing query string etc. will be able in the calling page. In response.redirect you can maintain the state. You can but has lot of drawbacks. If you are navigating with in the same website use “Server.transfer” or else go for “response.redirect()”

69. Can you explain in brief how the ASP.NET authentication process works?

Ans : ASP.NET does not run by itself it run inside the process of IIS. So there are two authentication layers which exist in ASP.NET system. First authentication happens at the IIS level and then at the ASP.NET level depending on the WEB.CONFIG file. Below is how the whole process works:-

- IIS first checks to make sure the incoming request comes from an IP address that is allowed access to the domain. If not it denies the request. √ Next IIS performs its own user authentication if it configured to do so. By default IIS allows anonymous access, so requests are automatically authenticated, but you can change this default on a per – application basis with in IIS.
- If the request is passed to ASP.net with an authenticated user, ASP.net checks to see whether impersonation is enabled. If impersonation is enabled, ASP.net acts as though it were the authenticated user. If not ASP.net acts with its own configured account.
- Finally the identity from step 3 is used to request resources from the operating system. If ASP.net authentication can obtain all the necessary resources it grants the users request otherwise it is denied. Resources can include much more than just the ASP.net page itself you can also use .Net’s code access security features to extend this authorization step to disk files, Registry keys and other resources.

70. What are the various ways of authentication techniques in ASP.NET?

Ans : Selecting an authentication provider is as simple as making an entry in the web.config file for the application. You can use one of these entries to select the corresponding built in authentication provider:

- <authentication mode=”windows”>
- <authentication mode=”passport”>
- <authentication mode=”forms”>
- Custom authentication where you might install an ISAPI filter in IIS that compares incoming requests to list of source IP addresses, and considers requests to be authenticated if they come from an acceptable address. In that case, you would set the authentication mode to none to prevent any of the .net authentication providers from being triggered.

Windows authentication and IIS If you select windows authentication for your ASP.NET application, you also have to configure authentication within IIS. This is because IIS provides Windows authentication. IIS gives you a choice for four different authentication methods: Anonymous, basic digest and windows integrated If you select anonymous authentication, IIS doesn’t perform any authentication, Any one is allowed to access the ASP.NET application. If you select basic authentication, users must provide a windows username and password to connect. How ever this information is sent over the network in clear text, which makes basic authentication very much insecure over the internet. If you select digest authentication, users must still provide a windows user name and password to connect. However the password is hashed before it is sent across the network. Digest authentication requires that all users be running Internet Explorer 5 or later and that windows accounts to stored in active directory. If you select windows integrated authentication, passwords never cross the network. Users must still have a username and password, but the application uses either the Kerberos or challenge/response protocols authenticate the user. Windows-integrated authentication requires that all users be running internet explorer 3.01 or later Kerberos is a network authentication protocol. It is designed to provide strong authentication for client/server applications by using secret-key cryptography. Kerberos is a solution to network security problems. It provides the tools of authentication and strong cryptography over the network to help to secure information in systems across entire enterprise

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

Passport authentication Passport authentication lets you to use Microsoft's passport service to authenticate users of your application. If your users have signed up with passport, and you configure the authentication mode of the application to the passport authentication, all authentication duties are off-loaded to the passport servers. Passport uses an encrypted cookie mechanism to indicate authenticated users. If users have already signed into passport when they visit your site, they'll be considered authenticated by ASP.NET. Otherwise they'll be redirected to the passport servers to log in. When they are successfully log in, they'll be redirected back to your site To use passport authentication you have to download the Passport Software Development Kit (SDK) and install it on your server. The SDK can be found at http:// msdn.microsoft.com/library/default.asp?url=/downloads/list/websrvpass.aps.It includes full details of implementing passport authentication in your own applications.

Forms authentication Forms authentication provides you with a way to handle authentication using your own custom logic with in an ASP.NET application. The following applies if you choose forms authentication.
When a user requests a page for the application, ASP.NET checks for the presence of a special session cookie. If the cookie is present, ASP.NET assumes the user is authenticated and processes the request.
If the cookie isn't present, ASP.NET redirects the user to a web form you provide

71. How does authorization work in ASP.NET?

ASP.NET impersonation is controlled by entries in the applications web.config file. The default setting is "no impersonation". You can explicitly specify that ASP.NET shouldn't use impersonation by including the following code in the file <identity impersonate="false"/> It means that ASP.NET will not perform any authentication and runs with its own privileges. By default ASP.NET runs as an unprivileged account named ASPNET. You can change this by making a setting in the processModel section of the machine.config file. When you make this setting, it automatically applies to every site on the server. To user a high-privileged system account instead of a low-privileged, set the userName attribute of the processModel element to SYSTEM. Using this setting is a definite security risk, as it elevates the privileges of the ASP.NET process to a point where it can do bad things to the operating system. When you disable impersonation, all the request will run in the context of the account running ASP.NET: either the ASPNET account or the system account. This is true when you are using anonymous access or authenticating users in some fashion. After the user has been authenticated, ASP.NET uses it own identity to request access to resources. The second possible setting is to turn on impersonation. <identity impersonate ="true"/> In this case, ASP.NET takes on the identity IIS passes to it. If you are allowing anonymous access in IIS, this means ASP.NET will impersonate the IUSR_ComputerName account that IIS itself uses. If you aren't allowing anonymous access, ASP.NET will take on the credentials of the authenticated user and make requests for resources as if it were that user. Thus by turning impersonation on and using a non-anonymous method of authentication in IIS, you can let users log on and use their identities within your ASP.NET application. Finally, you can specify a particular identity to use for all authenticated requests <identity impersonate="true" username="DOMAIN\username" password="password"/ With this setting, all the requests are made as the specified user (Assuming the password it correct in the configuration file). So, for example you could designate a user for a single application, and use that user's identity every time someone authenticates to the application. The drawback to this technique is that you must embed the user's password in the web.config file in plain text. Although ASP.NET won't allow anyone to download this file, this is still a security risk if anyone can get the file by other means.

72. What's difference between Datagrid, Datalist and repeater?

Ans : A Datagrid, Datalist and Repeater are all ASP.NET data Web controls.
They have many things in common like DataSource Property , DataBind Method ItemDataBound and ItemCreated. When you assign the DataSource Property of a Datagrid to a DataSet then each DataRow present in the DataRow Collection of DataTable is assigned to a corresponding DataGridItem and this is same for the rest of the two controls also.But The HTML code generated for a Datagrid has an HTML TABLE <ROW> element created for the particular DataRow and its a Table form representation with Columns and Rows. For a Datalist its an Array of Rows and based on the Template Selected and the RepeatColumn Property value We can specify how many DataSource records should appear per HTML <table> row. In short in datagrid we have one record per row, but in datalist we can have five or six rows per row. For a

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

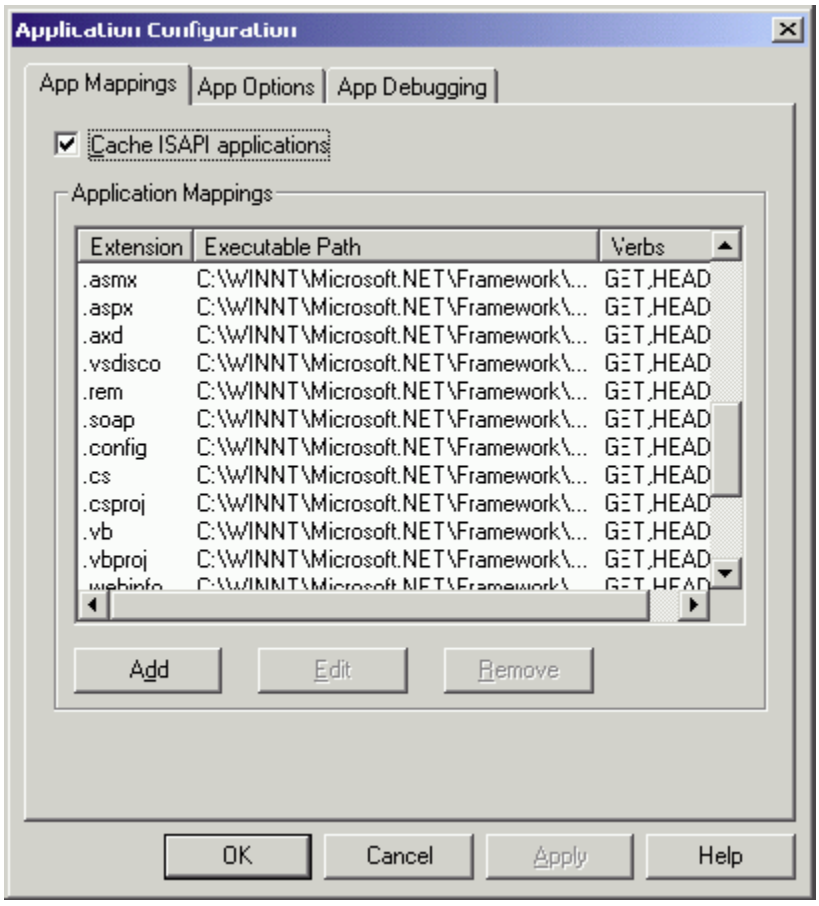
Repeater Control, The Datarecords to be displayed depends upon the Templates specified and the only HTML generated is the due to the Templates. In addition to these, Datagrid has a in-built support for Sort, Filter and paging the Data ,which is not possible when using a DataList and for a Repeater Control we would require to write an explicit code to do paging.

73. What exactly happens when ASPX page is requested from Browser?

Ans : **Note**: - Here the interviewer is expecting complete flow of how an ASPX page is processed with respect to IIS and ASP.NET engine.

Following are the steps which occur when we request an ASPX page:-

- The browser sends the request to the webserver.let's assume that the webserver at the other end is IIS.
- Once IIS receives the request he looks on which engine can serve this request. When I mean engine means the DLL who can parse this page or compile and send a response back to browser. Which request to map to is decided by file extension of the page requested. Depending on file extension following are some mapping
- .aspx, for ASP.NET Web pages,
- .asmx, for ASP.NET Web services,
- .config, for ASP.NET configuration files,
- .ashx, for custom ASP.NET HTTP handlers,
- .rem, for remoting resources
- Etc you can also configure the extension mapping to which engine it can route by using the IIS engine.



Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

Figure: - 7.1 Following screen shows some IIS mappings

Example a ASP page will be sent to old classic ASP.DLL to compile. While .ASPX pages will be routed to ASP.NET engine for compilation.

- As this book mainly will target ASP.NET we will look in to how ASP.NET pages that is ASPX pages generation sequence occurs. Once IIS passes the request to ASP.NET engine page has to go through two section HTTP module section and HTTP handler section. Both these section have there own work to be done in order that the page is properly compiled and sent to the IIS. HTTP modules inspect the incoming request and depending on that they can change the internal workflow of the request. HTTP handler actually compiles the page and generates output. If you see your machine.config file you will see following section of HTTP modules <httpModules> <add name="OutputCache" type="System.Web.Caching.OutputCacheModule" /> <add name="Session" type="System.Web.SessionState.SessionStateModule" /> <add name="WindowsAuthentication" type="System.Web.Security.WindowsAuthenticationModule" /> <add name="FormsAuthentication" type="System.Web.Security.FormsAuthenticationModule" /> <add name="PassportAuthentication" type="System.Web.Security.PassportAuthenticationModule" /> <add name="UrlAuthorization" type="System.Web.Security.UrlAuthorizationModule" /> <add name="FileAuthorization" type="System.Web.Security.FileAuthorizationModule" /> <add name="ErrorHandlerModule" type="System.Web.Mobile.ErrorHandlerModule, System.Web.Mobile, Version=1.0.5000.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a" / ></httpModules>133 The above mapping shows which functionality is handled by which Namespace. Example FormsAthuentication is handled by “System.Web.Security.FormsAuthenticationModule”. If you look at the web.config section HTTP module is where authentication and authorization happens. Ok now the HTTP handler is where the actual compilation takes place and the output is generated.Following is a paste from HTTP handler section of WEB.CONFIG file. <httpHandlers> <add verb="*" path="*.vjsproj" type="System.Web.HttpForbiddenHandler" /> <add verb="*" path="*.java" type="System.Web.HttpForbiddenHandler" /> <add verb="*" path="*.jsl" type="System.Web.HttpForbiddenHandler" /> <add verb="*" path="trace.axd" type="System.Web.Handlers.TraceHandler" /> <add verb="*" path="*.aspx" type="System.Web.UI.PageHandlerFactory" /> <add verb="*" path="*.ashx" type="System.Web.UI.SimpleHandlerFactory" /> ... </httpHandlers>
- Depending on the File extension handler decides which Namespace will generate the output. Example all .ASPX extension files will be compiled by System.Web.UI.PageHandlerFactory
- Once the file is compiled it send back again to the HTTP modules and from there to IIS and then to the browser.

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

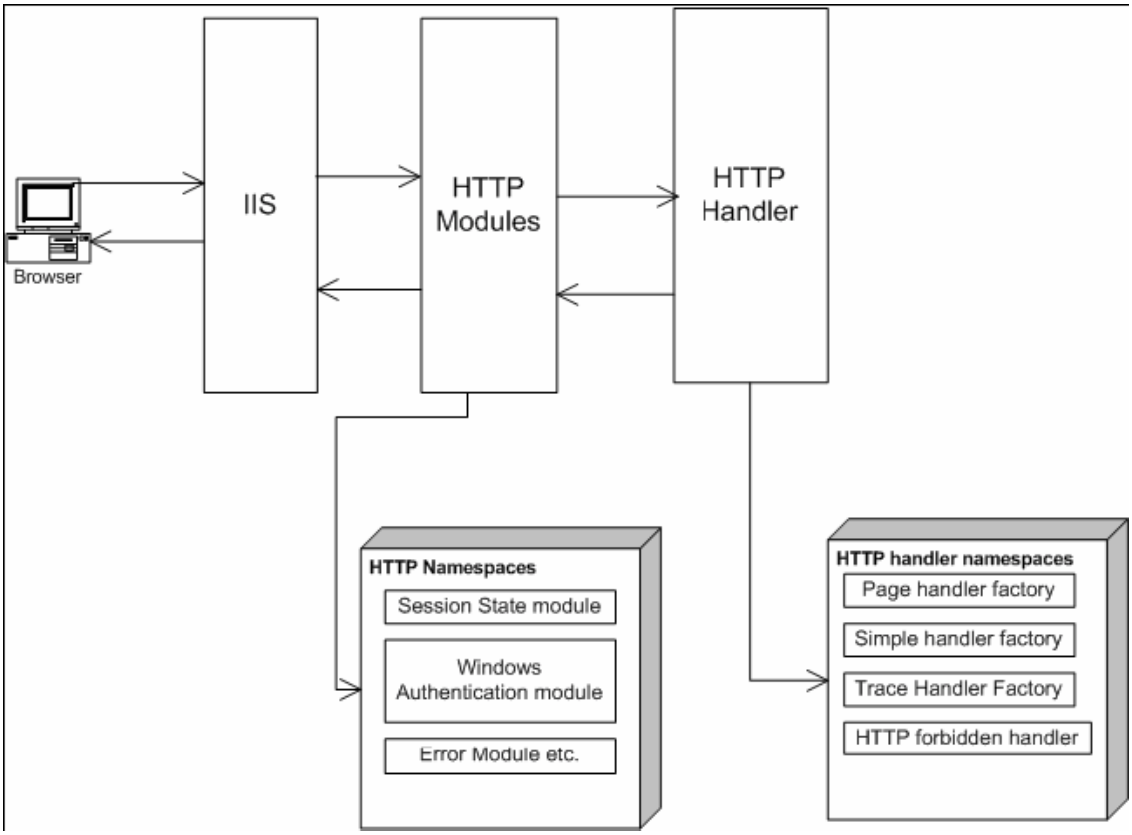


Figure :- 7.2 IIS flow from various sections.

74. What is the result of "select firstname, secondname from emp order by 2"?

Ans: it will sort according to second field name.

75. How can we create proxy class without VS?

Ans: We can create proxy class with the use of WSDL.exe application in dot net framework.

76. How can we create overloaded methods in Web Service?

Web services are also classes just like any other .NET classes. Nevertheless they have methods marked as WebMethods that can be exposed by the WebServices to be consumed by the outside world. Apart from these WebMethods they can also have normal methods like any other classes have.
Since a web service is a class it can utilize all the OO features like method overloading. However to use this feature on WebMethods we need to do something more that is explained in this article.
Creating WebMethods:

Let us create a simple WebService that has the following overloaded methods:
public string GetGreeting()

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

public string GetGreeting(string p_Name)
public string GetGreeting(string p_Name, string p_Message)

All these three methods return variants of a Greeting message to the WebClient. Let us now mark the methods as Web Methods. To achieve this apply the [WebMethod] attribute to the public methods.

```
[WebMethod]
public string GetGreeting()
{
   return "Hi Guest";
}


[WebMethod]
public string GetGreeting(string p_Name)
{
   return "Hi " + p_Name + "!";
}


[WebMethod]
public string GetGreeting(string p_Name, string p_Message)
{
   return "Hi " + p_Name + "!" + p_Message;
}
```

This would compile fine. Run the WebService in the browser. That should give an error saying that the GetGreeting() mthods use the same message name 'GetGreeting' and asking to use the MessageName property of the WebMethod.

Adding the MessageName property:

Add the MessageName property to the WebMethod attribute as shown below:

```
[WebMethod]public string GetGreeting()
{
   return "Hi Guest";
}
[WebMethod (MessageName="WithOneString")]
public string GetGreeting(string p_Name)
{
   return "Hi " + p_Name + "!";
}
[WebMethod (MessageName="WithTwoStrings")]
public string GetGreeting(string p_Name, string p_Message)
{
   return "Hi " + p_Name + "!" + p_Message;
}
```

Now compile the WebService and run in the browser. You can see that the first method is displayed as GetGreeting wherein for the second and third method the alias we set using the MessageName property is displayed.

77. How can we maintain State in WebService?

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

Ans : Webservices as such do not have any mechanism by which they can maintain state.Webservices can access ASP.NET intrinsic objects like Session , application etc. if they inherit from “WebService” base class.82 <%@ Webservice class="TestWebServiceClass" %> Imports System.Web.Services Public class TestWebServiceClass Inherits WebService <WebMethod> Public Sub SetSession (value As String) session("Val") = Value End Sub end class Above is a sample code which sets as session object calles as “val”. TestWebserviceClass is inheriting from WebService to access the session and application objects.

78. ASP.Net Page Life Cycle ?

Ans : General Page Life-cycle Stages

Stage	Description
Page request	The page request occurs before the page life cycle begins. When the page is requested by a user, ASP.NET determines whether the page needs to be parsed and compiled (therefore beginning the life of a page), or whether a cached version of the page can be sent in response without running the page.
Start	In the start step, page properties such as Request and Response are set. At this stage, the page also determines whether the request is a postback or a new request and sets the IsPostBack property. Additionally, during the start step, the page's UICulture property is set.
Page initialization	During page initialization, controls on the page are available and each control's UniqueID property is set. Any themes are also applied to the page. If the current request is a postback, the postback data has not yet been loaded and control property values have not been restored to the values from view state.
Load	During load, if the current request is a postback, control properties are loaded with information recovered from view state and control state.
Validation	During validation, the Validate method of all validator controls is called, which sets the IsValid property of individual validator controls and of the page.
Postback event handling	If the request is a postback, any event handlers are called.
Rendering	Before rendering, view state is saved for the page and all controls. During the rendering phase, the page calls the Render method for each control, providing a text writer that writes its output to the OutputStream of the page's Response property.
Unload	Unload is called after the page has been fully rendered, sent to the client, and is ready to be discarded. At this point, page properties such as Response and Request are unloaded and any cleanup is performed.

Life-cycle Events

Within each stage of the life cycle of a page, the page raises events that you can handle to run your own code. For control events, you bind the event handler to the event, either declaratively using attributes such as onclick, or in code.

Pages also support automatic event wire-up, meaning that ASP.NET looks for methods with particular names and automatically runs those methods when certain events are raised. If the AutoEventWireup attribute of the @ Page directive is set to true (or if it is not defined, since by default it is true), page events are automatically bound to methods that use the naming convention of Page_event, such as Page_Load and Page_Init. For more information on automatic event wire-up, see ASP.NET Web Server Control Event Model.

The following table lists the page life-cycle events that you will use most frequently. There are more events than those listed; however, they are not used for most page processing scenarios. Instead, they are primarily used by server controls on the ASP.NET Web page to initialize and render themselves. If you want to write your own ASP.NET server controls, you need to understand more about these stages. For information about creating custom controls, see Developing Custom ASP.NET Server Controls.

Page Event	Typical Use
PreInit	Use this event for the following: Check the IsPostBack property to determine whether this is the first time the page is being processed. Create or re-create dynamic controls.

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

	Set a master page dynamically. Set the Theme property dynamically. Read or set profile property values. Note If the request is a postback, the values of the controls have not yet been restored from view state. If you set a control property at this stage, its value might be overwritten in the next event.
Init	Raised after all controls have been initialized and any skin settings have been applied. Use this event to read or initialize control properties.
InitComplete	Raised by the Page object. Use this event for processing tasks that require all initialization be complete.
PreLoad	Use this event if you need to perform processing on your page or control before the Load event. After the Page raises this event, it loads view state for itself and all controls, and then processes any postback data included with the Request instance.
Load	The Page calls the OnLoad event method on the Page, then recursively does the same for each child control, which does the same for each of its child controls until the page and all controls are loaded. Use the OnLoad event method to set properties in controls and establish database connections.
Control events	Use these events to handle specific control events, such as a Button control's Click event or a TextBox control's TextChanged event. Note In a postback request, if the page contains validator controls, check the IsValid property of the Page and of individual validation controls before performing any processing.
LoadComplete	Use this event for tasks that require that all other controls on the page be loaded.
PreRender	Before this event occurs: The Page object calls EnsureChildControls for each control and for the page. Each data bound control whose DataSourceID property is set calls its DataBind method. For more information, see Data Binding Events for Data-Bound Controls below. The PreRender event occurs for each control on the page. Use the event to make final changes to the contents of the page or its controls.
SaveStateComplete	Before this event occurs, ViewState has been saved for the page and for all controls. Any changes to the page or controls at this point will be ignored. Use this event perform tasks that require view state to be saved, but that do not make any changes to controls.
Render	This is not an event; instead, at this stage of processing, the Page object calls this method on each control. All ASP.NET Web server controls have a Render method that writes out the control's markup that is sent to the browser. If you create a custom control, you typically override this method to output the control's markup. However, if your custom control incorporates only standard ASP.NET Web server controls and no custom markup, you do not need to override the Render method. For more information, see Developing Custom ASP.NET Server Controls. A user control (an .ascx file) automatically incorporates rendering, so you do not need to explicitly render the control in code.
Unload	This event occurs for each control and then for the page. In controls, use this event to do final cleanup for specific controls, such as closing control-specific database connections. For the page itself, use this event to do final cleanup work, such as closing open files and database connections, or finishing up logging or other request-specific tasks. Note During the unload stage, the page and its controls have been rendered, so you cannot make further changes to the response stream. If you attempt to call a method such as the Response.Write method, the page will throw an exception.

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

79. What are the different data types in dot net?

There is primitive, reference, and value types of data type in dot net.
Primitive
Primitive types map directly to types existing in the Framework Class Library (FCL). For example, in C#, an int maps directly to the **System.Int32.**
Reference Type
Reference types are always allocated from the managed heap, and the C# new operator returns the memory address of the object—the memory address refers to the object's bits. You need to bear in mind some performance considerations when you're working with reference types. First, consider these facts:
• The memory must be allocated from the managed heap.
• Each object allocated on the heap has some additional overhead members associated with it that must be initialized.
• The other bytes in the object (for the fields) are always set to zero.
• Allocating an object from the managed heap could force a garbage collection to occur.

Value Type

Value Type variables are loaded in stack. Value type instances don't come under the control of the garbage collector, so their use reduces pressure in the managed heap and reduces the number of collections an application requires over its lifetime. Eg. Struct, enum etc. all of the structures are immediately derived from the System.ValueType abstract type. System.ValueType is itself immediately derived from the System.Object type. By definition, all value types must be derived from System.ValueType. All enumerations are derived from the System.Enum abstract type, which is itself derived from System.ValueType. The CLR and all programming languages give enumerations special treatment. In addition, all value types are sealed, which prevents a value type from being used as a base type for any other reference type or value type. So, for example, it's not possible to define any new types using Boolean, Char, Int32, UInt64, Single, Double, Decimal, and so on as base types.

Difference between Value Type and Reference Type

- Value type objects have two representations: an unboxed form and a boxed form.
- Value types are derived from System.ValueType. This type offers the same methods as defined by System.Object. However, System.ValueType overrides the Equals method so that it returns true if the values of the two objects' fields match. In addition, System .ValueType overrides the GetHashCode method to produce a hash code value by using an algorithm that takes into account the values in the object's instance fields. Due to performance issues with this default implementation, when defining your own value types, you should override and provide explicit implementations for the Equals and GetHashCode methods.
- Because you can't define a new value type or a new reference type by using a value type as a base class, you shouldn't introduce any new virtual methods into a value type. No methods can be abstract, and all methods are implicitly sealed (can't be overridden).
- Reference type variables contain the memory address of objects in the heap. By default,when a reference type variable is created, it is initialized to null, indicating that the reference type variable doesn't currently point to a valid object. Attempting to use a null reference type variable causes a NullReferenceException to be thrown. By contrast,value type variables always contain a value of the underlying type, and all members of the value type are initialized to 0. Since a value type variable isn't a pointer, it's not possible to generate a NullReferenceException when accessing a value type. The CLR does offer a special feature that adds the notion of nullability to a value type.
- When you assign a value type variable to another value type variable, a field-by-field copy is made. When you assign a reference type variable to another reference type variable, only the memory address is copied.
- Because of the previous point, two or more reference type variables can refer to a single object in the heap, allowing operations on one variable to affect the object referenced by the other variable. On the other hand, value type variables are distinct objects, and it's not possible for operations on one value type variable to affect another.
- Because unboxed value types aren't allocated on the heap, the storage allocated for them is freed as soon as the method that defines an instance of the type is no longer active. This means that a value type instance doesn't receive a notification (via a Finalize method) when its memory is reclaimed.

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

80. What is Static class?

Ans: There are certain classes that are never intended to be instantiated, such as Console, Math, Environment, and ThreadPool. These classes have only static members and, in fact, the classes exist simply as a way to group a set of related members together The compiler enforces many restrictions on a static class:

- The class must be derived directly from System.Object because deriving from any other base class makes no sense since inheritance applies only to objects, and you cannot create an instance of a static class.
- The class must not implement any interfaces since interface methods are callable only when using an instance of a class.
- The class must define only static members (fields, methods, properties, and events).Any instance members cause the compiler to generate an error.
- The class cannot be used as a field, method parameter, or local variable because all of these would indicate a variable that refers to an instance and this is not allowed. If the compiler detects any of these uses, the compiler issues an error.

81. How can you increase SQL performance?

Following are tips which will increase your SQl performance :-

- Every index increases the time in takes to perform INSERTS, UPDATES and DELETES, so the number of indexes should not be very much. Try to use maximum 4-5 indexes on one table, not more. If you have read-only table, then the number of indexes may be increased.
- Keep your indexes as narrow as possible. This reduces the size of the index and reduces the number of reads required to read the index.
- Try to create indexes on columns that have integer values rather than character values.
- If you create a composite (multi-column) index, the order of the columns in the key are very important. Try to order the columns in the key as to enhance selectivity, with the most selective columns to the leftmost of the key.
- If you want to join several tables, try to create surrogate integer keys for this purpose and create indexes on their columns.
- Create surrogate integer primary key (identity for example) if your table will not have many insert operations.
- Clustered indexes are more preferable than nonclustered, if you need to select by a range of values or you need to sort results set with GROUP BY or ORDER BY.
- If your application will be performing the same query over and over on the same table, consider creating a covering index on the table.
- You can use the SQL Server Profiler Create Trace Wizard with "Identify Scans of Large Tables" trace to determine which tables in your database may need indexes. This trace will show which tables are being scanned by queries instead of using an index.

82. What is ACID fundamental and what are transactions in SQL SERVER?

A transaction is a sequence of operations performed as a single logical unit of work. A logical unit of work must exhibit four properties, called the ACID (Atomicity, Consistency, Isolation, and Durability) properties, to qualify as a transaction:

- Atomicity: A transaction must be an atomic unit of work; either all of its data modifications are performed or none of them is performed.
- Consistency: When completed, a transaction must leave all data in a consistent state. In a relational database, all rules must be applied to the transaction's modifications to maintain all data integrity. All internal data structures, such as B-tree indexes or doubly-linked lists, must be correct at the end of the transaction.
- Isolation: Modifications made by concurrent transactions must be isolated from the modifications made by any other concurrent transactions. A transaction either sees data in the state it was in before another concurrent completed, but it does not see an intermediate state. This is referred to as serializability because it results in the

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

ability to reload the starting data and replay a series of transactions to end up with the data in the same state it was in after the original transactions were performed.

- Durability: After a transaction has completed, its effects are permanently in place in the system. The modifications persist even in the event of a system failure.

83. If we have multiple AFTER Triggers on table how can we define the sequence of the triggers?

If a table has multiple AFTER triggers, then you can specify which trigger should be executed first and which trigger should be executed last using the stored procedure sp_settriggerorder. All the other triggers are in an undefined order which you cannot control.

84. Give the output of following code ?

```
using System;
class Parent{
        public virtual void callme()
        {
        Console.WriteLine("Called Parent");
        }
}
class Child : Parent
{
  public override void callme()
        {
          Console.WriteLine("Called Child");
        }
}
class implementpoly
        {
          public static void Main()
        {
           Parent objPar=new Child ();
           objPar.callme();
        }
}
```

Ans : Called Child. It is example of runtime polymorphism.

85. Give the output of following code?

```
using System;
class Parent
{
 public virtual void callme()
        {
        Console.WriteLine("Called Parent");
        }
```


Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

```
}

class Child : Parent
{
   public override void callme()
            {
              Console.WriteLine("Called Child");
            }
}

class implementpoly
{
   public static void Main()
{
Child objPar=new Parent();
objPar.callme();
}
                                  }
```

Ans : Compile Time Error : runtimepolymorphism.cs(24,15): error CS0266: Cannot implicitly convert type 'Parent' to 'Child'. An explicit conversion exists (are you missing a cast?)

86. Give output of following code?

```
using System;
class Parent
{
 public virtual void callme()
            {
            Console.WriteLine("Called Parent");
            }
}
class Child : Parent
{
   public new void callme()
            {
              Console.WriteLine("Called Child");
            }
}



class implementpoly
{
   public static void Main()
{
    Parent objPar= new Child();
    objPar.callme();
}
}
```

Ans : Called Parent

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

87. What is Performance tips and tricks in .net application?

Ans : Performance Tips for All Applications

There are a few tips to remember when working on the CLR in any language. These are relevant to everyone, and should be the first line of defense when dealing with performance issues.

Throw Fewer Exceptions

Throwing exceptions can be very expensive, so make sure that you don't throw a lot of them. Use Perfmon to see how many exceptions your application is throwing. It may surprise you to find that certain areas of your application throw more exceptions than you expected. For better granularity, you can also check the exception number programmatically by using Performance Counters.

Finding and designing away exception-heavy code can result in a decent perf win. Bear in mind that this has nothing to do with try/catch blocks: you only incur the cost when the actual exception is thrown. You can use as many try/catch blocks as you want. Using exceptions gratuitously is where you lose performance. For example, you should stay away from things like using exceptions for control flow.

Here's a simple example of how expensive exceptions can be: we'll simply run through a For loop, generating thousands or exceptions and then terminating. Try commenting out the throw statement to see the difference in speed: those exceptions result in tremendous overhead.

Copy Code

```
public static void Main(string[] args){ int j = 0; for(int i = 0; i < 10000; i++){ try{ j = i; throw new System.Exception(); } catch {} } System.Console.Write(j); return; }
```

Beware! The run time can throw exceptions on its own! For example, Response.Redirect() throws a ThreadAbort exception. Even if you don't explicitly throw exceptions, you may use functions that do. Make sure you check Perfmon to get the real story, and the debugger to check the source.

To Visual Basic developers: Visual Basic turns on int checking by default, to make sure that things like overflow and divide-by-zero throw exceptions. You may want to turn this off to gain performance.

If you use COM, you should keep in mind that HRESULTS can return as exceptions. Make sure you keep track of these carefully.

Make Chunky Calls

A chunky call is a function call that performs several tasks, such as a method that initializes several fields of an object. This is to be viewed against chatty calls, that do very simple tasks and require multiple calls to get things done (such as setting every field of an object with a different call). It's important to make chunky, rather than chatty calls across methods where the overhead is higher than for simple, intra-AppDomain method calls. P/Invoke, interop and remoting calls all carry overhead, and you want to use them sparingly. In each of these cases, you should try to design your application so that it doesn't rely on small, frequent calls that carry so much overhead.

A transition occurs whenever managed code is called from unmanaged code, and vice versa. The run time makes it extremely easy for the programmer to do interop, but this comes at a performance price. When a transition happens, the following steps needs to be taken:

- Perform data marshalling
- Fix Calling Convention
- Protect callee-saved registers
- Switch thread mode so that GC won't block unmanaged threads
- Erect an Exception Handling frame on calls into managed code
- Take control of thread (optional)

To speed up transition time, try to make use of P/Invoke when you can. The overhead is as little as 31 instructions plus the cost of marshalling if data marshalling is required, and only 8 otherwise. COM interop is much more expensive, taking upwards of 65 instructions.

Data marshalling isn't always expensive. Primitive types require almost no marshalling at all, and classes with explicit layout are also cheap. The real slowdown occurs during data translation, such as text conversion from ASCI to Unicode. Make sure that data that gets passed across the managed boundary is only converted if it needs to be: it may turn out that simply by agreeing on a certain datatype or format across your program you can cut out a lot of marshalling overhead.

The following types are called blittable, meaning they can be copied directly across the managed/unmanaged boundary with no marshalling whatsoever: sbyte, byte, short, ushort, int, uint, long, ulong, float and double. You can pass these for

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

free, as well as ValueTypes and single-dimensional arrays containing blittable types. The gritty details of marshalling can be explored further on the MSDN Library. I recommend reading it carefully if you spend a lot of your time marshalling.

Design with ValueTypes

Use simple structs when you can, and when you don't do a lot of boxing and unboxing. Here's a simple example to demonstrate the speed difference:

```
using System;
namespace ConsoleApplication{
```

Copy Code

```
public struct foo{ public foo(double arg){ this.y = arg; } public double y; } public class bar{ public bar(double arg){ this.y = arg; } public double y; } class Class1{ static void Main(string[] args){ System.Console.WriteLine("starting struct loop..."); for(int i = 0; i < 50000000; i++) {foo test = new foo(3.14);} System.Console.WriteLine("struct loop complete. starting object loop..."); for(int i = 0; i < 50000000; i++) {bar test2 = new bar(3.14); } System.Console.WriteLine("All done"); } } }
```

When you run this example, you'll see that the struct loop is orders of magnitude faster. However, it is important to beware of using ValueTypes when you treat them like objects. This adds extra boxing and unboxing overhead to your program, and can end up costing you more than it would if you had stuck with objects! To see this in action, modify the code above to use an array of foos and bars. You'll find that the performance is more or less equal.

Tradeoffs ValueTypes are far less flexible than Objects, and end up hurting performance if used incorrectly. You need to be very careful about when and how you use them.

Try modifying the sample above, and storing the foos and bars inside arrays or hashtables. You'll see the speed gain disappear, just with one boxing and unboxing operation.

You can keep track of how heavily you box and unbox by looking at GC allocations and collections. This can be done using either Perfmon externally or Performance Counters in your code.

See the in-depth discussion of ValueTypes in Performance Considerations of Run-Time Technologies in the .NET Framework.

Use AddRange to Add Groups

Use AddRange to add a whole collection, rather than adding each item in the collection iteratively. Nearly all windows controls and collections have both Add and AddRange methods, and each is optimized for a different purpose. Add is useful for adding a single item, whereas AddRange has some extra overhead but wins out when adding multiple items. Here are just a few of the classes that support Add and AddRange:

- StringCollection, TraceCollection, etc.
- HttpWebRequest
- UserControl
- ColumnHeader

Trim Your Working Set

Minimize the number of assemblies you use to keep your working set small. If you load an entire assembly just to use one method, you're paying a tremendous cost for very little benefit. See if you can duplicate that method's functionality using code that you already have loaded.

Keeping track of your working set is difficult, and could probably be the subject of an entire paper. Here are some tips to help you out:

Use vadump.exe to track your working set. This is discussed in another white paper covering various tools for the managed environment.

Look at Perfmon or Performance Counters. They can give you detail feedback about the number of classes that you load, or the number of methods that get JITed. You can get readouts for how much time you spend in the loader, or what percent of your execution time is spent paging.

Use For Loops for String Iteration—version 1

In C#, the foreach keyword allows you to walk across items in a list, string, etc. and perform operations on each item. This is a very powerful tool, since it acts as a general-purpose enumerator over many types. The tradeoff for this generalization is speed, and if you rely heavily on string iteration you should use a For loop instead. Since strings are simple character arrays, they can be walked using much less overhead than other structures. The JIT is smart enough (in many cases) to optimize away bounds-checking and other things inside a For loop, but is prohibited from doing this on foreach walks. The end result is that in version 1, a For loop on strings is up to five times faster than using foreach. This will change in future versions, but for version 1 this is a definite way to increase performance.

Here's a simple test method to demonstrate the difference in speed. Try running it, then removing the For loop and uncommenting the foreach statement. On my machine, the For loop took about a second, with about 3 seconds for the foreach statement.

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

Copy Code
public static void Main(string[] args) { string s = "monkeys!"; int dummy = 0; System.Text.StringBuilder sb = new System.Text.StringBuilder(s); for(int i = 0; i < 1000000; i++) sb.Append(s); s = sb.ToString(); //foreach (char c in s) dummy++; for (int i = 0; i < 1000000; i++) dummy++; return; } }

Tradeoffs Foreach is far more readable, and in the future it will become as fast as a For loop for special cases like strings. Unless string manipulation is a real performance hog for you, the slightly messier code may not be worth it.

Use StringBuilder for Complex String Manipulation

When a string is modified, the run time will create a new string and return it, leaving the original to be garbage collected. Most of the time this is a fast and simple way to do it, but when a string is being modified repeatedly it begins to be a burden on performance: all of those allocations eventually get expensive. Here's a simple example of a program that appends to a string 50,000 times, followed by one that uses a StringBuilder object to modify the string in place. The StringBuilder code is much faster, and if you run them it becomes immediately obvious.

Copy Code namespace ConsoleApplication1.Feedback Copy Code using System; public class Feedback{ public Feedback(){ text = "You have ordered: \n"; } public string text; public static int Main(string[] args) { Feedback test = new Feedback(); String str = test.text; for(int i=0;i<50000;i++){ str = str + "blue_toothbrush"; } System.Console.Out.WriteLine("done"); return 0; } } }	Copy Code namespace ConsoleApplication1.Feedback Copy Code using System; public class Feedback{ public Feedback(){ text = "You have ordered: \n"; } public string text; public static int Main(string[] args) { Feedback test = new Feedback(); System.Text.StringBuilder SB = new System.Text.StringBuilder(test.text); for(int i=0;i<50000;i++){ SB.Append("blue_toothbrush"); } System.Console.Out.WriteLine("done"); return 0; } } }

Try looking at Perfmon to see how much time is saved without allocating thousands of strings. Look at the "% time in GC" counter under the .NET CLR Memory list. You can also track the number of allocations you save, as well as collection statistics.

Tradeoffs There is some overhead associated with creating a StringBuilder object, both in time and memory. On a machine with fast memory, a StringBuilder becomes worthwhile if you're doing about five operations. As a rule of thumb, I would say 10 or more string operations is a justification for the overhead on any machine, even a slower one.

Precompile Windows Forms Applications

Methods are JITed when they are first used, which means that you pay a larger startup penalty if your application does a lot of method calling during startup. Windows Forms use a lot of shared libraries in the OS, and the overhead in starting them can be much higher than other kinds of applications. While not always the case, precompiling Windows Forms applications usually results in a performance win. In other scenarios it's usually best to let the JIT take care of it, but if you are a Windows Forms developer you might want to take a look.

Microsoft allows you to precompile an application by calling ngen.exe. You can choose to run ngen.exe during install time or before you distribute you application. It definitely makes the most sense to run ngen.exe during install time, since you can make sure that the application is optimized for the machine on which it is being installed. If you run ngen.exe before you ship the program, you limit the optimizations to the ones available on your machine. To give you an idea of how much precompiling can help, I've run an informal test on my machine. Below are the cold startup times for ShowFormComplex, a winforms application with roughly a hundred controls.

Code State	Time
Framework JITed ShowFormComplex JITed	3.4 sec
Framework Precompiled, ShowFormComplex JITed	2.5 sec
Framework Precompiled, ShowFormComplex Precompiled	2.1sec

Each test was performed after a reboot. As you can see, Windows Forms applications use a lot of methods up front, making it a substantial performance win to precompile.

Use Jagged Arrays—Version 1

The v1 JIT optimizes jagged arrays (simply 'arrays-of-arrays') more efficiently than rectangular arrays, and the difference is quite noticeable. Here is a table demonstrating the performance gain resulting from using jagged arrays in place of rectangular ones in both C# and Visual Basic (higher numbers are better):

	C#	Visual Basic 7

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

Assignment (jagged) Assignment (rectangular)	14.16 8.37	12.24 8.62
Neural Net (jagged) Neural net (rectangular)	4.48 3.00	4.58 3.13
Numeric Sort (jagged) Numeric Sort (rectangular)	4.88 2.05	5.07 2.06

The assignment benchmark is a simple assignment algorithm, adapted from the step-by-step guide found in Quantitative Decision Making for Business (Gordon, Pressman, and Cohn; Prentice-Hall; out of print). The neural net test runs a series of patterns over a small neural network, and the numeric sort is self-explanatory. Taken together, these benchmarks represent a good indication of real-world performance.
As you can see, using jagged arrays can result in fairly dramatic performance increases. The optimizations made to jagged arrays will be added to future versions of the JIT, but for v1 you can save yourself a lot of time by using jagged arrays.

Keep IO Buffer Size Between 4KB and 8KB

For nearly every application, a buffer between 4KB and 8KB will give you the maximum performance. For very specific instances, you may be able to get an improvement from a larger buffer (loading large images of a predictable size, for example), but in 99.99% of cases it will only waste memory. All buffers derived from BufferedStream allow you to set the size to anything you want, but in most cases 4 and 8 will give you the best performance.

Be on the Lookout for Asynchronous IO Opportunities

In rare cases, you may be able to benefit from Asynchronous IO. One example might be downloading and decompressing a series of files: you can read the bits in from one stream, decode them on the CPU and write them out to another. It takes a lot of effort to use Asynchronous IO effectively, and it can result in a performance loss if it's not done right. The advantage is that when applied correctly, Asynchronous IO can give you as much as ten times the performance.
An excellent example of a program using Asynchronous IO is available on the MSDN Library.

- One thing to note is that there is a small security overhead for asynchronous calls: Upon invoking an async call, the security state of the caller's stack is captured and transferred to the thread that'll actually execute the request. This may not be a concern if the callback executes lot of code, or if async calls aren't used excessively

Tips for Database Access

The philosophy of tuning for database access is to use only the functionality that you need, and to design around a 'disconnected' approach: make several connections in sequence, rather than holding a single connection open for a long time. You should take this change into account and design around it.
Microsoft recommends an N-Tier strategy for maximum performance, as opposed to a direct client-to-database connection. Consider this as part of your design philosophy, as many of the technologies in place are optimized to take advantage of a multi-tired scenario.

Use the Optimal Managed Provider

Make the correct choice of managed provider, rather than relying on a generic accessor. There are managed providers written specifically for many different databases, such as SQL (System.Data.SqlClient). If you use a more generic interface such as System.Data.Odbc when you could be using a specialized component, you will lose performance dealing with the added level of indirection. Using the optimal provider can also have you speaking a different language: the Managed SQL Client speaks TDS to a SQL database, providing a dramatic improvement over the generic OleDbprotocol.

Pick Data Reader Over Data Set When You Can

Use a data reader whenever when you don't need to keep the data lying around. This allows a fast read of the data, which can be cached if the user desires. A reader is simply a stateless stream that allows you to read data as it arrives, and then drop it without storing it to a dataset for more navigation. The stream approach is faster and has less overhead, since you can start using data immediately. You should evaluate how often you need the same data to decide if the caching for navigation makes sense for you. Here's a small table demonstrating the difference between DataReader and DataSet on both ODBC and SQL providers when pulling data from a server (higher numbers are better):

	ADO	SQL
DataSet	801	2507
DataReader	1083	4585

As you can see, the highest performance is achieved when using the optimal managed provider along with a data reader. When you don't need to cache your data, using a data reader can provide you with an enormous performance boost.

Use Mscorsvr.dll for MP Machines

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

For stand-alone middle-tier and server applications, make sure mscorsvr is being used for multiprocessor machines. Mscorwks is not optimized for scaling or throughput, while the server version has several optimizations that allow it to scale well when more than one processor is available.

Use Stored Procedures Whenever Possible

Stored procedures are highly optimized tools that result in excellent performance when used effectively. Set up stored procedures to handle inserts, updates, and deletes with the data adapter. Stored procedures do not have to be interpreted, compiled or even transmitted from the client, and cut down on both network traffic and server overhead. Be sure to use CommandType.StoredProcedure instead of CommandType.Text

Be Careful About Dynamic Connection Strings

Connection pooling is a useful way to reuse connections for multiple requests, rather than paying the overhead of opening and closing a connection for each request. It's done implicitly, but you get one pool per unique connection string. If you're generating connection strings dynamically, make sure the strings are identical each time so pooling occurs. Also be aware that if delegation is occurring, you'll get one pool per user. There are a lot of options that you can set for the connection pool, and you can track the performance of the pool by using the Perfmon to keep track of things like response time, transactions/sec, etc.

Turn Off Features You Don't Use

Turn off automatic transaction enlistment if it's not needed. For the SQL Managed Provider, it's done via the connection string:

Copy Code

```
SqlConnection conn = new SqlConnection( "Server=mysrv01; Integrated Security=true; Enlist=false");
```

When filling a dataset with the data adapter, don't get primary key information if you don't have to (e.g. don't set MissingSchemaAction.Add with key):

Copy Code

```
public DataSet SelectSqlSrvRows(DataSet dataset,string connection,string query){ SqlConnection conn = new SqlConnection(connection); SqlDataAdapter adapter = new SqlDataAdapter(); adapter.SelectCommand = new SqlCommand(query, conn); adapter.MissingSchemaAction = MissingSchemaAction.AddWithKey; adapter.Fill(dataset); return dataset; }
```

Avoid Auto-Generated Commands

When using a data adapter, avoid auto-generated commands. These require additional trips to the server to retrieve meta data, and give you a lower level of interaction control. While using auto-generated commands is convenient, it's worth the effort to do it yourself in performance-critical applications.

Beware ADO Legacy Design

Be aware that when you execute a command or call fill on the adapter, every record specified by your query is returned. If server cursors are absolutely required, they can be implemented through a stored procedure in t-sql. Avoid where possible because server cursor-based implementations don't scale very well.

If needed, implement paging in a stateless and connectionless manner. You can add additional records to the dataset by:

Making sure PK info is present

Changing the data adapter's select command as appropriate, and

Calling Fill

Keep Your Datasets Lean

Only put the records you need into the dataset. Remember that the dataset stores all of its data in memory, and that the more data you request, the longer it will take to transmit across the wire.

Use Sequential Access as Often as Possible

With a data reader, use CommandBehavior.SequentialAccess. This is essential for dealing with blob data types since it allows data to be read off of the wire in small chunks. While you can only work with one piece of the data at a time, the latency for loading a large data type disappears. If you don't need to work the whole object at once, using Sequential Access will give you much better performance.

Performance Tips for ASP.NET Applications

Cache Aggressively

When designing an app using ASP.NET, make sure you design with an eye on caching. On server versions of the OS, you have a lot of options for tweaking the use of caches on the server and client side. There are several features and tools in ASP that you can make use of to gain performance.

Output Caching—Stores the static result of an ASP request. Specified using the <@% OutputCache %> directive:

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

- Duration—Time item exists in the cache
- VaryByParam—Varies cache entries by Get/Post params
- VaryByHeader—Varies cache entries by Http header
- VaryByCustom—Varies cache entries by browser
- Override to vary by whatever you want:
- Fragment Caching—When it is not possible to store an entire page (privacy, personalization, dynamic content), you can use fragment caching to store parts of it for quicker retrieval later.
    - VaryByControl—Varies the cached items by values of a control
- Cache API—Provides extremely fine granularity for caching by keeping a hashtable of cached objects in memory (System.web.UI.caching). It also:
    - Includes Dependencies (key, file, time)
    - Automatically expires unused items
    - Supports Callbacks

Caching intelligently can give you excellent performance, and it's important to think about what kind of caching you need. Imagine a complex e-commerce site with several static pages for login, and then a slew of dynamically-generated pages containing images and text. You might want to use Output Caching for those login pages, and then Fragment Caching for the dynamic pages. A toolbar, for example, could be cached as a fragment. For even better performance, you could cache commonly used images and boilerplate text that appear frequently on the site using the Cache API. For detailed information on caching (with sample code), check out the ASP. NET Web site.

Use Session State Only If You Need To

One extremely powerful feature of ASP.NET is its ability to store session state for users, such as a shopping cart on an e-commerce site or a browser history. Since this is on by default, you pay the cost in memory even if you don't use it. If you're not using Session State, turn it off and save yourself the overhead by adding <@% EnabledSessionState = false %> to your asp. This comes with several other options, which are explained at the ASP. NET Web site.

For pages that only read session state, you can choose EnabledSessionState=readonly. This carries less overhead than full read/write session state, and is useful when you need only part of the functionality and don't want to pay for the write capabilities.

Use View State Only If You Need To

An example of View State might be a long form that users must fill out: if they click Back in their browser and then return, the form will remain filled. When this functionality isn't used, this state eats up memory and performance. Perhaps the largest performance drain here is that a round-trip signal must be sent across the network each time the page is loaded to update and verify the cache. Since it is on by default, you will need to specify that you do not want to use View State with <@% EnabledViewState = false %>. You should read more about View State on the the ASP. NET Web site to learn about some of the other options and settings to which you have access.

Avoid STA COM

Apartment COM is designed to deal with threading in unmanaged environments. There are two kinds of Apartment COM: single-threaded and multithreaded. MTA COM is designed to handle multithreading, whereas STA COM relies on the messaging system to serialize thread requests. The managed world is free-threaded, and using Single Threaded Apartment COM requires that all unmanaged threads essentially share a single thread for interop. This results in a massive performance hit, and should be avoided whenever possible. If you can't port the Apartment COM object to the managed world, use <@%AspCompat = "true" %> for pages that use them. For a more detailed explanation of STA COM, see the MSDN Library.

Batch Compile

Always batch compile before deploying a large page into the Web. This can be initiated by doing one request to a page per directory and waiting until the CPU idles again. This prevents the Web server from being bogged down with compilations while also trying to serve out pages.

Remove Unnecessary Http Modules

Depending on the features used, remove unused or unnecessary http modules from the pipeline. Reclaiming the added memory and wasted cycles can provide you with a small speed boost.

Avoid the Autoeventwireup Feature

Instead of relying on autoeventwireup, override the events from Page. For example, instead of writing a Page_Load() method, try overloading the public void OnLoad() method. This allows the run time from having to do a CreateDelegate() for every page.

Encode Using ASCII When You Don't Need UTF

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

By default, ASP.NET comes configured to encode requests and responses as UTF-8. If ASCII is all your application needs, eliminated the UTF overhead can give you back a few cycles. Note that this can only be done on a per-application basis.
Use the Optimal Authentication Procedure
There are several different ways to authenticate a user and some of more expensive than others (in order of increasing cost: None, Windows, Forms, Passport). Make sure you use the cheapest one that best fits your needs.

88. How Garbage Collector Works?

.Net is the much hyped revolutionary technology gifted to the programmer's community by Microsoft. Many factors make it a must use for most developers. In this article we would like to discuss one of the primary advantages of .NET framework - the ease in memory and resource management.
Every program uses resources of one sort or another-memory buffers, network connections, database resources, and so on. In fact, in an object-oriented environment, every type identifies some resource available for a program's use. To use any of these resources, memory must be allocated to represent the type.
The steps required to access a resource are as follows:

- Allocate memory for the type that represents the resource.
- Initialize the memory to set the initial state of the resource and to make the resource usable.
- Use the resource by accessing the instance members of the type (repeat as necessary).
- Tear down the state of the resource to clean up.
- Free the memory.

The garbage collector (GC) of .NET completely absolves the developer from tracking memory usage and knowing when to free memory.
The Microsoft® .NET CLR (Common Language Runtime) requires that all resources be allocated from the managed heap. You never free objects from the managed heap-objects are automatically freed when they are no longer needed by the application.
Memory is not infinite. The garbage collector must perform a collection in order to free some memory. The garbage collector's optimizing engine determines the best time to perform a collection, (the exact criteria is guarded by Microsoft) based upon the allocations being made. When the garbage collector performs a collection, it checks for objects in the managed heap that are no longer being used by the application and performs the necessary operations to reclaim their memory.
However for automatic memory management, the garbage collector has to know the location of the roots i.e. it should know when an object is no longer in use by the application. This knowledge is made available to the GC in .NET by the inclusion of a concept know as metadata. Every data type used in .NET software includes metadata that describes it. With the help of metadata, the CLR knows the layout of each of the objects in memory, which helps the Garbage Collector in the compaction phase of Garbage collection. Without this knowledge the Garbage Collector wouldn't know where one object instance ends and the next begins.

Garbage Collection Algorithm
Application Roots
Every application has a set of roots. Roots identify storage locations, which refer to objects on the managed heap or to objects that are set to null.
For example:
¨ All the global and static object pointers in an application.
¨ Any local variable/parameter object pointers on a thread's stack.
¨ Any CPU registers containing pointers to objects in the managed heap.
¨ Pointers to the objects from Freachable queue
The list of active roots is maintained by the just-in-time (JIT) compiler and common language runtime, and is made accessible to the garbage collector's algorithm.

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

Implementation
Garbage collection in .NET is done using tracing collection and specifically the CLR implements the Mark/Compact collector.
This method consists of two phases as described below.
Phase I: Mark
Find memory that can be reclaimed.
When the garbage collector starts running, it makes the assumption that all objects in the heap are garbage. In other words, it assumes that none of the application's roots refer to any objects in the heap.
The following steps are included in Phase I:
1. The GC identifies live object references or application roots.
2. It starts walking the roots and building a graph of all objects reachable from the roots.
3. If the GC attempts to add an object already present in the graph, then it stops walking down that path. This serves two purposes. First, it helps performance significantly since it doesn't walk through a set of objects more than once. Second, it prevents infinite loops should you have any circular linked lists of objects. Thus cycles are handles properly.
Once all the roots have been checked, the garbage collector's graph contains the set of all objects that are somehow reachable from the application's roots; any objects that are not in the graph are not accessible by the application, and are therefore considered garbage.
Phase II: Compact
Move all the live objects to the bottom of the heap, leaving free space at the top.
Phase II includes the following steps:
1. The garbage collector now walks through the heap linearly, looking for contiguous blocks of garbage objects (now considered free space).
2. The garbage collector then shifts the non-garbage objects down in memory, removing all of the gaps in the heap.
3. Moving the objects in memory invalidates all pointers to the objects. So the garbage collector modifies the application's roots so that the pointers point to the objects' new locations.
4. In addition, if any object contains a pointer to another object, the garbage collector is responsible for correcting these pointers as well.
After all the garbage has been identified, all the non-garbage has been compacted, and all the non-garbage pointers have been fixed-up, a pointer is positioned just after the last non-garbage object to indicate the position where the next object can be added.
Finalization
.Net Framework's garbage collection implicitly keeps track of the lifetime of the objects that an application creates, but fails when it comes to the unmanaged resources (i.e. a file, a window or a network connection) that objects encapsulate. The unmanaged resources must be explicitly released once the application has finished using them. .Net Framework provides the Object.Finalize method: a method that the garbage collector must run on the object to clean up its unmanaged resources, prior to reclaiming the memory used up by the object. Since Finalize method does nothing, by default, this method must be overridden if explicit cleanup is required.
It would not be surprising if you will consider Finalize just another name for destructors in C++. Though, both have been assigned the responsibility of freeing the resources used by the objects, they have very different semantics. In C++, destructors are executed immediately when the object goes out of scope whereas a finalize method is called once when Garbage collection gets around to cleaning up an object.
The potential existence of finalizers complicates the job of garbage collection in .Net by adding some extra steps before freeing an object.
Whenever a new object, having a Finalize method, is allocated on the heap a pointer to the object is placed in an internal data structure called Finalization queue. When an object is not reachable, the garbage collector considers the object garbage. The garbage collector scans the finalization queue looking for pointers to these objects. When a pointer is found, the pointer is removed from the finalization queue and appended to another internal data structure called Freachable queue, making the object no longer a part of the garbage. At this point, the garbage collector has finished identifying garbage. The garbage collector compacts the reclaimable memory and the special runtime thread empties the freachable queue, executing each object's Finalize method.
The next time the garbage collector is invoked, it sees that the finalized objects are truly garbage and the memory for those objects is then, simply freed.
Thus when an object requires finalization, it dies, then lives (resurrects) and finally dies again. It is recommended to avoid using Finalize method, unless required. Finalize methods increase memory pressure by not letting the memory and the

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

resources used by that object to be released, until two garbage collections. Since you do not have control on the order in which the finalize methods are executed, it may lead to unpredictable results.

Garbage Collection Performance Optimizations

· Weak references
· Generations

Weak References

Weak references are a means of performance enhancement, used to reduce the pressure placed on the managed heap by large objects.

When a root points to an abject it's called a strong reference to the object and the object cannot be collected because the application's code can reach the object.

When an object has a weak reference to it, it basically means that if there is a memory requirement & the garbage collector runs, the object can be collected and when the application later attempts to access the object, the access will fail. On the other hand, to access a weakly referenced object, the application must obtain a strong reference to the object. If the application obtains this strong reference before the garbage collector collects the object, then the GC cannot collect the object because a strong reference to the object exists.

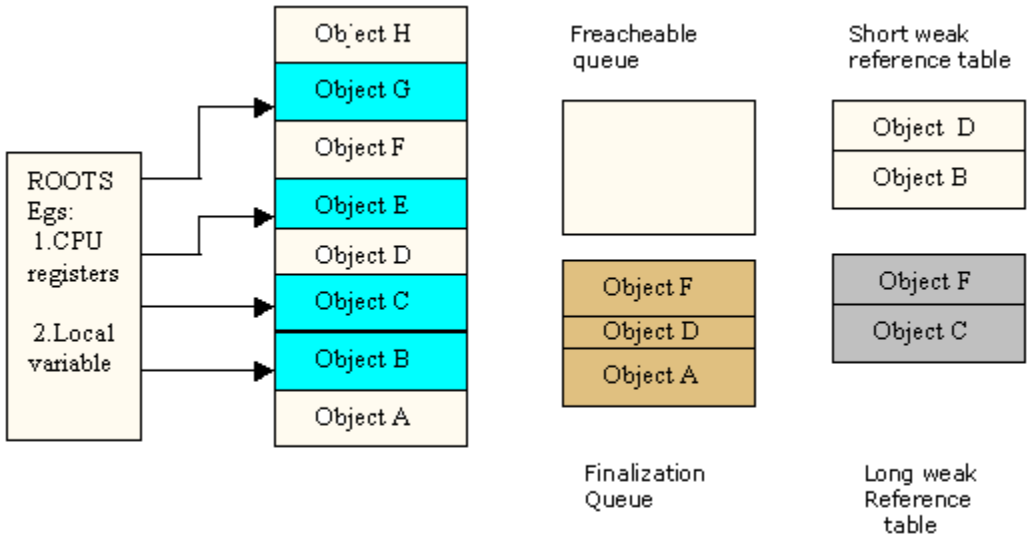
The managed heap contains two internal data structures whose sole purpose is to manage weak references: the short weak reference table and the long weak reference table.

- **Weak references are of two types:**
  A short weak reference doesn't track resurrection.
  i.e. the object which has a short weak reference to itself is collected immediately without running its finalization method.
- A long weak reference tracks resurrection.
  i.e. the garbage collector collects object pointed to by the long weak reference table only after determining that the object's storage is reclaimable. If the object has a Finalize method, the Finalize method has been called and the object was not resurrected.

These two tables simply contain pointers to objects allocated within the managed heap. Initially, both tables are empty. When you create a WeakReference object, an object is not allocated from the managed heap. Instead, an empty slot in one of the weak reference tables is located; short weak references use the short weak reference table and long weak references use the long weak reference table.

Consider an example of what happens when the garbage collector runs. The diagrams (Figure 1 & 2) below show the state of all the internal data structures before and after the GC runs.

Figure 1 Scenario before the GC runs

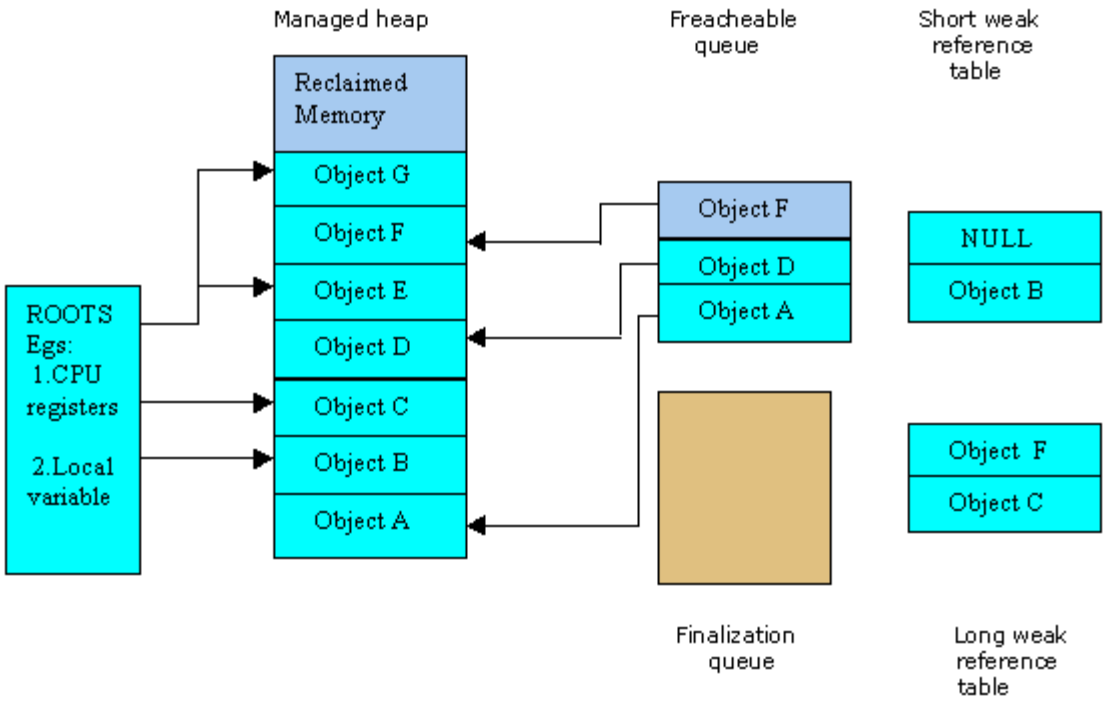


Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

Now, here's what happens when a garbage collection (GC) runs:

1. The garbage collector builds a graph of all the reachable objects. In the above example, the graph will include objects B, C, E, G.
2. The garbage collector scans the short weak reference table. If a pointer in the table refers to an object that is not part of the graph, then the pointer identifies an unreachable object and the slot in the short weak reference table is set to null. In the above example, slot of object D is set to null since it is not a part of the graph.
3. The garbage collector scans the finalization queue. If a pointer in the queue refers to an object that is not part of the graph, then the pointer identifies an unreachable object and the pointer is moved from the finalization queue to the freachable queue. At this point, the object is added to the graph since the object is now considered reachable. In the above example, though objects A, D, F are not included in the graph they are treated as reachable objects because they are part of the finalization queue. Finalization queue thus gets emptied.
4. The garbage collector scans the long weak reference table. If a pointer in the table refers to an object that is not part of the graph (which now contains the objects pointed to by entries in the freachable queue), then the pointer identifies an unreachable object and the slot is set to null. Since both the objects C and F are a part of the graph (of the previous step), none of them are set to null in the long reference table.
5. The garbage collector compacts the memory, squeezing out the holes left by the unreachable objects. In the above example, object H is the only object that gets removed from the heap and it's memory is reclaimed.

Figure 2 Scenario after the GC runs



Generations

Since garbage collection cannot complete without stopping the entire program, they can cause arbitrarily long pauses at arbitrary times during the execution of the program. Garbage collection pauses can also prevent programs from responding to events quickly enough to satisfy the requirements of real-time systems.

One feature of the garbage collector that exists purely to improve performance is called generations. A generational garbage collector takes into account two facts that have been empirically observed in most programs in a variety of languages:

1. newly created objects tend to have short lives.
2. The older an object is, the longer it will survive.

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

Generational collectors group objects by age and collect younger objects more often than older objects. When initialized, the managed heap contains no objects. All new objects added to the heap can be said to be in generation 0, until the heap gets filled up which invokes garbage collection. As most objects are short-lived, only a small percentage of young objects are likely to survive their first collection. Once an object survives the first garbage collection, it gets promoted to generation 1.Newer objects after GC can then be said to be in generation 0.The garbage collector gets invoked next only when the sub-heap of generation 0 gets filled up. All objects in generation 1 that survive get compacted and promoted to generation 2. All survivors in generation 0 also get compacted and promoted to generation 1. Generation 0 then contains no objects, but all newer objects after GC go into generation 0.
Thus, as objects "mature" (survive multiple garbage collections) in their current generation, they are moved to the next older generation. Generation 2 is the maximum generation supported by the runtime's garbage collector. When future collections occur, any surviving objects currently in generation 2 simply stay in generation 2.
Thus, dividing the heap into generations of objects and collecting and compacting younger generation objects improves the efficiency of the basic underlying garbage collection algorithm by reclaiming a significant amount of space from the heap and also being faster than if the collector had examined the objects in all generations.
A garbage collector that can perform generational collections, each of which is guaranteed (or at least very likely) to require less than a certain maximum amount of time, can help make runtime suitable for real-time environment and also prevent pauses that are noticeable to the user.

Myths Related To Garbage Collection

GC is necessarily slower than manual memory management.

Counter Explanation:
Not necessarily. Modern garbage collectors appear to run as quickly as manual storage allocators (malloc/free or new/delete). Garbage collection probably will not run as quickly as customized memory allocator designed for use in a specific program. On the other hand, the extra code required to make manual memory management work properly (for example, explicit reference counting) is often more expensive than a garbage collector would be.

GC will necessarily make my program pause.

Counter Explanation:
Since garbage collectors usually stop the entire program while seeking and collecting garbage objects, they cause pauses long enough to be noticed by the users. But with the advent of modern optimization techniques, these noticeable pauses can be eliminated.

Manual memory management won't cause pauses.

Counter Explanation:
Manual memory management does not guarantee performance. It may cause pauses for considerable periods either on allocation or deallocation.

Programs with GC are huge and bloated; GC isn't suitable for small programs or systems.

Counter Explanation:
Though using garbage collection is advantageous in complex systems, there is no reason for garbage collection to introduce any significant overhead at any scale.

I've heard that GC uses twice as much memory.

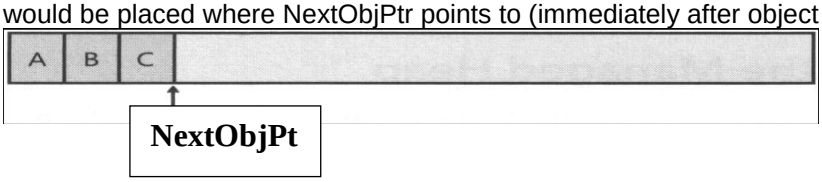
Counter Explanation:
This may be true of primitive GCs, but this is not generally true of garbage collection. The data structures used for GC need be no larger than those for manual memory management.

89. How objects are allocated on heap?

The CLR requires that all resources to be allocated from a heap called the managed heap. This heap is similar to a C-runtime heap, except that you never delete objects from the managed heap—objects are automatically deleted when the application no longer needs them. This, of course, raises the question, "How does the managed heap know when the application is no longer using an object?" I'll address this question shortly. Several garbage-collection algorithms are in use today. Each algorithm is fine-tuned for a particular environment to provide the best performance. In this chapter, I'll concentrate on the garbage-collection algorithm used by the Microsoft .NET Framework's CLR. Let's start off with the basic concepts.When a process is initialized, the CLR reserves a contiguous region of address space that initially contains no backing storage. This address space region is the

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

managed heap. The heap also maintains a pointer, which I'll call NextObjPtr. This pointer indicates where the next object is to be allocated within the heap. Initially, NextObjPtr is set to the base address of the reserved address space region.The newobj intermediate language (IL) instruction creates an object. Many languages (including C#, C++/CLI, and Microsoft Visual Basic) offer a new operator that causes the compiler to emit a newobj instruction into the method's IL code. The newobj instruction causes the CLR to perform the following steps:
1. Calculate the number of bytes required for the type's (and all of its base type's) fields.
2. Add the bytes required for an object's overhead. Each object has two overhead fields: fields requires 32 bits, adding 8 bytes to each object. For a 64-bit application, each field is 64 bits, adding 16 bytes to each object. The CLR then checks that the bytes required to allocate the object are available in the reserved region (committing storage if necessary). If there is enough free space in the managed heap, the object will fit, starting at the address pointed to by NextObjPtr, and these bytes are zeroed out. The type's constructor is called (passing NextObjPtr for the this parameter), and the newobj IL instruction (or C#'s new operator) returns the address of the object. Just before the address is returned, NextObjPtr is advanced past the object and now points to the address where the next object will be placed in the heap.
Figure shows a managed heap consisting of three objects: A, B, and C. If a new object were to be allocated, it would be placed where NextObjPtr points to (immediately after object C).



The managed heap gains these advantages because it makes one really big assumption: that address space and storage are infinite. Obviously, this assumption NextObjPtr is ridiculous, and the managed heap must employ a mechanism to allow it to make this assumption. This mechanism is the garbage collector. Here's how it works: When an application calls the new operator to create an object, there might not be enough address space left in the region to allocate to the object. The heap detects this lack of space by adding the bytes that the object requires to the address in NextObjPtr. If the resulting value is beyond the end of the address space region, the heap is full, and a garbage collection must be performed.

90. What causes finalize methods to be called?

Finalize methods are called at the completion of a garbage collection, which is started by one of the following five events:

- Generation 0 is full When generation 0 is full, a garbage collection starts. This event is by far the most common way for Finalize methods to be called because it occurs naturally as the application code runs, allocating new objects. I'll discuss generations later in this chapter.
- Code explicitly calls System.GC's static Collect method Code can explicitly request that the CLR perform a collection. Although Microsoft strongly discourages such requests, at times it might make sense for an application to force a collection.
- Windows is reporting low memory conditions The CLR internally use the Win32
- CreateMemoryResourceNotification and QueryMemoryResourceNotification functions to monitor system memory overall. If Windows reports low memory, the CLR will force a garbage collection in an effort to free up dead objects to reduce the size of a process' working set.
- The CLR is unloading an AppDomain When an AppDomain unloads, the CLR considers nothing in the AppDomain to be a root, and a garbage collection consisting of all generations is performed. I'll discuss AppDomains in Chapter 21, "CLR Hosting and AppDomains."
- The CLR is shutting down The CLR shuts down when a process terminates normally (as opposed to an external shutdown via Task Manager, for example). During this shutdown, the CLR considers nothing in the process to be a root and calls the Finalize method for all objects in the managed heap. Note that the CLR does not attempt to compact or free memory here because the whole process is terminating, and Windows will
- Reclaim all of the processes' memory.

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

The CLR uses a special, dedicated thread to call Finalize methods. For the first four events, if a Finalize method enters an infinite loop, this special thread is blocked, and no more Finalize methods can be called. This is a very bad situation because the application will never be able to reclaim the memory occupied by the finalizable objects—the application will leak memory as long as it runs. For the fifth event, each Finalize method is given approximately two seconds to return. If a Finalize method doesn't return within two seconds, the CLR just kills the process—no more Finalize methods are called. Also, if it takes more than 40 seconds to call all objects'
Finalize methods; again, the CLR just kills the process.
Note These timeout values were correct at the time I wrote this text, but Microsoft might change them in the future.
Code in a Finalize method can construct new objects. If this happens during CLR shutdown, the CLR continues collecting objects and calling their Finalize methods until no more objects exist or until the 40 seconds have elapsed.

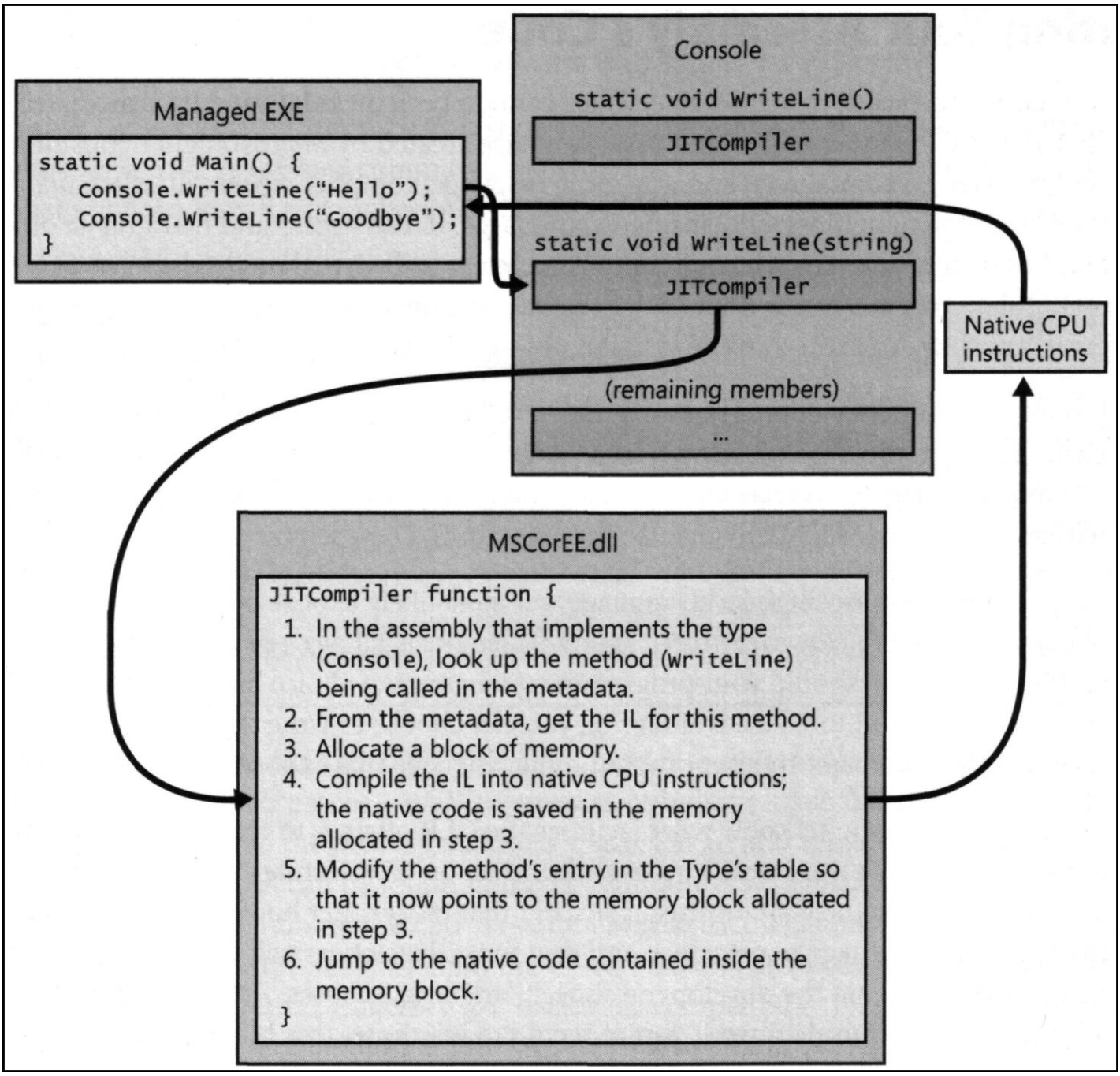
91. What is Sync Block Index and object pointer?

Every object on the heap requires some additional members—called the type object pointer and the sync block index—used by the CLR to manage the object. Sync block index has bit value while garbage collection's marking phase. Garbage collector mark alive objects, it sets sync block index value. While object pointer is used for tracking between object on heap and stack.

92. What is JIT compiler? What is its role?

To execute a method, its IL must first be converted to native CPU instructions. This is the job of the CLR's JIT (just-in-time) compiler.
Figure shows what happens the first time a method is called.

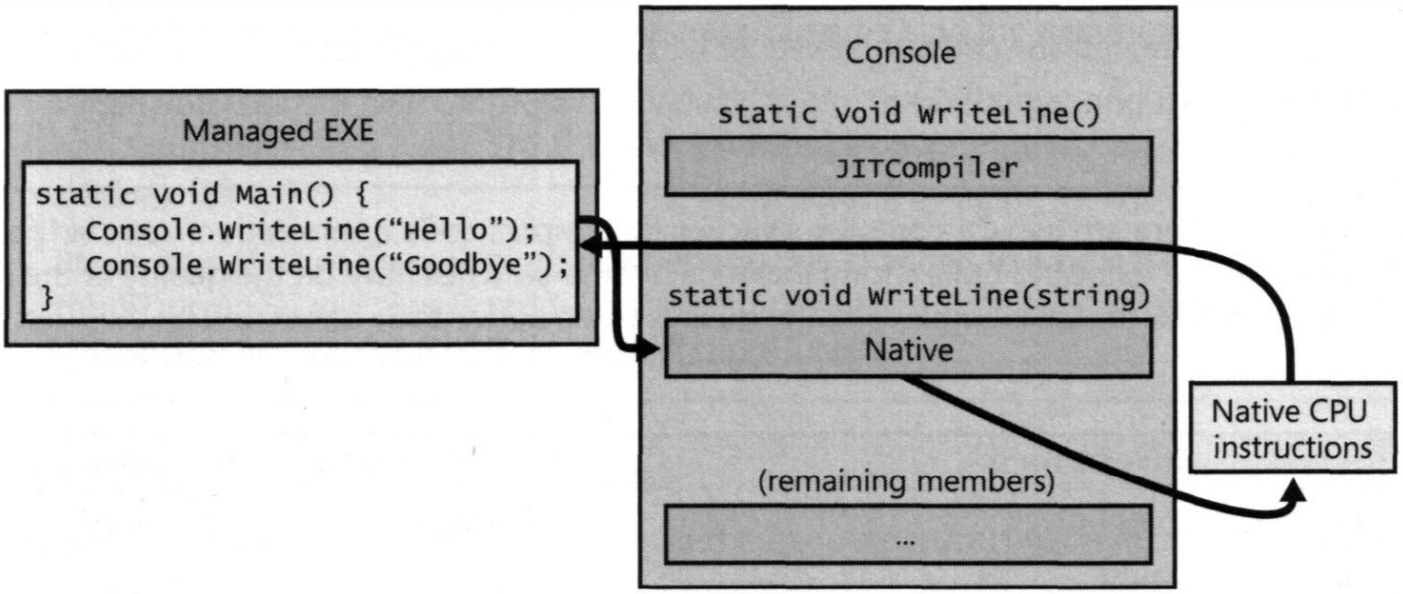
Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

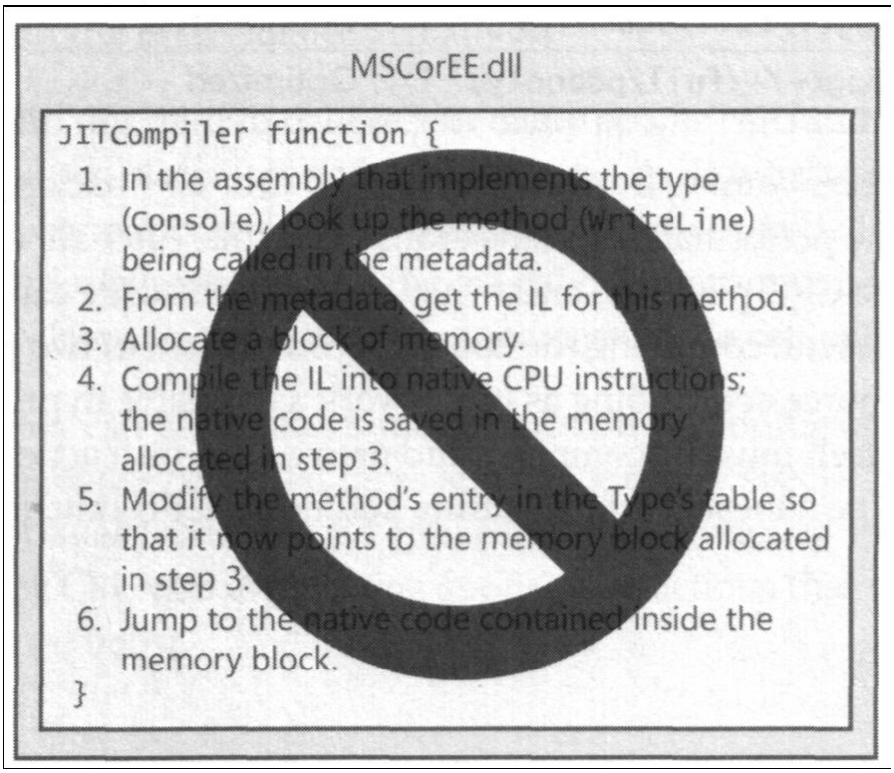


Just before the Main method executes, the CLR detects all of the types that are referenced by Main's code. This causes the CLR to allocate an internal data structure that is used to manage access to the referenced types. In Figure 1-4, the Main method refers to a single type, Console, causing the CLR to allocate a single internal structure. This internal data structure contains an entry for each method defined by the Console type. Each entry holds the address where the method's implementation can be found. When initializing this structure, the CLR sets each entry to an internal, Undocumented function contained inside the CLR itself. I call this function JITCompiler. When Main makes its first call to WriteLine, the JITCompiler function is called. The JITCompiler function is responsible for compiling a method's IL code into native CPU instructions. Because the IL is being compiled "just in time," this component of the CLR is frequently
referred to as a JITter or a JIT compiler. When called, the JITCompiler function knows what method is being called and what type defines this method. The JITCompiler function then searches the defining assembly's metadata for the called

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

method's IL. JITCompiler next verifies and compiles the IL code into native CPU instructions. The native CPU instructions are saved in a dynamically allocated block of memory. Then, JITCompiler goes back to the entry for the called method in the type's internal data structure created by the CLR and replaces the reference that called it in the first place with the address of the block of memory containing the native CPU instructions it just compiled. Finally, the JITCompiler function jumps to the code in the memory block. This code is the implementation of the WriteLine method (the version that takes a String parameter). When this code returns, it returns to the code in Main, which continues execution as normal. Main now calls WriteLine a second time. This time, the code for WriteLine has already been verified and compiled. So the call goes directly to the block of memory, skipping the JITCompiler function entirely. After the WriteLine method executes, it returns to Main.
Below Figure shows what the process looks like when WriteLine is called the second time.





Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

A performance hit is incurred only the first time a method is called. All subsequent calls to the method execute at the full speed of the native code because verification and compilation to native code don't need to be performed again. The JIT compiler stores the native CPU instructions in dynamic memory. This means that the compiled code is discarded when the application terminates. So if you run the application again in the future or if you run two instances of the application simultaneously (in two different operating system processes), the JIT compiler will have to compile the IL to native instructions again. For most applications, the performance hit incurred by JIT compiling isn't significant. Most applications tend to call the same methods over and over again. These methods will take the performance hit only once while the application executes. It's also likely that more time is spent inside the method than calling the method. You should also be aware that the CLR's JIT compiler optimizes the native code just as the back end of an unmanaged C++ compiler does. Again, it may take more time to produce the optimized code, but the code will execute with much better performance than if it hadn't been optimized.

Note There are two C# compiler switches that impact code optimization: /optimize and /debug. The following table shows the impact these switches have on the quality of the IL code generated by the C# compiler and the quality of the native code generated by the JIT compiler:

Compiler Switch Settings	**C# IL Code Quality**	**JIT Native Code Quality**
/optimize- /debug- (this is the default)	Unoptimized	Optimized
/optimize- /debug(+/full/pdbonly)	Unoptimized	Unoptimized
/optimize+ /debug (-/+/full /pdbonly)	Optimized	Optimized

For those developers coming from an unmanaged C or C++ background, you're probably thinking about the performance ramifications of all this. After all, unmanaged code is compiled for a specific CPU platform, and, when invoked, the code can simply execute. In this managed environment, compiling the code is accomplished in two phases. First, the compiler passes over the source code, doing as much work as possible in producing IL. But to execute the code, the IL itself must be compiled into native CPU instructions at run time, requiring more memory to be allocated and requiring additional CPU time to do the work.

Compiler Switch Settings C# IL Code Quality JIT Native Code Quality

If you too are skeptical, you should certainly build some applications and test the performance for yourself. In addition, you should run some nontrivial managed applications Microsoft or others have produced, and measure their performance. I think you'll be surprised at how good the performance actually is. You'll probably find this hard to believe, but many people (including me) think that managed applications could actually outperform unmanaged applications. There are many reasons to believe this. For example, when the JIT compiler compiles the IL code into native code at run time, the compiler knows more about the execution environment than an unmanaged compiler would know. Here are some ways that managed code can outperform unmanaged code:

- A JIT compiler can determine if the application is running on an Intel Pentium 4 CPU and produce native code that takes advantage of any special instructions offered by the Pentium 4. Usually, unmanaged applications are compiled for the lowest-common denominator CPU and avoid using special instructions that would give the application a performance boost.
- A JIT compiler can determine when a certain test is always false on the machine that it is running on. For example, consider a method that contains the following code:

```
if (numberOfCPUs > 1) {
...
}
```

- This code could cause the JIT compiler to not generate any CPU instructions if the host machine has only one CPU. In this case, the native code would be fine-tuned for the host machine; the resulting code is smaller and executes faster.
- The CLR could profile the code's execution and recompile the IL into native code while the application runs. The recompiled code could be reorganized to reduce incorrect branch predictions depending on the observed execution patterns.

These are only a few of the reasons why you should expect future managed code to execute better than today's unmanaged code. As I said, the performance is currently quite good for most applications, and it promises to improve as time goes on.

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

Note: *When producing unoptimized IL code, the C# compiler will emit NOP (no-operation) instructions into the code. The NOP instructions are emitted to enable the edit-and-continuefeature while debugging. These NOP instructions also make code easier to debug by allowing breakpoints to be set on control flow instructions such as for, while, do, if, else, try, catch, and finally statement blocks. This is supposed to be a feature, but I've actually found these NOP instructions to be annoying at times because I have to single-step over them, which can actually slow me down while trying to debug some code. When producing optimized IL code,*
the C# compiler will remove these NOP instructions. Be aware that when the JIT compiler produces optimized native code, the code will be much harder to single-step through in a debugger, and control flow will be optimized. Also, some function evaluations may not work when performed inside the debugger. When you create a new C# project in Visual Studio, the Debug configuration of the project
has /optimize- and /debug:full, and the Release configuration has /optimize+ and /debug:pdbonly. Regardless of all of these settings, if a debugger is attached to a process containing the CLR, the JIT compiler will always produce unoptimized native code to help debugging; but, of course, the performance of the code will be reduced.

93. How Types are stored in dot net?

Value Types DataTypes
Windows uses a system known as virtual addressing, in which the mapping from the memory address seen by your program to the actual location in hardware memory is entirely managed by Windows. Somewhere inside a process' virtual memory is an area known as the stack. The stack stores value data types that are not members of objects. In addition, when you call a method, the stack is used to hold a copy of any parameters passed to the method. In order to understand how the stack works, we need to understand the importance of variable scope in C#. It is always the case that if a variable a goes into scope before variable b, then b will go out of scope first.
We don't know exactly where in the address space the stack is—we don't need to know for C# development. A stack pointer (a variable maintained by the operating system) identifies the next free location on the stack. When your program first starts running, the stack pointer will point to just past the end of the block of memory that is reserved for the stack. The stack actually fills downward, from high memory addresses to low addresses. As data is put on the stack, the stack pointer is adjusted accordingly, so it always points to just past the next free location. This illustrated in Figure 7-1, which shows a stack pointer with a value of 800000 (0xC3500 in hex) and the next free location is the address

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

9999.

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

Disclai

Reference Data Types

While the stack gives very high performance, it is not flexible enough to be used for all variables. The requirement that the lifetimes of variables must be nested is too restrictive for many purposes. Often, you will want to use a method to allocate memory to store some data and be able to keep that data available long after that method has exited. This possibility exists whenever storage space is requested with the new operator—as is the case for all reference types. That's where the managed heap comes in.

The managed heap (or heap for short) is just another area of memory from the process's available 4GB. The following code demonstrates how the heap works and how memory is allocated for reference data types:

```
void DoWork()
{
Customer arabel;
arabel = new Customer();
Customer mrJones = new Nevermore60Customer();
}
```

In this code, we have assumed the existence of two classes, Customer and Nevermore60Customer. These classes are in fact taken from the Mortimer Phones examples in Appendix A (which is posted at www.wrox.com).

First, we declare a Customer reference called arabel. The space for this will be allocated on the stack, but remember that this is only a reference, not an actual Customer object. The arabel reference takes up 4 bytes, enough space to hold the address at which a Customer object will be stored. (We need 4 bytes to represent a memory address as an integer value between 0 and 4GB.)

Then we get to the next line:

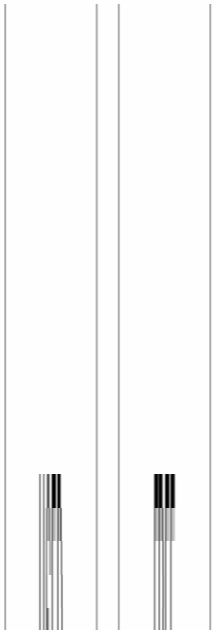
```
arabel = new Customer();
```

This line of code does several things. First, it allocates memory on the heap to store a Customer object (a real object, not just an address). Then, it sets the value of the variable arabel to the address of the memory it has allocated to the new Customer object. (It also calls the appropriate Customer() constructor to

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

initialize the fields in the class instance, but we won't worry about that here

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.



Disclaimer: So *es are from different sources and may be incomplete.*

94. Explain structure of GAC in windows?

The GAC -- Global Assembly Cache -- is a popular topic among managed code developers. It serves two main purposes: (A) as a catalogue of the globally publicly available assemblies available on the system, indexed by strong name, and (B) the location of those same files. It is not my intent to describe the GAC, however; look here for more info. My intent is to describe some interesting changes that have been made to the GAC for Whidbey and to promote some good practices. This is part I and is dedicated to processor architecture; we'll see how far we get with the series ;)
Whidbey is the first CLR release in which we support 64-bit execution. The CLR team has been working on porting the CLR to 64-bit for several years now and is now releasing it to the world with Whidbey. Woohoo! Not surprisingly, 64-bit has required a lot of changes all over the CLR, as it imports a bunch of new concepts onto us. My favourite is the general concept of processor architecture, of which there are four that we now support: X86, X64, IA64 and MSIL. We always supported the first and the last, but there was never a strong reason to differentiate them considerably as we only ran on a 32-bit system.
With respect to the v1.x GAC (v1.0 and Everett share the same GAC), you can see pretty clearly that processor architecture was not a design point.

V1.x GAC location
C:\Windows\assembly\GAC
V1.x System.dll location
C:\WINDOWS\assembly\GAC\System\1.0.5000.0__b77a5c561934e089\System.dll
Where would you have tagged the processor architecture? Nowhere that I can see. As a result, we changed the GAC structure in Whidbey to accommodate processor architecture, which is now an incredible important aspect of the GAC.
V2.0 GAC locations on an x86 machine
C:\Windows\assembly\GAC_MSIL
C:\Windows\assembly\GAC_32
V2.0 System.dll and mscorlib.dll locations on an x86 machine
C:\WINDOWS\assembly\GAC_MSIL\System\2.0.0.0__b77a5c561934e089\System.dll
C:\WINDOWS\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\mscorlib.dll
V2.0 GAC locations on an x64 or IA64 machine
C:\Windows\assembly\GAC_MSIL
C:\Windows\assembly\GAC_32
C:\Windows\assembly\GAC_64
V2.0 System.dll and mscorlib.dll locations on an x64 machine
C:\WINDOWS\assembly\GAC_MSIL\System\2.0.0.0__b77a5c561934e089\System.dll
C:\WINDOWS\assembly\GAC_32\mscorlib\2.0.0.0__b77a5c561934e089\mscorlib.dll
C:\WINDOWS\assembly\GAC_64\mscorlib\2.0.0.0__b77a5c561934e089\mscorlib.dll
You can now see that it is easy to tag assemblies with processor architecture. We've created three GACs to do just that! As you can see, the GAC_64 directory only shows up on 64-bit machines, which is hopefully quite intuitive.
I've run out of time to get any further in this post, but hopefully it gives you a good idea of how and why the GAC changed, at least with respect to processor architecture in Whidbey. I also hope I left you with the opinion that these changes were a good idea ;)
A question to ponder: If you call System.Reflection.Assembly.Load("System, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089") and there is both a copy in the bit-specific GAC and another in the bit-agnostic (MSIL) GAC, which do you get?

95. What is Advantage and disadvantage of GC?

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

GC Advantage/Disadvantage:

- Advantage: Minimal house-keeping overhead (just one freelist)
- Disadvantage: Every allocation request requires a walk thru the freelist, makes allocations slow
- Disadvantage: Heap fragmentation?there may not be a single contiguous block to accommodate a request for a large block

96. What is Reflection? Explain about reflection performance.

As you know, metadata is stored in a bunch of tables. When you build an assembly or a module, the compiler that you're using creates a type definition table, a field definition table, a method definition table, and so on. The System.Reflection namespace contains several types that allow you to write code that reflects over (or parses) these metadata tables. In effect, the types in this namespace offer an object model over the metadata contained in an assembly or a module.
Using these object model types, you can easily enumerate all of the types in a type definition metadata table. Then for each type, you can obtain its base type, the interfaces it implements, and the flags that are associated with the type.
In reality, very few applications will have the need to use the reflection types. Reflection is typically used by class libraries that need to understand a type's definition in order to provide some rich functionality. For example, the FCL's serialization mechanism uses reflection to determine what fields a type defines. The serialization formatter can then obtain the values of these fields and write them into a byte stream that is used for sending across the Internet, saving to a file, or copying to the clipboard. Similarly, Microsoft Visual Studio's designers use reflection to determine which properties should be shown to developers when laying out controls on their Web Forms or Windows Forms at design time.
Reflection is also used when an application needs to load a specific type from a specific assembly at run time to accomplish some task. For example, an application might ask the user to provide the name of an assembly and a type. The application could then explicitly load the assembly, construct an instance of the type, and call methods defined in the type. This usage is conceptually similar to calling Win32's LoadLibrary and GetProcAddress functions. Binding to types and calling methods in this way is frequently referred to as late binding. (Early binding is when the types and methods used by an application are determined at compile time.)

Reflection performance

- Reflection is an extremely powerful mechanism because it allows you to discover and use types and members at run time that you did not know about at compile time. This power does come with two main drawbacks:
- Reflection prevents type safety at compile time. Since reflection uses strings heavily, you lose type safety at compile time. For example, if you call Type.GetType("Jef"); to ask reflection to find a type called "Jef" in an assembly that has a type called "Jeff," the code compiles but produces an error at run time because you accidentally misspelled the type name passed as the argument.
- Reflection is slow. When using reflection, the names of types and their members are not known at compile time; you discover them at run time by using a string name to identify each type and member. This means that reflection is constantly performing string searches as the types in the System.Reflection namespace scan through an assembly's metadata. Often, the string searches are case-insensitive comparisons, which can slow this down even more.
- Invoking a member by using reflection will also hurt performance. When using reflection to invoke a method, you must first package the arguments into an array; internally, reflection must unpack these on to the thread's stack. Also, the CLR must check that the arguments are of the correct data type before invoking a method. Finally, the CLR ensures that the caller has the proper security permission to access the member being invoked.

For all of these reasons, it's best to avoid using reflection to access a member. If you're writing an application that will dynamically discover and construct type instances, you should take one of the following approaches:

- Have the types derive from a base type that is known at compile time. At run time, construct an instance of the derived type, place the reference in a variable that is of the base type (by way of a cast), and call virtual methods defined by the base type.

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

- Have the type implement an interface that is known at compile time. At run time, construct an instance of the type, place the reference in a variable that is of the interface type (by way of a cast), and call the methods defined by the interface. I prefer this technique over the base type technique because the base type technique doesn't allow the developer to choose the base type that works best in a particular situation.

97. What are members of a type?

Constants: A constant is a symbol that identifies a never-changing data value. These symbols are typically used to make code more readable and maintainable. Constants are always associated with a type, not an instance of a type. Logically, constants are always static members.
Fields: A field represents a read-only or read/write data value. A field can be static, in which case the field is considered part of the type's state. A field can also be instance (nonstatic), in which case it's considered part of an object's state. I strongly encourage you to make fields private so that the state of the type or object can't be corrupted by code outside of the defining type.
Instance constructors: An instance constructor is a special method used to initialize a new object's instance fields to a good initial state.
Type constructors: A type constructor is a special method used to initialize a type's static fields to a good initial state.
Methods: A method is a function that performs operations that change or query the state of a type (static method) or an object (instance method). Methods typically read and write to the fields of the type or object.
Operator overloads: An operator overload is a method that defines how an object should be manipulated when certain operators are applied to the object. Because not all programming languages support operator overloading, operator overload methods are not part of the Common Language Specification (CLS).
Conversion operators: A conversion operator is a method that defines how to implicitly or explicitly cast or convert an object from one type to another type. As with operator overload methods, not all programming languages support conversion operators, so they're not part of the CLS.
Properties: A property is a mechanism that allows a simple, field-like syntax for setting or querying part of the logical state of a type (static property) or object (instance property) while ensuring that the state doesn't become corrupt. Properties can be parameterless (very common) or parameterful (fairly uncommon but used frequently with collection classes).
Events: A static event is a mechanism that allows a type to send a notification to listening types or listening objects. An instance (nonstatic) event is a mechanism that allows an object to send a notification to listening types or listening objects. Events are usually raised in response to a state change occurring in the type or object offering the event. An event consists of two methods that allow types or objects ("listeners") to register and unregister interest in the event. In addition to the two methods, events typically use a delegate field to maintain the set of registered listeners.
Types: A type can define other types nested within it. This approach is typically used to break a large, complex type down into smaller building blocks to simplify the implementation.

98. Explain bridge between ISAPI and Application Domains?

ASP.NET inner workings are sometimes seen as black magic occurring behind the scenes, providing developers with a high level abstraction which allows them to concentrate on the logic of the application, instead of having to deal with low level mechanisms of the HTTP protocol. ASP.NET is built for and with the .NET framework, thus mostly in managed code. The entry point for web requests, however, are nowadays web servers written in native code, therefore a communication mechanism is in order. In this article I'm going to describe the bridge between

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

managed and unmanaged worlds and how the two collaborate to set up the processing environment needed by ASP.NET requests to be processed.

Introduction

In the previous as well as first article of this series I've introduced the first steps performed by a generic web request once accepted by the web server, and what route it takes when it is identified as an ASP.NET resource request. You've seen how different releases of IIS deal with incoming ASP.NET-related requests until they finally dispatch them to an unmanaged Win32 component called aspnet_isapi.dll, which acts as a bridge between the web server and the managed ASP.NET infrastructure.
In this article I'm going to resume the discussion exactly where I left it and start delving into the managed part of the processing environment I've presented in the previous article as those black boxes labeled ASP.NET Http Runtime Environment.
Note: The ASP.NET Http Runtime Environment is sometimes also referred to as the ASP.NET Pipeline, Http Runtime Pipeline or a mix of these words.

Leaving the aspnet_isapi.dll extension for the managed world

In the previous article I've explained how IIS 5 and IIS 6 manage ASP.NET requests. Regardless of how they deal with worker processes generation, management and recycling, all the information pertaining to the request is in the end forwarded to the aspnet_isapi.dll extension.
This is the bridge between the unmanaged and managed world and is the least documented part of all the ASP.NET architecture.
Note: At the moment of the writing of this article IIS 7 has just been released with a Go Live license along with Longhorn Server Beta 3. A lot of things will change with IIS 7, and although these first articles have been focusing solely on previous versions of IIS, a future article will be dedicated to changes and improvements introduced by IIS 7.
As for the lack of documentation available on most parts of this topic, what I'm going to explain may not be completely correct, especially for what concerns the unmanaged parts of the infrastructure. However, my considerations on the undocumented topics are based on some tools which helped a lot in understanding and making sense of the inner workings of the framework:

- Lutz Roeder's .NET Reflector, to analyze statically and decompile the code of .NET managed classes.
- Microsoft CLR Profiler, to analyze the dynamic behavior of the framework by looking at method calls, memory allocation, class instantiation and a whole big number of other features it provides.
- JetBrains dotTrace Profiler, a commercial profiler for .NET Applications. A free time limited version is available too.
- Red Gate ANTS Profiler, another commercial profiler for .NET applications. A free time limited version is available.

Going back to the bridge between the unmanaged and managed worlds, once the CLR has been loaded - either by the ISAPI Extension in case of IIS 6 process model or by the worker process in case of IIS 5 process model - the black magic occurs through a bunch of unmanaged COM interfaces calls that the ASP.NET ISAPI component makes to a couple of managed classes contained in the System.Web.Hosting namespace, the AppManagerAppDomainFactory class and the ISAPIRuntime class, which expose some methods via COM-callable interfaces.
Note: The Common Language Runtime - CLR - represents the execution environment of every .NET application. It's what actually provides both environment and services for running managed applications. It has to be hosted inside a Win32 process and ASP.NET is one of the available hosts for the CLR provided by the .NET framework. More specifically, the ASP.NET worker process (aspnet_wp.exe in IIS 5 and w3wp.exe in IIS 6) is the process who hosts the CLR in ASP.NET.
Before delving into the technical details of the interactions occurring between these classes let's see on the surface what actually happens for a request to be processed. I have introduced the previous two classes because the entry points in the processing of a request can be roughly grouped into two categories:

1. Setup of an AppDomain, in case one doesn't exist yet, to represent the application to which the request is targeted - this is accomplished by the AppManagerAppDomain class.

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

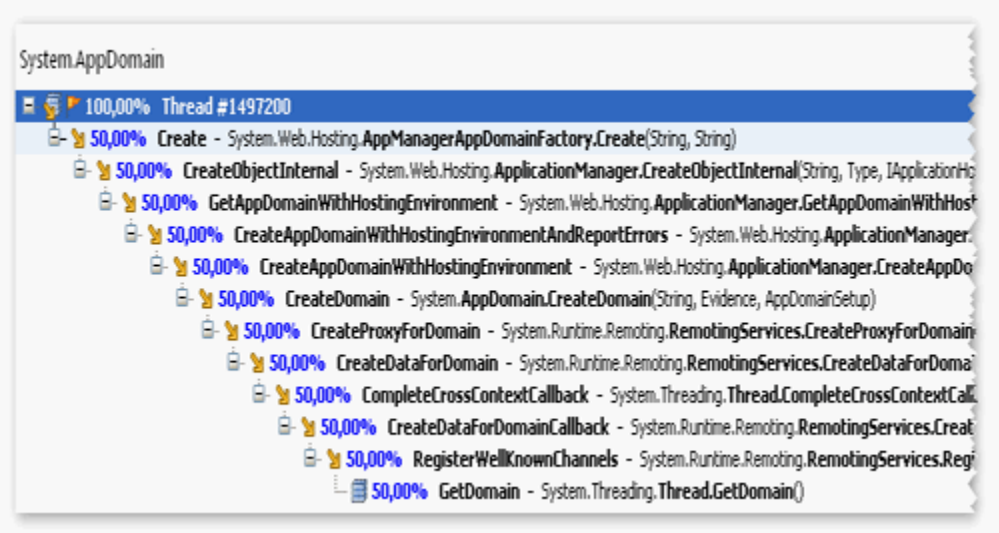
2. Handling and processing of the request - accomplished by the ISAPIRuntime class.

Though the two categories are equally important, the first consists of interactions which are supposed to happen without the interference of the developer and which concern mostly the hosting environment of the application, while the second is the most configurable part of the infrastructure, which I will delve into completely in the following article of this series.
Under another point of view, the first category comprises operations performed only once during the lifecycle of an application, that is, at startup, while the second consists of interactions occurring at each request targeting that specific application.

Setting up an AppDomain

As seen in the previous article, an ASP.NET application is reserved and wrapped into an entity called Application Domain, aka AppDomain, which turns out to be represented by a class of the ASP.NET architecture, the AppDomain class. When a request to a particular application arrives, an AppDomain has to be setup, if it doesn't exist. This usually happens if the incoming request is the first one targeting that particular application, or if for some reasons the corresponding AppDomain has been shut down, which may happen for several reasons that I will talk about later. Note that only one AppDomain exists for a single ASP.NET application, which is itself mapped one-to-one with an IIS application - either created over a physical/virtual directory. So how does an instance of this class get created?

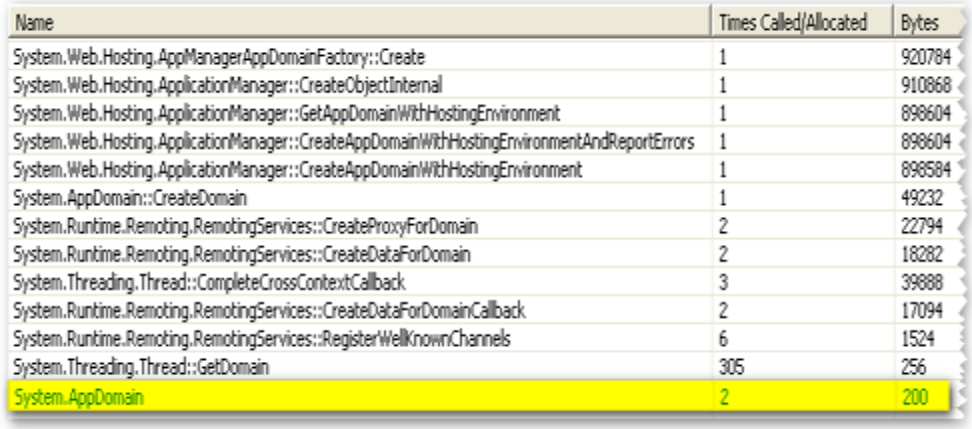
Figure 1: Call stack generated by the creation of an AppDomain instance as shown by JetBrains dotTrace Profiler



Using Reflector it's possible to realize that the AppDomain class has a private constructor which throws a NotSupportedException exception. Therefore this is obviously not the way to go. The entry point for the instantiation of an AppDomain is in fact the AppManagerAppDomainFactory.Create method. To find it out it's necessary to use a profiler which keeps track of the call stack generated when instantiating the object. Figure 1 displays a screenshot of JetBrains dotTrace Profiler, which shows the call stack generated to instantiate the AppDomain of a web application hosted on IIS. A lot of classes participate in the process, but it's something a developer will probably never have to put his hands on.
Note that the AppManagerAppDomainFactory class is instantiated only once during and by the CLR initialization process. As its name suggests, it's used as the entry point for creating AppDomains as well as other critical objects.
Figure 2 shows the same call stack as the previous image, this time captured from the output of the Call Tree view of Microsoft CLR Profiler. This provides some more information about the instantiation of the AppDomain. Actually, the highlighted line shows that there are two instances of the AppDomain class created by the principal thread.

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

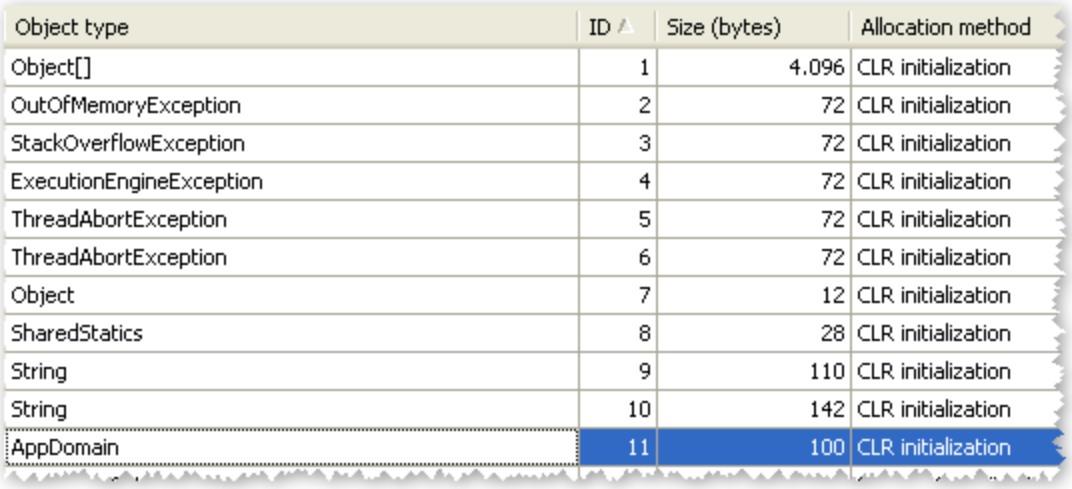
Figure 2: Call stack generated by the creation of an AppDomain instance as shown by the Call Tree view of Microsoft CLR Profiler



Name	Times Called/Allocated	Bytes
System.Web.Hosting.AppManagerAppDomainFactory::Create	1	920784
System.Web.Hosting.ApplicationManager::CreateObjectInternal	1	910868
System.Web.Hosting.ApplicationManager::GetAppDomainWithHostingEnvironment	1	898604
System.Web.Hosting.ApplicationManager::CreateAppDomainWithHostingEnvironmentAndReportErrors	1	898604
System.Web.Hosting.ApplicationManager::CreateAppDomainWithHostingEnvironment	1	898584
System.AppDomain::CreateDomain	1	49232
System.Runtime.Remoting.RemotingServices::CreateProxyForDomain	2	22794
System.Runtime.Remoting.RemotingServices::CreateDataForDomain	2	18282
System.Threading.Thread::CompleteCrossContextCallback	3	39888
System.Runtime.Remoting.RemotingServices::CreateDataForDomainCallback	2	17094
System.Runtime.Remoting.RemotingServices::RegisterWellKnownChannels	6	1524
System.Threading.Thread::GetDomain	305	256
System.AppDomain	2	200

So, how does this additional instance get created, and why? Unluckily it's not an easy question. The call stack shown in the previous images represents the steps followed for instantiating the AppDomain which corresponds to the actual application being executed, so this additional instance looks like a helper object used for some purpose which is difficult to understand - as far as I can guess, it's used to host all the objects which don't belong to a specific application, and therefore are hosted in an isolated AppDomain, like the AppManagerAppDomainFactory class. As for the AppManagerAppDomainFactory class, this helper AppDomain is instantiated only once during and by the CLR initialization, and very very early during this stage, as shown in Figure 3.

Figure 3: Order and allocation method of the additional AppDomain instance



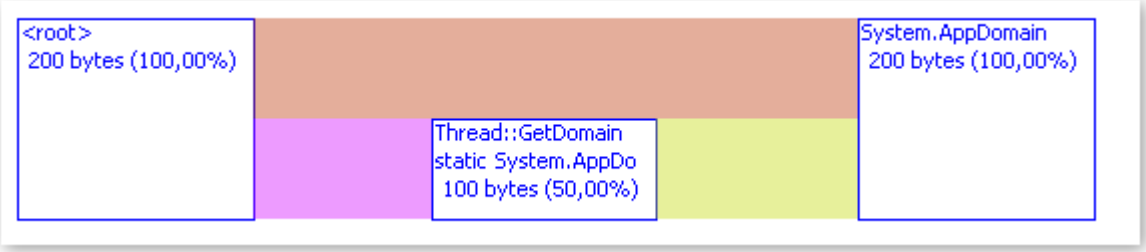
Object type	ID	Size (bytes)	Allocation method
Object[]	1	4.096	CLR initialization
OutOfMemoryException	2	72	CLR initialization
StackOverflowException	3	72	CLR initialization
ExecutionEngineException	4	72	CLR initialization
ThreadAbortException	5	72	CLR initialization
ThreadAbortException	6	72	CLR initialization
Object	7	12	CLR initialization
SharedStatics	8	28	CLR initialization
String	9	110	CLR initialization
String	10	142	CLR initialization
AppDomain	11	100	CLR initialization

Figure 3 represents a screenshot of Red Gate ANTS Profiler. It shows that the additional AppDomain instance is created very early and during the initialization of the CLR. It is obviously created earlier than the AppManagerAppDomainFactory class, since it is probably its container. The ID of the singleton AppManagerAppDomainFactory instance, in fact, in the profiling session of the previous figure turns out to be 52.

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

Another way to see the instantiation process of the two AppDomains - which I present for completeness - is the Allocation Graph view provided by Microsoft CLR Profiler. It creates a graphical flow representation of the classes taking place in the process, as shown in Figure 4.

Figure 4: Instantiation of two AppDomains shown with the Allocation Graph view of Microsoft CLR Profiler



This represents a compressed view which cuts out all the classes between the <root> element - the CLR - and the Thread class. However, it shows clearly the two routes followed when creating the two AppDomain instances. Another information about AppDomain instances that can be obtained from the previous figures is that each instance occupies exactly 100 bytes in memory. Not that it matters much, but one might think that an AppDomain is a huge class since it has to host an entire application. Actually, it provides a lot of services but doesn't store a lot of data.

By now, the helper AppDomain instance has been created, as well as an instance of the AppManagerAppDomainFactory class, whose Create method is the entry point of the call stack shown in figures 1 and 2.

This method is exposed and called via COM, thus unmanaged code. The AppManagerAppDomainFactory class implements the IAppManagerAppDomainFactory interface, whose structure is represented in Figure 5.

Figure 5: Structure of the IAppManagerAppDomainFactory interface

```
[ComImport, InterfaceType(ComInterfaceType.InterfaceIsIUnknown), Guid("02998279-7175-4d59-aa5a-fb8e44d4ca9d"), AspNet
public interface IAppManagerAppDomainFactory
{
    [return: MarshalAs(UnmanagedType.Interface)]
    object Create([In, MarshalAs(UnmanagedType.BStr)] string appId, [In, MarshalAs(UnmanagedType.BStr)] string appPath);
    void Stop();
}
```

The interface is decorated with the ComImport and InterfaceType attribute, which bind the type to an unmanaged interface type. When this method is called it produces the call stack of figures 1 and 2 which in the end triggers the creation of an instance of the AppDomain class, the one used to host the web application targeted by the request.

Once the AppDomain is up and running all that's left to do - and it's much more than what's been done so far - is processing the request. This is probably the most interesting part of the ASP.NET architecture, but before approaching the fun part I wanted to introduce some core topics.

Summary

In this article I've introduced some very low level topics about the ASP.NET infrastructure, those concerning the interface between the unmanaged world - represented by the ASP.NET ISAPI extension - and the managed world, represented by the AppDomain hosting the web application. I've presented the mechanisms by which an AppDomain gets instantiated and which classes take part in the process. In the next article I will discuss of managed code only by presenting the HTTP Pipeline, which is in my opinion the most attractive chapter of the ASP.NET architecture, and what actually makes ASP.NET different from all the other web development

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

frameworks out there. I hope you enjoyed reading this article as much as I did writing it. My advice is to fire up some profiler and try out this stuff by yourself, which is the best way to fully understand how it all works.

99. How Securely Implement Request Processing, Filtering, and Content Redirection with HTTP Pipelines in ASP.NET?

Most people think of ASP.NET in terms of pages—that is, executable templates for creating HTML to return to browsers. But that is just one of many possible ways to use the ASP.NET core infrastructure, the HTTP pipeline. The pipeline is the general-purpose framework for server-side HTTP programming that serves as the foundation for ASP.NET pages as well as Web Services. To qualify as a serious ASP.NET developer, you must understand how the pipeline works. This article explains how the HTTP pipeline processes requests.

The Pipeline Object Model

The types defined in the System.Web namespace process HTTP requests using a pipeline model. The general structure of the pipeline is shown in Figure 1. HTTP requests are passed to an instance of the HttpRuntime class, which represents the beginning of the pipe. The HttpRuntime object examines the request and figures out which application it was sent to (from the pipeline's perspective, a virtual directory is an application). Then it uses an HttpApplicationFactory to either find or create an HttpApplication object to process the request. An HttpApplication holds a collection of HTTP module objects, implementations of the IHttpModule interface. HTTP modules are filters that can examine and modify the contents of HTTP request and response messages as they pass through the pipeline. The HttpApplication object uses an HTTP handler factory to either find or create an HTTP handler object. HTTP handlers are endpoints for HTTP communication that process request messages and generate corresponding response messages. HTTP handlers and handler factories implement the IHttpHandler and IHttpHandlerFactory interfaces, respectively.

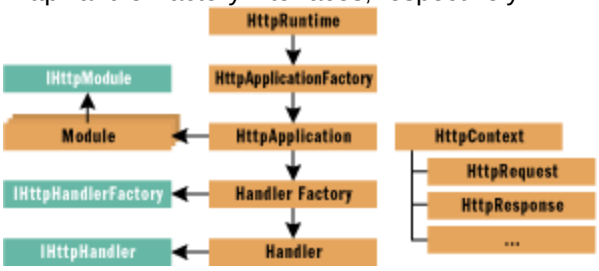


Figure 1 HTTP Pipeline Processing

An HttpApplication, its modules, and its handler will only be used to process one request at a time. If multiple requests targeting the same application arrive simultaneously, multiple HttpApplication objects will be used. The HttpApplicationFactory and HttpHandlerFactory classes pool HttpApplication objects and HTTP handler objects, respectively, for efficiency's sake.

The pipeline uses an HttpContext object to represent each request/response pair. The object is passed to the HttpApplication, which in turn passes it to the handler. Each module can access the current HttpContext as well. The HttpContext object exposes properties representing the HTTP request and response messages, which are instances of the HttpRequest and HttpResponse classes, respectively. The HttpContext object also exposes properties representing security information and per-call, per-session, and per-application state. Figure 2 shows most of the interesting properties of the HttpContext class.

The ASP.NET HTTP pipeline is fully extensible. You can implement your own HTTP modules, handlers, and handler factories. You can also extend the behavior of the HttpApplication class.

The Pipeline Process Model

The ASP.NET HTTP pipeline relies on Microsoft® Internet Information Services (IIS) to receive the requests it is going to process (it can also be integrated with other Web servers). When IIS receives an HTTP request, it

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

examines the extension of the file identified by the target URL. If the file extension is associated with executable code, IIS invokes that code in order to process the request. Mappings from file extensions to pieces of executable code are recorded in the IIS metabase. When ASP.NET is installed, it adds entries to the metabase associating various standard file extensions, including .aspx and .asmx, with a library called aspnet_isapi.dll.
When IIS receives an HTTP request for one of these files, it invokes the code in aspnet_isapi.dll, which in turn funnels the request into the HTTP pipeline. Aspnet_isapi.dll uses a named pipe to forward the request from the IIS service where it runs, inetinfo.exe, to an instance of the ASP.NET worker process, aspnet_wp.exe. (In Windows® .NET Server, ASP.NET integrates with the IIS 6.0 kernel-mode HTTP listener, allowing requests to pass from the operating system directly to the worker process without passing through inetinfo.exe.) The worker process uses an instance of the HttpRuntime class to process the request. Figure 3 illustrates the entire architecture.

Figure 3 ASP.NET Pipeline Architecture

The HTTP pipeline always processes requests in an instance of the worker processes. By default, there will only be one worker process in use at a time. (If your Web server has multiple CPUs, you can configure the pipeline to use multiple worker processes, one per CPU.) This is a notable change from native IIS, which uses multiple worker processes in order to isolate applications from one another. The pipeline's worker process achieves isolation by using AppDomains. You can think of an AppDomain as a lightweight process within a process. The pipeline sends all HTTP requests targeting the same virtual directory to a single AppDomain. In other words, each virtual directory is treated as a separate application. This is another notable change from native IIS, which allowed multiple virtual directories to be part of the same application.
ASP.NET supports recycling worker processes based on a number of criteria, including age, time spent idle, number of requests serviced, number of requests queued, and amount of physical memory consumed. The global .NET configuration file, machine.config, sets thresholds for these values (see the processModel element). When an instance of aspnet_wp.exe crosses one of these thresholds, aspnet_isapi.dll launches a new instance of the worker process and starts sending it requests. The old instance terminates when it finishes processing pending requests. Recycling of worker processes promotes reliability by killing off processes before their performance begins to degrade from resource leaks or other runtime phenomena.

HTTP Handlers

HTTP handlers are simply classes that implement the IHttpHandler interface, as shown in the following lines of code:

```
  interface IHttpHandler
{
 // called to process request and generate response
 void ProcessRequest(HttpContext ctx);
 // called to see if handler can be pooled
 bool IsReuseable { get; }
}
```

Handlers can also implement the IHttpAsyncHandler interface if they want to support asynchronous invocation. The ProcessRequest method is called by an HttpApplication object when it wants the handler to process the current HTTP request and to generate a response. The IsReuseable property is accessed in order to determine whether a handler can be used more than once.
The code in Figure 4 implements a simple reusable HTTP handler that responds to all requests by returning the current time in an XML tag. You should note the use of the HttpContext object's Response property to set the response message's MIME type and to write out its content.
Once an HTTP handler class is implemented, it must be deployed. Deployment involves three steps. First, you have to put the compiled code someplace where the ASP.NET worker process can find it. In general, that means you place your compiled .NET assembly (typically a DLL) in the bin subdirectory of your Web server's virtual directory or in the Global Assembly Cache (GAC).
Next, you have to tell the HTTP pipeline to execute your code when an HTTP request that meets some basic criteria arrives. You do this by adding an <httpHandlers> section to your virtual directory's Web.config file:

```
<configuration>
```


Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

```
 <system.web>
  <httpHandlers>
   <add verb="GET" path="*.time"
     type="Pipeline.TimeHandler,
     Pipeline"
   />
  </httpHandlers>
 </system.web>
</configuration>
```

This information is treated as an addendum to the configuration details specified in the global .NET machine.config file. In this example, the Web.config file tells the ASP.NET HTTP pipeline to process HTTP GET requests for .time files by invoking the Pipeline.TimeHandler class in the Pipeline assembly.
Finally, you have to tell IIS to route requests for .time files to the aspnet_isapi.dll library so that they can be funneled into the pipeline in the first place. This requires adding a new file mapping to the IIS metabase. The easiest way to do this is using the IIS management console, which shows a virtual directory's file extension mappings on the Mappings tab of the Application Configuration dialog (see Figure 5).

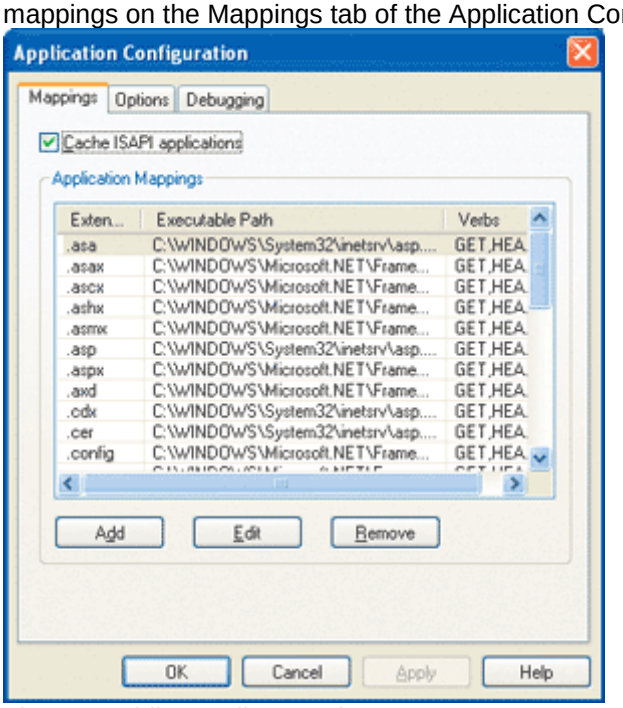


Figure 5 Adding a File Mapping

In addition to implementing custom handlers, you can also write your own handler factories. A handler factory is a simple class that implements the IHttpHandlerFactory interface. Handler factories are deployed the same way handlers are; the only difference is that the entry in the Web.config file refers the factory class instead of the handler class that the factory instantiates. If you implement a custom HTTP handler without implementing a handler factory, an instance of the pipeline-provided default factory class, HandlerFactoryWrapper, is used instead.

Standard Handlers

The higher-level ASP.NET technologies, such as pages and Web Services, are built directly on top of the HTTP handlers. A quick peek at the global .NET machine.config file reveals the following <httpHandlers> entries:

```
<httpHandlers>
 <add verb="*" path="*.ashx"
  type="System.Web.UI.SimpleHandlerFactory"
 />
 <add verb="*" path="*.aspx"
    type="System.Web.UI.PageHandlerFactory"
   />
```


Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

```
  <add verb="*" path="*.asmx"
   type="System.Web.Services.Protocols.
   WebServiceHandlerFactory ... "
  />
 </httpHandlers>
```

The first entry maps the .ashx file extension to the SimpleHandlerFactory class, an HTTP handler factory that knows how to compile and instantiate an implementation of IHttpHandler from the source code in an .ashx file. The resulting object can be used directly by the HTTP pipeline.
Figure 6 shows the TimeHandler example rewritten as an .ashx file. The @WebHandler directive tells the SimpleHandlerFactory the name of the HTTP handler class to instantiate after the source code has been compiled. The benefit of this approach is that it simplifies deployment: all you need to do is copy the .ashx file to your virtual directory. There is no need to create or modify a Web.config file or to update IIS—the requisite configuration was done when .NET was installed.
The second <httpHandlers> entry maps the .aspx file extension to the PageHandlerFactory class, which is an HTTP handler factory that knows how to compile the source code in an .aspx file into a System.Web.UI.Page-derived class and instantiate it. The Page class implements the IHttpHandler interface, so the resulting object can be used directly by the HTTP pipeline.
The third entry maps the .asmx file extension to the WebServiceHandlerFactory class, which is an HTTP handler factory that knows how to compile the source code in an .asmx file into a class and instantiate it. Then it wraps the object with an instance of a standard HTTP handler (SyncSessionlessHandler by default) that uses reflection in order to translate SOAP messages into method invocations. Once again, the resulting object can be used directly by the HTTP pipeline.
It is important to note that the PageHandlerFactory, WebServiceHandlerFactory, and SimpleHandlerFactory classes do not compile .aspx, .asmx, and .ashx files on every request. Instead, the compiled code is cached in the Temporary ASP.NET Files subdirectory of the .NET installation directory. The code will only be recompiled when its corresponding source file changes.

HTTP Modules

HTTP handlers are endpoints for communication. Instances of handler classes consume HTTP requests and produce HTTP responses. HTTP modules are filters that process HTTP request and response messages as they pass through the pipeline, examining and possibly modifying their content. The pipeline uses HTTP modules to implement its own infrastructure, most notably security and session management.
HTTP modules are simply classes that implement the IHttpModule interface:

```
interface IHttpModule
{
  // called to attach module to app events
  void Init(HttpApplication app);
  // called to clean up
  void Dispose()
}
```

The Init method is called by an HttpApplication object when the module is first created. It gives the module the opportunity to attach one or more event handlers to the events exposed by the HttpApplication object.
The code in Figure 7 implements a simple HTTP module that handles its HttpApplication object's BeginRequest and EndRequest events and measures the elapsed time between them. In this example, the Init method uses normal .NET techniques to attach the module's OnBeginRequest and OnEndRequest handlers to the events fired by the HttpApplication object the module is being attached to. The implementation of OnBeginRequest simply stores the time when the event fired in the member variable starts. The implementation of OnEndRequest measures the elapsed time since OnBeginRequest was called and adds that information to the response message using a custom HTTP header. The OnEndRequest method takes advantage of the fact that the first parameter passed to an event handler is a reference to the object that fired the event; in this case, it is the HttpApplication object that the module is attached to. The HttpApplication object exposes the Http-Context object for the current message exchange as a property, which is exactly how OnEndRequest is able to manipulate the HTTP response message.

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

Once an HTTP module class is implemented, it must be deployed. Deployment involves two steps. As with an HTTP handler, you have to put the compiled module code either in the bin subdirectory of your Web server's virtual directory or in the GAC so that the ASP.NET worker process can find it.
Then you have to tell the HTTP pipeline to create your module whenever a new HttpApplication object is created to handle a request sent to your application. You do this by adding an <httpModules> section to your virtual directory's Web.config file, as shown here:

```
<configuration>
 <system.web>
  <httpModules>
   <add
    name="Elapsed"
    type="Pipeline.ElapsedTimeModule, Pipeline"
   />
  </httpModules>
 </system.web>
</configuration>
```

In this example, the Web.config file tells the ASP.NET HTTP pipeline to attach an instance of the Pipeline.ElapsedTimeModule class to every HttpApplication object instantiated to service requests that target this virtual directory.

The Pipeline Event Model

The ElapsedTimeModule in the previous example implements two event handlers, OnBeginRequest and OnEndRequest. These are just two of the events that an HttpApplication object fires in the course of processing an HTTP message exchange. The complete list of events is shown in Figure 8. Note that the HTTP handler object that an instance of the HttpApplication class uses to ultimately process a request message is created between the ResolveRequestCache and AcquireRequestState events. The user session state, if any, is acquired during the AcquireRequestState event. Finally, note that the handler is invoked between the PreRequestHandlerExecute and PostRequestHandlerExecute events.
The HttpApplication class exposes all these events using multicast delegates so that multiple HTTP modules can register for each one. HTTP modules can register event handlers for as many of their HttpApplication objects' events as they like. However, modules should register for as few events as possible for efficiency's sake. Since an HttpApplication object and its modules will only be used to process one HTTP request at a time, individual HTTP module objects can store any per-request state they need across multiple events.
In some cases, an HTTP module may want to influence the flow of processing in the pipeline. For example, a module that implements a security scheme might want to abort normal message processing and redirect the client to a login URL when it detects that the HTTP request message does not include a cookie identifying the user. The HttpApplication object exposes the CompleteRequest method. If an HTTP module's event handler calls HttpApplication.CompleteRequest, normal pipeline processing is interrupted after the current event completes (including the processing of any other registered event handlers). A module that terminates the normal processing of a message is expected to generate an appropriate HTTP response message.
The code in Figure 9 provides an example of a module that uses CompleteRequest to abort the normal processing of a Web Service invocation. The SOAP specification's HTTP binding requires that an HTTP message carrying a SOAP message include a custom header called SOAPAction. The EnableWebServiceModule class's OnBeginRequest event handler examines the request message and if a SOAPAction header is present and the class's static enabled field is false, it stops further processing.
Figure 10 contains the source code for an HTTP handler called EnableWebServicesHandler that toggles the EnableWebServiceModule class's static enabled field whenever its ProcessRequest method is invoked.
Assuming that the source code for both the HTTP module and handler are compiled into a .NET assembly called Pipeline, the following entries in the Web.config file would be necessary for configuration:

```
<configuration>
 <system.web>
  <httpModules>
   <add name="WebServicesEnabledModule"
    type="Pipeline.EnableWebServicesModule, Pipeline"
   />
```


Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

```
  </httpModules>
  <httpHandlers>
   <add verb="*" path="toggle.switch"
    type="Pipeline.EnableWebServicesHandler, Pipeline"
  </httpHandlers>
 </system.web>
</configuration>
```

An IIS metabase entry associating the .switch extension with aspnet_isapi.dll would also have to be created and the Pipeline assembly would have to be deployed in the bin subdirectory of the Web server's virtual directory or in the GAC.

There is one other important point to mention about HTTP modules and the HttpApplication.CompleteRequest method. If a module aborts normal message handling during an event handler by calling CompleteRequest, the ASP.NET HTTP pipeline interrupts processing after that event completes. However, EndRequest and the events that follow are still fired. Thus, any modules that acquired valuable resources before processing was terminated have a chance to clean up those resources. For instance, if a module acquired a lock against shared state in its BeginRequest event handler, it can release the lock in its EndRequest event handler and be confident that the right thing will happen. The EndRequest event will fire even if some other module calls CompleteRequest and the HTTP message is never delivered to a handler.

HTTP Applications

As we mentioned, the ASP.NET HTTP pipeline treats each virtual directory as an application. When a request for a URL in a given virtual directory arrives, the HttpRuntime object that dispatches the message uses an HttpApplicationFactory object to find or create an HttpApplication object to process the request. A given HttpApplication object will only be used to service requests sent to a single virtual directory, and there can be multiple pooled instances of HttpApplication for the same virtual directory.

If you want, you can customize the behavior of the HttpApplication class for your application (virtual directory). You do this by writing a global.asax file. If the HTTP pipeline detects a global.asax file in your virtual directory, it compiles it into an HttpApplication-derived class. Then it instantiates your specialized HttpApplication subclass and uses it to service requests.

The most interesting use of a global.asax file is to implement a subclass of HttpApplication that handles events fired by the HTTP pipeline. These events fall into two categories, one of which we've already discussed. First, you can use a global.asax file to implement handlers for events fired by an HttpApplication object itself (the events listed in Figure 8). Normally HTTP modules handle these events, but in some cases implementing and deploying modules is not necessary—especially if the behavior you want to implement is application-specific.

For instance, you could implement a handler for the BeginRequest event in a global.asax file, as shown here:

```
<%@ import namespace="System.Web" %>

<!-- this code will be added to a new
     HttpApplication-derived class -->
<script language="C#" runat=server>

public void Application_BeginRequest(
          object obj, EventArgs ea)
{
  string s = Context.Request.Headers["SOAPAction"];
  Context.Items["IsSOAP"] = (s != null);
}

</script>
```

In this example, the Application_BeginRequest event handler detects the presence of a Web Service invocation based on the SOAPAction header. It records the fact that a SOAP invocation is being made in the Items property of the current HttpContext object (which is available as a property of the HttpApplication class, from which this code derives). The Items property is used to store a per-request state that you want to make available to multiple modules and the ultimate message handler. Its contents are flushed when a request completes. An HTTP module could provide this behavior, but if you only need it in one application it is simpler to implement it this way. (It might

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

seem odd to have an HttpApplication-derived class handling events fired by its base class, which is what is happening here, but it is very convenient!)
Two additional application-level events are not listed in Figure 8 and are not made available to HTTP modules in the normal way, namely, Application_OnStart and Application_OnEnd. These events are familiar to classic ASP programmers. They are called when an application is first accessed and when it shuts down, respectively. Here is a simple example:

```
<%@ import namespace="System.Web" %>

<script language="C#" runat=server>

public void Application_OnStart()
{
  ... // set up application here
}

public void Application_OnEnd()
{
  ... // clean up application here
}

</script>
```

The other category of events that an HttpApplication-derived class might handle is events fired by HTTP modules. In fact, this is how the pipeline implements the classic ASP Session_OnStart and Session_OnEnd events, both of which are fired by the SessionStateModule class.
Consider the EnableWebServicesModule presented earlier that conditionally rejects Web Service invocations based on the state of a static field. When it rejects a request, it does so brusquely, with a hardcoded, somewhat curt message. It might be better if the module allowed the application it is being used with to tailor the message for its own purposes. One way to do this is to have the HTTP module fire an event when a Web Service request is rejected. Figure 11 shows a new version of the EnableWebServicesModule that fires a Rejection event when a Web Service request is rejected. The modified implementation of the module's OnBeginRequest event handler checks to see if there are any handlers registered for the Rejection event by comparing the property to null. If one is registered, the module fires the event and expects the handler to produce an appropriate HTTP response message. If no handlers are registered, the module generates its own HTTP response message with the same abrupt tone.
An application can handle the events fired by a module simply by implementing a method with the correct signature. The syntax is based on the name assigned to the HTTP module in the Web.config file when the module was deployed and the name of the event the module fires. In the previous example, the module was given the name EnableWebServicesModule (which also happens to be its class name, but that is just coincidence) and the event is called Rejection. Based on that, the signature for the HttpApplication subclass's handler for the event is:

```
public void EnableWebServicesModule_Rejection(
        object o, EventsArgs ea);
```

Here is an implementation:

```
<%@ import namespace="System.Web" %>

<script language="C#" runat=server>

public void EnableWebServicesModule_Rejection(
        object o, EventArgs ea)
{
  Context.Response.StatusCode = 403;
  ctx.Response.StatusDescription = "Forbidden";
  ctx.Response.ContentType = "text/plain";
  ctx.Response.Write("Unfortunately, web " +
      "services are not available now, " +
      "please try your request again");
```


Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

```
}

</script>
```

The pipeline plumbing knows how to wire up this event handler based on its name. Now when the HTTP module rejects Web Service invocations, it will fire the Rejection event and the application will have a chance to generate a friendlier HTTP response message. The entire new architecture, including the handler for controlling the module's behavior, is shown in Figure 12.

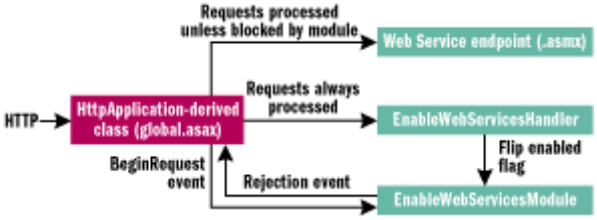


Figure 12 EnableWebServicesModule Architecture

Security in the Pipeline

One of the most common uses of HttpModules is to implement security features such as authentication and authorization, a healthy dose of which can be layered on top of an application quite transparently. In fact, take a look at the default list of HttpModules installed for all Web applications by machine.config (We've omitted the type names for brevity):

```
<httpModules>
  <add name="OutputCache"           type="..."/>
  <add name="Session"               type="..."/>
  <add name="WindowsAuthentication"  type="..."/>
  <add name="FormsAuthentication"    type="..."/>
  <add name="PassportAuthentication" type="..."/>
  <add name="UrlAuthorization"       type="..."/>
  <add name="FileAuthorization"      type="..."/>
</httpModules>
```

Aside from the output caching and session state management modules, these modules are there to help implement security. Also note that the order of the modules is important. Authentication answers the question "Who are you?", while authorization answers the question "Are you allowed to do this?" Clearly authentication must happen before authorization, thus the order of the modules shown previously.

The three authentication modules correspond to the three options in web.config for performing authentication:

```
<authentication mode='None|Windows|Forms|Passport'>
```

By selecting a mode other than None, you enable the corresponding authentication module to do its work. The job of these modules is to perform an authentication handshake with the client and possibly a trusted authority such as passport.com. Once authenticated, these modules create an implementation of IIdentity and IPrincipal that can be used by the authorization modules downstream to determine if the request should be granted or denied. This information is hung on the HttpContext.User property. The HttpHandler at the end of the pipeline can also use this information.

To illustrate, imagine your web.config file was written this way:

```
<configuration>
  <web.config>
    <authentication mode='Forms'/>
    <authorization>
      <deny users='?'/>
      <allow roles='Managers, Staff'/>
      <deny users='*'/>
    </authorization>
  </web.config>
```


Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

</configuration>
Now imagine that a user tries to access the Web application, but is not currently logged on via Forms authentication and thus is considered anonymous. The FormsAuthentication module first processes the request and notices that the client has not sent the special cookie that represents a successful prior login. Thus it constructs an IIdentity object that indicates the user is anonymous and an IPrincipal object that binds an empty set of roles to that identity, attaching this to the HttpContext.User property. When the UrlAuthorization module gets the request, it notes that anonymous access to the directory has been denied in web.config. The <deny users='?'/> tag in web.config represents denial of any anonymous requests. The module checks the user associated with the current context via the HttpContext.User property, sees that it's anonymous, and therefore completes the request and indicates that access is forbidden. Remember though, we're not done yet. The FormsAuthenticationModule now gets to see this forbidden request being sent back to the client, notes that the user is not authenticated, and therefore changes the response into a redirect to the default login page, login.aspx. Assuming the user submits valid credentials to the login page, the login page handler calls FormsAuthentication.RedirectFromLoginPage, which redirects the user back to the page she was after in the first place while sending her an encrypted cookie containing the authenticated user's name. When the redirection causes the client to request the original page again, the FormsAuthentication module decrypts and validates the cookie, then constructs an IIdentity for an authenticated user along with her name. This identity is bound with an empty set of roles and attached to HttpContext.User.
We didn't mention this before, but after setting HttpContext.User, the FormsAuthentication module causes the AuthenticateRequest event to fire. If you've implemented a handler for this event in your global.asax file, you'll now have a chance to take the IIdentity produced by the FormsAuthentication module and bind it to a set of application-defined roles. The code in Figure 13 shows an example of this.
Now that the FormsAuthentication module is finished processing the request, the UrlAuthorization module has its turn. It notes that the request is authenticated, so the first line in the <authorization> section of web.config is satisfied. Now it looks to see if the principal is in either the Managers or Staff role. If so, the request will be allowed; otherwise, the last line of the <authorization> section will cause the request to be denied. In the <authorization> section of web.config, the wildcard character (?) indicates unauthenticated requests; the star character (*) indicates all requests.
This example illustrates how the flexibility of the HTTP pipeline makes it possible to layer security—specifically authentication and authorization—onto many different Web applications without much effort.

Conclusion

This article introduced the ASP.NET pipeline, a very flexible infrastructure for server-side HTTP development. The HTTP pipeline integrates with IIS and provides a rich programming model based on applications, modules, and handlers—all of which you can implement if you want. The HTTP pipeline is a large piece of plumbing and there are many important aspects to it that we did not have space to mention, including support for state management, which is a feature-length topic in its own right. Hopefully this article will help you better understand how the pipeline works and how you can use it in your HTTP-based .NET applications.

100. Is there any difference in the way garbage collection works when it is called automatically by the Runtime environment, and when it is invoked intentionally by the programmer?

GC.Collect () method has the flexibility to even collect those object that are of older generations.

101. What is probing?

After the correct assembly version has been determined by using the information in the calling assembly's reference and in the configuration files, and after it has checked in the global assembly cache (only for strong-named

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

assemblies), the common language runtime attempts to find the assembly. The process of locating an assembly involves the following steps:

1. If a codebase is found in the application configuration file, the runtime checks the specified location. If a match is found, that assembly is used and no probing occurs. If the assembly is not found there, the binding request fails.
2. The runtime then probes for the referenced assembly using the rules specified later in this section.

Locating the Assembly through Codebases

Codebase information can be provided by using a <codeBase> element in a configuration file. This codebase is always checked before the runtime attempts to probe for the referenced assembly. If a publisher policy file containing the final version redirect also contains a codebase, that codebase is the one that is used. For example, if your application configuration file specifies a codebase, and a publisher policy file that is overriding the application information also specifies a codebase, the codebase in the publisher policy file is used.

If no match is found at the location specified by the <codeBase> element, the bind request fails and no further steps are taken. If the runtime determines that an assembly matches the calling assembly's criteria, it uses that assembly. When the file at the given codebase is loaded, the runtime checks to make sure that the name, version, culture, and public key match the calling assembly's reference.

Note Referenced assemblies outside the application's root directory must have strong names and must either be installed in the global assembly cache or specified using the <codebase> element.

Locating the Assembly through Probing

If there is no <codeBase> element in the application configuration file, the runtime probes for the assembly using four criteria:

- Application base, which is the root location where the application is being executed.
- Culture, which is the culture attribute of the assembly being referenced.
- Name, which is the name of the referenced assembly.
- Private binpath, which is the user-defined list of subdirectories under the root location. This location can be specified in the application configuration file and in managed code using the AppendPrivatePath property for an application domain. When specified in managed code, the managed code privatePath is probed first, followed by the path specified in the application configuration file.

Probing the Application Base and Culture Directories

The runtime always begins probing in the application's base, which can be either a URL or the application's root directory on a computer. If the referenced assembly is not found in the application base and no culture information is provided, the runtime searches any subdirectories with the assembly name. The directories probed include:

[application base] / [assembly name].dll
[application base] / [assembly name] / [assembly name].dll

If culture information is specified for the referenced assembly, only the following directories are probed:

[application base] / [culture] / [assembly name].dll
[application base] / [culture] / [assembly name] / [assembly name].dll

Probing the Private binpath

In addition to the culture subdirectories and the subdirectories named for the referenced assembly, the runtime also probes directories specified using the private binpath parameter. The directories specified using the private binpath parameter must be subdirectories of the application's root directory. The directories probed vary depending on whether culture information is included in the referenced assembly request. If culture is included, the following directories are probed:

[application base] / [binpath] / [culture] / [assembly name].dll
[application base] / [binpath] / [culture] / [assembly name] / [assembly name].dll

If culture information is not included, the following directories are probed:

[application base] / [binpath] / [assembly name].dll
[application base] / [binpath] / [assembly name] / [assembly name].dll

Probing Example

Given the following information:

Referenced assembly name: myAssembly
Application root directory: http://www.code.microsoft.com
<probing> element in configuration file specifies: bin

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

Culture: de
The runtime probes the following URLs:
http://www.code.microsoft.com/de/myAssembly.dll
http://www.code.microsoft.com/de/myAssembly/myAssembly.dll
http://www.code.microsoft.com/bin/de/myAssembly.dll
http://www.code.microsoft.com/bin/de/myAssembly/myAssembly.dll
Other Locations Probed
Assembly location can also be determined using the current binding context. This most often occurs when the Assembly.LoadFrom method is used and in COM interop scenarios. If an assembly uses the Assembly.LoadFrom method to reference another assembly, the calling assembly's location is considered to be a hint about where to find the referenced assembly. If a match is found, that assembly is loaded. If no match is found, the runtime continues with its search semantics and then queries the Windows Installer to provide the assembly. If no assembly is provided that matches the binding request, an exception is thrown. This exception is a TypeLoadException in managed code if a type was referenced, or a FileNotFoundException if an assembly being loaded was not found.

102. What is compilation and execution procedure for asp.net page?

Compilation of ASP .Net Pages:

It is a well known fact that ASP .Net pages functional logic can be written in two ways. The ASP .Net code can either be written inside the ASPX page or it can be included as a asp_net_code.cs or vb .net file.

When the ASPX page is embedded with the code of either C# or VB .Net, the ASP .Net run time automatically compiles the code into an assembly and loads it. If the code is kept in a separate source file either as a VB or C# file, it has to be compiled by the programmer, which will be used by the run time for further execution.

Compilation of Script Inside the aspx pages:

When a page is created with the tags inside the .aspx page itself, the code can be written at two locations. It can either be placed inside the <script runat="server">...</script> tag or any where inside the page within the <%.. %> server code blocks. When the code is placed inside the script tag, it is treated to be class level code and any other code found is considered to be a part of the method that renders the web page.

When a the aspnet_wp.exe gets a request for an aspx page which is written without a code behind class file, it generates a class file dynamically for the corresponding page and places it inside the Temporary ASP .Net Files somewhere under the tree structure of the .Net installation path. Then it compiles this into a DLL and finally deletes the class file after successful compilation. It will render the pages from the same binary DLL for any subsequent requests. If it sees any change in the time stamp of the .aspx file, it recognizes that there is some change and recompiles it again for further use.

So ultimately the compilation is only once and all the subsequent requests are entertained only by using the compiled code/DLL.

Writing ASP .Net Apps with Code behind files:

The compilation of these Code Behind files is usually done manually either using the csc.exe command line compiler or by using the Build feature in Microsoft Visual Studio .Net, which produces an output as a library with an extension of .DLL. Now the job of aspnet_wp.exe is very simple. It can directly execute the compiled code and return the HTML page to the web server.

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

Execution Flow of ASPX Pages by IIS:

The execution of ASP .Net pages are not singly handled by the Internet Information Server or in short hand form IIS. It is taken care by the worker process aspnet_wp.exe. Whenever the IIS receives a request from a web browser or a client requesting for a page, it delegates the job to the aspnet_wp.exe process, which takes care of the subsequent jobs and finally returns the HTML page back to the IIS.

When ASP .Net is installed, installation process creates an association for .aspx files with the aspnet_isapi.dll files. When the IIS receives a request from the clients or web browsers for an aspx page, the IIS web server hands this request over to the aspnet_isapi.dll, which in turn instantiates the aspnet_wp.exe job. This aspnet_wp.exe finalizes any unfinished jobs like run time compilation etc., as explained above and then executes the asp .net application in a new application domain. Finally the output page is generated and returned back to the web server, which in-turn sends the file over to the client.

103. If need an array of objects then which option when better ArrayList Class or List Generic Class ? Array[] / ArrayList / List<T> Difference i.e. string[] arraylist.add(string) / List<string> ?

Arrays are one of the simplest and most widely used data structures in computer programs. Arrays in any programming language all share a few common properties:
The contents of an array are stored in contiguous memory.
All of the elements of an array must be of the same type; hence arrays are referred to as homogeneous data structures.
Array elements can be directly accessed.
Behind the scenes the ArrayList uses a System.Array of type object. Since all types are derived either directly or indirectly from object, an object array can hold elements of any type. By default, an ArrayList creates a 16-element object array, although the precise size can be specified through a parameter in the constructor or the Capacity property. When adding an element thought the Add() method, the number of elements in the internal array is checked with the array's capacity. If adding the new element causes the count to exceed the capacity, the capacity is automatically doubled and the array is redimensioned.
An arraylist is dynamically sized whereas an array is not. You can iterate over an array list using a foreach and you can call the Add() method on an arraylist.

Performance wise an array is typically better for performance. With an array list it is similar internally to an object[] which requires a cast to get it to the correct object. With .NET 2.0 and generics you can avoid these casts by using some of the generic collections or creating your own from IList<T>, etc.

Detailed

Everyone's Favorite Linear, Direct Access, Homogeneous Data Structure—the Array

Arrays are one of the simplest and most widely used data structures in computer programs. Arrays in any programming language all share a few common properties:

- The contents of an array are stored in contiguous memory.
- All of the elements of an array must be of the same type; hence arrays are referred to as homogeneous data structures.
- Array elements can be directly accessed. (This is not necessarily the case for many other data structures. For example, in part 4 of this article series we'll examine a data structure called the SkipList. To access a particular element of a SkipList you must search through other elements until you find the element for which you're looking. With arrays, however, if you know you want to access the ith element, you can simply use one line of code: arrayName[i].)

The common operations performed on arrays are:

- Allocation

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

- Accessing
- Redimensioning

When an array is initially declared in C# it has a null value. That is, the following line of code simply creates a variable named booleanArray that equals null:
bool [] booleanArray;
Before we can begin to work with the array, we must allocate a specified number of elements. This is accomplished using the following syntax:

booleanArray = new bool[10];

Or more generically:

arrayName = new arrayType[allocationSize];

This allocates a contiguous block of memory in the CLR-managed heap large enough to hold the allocationSize number of arrayTypes. If arrayType is a value type, then allocationSize number of unboxed arrayType values are created. If arrayType is a reference type, then allocationSize number of arrayType references are created. (If you are unfamiliar with the difference between reference and value types and the managed heap versus the stack,)
To help hammer home how the .NET Framework stores the internals of an array, consider the following example:

bool [] booleanArray; FileInfo [] files; booleanArray = new bool[10]; files = new FileInfo[10];

Here, the booleanArray is an array of the value type System.Boolean, while the files array is an array of a reference type, System.IO.FileInfo. Figure 1 shows a depiction of the CLR-managed heap after these four lines of code have executed.

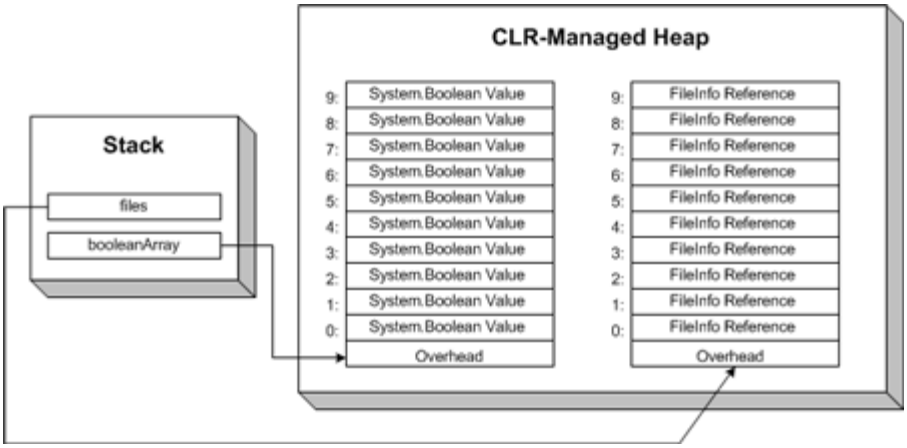


Figure 1. The contents of an array are laid out contiguously in the managed heap.
The thing to keep in mind is that the ten elements in the files array are references to FileInfo instances. Figure 2 hammers home this point, showing the memory layout if we assign some of the values in the files array to FileInfo instances.

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

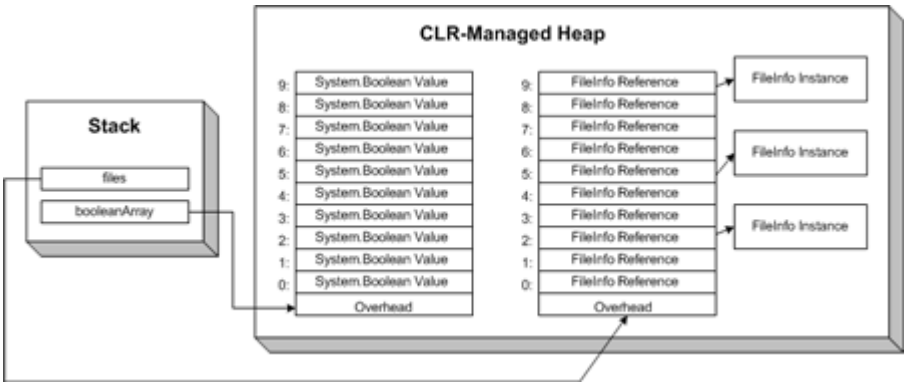


Figure 2. The contents of an array are laid out contiguously in the managed heap.

All arrays in .NET provide allow their elements to both be read and written to. The syntax for accessing an array element is:

// Read an array element bool b = booleanArray[7]; // Write to an array element booleanArray[0] = false;

The running time of an array access is denoted O(1) because it is constant. That is, regardless of how many elements are stored in the array, it takes the same amount of time to lookup an element. This constant running time is possible solely because an array's elements are stored contiguously; hence a lookup just requires knowledge of the array's starting location in memory, the size of each array element, and the element to be indexed.

Realize that in managed code, array lookups are a bit more involved than this because with each array access the CLR checks to ensure that the index being requested is within the array's bounds. If the array index specified is out of bounds, an IndexOutOfRangeException is thrown. This check helps ensure that when stepping through an array we do not accidentally step past the last array index and into some other memory. This check, though, does not affect the running time of an array access because the time to perform such checks does not increase as the size of the array increases.

Note This index-bounds check comes at a slight cost of performance for applications that make a large number of array accesses. With a bit of unmanaged code, though, this index out of bounds check can be bypassed. For more information, refer to Chapter 14 of Applied Microsoft .NET Framework Programming by Jeffrey Richter.

When working with an array, you might need to change the number of elements it holds. To do so, you'll need to create a new array instance of the specified size and then copy over the contents of the old array into the new, resized array. This process is called redimensioning, and can be accomplished with the following code:

```
using System;
using System.Collections;
 public class MyClass {
 public static void Main()
          { // Create an integer array with three elements
          int [] fib = new int[3];
          fib[0] = 1;
          fib[1] = 1;
          fib[2] = 2; // Redimension message to a 10 element array
         int [] temp = new int[10];
         // Copy the fib array to temp
         fib.CopyTo(temp, 0);
          // Assign temp to fib
         fib = temp;
          }
    }
```

After the last line of code, fib references a ten-element Int32 array. The elements 3 through 9 in the fib array will have the default Int32 value—0.

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

Arrays are excellent data structures to use when storing a collection of heterogeneous types that you only need to access directly. Searching an unsorted array has linear running time. While this is acceptable when working with small arrays, or when performing very few searches, if your application is storing large arrays that are searched frequently, there are a number of other data structures better suited for the job. We'll look at some such data structures in upcoming pieces of this article series. (Realize that if you are searching an array on some property and the array is sorted by that property, you can use an algorithm called binary search to search the array in O(log n) running time, which is on par with the search times for binary search trees. In fact, the Array class contains a static BinarySearch() method. For more information on this method, check out an earlier article of mine, Efficiently Searching a Sorted Array.

Note The .NET Framework allows for multi-dimensional arrays as well. Multi-dimensional arrays, like single-dimensional arrays, offer a constant running time for accessing elements. Recall that the running time to search through an n-element single dimensional array was denoted O(n). For an nxn two-dimensional array, the running time is denoted O(n2) because the search must check n2 elements. More generally, a k-dimensional array has a search running time of O(nk).

The ArrayList: a Heterogeneous, Self-Redimensioning Array

While arrays definitely have their time and place, arrays create some limitations on design because a single array can only store elements of one type (homogeneity), and when using arrays you must specifically allocate a certain number of elements. Oftentimes, though, developers want something more flexible—a simple collection of objects of potentially different types that can be easily managed without having to worry about allocation issues. The .NET Framework Base Class Library provides such a data structure called the System.Collections.ArrayList.

An example of the ArrayList in action can be seen in the code snippet below. Note that with the ArrayList any type can be added and no allocation step has to be performed, and in fact, elements of different types can be added to the ArrayList. Furthermore, at no point do we have to concern ourselves with redimensioning the ArrayList. All of this is handled behind the scenes for us.

```
ArrayList countDown = new ArrayList(); countDown.Add(5); countDown.Add(4); countDown.Add(3);
countDown.Add(2); countDown.Add(1); countDown.Add("blast off!"); countDown.Add(new ArrayList());
```

Behind the scenes the ArrayList uses a System.Array of type object. Since all types are derived either directly or indirectly from object, an object array can hold elements of any type. By default, an ArrayList creates a 16-element object array, although the precise size can be specified through a parameter in the constructor or the Capacity property. When adding an element thought the Add() method, the number of elements in the internal array is checked with the array's capacity. If adding the new element causes the count to exceed the capacity, the capacity is automatically doubled and the array is redimensioned.

The ArrayList, like the array, can be directly indexed using the same syntax:

```
// Read access int x = (int) countDown[0]; string y = (string) countDown[5]; // Write access countDown[1] = 5; // **
WILL GENERATE AN ArgumentOutOfRange EXCEPTION ** countDown[7] = 5;
```

Since the ArrayList stores an array of objects, when reading the value from an ArrayList you need to explicitly cast it to the data type being stored in the specified location. Also, note that if you try to reference an ArrayList element greater than the ArrayList's size, a System.ArgumentOutOfRange exception will be thrown.

While the ArrayList provides added flexibility over the standard array, this flexibility comes at the cost of performance, especially when storing value types in an ArrayList. Recall that an array of a value type—such as a System.Int32, System.Double, System.Boolean, and so on—is stored contiguously in the managed heap in its unboxed form. The ArrayList's internal array, however, is an array of object references. Therefore, even if you have an ArrayList that stores nothing but value types, each ArrayList element is a reference to a boxed value type, as shown in Figure 3.

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

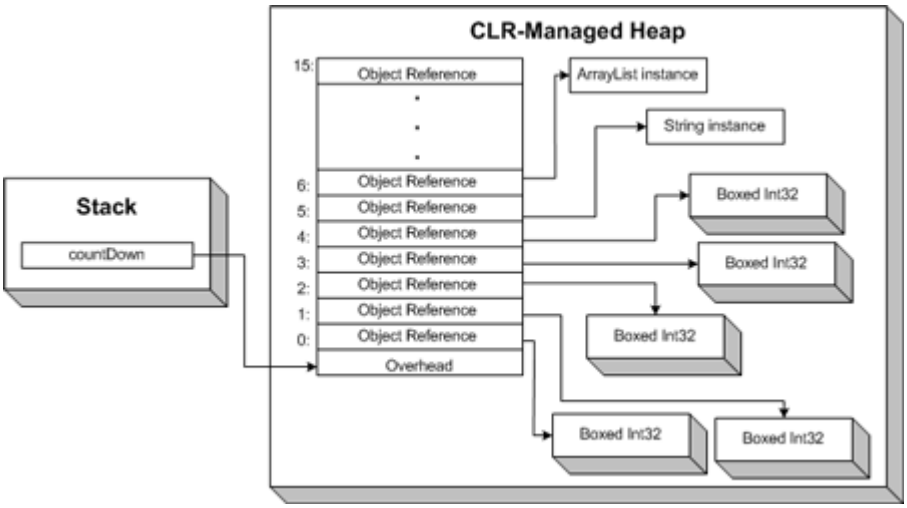


Figure 3. The ArrayList contains a contiguous block of object references

The boxing and unboxing, along with the extra level of indirection that comes with using value types in an ArrayList, can hamper the performance of your application when using large ArrayLists with many reads and writes. As Figure 3 illustrates, the same memory layout occurs for reference types in both ArrayLists and arrays. The ArrayList's self-redimensioning shouldn't cause any sort of performance degradation in comparison to an array. If you know the precise number of elements that need to be stored in the ArrayList, you can essentially turn off self-redimensioning by specifying the initial capacity in the ArrayList's constructor. If you don't know the precise size, even with an array you may have to redimension the array should the number of elements inserted exceed the array's size. A classic computer science problem is determining how much new space to allocate when running out of a space in some buffer. One option when redimensioning an array is to allocate just one more element in the resized array. That is, if the array is initially allocated with five elements, before the sixth element is inserted, the array is redimensioned to six elements. Clearly, this approach conserves the most memory, but can become costly because each insert following the first redimensioning results in another redimensioning.Another option, at the opposite end of the spectrum, is to redimension the array 100 times larger than its current size. That is, if an array is initially allocated with five elements, before the sixth element is inserted, the array is redimensioned to 500 elements. Clearly, this approach greatly reduces the number of redimensionings that needs to occur, but, if only a few more elements are added to the array then hundreds of array elements have been unused, resulting in wasted space.A tried and true compromise to this problem is to simply double the existing size of the array when free space becomes exhausted. So, for an array initially allocated with five elements, adding a sixth element would cause the array to be redimensioned to a size of 10. This is precisely the approach the ArrayList class takes and, best of all, it is all performed for you.The asymptotic running time of the ArrayList's operations is the same as those of the standard array. While the ArrayList does indeed have more overhead, especially when storing value types, the relationship between the number of elements in the ArrayList and the cost per operation is the same as the standard array.

Conclusion

This article started our discussion on data structures by identifying why studying data structures was important, and by providing a means of how to analyze the performance of data structures. This material is important to understand as being able to analyze the running times of various data structure operations is a useful tool when deciding what data structure to use for a particular programming problem.

After studying how to analyze data structures, we turned to examining two of the most common data structures in the .NET Framework Base Class Library—System.Array class and System.Collections.ArrayList. Arrays allow for a contiguous block of homogeneous types. Their main benefit is that they provide lightning-fast access to reading and writing array elements. Their weak point lies in searching arrays, as each and every element must potentially be visited (in an unsorted array).

The ArrayList provides a more flexible array-like data structure. Rather than enforcing homogeneous types, the ArrayList allows for heterogeneous types to be stored by using an array of objects. Furthermore, the ArrayList does not require explicit allocation and can gracefully grow as the more elements are added.

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

104. Custom Paging in ASP.NET 2.0 with SQL Server 2005

Introduction
A common pattern in web development is providing paged access to data. Rather than displaying the entire contents of a report or database table to an end user, developers often show only a subset of records per web page, with controls for moving from page to page. With ASP.NET 1.x, the DataGrid made paging incredibly simple - just set the AllowPaging property to True and add a few lines of code in the PageIndexChanged event handler and you were done! ASP.NET 2.0's GridView makes the process even simpler - just check the Enable Paging option from the GridView's smart tag - no code needed.

Of course nothing is free in life, and the tradeoff you make with the ease of checking a checkbox to enable paging (or, in the DataGrid's case, writing a couple lines of code) is performance. Out of the box, the DataGrid and GridView use default paging, which is a simple paging model that returns all of the records for each every page of data shown. When paging through small amounts of data (dozens to a hundred or so records), this inefficiency is likely outweighed by the ease of adding the feature. However, if you want to page through thousands, tens of thousands, or hundreds of thousands of records the default paging model is not viable.

The alternative to default paging is custom paging, in which you are tasked with writing code that intelligently grabs the correct subset of data. It requires a bit more work, but is essential when dealing with sufficiently-sized data. I discuss how to implement custom paging in ASP.NET 1.x in my book ASP.NET Data Web Controls Kick Start. In this article we'll look at how to implement custom paging in ASP.NET 2.0 using SQL Server 2005's new ROW_NUMBER() feature. (For more information on SQL Server's new ranking features, including ROW_NUMBER(), see Returning Ranked Results with Microsoft SQL Server 2005.)

Default Paging vs. Custom Paging
The GridView in 2.0 (and the DataGrid in 1.x) offers two paging models: default paging and custom paging. The two models provide a tradeoff between performance and ease of setting up/configuring/using. The SqlDataSource control uses default paging (although you can wrestle it into using custom paging); the ObjectDataSource uses default paging by default, but has an easy mechanism to indicate that it should use custom paging. Keep in mind that the GridView merely displays data; it's the GridView's data source control that is actually retrieving data from the database.

With default paging, each time a new page of data in displayed in the GridView, all of the data is requeried from the and returned from the GridView's data source. Once all of the data has been returned, the GridView selectively displays part of the entire set of data, based on the page of data the user is viewing and how many records per page are displayed. The key thing to understand here is that every single time a page of data is loaded - be it on the first page visit when viewing the first page of data or when the user postsbacks after requesting to view a different page of data - the *entire* data result is retrieved.

For example, imagine that you work at an eCommerce company and you want to allow the user to page through a list of the 150 products your company sells. Specifically, you want to display 10 records per page. Now, when a user visits the web page, all 150 records will be returned by the data source control, but the GridView will display the first 10 products (products 1 to 10). Next, imagine that the user navigates to the next page of data. This will cause a postback, at which point the GridView will rerequest all 150 records from the data source control, but this time only display the second set of 10 (products 11 to 20).

Caching and the SqlDataSource

The SqlDataSource allows for the DataSet it returns to be cached by simply setting the EnableCaching property. With a cached DataSet, stepping to another page does not require the database be requiried since the data being paged through is cached in memory. However, on the initial page load the same problem arises - *all* of the data must be loaded into the cached DataSet. Furthermore, you must worry about stale data with this approach (although if you use SQL cache dependencies, then this point is moot).

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

Even with caching the DataSet, my unscientific tests found custom paging to be twice as fast... When we examine the performance metrics later, though, you'll see that this cached approach far outshines the non-cached approach. (But it still doesn't beat the custom paging approach!)

For more on caching the DataSet returned by the SqlDataSource see Caching Data With the SqlDataSource.

With custom paging, you, the developer, have to do a bit more work. Rather than just being able to blindly bind the GridView to a data source control and check the "Enable Paging" checkbox, you have to configure the data source control to selectively retrieve only those records that should be shown for the particular page. The benefit of this is that when displaying the first page of data, you can use a SQL statement that only retrieves products 1 through 10, rather than all 150 records. However, your SQL statement has to be "clever" enough to be able to know how to just snip out the right subset of records from the 150.

The Performance Edge of Custom Paging

Realize that custom paging provides better performance than default paging because only those database records that need to be displayed are retrieved. In our products example, we assumed there were 150 products, showing 10 per page. With custom paging, if the user stepped through all 15 pages of data, precisely 150 records would have been queried from the database. With default paging, however, for each page of data, 150 records would have been accessed, leading to a total number of retrieved records of 15 times 150, or 2,250!

While custom paging exhibits better performance, default paging is much easier to use. Therefore, I would encourage you to use default paging if the data you are paging through is relatively small and/or the database server is not heavily trafficked. If you have several hundred, thousands, or tens of thousands of records you are paging through, by all means use custom paging. However, for paging through something like the ASPFAQs.com database, which only has, currently, ~200 FAQs, default paging is sufficient.

(Of course, if you use default paging on a small table with, say, 75 records, you are assuming that over time the table's row count will stay low. There will be some unhappy customers if you use default paging on that small table which later grows to be a table with 7,500 records!)

Efficiently Getting Back a Page of Data with SQL Server 2005

As discussed in an earlier 4Guys article, Returning Ranked Results with Microsoft SQL Server 2005, SQL Server 2005 introduces a number of new keywords for returning ranked results. In particular, the ROW_NUMBER() keyword enables us to associate a sequentially-increasing row number for the results returned. We can use ROW_NUMBER(), then, to get a particular page of data using a query like the following:

```
SELECT ...
FROM
  (SELECT ...
      ROW_NUMBER() OVER(ORDER BY ColumnName) as RowNum
   FROM Employees e
  ) as DerivedTableName
WHERE RowNum BETWEEN @startRowIndex AND (@startRowIndex + @maximumRows) - 1
```

Here @startRowIndex is the index of the row to start from and @maximumRows is the maximum number of records to show per page. This query returns the subset of records whose ROW_NUMBER() is between the starting index and the starting index plus the page size.

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

To help concretize this concept, let's look at the following example. Imagine that we have an Employees table with 5,000 records (business is good!). The following query:

```
SELECT RowNum, EmployeeID, LastName, FirstName
FROM
  (SELECT EmployeeID, LastName, FirstName
     ROW_NUMBER() OVER(ORDER BY EmployeeID) as RowNum
   FROM Employees e
  ) as EmployeeInfo
```

Would return results like:

RowNum	EmployeeID	LastName	FirstName
1	1000	Smith	Frank
2	1001	Jackson	Lucy
3	1011	Lee	Sam
4	1012	Mitchell	Jisun
5	1013	Yates	Scott
6	1016	Props	Kathryn
...			
5000	6141	Jordan	DJ

Notice that even though the EmployeeID fields may have gaps and may not start at 1, the ROW_NUMBER() value starts at 1 for the first record and steadily increases. Therefore, if we want to view 10 records per page, and we want to see the third page, we know that we want records 31-40, and can accomplish that in a simple WHERE clause.

Configuring the ObjectDataSource for Custom Paging
As aforementioned, the SqlDataSource isn't designed to provide custom sorting capabilities. The ObjectDataSource, on the other hand, was designed to support this scenario. The ObjectDataSource is a data source control that's designed to access data from an object. The object can retrieve its data however it likes, be it from a Web Service, a database, the file system, an XML file... whatever. The ObjectDataSource doesn't care, it simply acts as a proxy between the data Web control that wants to consume the data (such as a GridView control) and the underlying data that the object provides. (For more information on the ObjectDataSource see the ObjectDataSource Control Overview.)

When binding a data Web control to an ObjectDataSource the "Enable Paging" option is available. If you've not specifically set up the ObjectDataSource to support custom paging, the paging provided will be of the default paging flavor. To setup custom paging with the ObjectDataSource you need to be using an object that provides the following functionality:

A method that takes in as its final two input parameters two integer values. The first integer value specifies the starting index from which to retrieve the data (it's zero-based), while the second integer value indicates the maximum number of records to retrieve per page. This method needs to return the precise subset of data being requested, namely the data starting at the specified index and not exceeding the total number of records indicated.

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

A method that returns an integer value specifying the total number of records that are being paged through. (This information is used by the data Web control when rendering the paging controls, since it needs to know how many total pages of data there are when showing page numbers or when deciding whether to enable the Next link.)

If you are using an underlying object that provides these features, configuring the ObjectDataSource to support custom paging is a breeze. Just set the following ObjectDataSource properties:

- Set EnablePaging to True
- Set SelectMethod to the method that accepts the starting index and maximum number of rows input parameters
- Set the StartRowIndexParameterName to the name of the integer input parameter in your SelectMethod that accepts the starting index; if you do not provide this value it defaults to startRowIndex
- Set the MaximumRowsParameterName to the name of the integer input parameter in your SelectMethod that accepts the maximum number of rows to return; if you do not provide this value it defaults to maximumRows
- Set SelectCountMethod to the method that returns the total number of records being paged through

That's it. Once you've done the above, the ObjectDataSource will be using the custom paging functionality. Of course, the hard part of this all is creating the underlying object that can intelligently grab the right subset of data. But once you have that object, configuring the ObjectDataSource to utilize custom paging is just a matter of setting a few properties.

Creating an Object That Supports Custom Paging
In order to bind an ObjectDataSource to a GridView we need to first have an underlying object that the ObjectDataSource will use, and this object must have methods for accessing a particular subset of the data and returning the number of rows to be paged through. As discussed in Joseph Chancellor's article, Using Strongly-Typed Data Access in Visual Studio 2005 and ASP.NET 2.0 and Brian Noyes's article Build a Data Access Layer with the Visual Studio 2005 DataSet Designer, creating objects that can be bound to the ObjectDataSource is a breeze in Visual Studio 2005. The first step is to define the stored procedures (or SQL queries) that will be used to populate the strongly-typed DataSets returned by these object's methods.

The download, available at the end of this article, has a sample database with 50,000 employee records (plus an easy way to add additional records in bulk). The database includes three stored procedures that are used by the two custom paging demos:

- GetEmployeesSubset(@startRowIndex int, @maximumRows int) - returns at most @maximumRows records from the Employees table starting at @startRowIndex when ordered by EmployeeID.
- GetEmployeesRowCount - returns the total number of records in the Employees table.
- GetEmployeesSubsetSorted(@sortExpression nvarchar(50), @startRowIndex int, @maximumRows int) - this sproc returns a page of data sorted by a specified sort expression. This allows the a page of data ordered by, say, Salary, to be returned. (GetEmployeesSubset returns records always ordered by EmployeeID.) This flexibility is needed if you want to create a sortable GridView that employs custom paging.

We won't be discussing implementing the sortable, custom pageable GridView in this article, although examples are included in this article's download; see Sorting Custom Paged Results for a look at how to create a custom paging and bi-directional sortable UI...

Once these stored procedures are created, I created the underlying object by adding a Typed DataSet to my project (Employees.xsd). I then added three methods, one against each of the stored procedures listed above. I ended up with an EmployeesTableAdapter object with methods GetEmployeesSubset(startRowIndex, maximumRows) and GetEmployeesRowCount() that can then be plugged into the ObjectDataSource's properties.

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

(For step-by-step instructions on creating the Typed DataSet, see Using Strongly-Typed Data Access in Visual Studio 2005 and ASP.NET 2.0 and Scott Guthrie's blog entry Building a DAL using Strongly Typed TableAdapters and DataTables in VS 2005 and ASP.NET 2.0.)

Comparing the Performance of Default Paging and Custom Paging
To compare the performance between default and custom paging against the database included in this article's download (which has a table with 50,000 records), I used both SQL Profile and ASP.NET tracing to ascertain relative performance differences. (These techniques were done very unscientifically on my computer, which had other processes running in the background and such. While the results can hardly be called conclusive, I think the performance differences between the two methods clearly highlights custom paging's advantages.)

Default Paging	
(Selecting *All* Records from Employees)	
Duration (sec)	**Reads**
1.455	383
1.405	383
1.434	383
1.394	383
1.365	383
Avg: 1.411	**Avg: 383**

Custom Paging	
(Selecting a Page of Records from Employees)	
Duration (sec)	**Reads**
0.003	29
0.000	29
0.000	29
0.003	29
0.003	29
Avg: 0.002	**Avg: 29**

Default Paging
(Selecting *All* Records from Employees)
Page Load Duration (sec)
2.34136852588807
2.35772228034569
2.43368277253115
2.43237562315881
2.33167064529151
Avg: 2.379363969

Custom Paging
(Selecting a Page of Records from Employees)
Page Load Duration (sec)
0.0259611207569677
0.0280046765720224
0.0359054013848129
0.0295534767686955
0.0300096800012292
Avg: 0.029886871

Cached SqlDataSource
(Selecting *All* Records, But Caching Them)
Page Load Duration (sec)
2.39666633608461
0.0431529705591074
0.0443528437273452
0.0442313199023898
0.0491523364002967
Avg: 0.515511161

As you can see, the custom paging is roughly two order of magnitudes faster than the default paging. At the database level, the GetEmployeesSubset(@startRowIndex int, @maximumRows int) stored procedure is about 470 times faster than the simple SELECT statement that returns all records from the Employees table. Custom paging is about 120 times faster than default paging at the ASP.NET level. The reduction is performance gain is probably due to expensive workloads common to both approaches, namely setting up the database connection and issuing the command. Regardless, two orders of magnitude is a very big difference in the world of performance. And this disparity would be more pronounced with larger data sets or a server that was experiencing any kind of load.

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

The cached SqlDataSource has a high cost when the cache is empty, as it must go to the database and get all of the records. The frequency that the cache needs to be reloaded depends upon free resources on the web server (if you have low resources available, the cached DataSet may get evicted) and your cache expiration policy. After the data has been cached, though, it greatly improves in performance and is comparable to the custom paging approach. The 0.516 second average time would be amortized to closer to 0.05 seconds as more requests were served with the cached data.

Conclusion
As with the DataGrid in ASP.NET 1.x, the GridView in 2.0 offers two flavors of paging: default and custom. Default paging is easier to setup, but involves requerying the database when viewing each and every page of data. Custom paging, however, more intelligently just grabs those records needing to be displayed and therefore affords much higher degree of performance. SQL Server 2005 simplifies obtaining the precise subset of records for an arbitrary page due to its ability to rank results, which includes the ROW_NUMBER() feature.

If you are building web applications that need to scale or either now or in the future will allow users to page through potentially large data sets, it behooves you to implement custom paging.

105. Why do I get errors when I try to serialize a Hashtable?

XmlSerializer will refuse to serialize instances of any class that implements IDictionary, e.g. Hashtable. SoapFormatter and BinaryFormatter do not have this restriction.

106. What is the difference between a Struct and a Class?

- We can't declare a structure as sealed or abstract.
- Structure is value type and class is reference type.
- Structure doesn't contain destructor while class contains.
- Structure can't be inherited. It can only inherit interfaces, while a class can inherit class and interface as well.
- Structure can contain parameterized constructor.

The struct type is suitable for representing lightweight objects such as Point, Rectangle, and Color. Although it is possible to represent a point as a class, a struct is more efficient in some scenarios. For example, if you declare an array of 1000 Point objects, you will allocate additional memory for referencing each object. In this case, the struct is less expensive. When you create a struct object using the new operator, it gets created and the appropriate constructor is called. Unlike classes, structs can be instantiated without using the new operator. If you do not use new, the fields will remain unassigned and the object cannot be used until all of the fields are initialized. It is an error to declare a default (parameterless) constructor for a struct. A default constructor is always provided to initialize the struct members to their default values.
It is an error to initialize an instance field in a struct.There is no inheritance for structs as there is for classes. A struct cannot inherit from another struct or class, and it cannot be the base of a class. Structs, however, inherit from the base class Object. A struct can implement interfaces, and it does that exactly as classes do.
A struct is a value type, while a class is a reference type.

107. What is the difference between XML Web Services using ASMX and .NET Remoting using SOAP?

ASP.NET Web services and .NET Remoting provide a full suite of design options for cross-process and cross-plaform communication in distributed applications. In general, ASP.NET Web services provide the highest levels of interoperability with full support for WSDL and SOAP over HTTP, while .NET Remoting is designed for common language runtime type-system fidelity and supports additional data format and communication channels. Hence if we

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

looking cross-platform communication than web services is the choice coz for .NET remoting .Net framework is requried which may or may not present for the other platform.

Serialization and Metadata
ASP.NET Web services rely on the System.Xml.Serialization.XmlSerializer class to marshal data to and from SOAP messages at runtime. For metadata, they generate WSDL and XSD definitions that describe what their messages contain. The reliance on pure WSDL and XSD makes ASP.NET Web services metadata portable; it expresses data structures in a way that other Web service toolkits on different platforms and with different programming models can understand. In some cases, this imposes constraints on the types you can expose from a Web service—XmlSerializer will only marshal things that can be expressed in XSD. Specifically, XmlSerializer will not marshal object graphs and it has limited support for container types.

.NET Remoting relies on the pluggable implementations of the IFormatter interface used by the System.Runtime.Serialization engine to marshal data to and from messages. There are two standard formatters, System.Runtime.Serialization.Formatters.Binary.BinaryFormatter and System.Runtime.Serialization.Formatters.Soap.SoapFormatter. The BinaryFormatter and SoapFormatter, as the names suggest, marshal types in binary and SOAP format respectively. For metadata, .NET Remoting relies on the common language runtime assemblies, which contain all the relevant information about the data types they implement, and expose it via reflection. The reliance on the assemblies for metadata makes it easy to preserve the full runtime type-system fidelity. As a result, when the .NET Remoting plumbing marshals data, it includes all of a class's public and private members; handles object graphs correctly; and supports all container types (e.g., System.Collections.Hashtable). However, the reliance on runtime metadata also limits the reach of a .NET Remoting system—a client has to understand .NET constructs in order to communicate with a .NET Remoting endpoint. In addition to pluggable formatters, the .NET Remoting layer supports pluggable channels, which abstract away the details of how messages are sent. There are two standard channels, one for raw TCP and one for HTTP. Messages can be sent over either channel independent of format.

Distributed Application Design: ASP.NET Web Services vs. .NET Remoting

ASP.NET Web services favor the XML Schema type system, and provide a simple programming model with broad cross-platform reach. .NET Remoting favors the runtime type system, and provides a more complex programming model with much more limited reach. This essential difference is the primary factor in determining which technology to use. However, there are a wide range of other design factors, including transport protocols, host processes, security, performance, state management, and support for transactions to consider as well.

Security
Since ASP.NET Web services rely on HTTP, they integrate with the standard Internet security infrastructure. ASP.NET leverages the security features available with IIS to provide strong support for standard HTTP authentication schemes including Basic, Digest, digital certificates, and even Microsoft® .NET Passport. (You can also use Windows Integrated authentication, but only for clients in a trusted domain.) One advantage of using the available HTTP authentication schemes is that no code change is required in a Web service; IIS performs authentication before the ASP.NET Web services are called. ASP.NET also provides support for .NET Passport-based authentication and other custom authentication schemes. ASP.NET supports access control based on target URLs, and by integrating with the .NET code access security (CAS) infrastructure. SSL can be used to ensure private communication over the wire.

Although these standard transport-level techniques to secure Web services are quite effective, they only go so far. In complex scenarios involving multiple Web services in different trust domains, you have to build custom ad hoc solutions. Microsoft and others are working on a set of security specifications that build on the extensibility of SOAP messages to offer message-level security capabilities. One of these is the XML Web Services Security Language (WS-Security), which defines a framework for message-level credential transfer, message integrity, and message confidentiality.

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

As noted in the previous section, the .NET Remoting plumbing does not secure cross-process invocations in the general case. A .NET Remoting endpoint hosted in IIS with ASP.NET can leverage all the same security features available to ASP.NET Web services, including support for secure communication over the wire using SSL. If you are using the TCP channel or the HTTP channel hosted in processes other than aspnet_wp.exe, you have to implement authentication, authorization and privacy mechanisms yourself.

One additional security concern is the ability to execute code from a semi-trusted environment without having to change the default security policy. ASP.NET Web Services client proxies work in these environments, but .NET Remoting proxies do not. In order to use a .NET Remoting proxy from a semi-trusted environment, you need a special serialization permission that is not given to code loaded from your intranet or the Internet by default. If you want to use a .NET Remoting client from within a semi-trusted environment, you have to alter the default security policy for code loaded from those zones. In situations where you are connecting to systems from clients running in a sandbox—like a downloaded Windows Forms application, for instance—ASP.NET Web Services are a simpler choice because security policy changes are not required.

108. What is Web Gardening? How would using it affect a design?

The Web Garden Model: The Web garden model is configurable through the section of the machine.config file. Notice that the section is the only configuration section that cannot be placed in an application-specific web.config file. This means that the Web garden mode applies to all applications running on the machine. However, by using the node in the machine.config source, you can adapt machine-wide settings on a per-application basis.

Two attributes in the section affect the Web garden model. They are webGarden and cpuMask. The webGarden attribute takes a Boolean value that indicates whether or not multiple worker processes (one per each affinitized CPU) have to be used. The attribute is set to false by default. The cpuMask attribute stores a DWORD value whose binary representation provides a bit mask for the CPUs that are eligible to run the ASP.NET worker process. The default value is -1 (0xFFFFFF), which means that all available CPUs can be used. The contents of the cpuMask attribute is ignored when the webGarden attribute is false. The cpuMask attribute also sets an upper bound to the number of copies of aspnet_wp.exe that are running.

Web gardening enables multiple worker processes to run at the same time. However, you should note that all processes will have their own copy of application state, in-process session state, ASP.NET cache, static data, and all that is needed to run applications. When the Web garden mode is enabled, the ASP.NET ISAPI launches as many worker processes as there are CPUs, each a full clone of the next (and each affinitized with the corresponding CPU). To balance the workload, incoming requests are partitioned among running processes in a round-robin manner. Worker processes get recycled as in the single processor case. Note that ASP.NET inherits any CPU usage restriction from the operating system and doesn't include any custom semantics for doing this.

All in all, the Web garden model is not necessarily a big win for all applications. The more stateful applications are, the more they risk to pay in terms of real performance. Working data is stored in blocks of shared memory so that any changes entered by a process are immediately visible to others.

However, for the time it takes to service a request, working data is copied in the context of the process. Each worker process, therefore, will handle its own copy of working data, and the more stateful the application, the higher the cost in performance. In this context, careful and savvy application benchmarking is an absolute must.

Changes made to the section of the configuration file are effective only after IIS is restarted. In IIS 6, Web gardening parameters are stored in the IIS metabase; the webGarden and cpuMask attributes are ignored.

109. What is view state? Where it stored? Can we disable it?

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

The web is state-less protocol, so the page gets instantiated, executed, rendered and then disposed on every round trip to the server. The developers code to add "statefulness" to the page by using Server-side storage for the state or posting the page to itself. When require to persist and read the data in control on webform, developer had to read the values and store them in hidden variable (in the form), which were then used to restore the values. With advent of .NET framework, ASP.NET came up with ViewState mechanism, which tracks the data values of server controls on ASP.NET webform. In effect, ViewState can be viewed as "hidden variable managed by ASP.NET framework!" When ASP.NET page is executed, data values from all server controls on page are collected and encoded as single string, which then assigned to page's hidden atrribute "< input type=hidden >", that is part of page sent to the client.

ViewState value is temporarily saved in the client's browser.ViewState can be disabled for a single control, for an entire page orfor an entire web application. The syntax is:

Disable ViewState for control (Datagrid in this example)
< asp:datagrid EnableViewState="false" ... / >

Disable ViewState for a page, using Page directive
< %@ Page EnableViewState="False" ... % >

Disable ViewState for application through entry in web.config
< Pages EnableViewState="false" ... / >

110. Can you debug a Windows Service? How?

Yes we can debug a Windows Service. Attach the WinDbg debugger to a service after the service starts This method is similar to the method that you can use to attach a debugger to a process and then debug a process.

Use the process ID of the process that hosts the service that you want to debug
1 to determine the process ID (PID) of the process that hosts the service that you want to debug, use one of the following methods.
• Method 1: Use the Task Manager
a. Right-click the taskbar, and then click Task Manager. The Windows Task Manager Dialog box appears.
b. Click the Processes tab of the Windows Task Manager dialog box.
c. Under Image Name, click the image name of the process that hosts the service that you want to debug. Note the process ID of this process as specified by the value of the corresponding PID field.
• Method 2: Use the Task List Utility (tlist.exe)
a. Click Start, and then click Run. The Run dialog box appears.
b. In the Open box, type cmd, and then click OK.
c. At the command prompt, change the directory path to reflect the location of the tlist.exe file on your computer.

Note The tlist.exe file is typically located in the following directory: C:\Program Files\Debugging Tools for Windows
d. At the command prompt, type tlist to list the image names and the process IDs of all processes that are currently running on your computer.

Note Make a note of the process ID of the process that hosts the service that you want to debug.

2 At a command prompt, change the directory path to reflect the location of the windbg.exe file on your computer.

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

Note If a command prompt is not open, follow steps a and b of Method 1. The windbg.exe file is typically located in the following directory: C:\Program Files\Debugging Tools for Windows.
3 At the command prompt, type windbg –p ProcessID to attach the WinDbg debugger to the process that hosts the service that you want to debug.

Note ProcessID is a placeholder for the process ID of the process that hosts the service that you want to debug.

Use the image name of the process that hosts the service that you want to debug

You can use this method only if there is exactly one running instance of the process that hosts the service that you want to run. To do this, follow these steps:
1 Click Start, and then click Run. The Run dialog box appears.
2 In the Open box, type cmd, and then click OK to open a command prompt.
3 At the command prompt, change the directory path to reflect the location of the windbg.exe file on your computer.

Note: the windbg.exe file is typically located in the following directory: C:\Program Files\Debugging Tools for Windows.
4 At the command prompt, type windbg –pn ImageName to attach the WinDbg debugger to the process that hosts the service that you want to debug.

NoteImageName is a placeholder for the image name of the process that hosts the service that you want to debug. The "-pn" command-line option specifies that the ImageName command-line argument is the image name of a process.
back to the top
Start the WinDbg debugger and attach to the process that hosts the service that you want to debug

1 Start Windows Explorer.
2 Locate the windbg.exe file on your computer.

Note The windbg.exe file is typically located in the following directory: C:\Program Files\Debugging Tools for Windows
3 Run the windbg.exe file to start the WinDbg debugger.
4 On the File menu, click Attach to a Process to display the Attach to Process dialog box.
5 Click to select the node that corresponds to the process that hosts the service that you want to debug, and then click OK.
6 In the dialog box that appears, click Yes to save base workspace information. Notice that you can now debug the disassembled code of your service.
Configure a service to start with the WinDbg debugger attached
You can use this method to debug services if you want to troubleshoot service-startup-related problems.
1 Configure the "Image File Execution" options. To do this, use one of the following methods:
• Method 1: Use the Global Flags Editor (gflags.exe)
a. Start Windows Explorer.
b. Locate the gflags.exe file on your computer.

Note The gflags.exe file is typically located in the following directory: C:\Program Files\Debugging Tools for Windows.
c. Run the gflags.exe file to start the Global Flags Editor.
d. In the Image File Name text box, type the image name of the process that hosts the service that you want to debug. For example, if you want to debug a service that is hosted by a process that has MyService.exe as the image name, type MyService.exe.
e. Under Destination, click to select the Image File Options option.
f. Under Image Debugger Options, click to select the Debugger check box.
g. In the Debugger text box, type the full path of the debugger that you want to use. For example, if you want to use the WinDbg debugger to debug a service, you can type a full path that is similar to the following: C:\Program

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

Files\Debugging Tools for Windows\windbg.exe
h. Click Apply, and then click OK to quit the Global Flags Editor.
• Method 2: Use Registry Editor
a. Click Start, and then click Run. The Run dialog box appears.
b. In the Open box, type regedit, and then click OK to start Registry Editor.
c. Warning If you use Registry Editor incorrectly, you may cause serious problems that may require you to reinstall your operating system. Microsoft cannot guarantee that you can solve problems that result from using Registry Editor incorrectly. Use Registry Editor at your own risk.

In Registry Editor, locate, and then right-click the following registry subkey:
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options
d. Point to New, and then click Key. In the left pane of Registry Editor, notice that New Key #1 (the name of a new registry subkey) is selected for editing.
e. Type ImageName to replace New Key #1, and then press ENTER.

Note ImageName is a placeholder for the image name of the process that hosts the service that you want to debug. For example, if you want to debug a service that is hosted by a process that has MyService.exe as the image name, type MyService.exe.
f. Right-click the registry subkey that you created in step e.
g. Point to New, and then click String Value. In the right pane of Registry Editor, notice that New Value #1, the name of a new registry entry, is selected for editing.
h. Replace New Value #1 with Debugger, and then press ENTER.
i. Right-click the Debugger registry entry that you created in step h, and then click Modify. The Edit String dialog box appears.
j. In the Value data text box, type DebuggerPath, and then click OK.

Note DebuggerPath is a placeholder for the full path of the debugger that you want to use. For example, if you want to use the WinDbg debugger to debug a service, you can type a full path that is similar to the following:
C:\Program Files\Debugging Tools for Windows\windbg.exe
2 For the debugger window to appear on your desktop, and to interact with the debugger, make your service interactive. If you do not make your service interactive, the debugger will start but you cannot see it and you cannot issue commands. To make your service interactive, use one of the following methods:
• Method 1: Use the Services console
a. Click Start, and then point to Programs.
b. On the Programs menu, point to Administrative Tools, and then click Services. The Services console appears.
c. In the right pane of the Services console, right-click ServiceName, and then click Properties.

Note ServiceName is a placeholder for the name of the service that you want to debug.
d. On the Log On tab, click to select the Allow service to interact with desktop check box under Local System account, and then click OK.
• Method 2: Use Registry Editor
a. In Registry Editor, locate, and then click the following registry subkey:
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\ServiceName
Note Replace ServiceName with the name of the service that you want to debug. For example, if you want to debug a service named MyService, locate and then click the following registry key:
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\MyService
b. Under the Name field in the right pane of Registry Editor, right-click Type, and then click Modify. The Edit DWORD Value dialog box appears.
c. Change the text in the Value data text box to the result of the binary OR operation with the binary value of the current text and the binary value, 0x00000100, as the two operands. The binary value, 0x00000100, corresponds to the SERVICE_INTERACTIVE_PROCESS constant that is defined in the WinNT.h header file on your computer. This constant specifies that a service is interactive in nature.
3 When a service starts, the service communicates to the Service Control Manager how long the service must have to start (the time-out period for the service). If the Service Control Manager does not receive a "service started" notice from the service within this time-out period, the Service Control Manager terminates the process

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

that hosts the service. This time-out period is typically less than 30 seconds. If you do not adjust this time-out period, the Service Control Manager ends the process and the attached debugger while you are trying to debug. To adjust this time-out period, follow these steps:
a. In Registry Editor, locate, and then right-click the following registry subkey:
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control
b. Point to New, and then click DWORD Value. In the right pane of Registry Editor, notice that New Value #1 (the name of a new registry entry) is selected for editing.
c. Type ServicesPipeTimeout to replace New Value #1, and then press ENTER.
d. Right-click the ServicesPipeTimeout registry entry that you created in step c, and then click Modify. The Edit DWORD Value dialog box appears.
e. In the Value data text box, type TimeoutPeriod, and then click OK

Note TimeoutPeriod is a placeholder for the value of the time-out period (in milliseconds) that you want to set for the service. For example, if you want to set the time-out period to 24 hours (86400000 milliseconds), type 86400000.
f. Restart the computer. You must restart the computer for Service Control Manager to apply this change.
4 Start your Windows service. To do this, follow these steps:
a. Click Start, and then point to Programs.
b. On the Programs menu, point to Administrative Tools, and then click Services. The Services console appears.
c. In the right pane of the Services console, right-click ServiceName, and then click Start.

Note ServiceName is a placeholder for the name of the service that you want to debug.

111. What are the different ways a method can be overloaded?

- Different parameter data types
- Different number of parameters
- Different order of parameters.

112. How do you handle data concurrency in .NET?

One of the key features of the ADO.NET DataSet is that it can be a self-contained and disconnected data store. It can contain the schema and data from several rowsets in DataTable objects as well as information about how to relate the DataTable objects-all in memory. The DataSet neither knows nor cares where the data came from, nor does it need a link to an underlying data source. Because it is data source agnostic you can pass the DataSet around networks or even serialize it to XML and pass it across the Internet without losing any of its features. However, in a disconnected model, concurrency obviously becomes a much bigger problem than it is in a connected model.

I'll explore how ADO.NET is equipped to detect and handle concurrency violations. I'll begin by discussing scenarios in which concurrency violations can occur using the ADO.NET disconnected model. Then I will walk through an ASP.NET application that handles concurrency violations by giving the user the choice to overwrite the changes or to refresh the out-of-sync data and begin editing again. Because part of managing an optimistic concurrency model can involve keeping a timestamp (rowversion) or another type of flag that indicates when a row was last updated, I will show how to implement this type of flag and how to maintain its value after each database update.

Is Your Glass Half Full?

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

There are three common techniques for managing what happens when users try to modify the same data at the same time: pessimistic, optimistic, and last-in wins. They each handle concurrency issues differently.

The pessimistic approach says: "Nobody can cause a concurrency violation with my data if I do not let them get at the data while I have it." This tactic prevents concurrency in the first place but it limits scalability because it prevents all concurrent access. Pessimistic concurrency generally locks a row from the time it is retrieved until the time updates are flushed to the database. Since this requires a connection to remain open during the entire process, pessimistic concurrency cannot successfully be implemented in a disconnected model like the ADO.NET DataSet, which opens a connection only long enough to populate the DataSet then releases and closes, so a database lock cannot be held.

Another technique for dealing with concurrency is the last-in wins approach. This model is pretty straightforward and easy to implement-whatever data modification was made last is what gets written to the database. To implement this technique you only need to put the primary key fields of the row in the UPDATE statement's WHERE clause. No matter what is changed, the UPDATE statement will overwrite the changes with its own changes since all it is looking for is the row that matches the primary key values. Unlike the pessimistic model, the last-in wins approach allows users to read the data while it is being edited on screen. However, problems can occur when users try to modify the same data at the same time because users can overwrite each other's changes without being notified of the collision. The last-in wins approach does not detect or notify the user of violations because it does not care. However the optimistic technique does detect violations.

In optimistic concurrency models, a row is only locked during the update to the database. Therefore the data can be retrieved and updated by other users at any time other than during the actual row update operation. Optimistic concurrency allows the data to be read simultaneously by multiple users and blocks other users less often than its pessimistic counterpart, making it a good choice for ADO.NET. In optimistic models, it is important to implement some type of concurrency violation detection that will catch any additional attempt to modify records that have already been modified but not committed. You can write your code to handle the violation by always rejecting and canceling the change request or by overwriting the request based on some business rules. Another way to handle the concurrency violation is to let the user decide what to do. The sample application that is shown in Figure 1 illustrates some of the options that can be presented to the user in the event of a concurrency violation.

Where Did My Changes Go?

When users are likely to overwrite each other's changes, control mechanisms should be put in place. Otherwise, changes could be lost. If the technique you're using is the last-in wins approach, then these types of overwrites are entirely possible.For example, imagine Julie wants to edit an employee's last name to correct the spelling. She navigates to a screen which loads the employee's information into a DataSet and has it presented to her in a Web page. Meanwhile, Scott is notified that the same employee's phone extension has changed. While Julie is correcting the employee's last name, Scott begins to correct his extension. Julie saves her changes first and then Scott saves his.Assuming that the application uses the last-in wins approach and updates the row using a SQL WHERE clause containing only the primary key's value, and assuming a change to one column requires the entire row to be updated, neither Julie nor Scott may immediatelyrealize the concurrency issue that just occurred. In this particular situation, Julie's changes were overwritten by Scott's changes because he saved last, and the last name reverted to the misspelled version.

So as you can see, even though the users changed different fields, their changes collided and caused Julie's changes to be lost. Without some sort of concurrency detection and handling, these types of overwrites can occur and even go unnoticed.When you run the sample application included in this column's download, you should open two separate instances of Microsoft® Internet Explorer. When I generated the conflict, I opened two instances to simulate two users with two separate sessions so that a concurrency violation would occur in the sample

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

application. When you do this, be careful not to use Ctrl+N because if you open one instance and then use the Ctrl+N technique to open another instance, both windows will share the same session.

Detecting Violations

The concurrency violation reported to the user in Figure 1 demonstrates what can happen when multiple users edit the same data at the same time. In Figure 1, the user attempted to modify the first name to "Joe" but since someone else had already modified the last name to "Fuller III," a concurrency violation was detected and reported. ADO.NET detects a concurrency violation when a DataSet containing changed values is passed to a SqlDataAdapter's Update method and no rows are actually modified. Simply using the primary key (in this case the EmployeeID) in the UPDATE statement's WHERE clause will not cause a violation to be detected because it still updates the row (in fact, this technique has the same outcome as the last-in wins technique). Instead, more conditions must be specified in the WHERE clause in order for ADO.NET to detect the violation.

The key here is to make the WHERE clause explicit enough so that it not only checks the primary key but that it also checks for another appropriate condition. One way to accomplish this is to pass in all modifiable fields to the WHERE clause in addition to the primary key. For example, the application shown in Figure 1 could have its UPDATE statement look like the stored procedure that's shown in Figure 2.

Notice that in the code in Figure 2 nullable columns are also checked to see if the value passed in is NULL. This technique is not only messy but it can be difficult to maintain by hand and it requires you to test for a significant number of WHERE conditions just to update a row. This yields the desired result of only updating rows where none of the values have changed since the last time the user got the data, but there are other techniques that do not require such a huge WHERE clause.

Another way to make sure that the row is only updated if it has not been modified by another user since you got the data is to add a timestamp column to the table. The SQL Server(tm) TIMESTAMP datatype automatically updates itself with a new value every time a value in its row is modified. This makes it a very simple and convenient tool to help detect concurrency violations.

A third technique is to use a DATETIME column in which to track changes to its row. In my sample application I added a column called LastUpdateDateTime to the Employees table.

ALTER TABLE Employees ADD LastUpdateDateTime DATETIME

There I update the value of the LastUpdateDateTime field automatically in the UPDATE stored procedure using the built-in SQL Server GETDATE function.

The binary TIMESTAMP column is simple to create and use since it automatically regenerates its value each time its row is modified, but since the DATETIME column technique is easier to display on screen and demonstrate when the change was made, I chose it for my sample application. Both of these are solid choices, but I prefer the TIMESTAMP technique since it does not involve any additional code to update its value.

Retrieving Row Flags

One of the keys to implementing concurrency controls is to update the timestamp or datetime field's value back into the DataSet. If the same user wants to make more modifications, this updated value is reflected in the DataSet so it can be used again. There are a few different ways to do this. The fastest is using output parameters within the stored procedure. (This should only return if @@ROWCOUNT equals 1.) The next fastest involves selecting the row again after the UPDATE within the stored procedure. The slowest involves selecting the row from another SQL statement or stored procedure from the SqlDataAdapter's RowUpdated event.

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

I prefer to use the output parameter technique since it is the fastest and incurs the least overhead. Using the RowUpdated event works well, but it requires me to make a second call from the application to the database. The following code snippet adds an output parameter to the SqlCommand object that is used to update the Employee information:

```
oUpdCmd.Parameters.Add(new SqlParameter("@NewLastUpdateDateTime",

  SqlDbType.DateTime, 8, ParameterDirection.Output,

  false, 0, 0, "LastUpdateDateTime", DataRowVersion.Current, null));

oUpdCmd.UpdatedRowSource = UpdateRowSource.OutputParameters;
```

The output parameter has its sourcecolumn and sourceversion arguments set to point the output parameter's return value back to the current value of the LastUpdateDateTime column of the DataSet. This way the updated DATETIME value is retrieved and can be returned to the user's .aspx page.

Saving Changes

Now that the Employees table has the tracking field (LastUpdateDateTime) and the stored procedure has been created to use both the primary key and the tracking field in the WHERE clause of the UPDATE statement, let's take a look at the role of ADO.NET. In order to trap the event when the user changes the values in the textboxes, I created an event handler for the TextChanged event for each TextBox control:

```
private void txtLastName_TextChanged(object sender, System.EventArgs e)

{

    // Get the employee DataRow (there is only 1 row, otherwise I could

    // do a Find)

    dsEmployee.EmployeeRow oEmpRow =

          (dsEmployee.EmployeeRow)oDsEmployee.Employee.Rows[0];

    oEmpRow.LastName = txtLastName.Text;

    // Save changes back to Session

    Session["oDsEmployee"] = oDsEmployee;

}
```

This event retrieves the row and sets the appropriate field's value from the TextBox. (Another way of getting the changed values is to grab them when the user clicks the Save button.) Each TextChanged event executes after the Page_Load event fires on a postback, so assuming the user changed the first and last names, when the user clicks the Save button, the events could fire in this order: Page_Load, txtFirstName_TextChanged, txtLastName_TextChanged, and btnSave_Click.

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

The Page_Load event grabs the row from the DataSet in the Session object; the TextChanged events update the DataRow with the new values; and the btnSave_Click event attempts to save the record to the database. The btnSave_Click event calls the SaveEmployee method (shown in Figure 3) and passes it a bLastInWins value of false since we want to attempt a standard save first. If the SaveEmployee method detects that changes were made to the row (using the HasChanges method on the DataSet, or alternatively using the RowState property on the row), it creates an instance of the Employee class and passes the DataSet to its SaveEmployee method. The Employee class could live in a logical or physical middle tier. (I wanted to make this a separate class so it would be easy to pull the code out and separate it from the presentation logic.)

Notice that I did not use the GetChanges method to pull out only the modified rows and pass them to the Employee object's Save method. I skipped this step here since there is only one row. However, if there were multiple rows in the DataSet's DataTable, it would be better to use the GetChanges method to create a DataSet that contains only the modified rows.

If the save succeeds, the Employee.SaveEmployee method returns a DataSet containing the modified row and its newly updated row version flag (in this case, the LastUpdateDateTime field's value). This DataSet is then merged into the original DataSet so that the LastUpdateDateTime field's value can be updated in the original DataSet. This must be done because if the user wants to make more changes she will need the current values from the database merged back into the local DataSet and shown on screen. This includes the LastUpdateDateTime value which is used in the WHERE clause. Without this field's current value, a false concurrency violation would occur.

Reporting Violations

If a concurrency violation occurs, it will bubble up and be caught by the exception handler shown in Figure 3 in the catch block for DBConcurrencyException. This block calls the FillConcurrencyValues method, which displays both the original values in the DataSet that were attempted to be saved to the database and the values currently in the database. This method is used merely to show the user why the violation occurred. Notice that the exDBC variable is passed to the FillConcurrencyValues method. This instance of the special database concurrency exception class (DBConcurrencyException) contains the row where the violation occurred. When a concurrency violation occurs, the screen is updated to look like Figure 1.

The DataSet not only stores the schema and the current data, it also tracks changes that have been made to its data. It knows which rows and columns have been modified and it keeps track of the before and after versions of these values. When accessing a column's value via the DataRow's indexer, in addition to the column index you can also specify a value using the DataRowVersion enumerator. For example, after a user changes the value of the last name of an employee, the following lines of C# code will retrieve the original and current values stored in the LastName column:

```
string sLastName_Before = oEmpRow["LastName", DataRowVersion.Original];

string sLastName_After = oEmpRow["LastName", DataRowVersion.Current];
```

The FillConcurrencyValues method uses the row from the DBConcurrencyException and gets a fresh copy of the same row from the database. It then displays the values using the DataRowVersion enumerators to show the original value of the row before the update and the value in the database alongside the current values in the textboxes.

User's Choice

Once the user has been notified of the concurrency issue, you could leave it up to her to decide how to handle it. Another alternative is to code a specific way to deal with concurrency, such as always handling the exception to let the user know (but refreshing the data from the database). In this sample application I let the user decide what to do next. She can cancel changes, cancel and reload from the database, save changes, or save anyway.

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

The option to cancel changes simply calls the RejectChanges method of the DataSet and rebinds the DataSet to the controls in the ASP.NET page. The RejectChanges method reverts the changes that the user made back to its original state by setting all of the current field values to the original field values. The option to cancel changes and reload the data from the database also rejects the changes but additionally goes back to the database via the Employee class in order to get a fresh copy of the data before rebinding to the control on the ASP.NET page.

The option to save changes attempts to save the changes but will fail if a concurrency violation is encountered. Finally, I included a "save anyway" option. This option takes the values the user attempted to save and uses the last-in wins technique, overwriting whatever is in the database. It does this by calling a different command object associated with a stored procedure that only uses the primary key field (EmployeeID) in the WHERE clause of the UPDATE statement. This technique should be used with caution as it will overwrite the record.

If you want a more automatic way of dealing with the changes, you could get a fresh copy from the database. Then overwrite just the fields that the current user modified, such as the Extension field. That way, in the example I used the proper LastName would not be overwritten. Use this with caution as well, however, because if the same field was modified by both users, you may want to just back out or ask the user what to do next. What is obvious here is that there are several ways to deal with concurrency violations, each of which must be carefully weighed before you decide on the one you will use in your application.

Wrapping It Up

Setting the SqlDataAdapter's ContinueUpdateOnError property tells the SqlDataAdapter to either throw an exception when a concurrency violation occurs or to skip the row that caused the violation and to continue with the remaining updates. By setting this property to false (its default value), it will throw an exception when it encounters a concurrency violation. This technique is ideal when only saving a single row or when you are attempting to save multiple rows and want them all to commit or all to fail.

113. What is difference between MetaData and Manifest?

Metadata and Manifest forms an integral part of an assembly (dll / exe) in .net framework. Out of which Metadata is a mandatory component, which as the name suggests gives the details about various components of IL code viz: Methods, properties, fields, class etc.

Essentially Metadata maintains details in form of tables like Methods Metadata tables, Properties Metadata tables, which maintains the list of given type and other details like access specifier, return type etc.

Now Manifest is a part of metadata only, fully called as “manifest metadata tables”, it contains the details of the references needed by the assembly of any other external assembly / type; it could be a custom assembly or standard System namespace.

Now for an assembly that can independently exists and used in the .Net world both the things (Metadata with Manifest) are mandatory , so that it can be fully described assembly and can be ported anywhere without any system dependency . Essentially .Net framework can read all assembly related information from assembly itself at runtime.

But for .Net modules, that can’t be used independently, until they are being packaged as a part of an assembly, they don’t contain Manifest but their complete structure is defined by their respective metadata.

Ultimately. .Net modules use Manifest Metadata tables of parent assembly which contain them.

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

114. What are jagged array?

A jagged array is an array whose elements are arrays. The elements of jagged array can be of different dimensions and sizes. A jagged array is sometimes called as “array-of-arrays”. It is called jagged because each of its rows is of different size so the final or graphical representation is not a square.

When you create a jagged array you declare the number of rows in your array. Each row will hold an array that will be on any length. Before filling the values in the inner arrays you must declare them.

Jagged array declaration in C#:

For e.g. : int [] [] myJaggedArray = new int [3][];

Declaration of inner arrays:

```
myJaggedArray[0] = new int[5] ;   // First inner array will be of length 5.
myJaggedArray[1] = new int[4] ;  // Second inner array will be of length 4.
myJaggedArray[2] = new int[3] ;   // Third inner array will be of length 3.
```

Now to access third element of second row we write:

```
int value = myJaggedArray[1][2];
```

Note that while declaring the array the second dimension is not supplied because this you will declare later on in the code.

Jagged array are created out of single dimensional arrays so be careful while using them. Don’t confuse it with multi-dimensional arrays because unlike them jagged arrays are not rectangular arrays.

115. Who host CLR? How windows identify where running assembly is managed or not?

MSCorEE is unmanaged dll is system32 folder of windows. It identifies whether runtime exe or dll is managed or not. It reads PE Header of managed assembly then it redirect its execution to particular framework that assembly is referring. It loads CLR for further execution.

116. Conceptual view of DotNet 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:

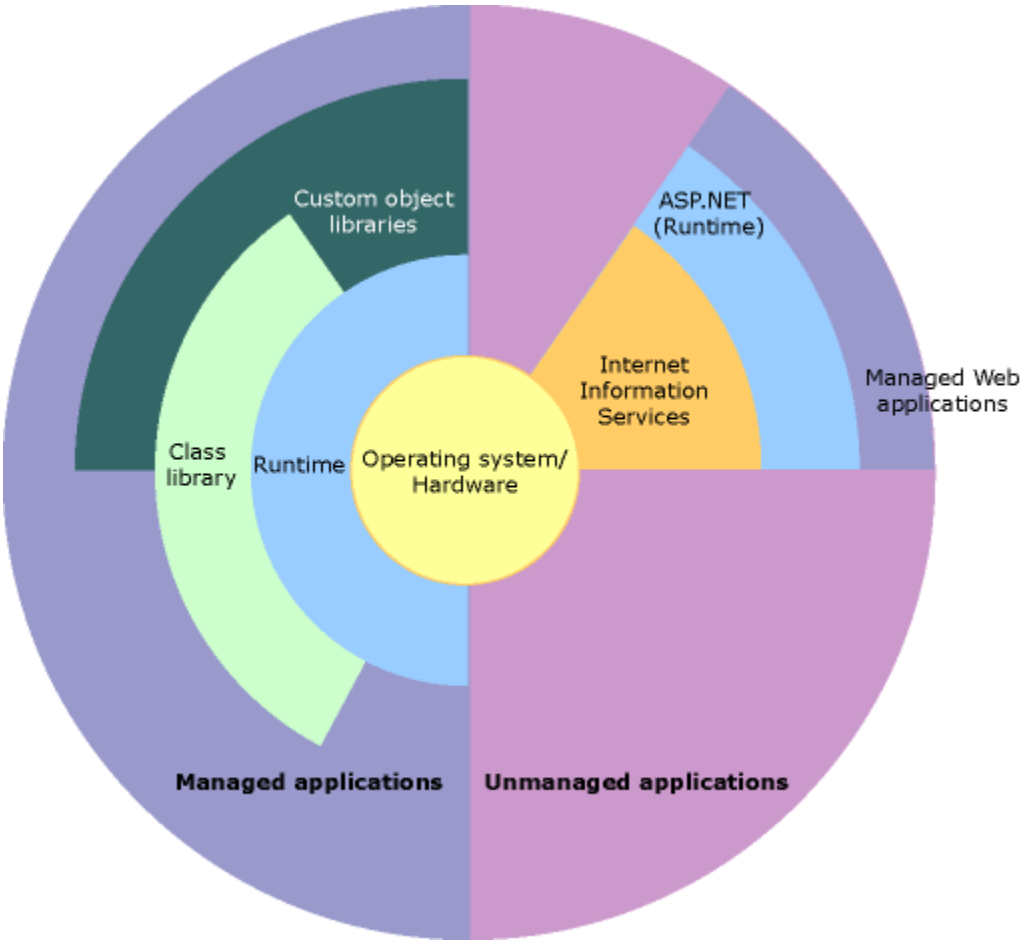
- To provide a consistent object-oriented programming environment whether object code is stored and executed locally, executed locally but Internet-distributed, or executed remotely.
- To provide a code-execution environment that minimizes software deployment and versioning conflicts.
- To provide a 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.

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

- 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

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.



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.

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

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.

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

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

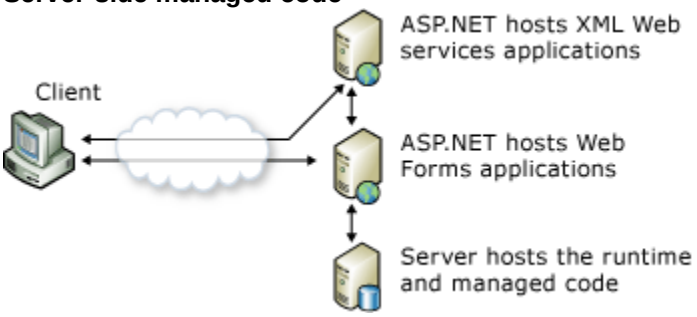
applications that once needed to be installed on a user's system can now be deployed through the Web. Your applications can implement the features of a local application while being deployed like a Web page.

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

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

117. Explain Delegates?

A delegate is a type that references a method. Once a delegate is assigned a method, it behaves exactly like that method. The delegate method can be used like any other method, with parameters and a return value, as in this example:

```
C#
Copy Code
public delegate int PerformCalculation(int x, int y);
```

Any method that matches the delegate's signature, which consists of the return type and parameters, can be assigned to the delegate. This makes is possible to programmatically change method calls, and also plug new code into existing classes. As long as you know the delegate's signature, you can assign your own delegated method.

This ability to refer to a method as a parameter makes delegates ideal for defining callback methods. For example, a sort algorithm could be passed a reference to the method that compares two objects. Separating the comparison code allows the algorithm to be written in a more general way.

Delegates Overview

Delegates have the following properties:

- Delegates are similar to C++ function pointers, but are type safe.
- Delegates allow methods to be passed as parameters.
- Delegates can be used to define callback methods.
- Delegates can be chained together; for example, multiple methods can be called on a single event.
- Methods don't need to match the delegate signature exactly. C# version 2.0 introduces the concept of Anonyms method which permit code blocks to be passed as parameters in place of a separately defined method.

Using Delegates (C# Programming Guide)

A delegate is a type that safely encapsulates a method, similar to a function pointer in C and C++. Unlike C function pointers, delegates are object-oriented, type safe, and secure. The type of a delegate is defined by the name of the delegate. The following example declares a delegate named Del that can encapsulate a method that takes a string as an argument and returns void:

```
C#
Copy Code
public delegate void Del(string message);
```

A delegate object is normally constructed by providing the name of the method the delegate will wrap, or with an anonymous Method. Once a delegate is instantiated, a method call made to the delegate will be passed by the delegate

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

to that method. The parameters passed to the delegate by the caller are passed to the method, and the return value, if any, from the method is returned to the caller by the delegate. This is known as invoking the delegate. An instantiated delegate can be invoked as if it were the wrapped method itself. For example:

```
C#
Copy Code
// Create a method for a delegate. public static void DelegateMethod(string message) {
System.Console.WriteLine(message); }
C#
Copy Code
// Instantiate the delegate. Del handler = DelegateMethod; // Call the delegate. handler("Hello World");
```

Delegate types are derived from the Delegate class in the .NET Framework. Delegate types are sealed—they cannot be derived from— and it is not possible to derive custom classes from **Delegate**. Because the instantiated delegate is an object, it can be passed as a parameter, or assigned to a property. This allows a method to accept a delegate as a parameter, and call the delegate at some later time. This is known as an asynchronous callback, and is a common method of notifying a caller when a long process has completed. When a delegate is used in this fashion, the code using the delegate does not need any knowledge of the implementation of the method being used. The functionality is similar to the encapsulation interfaces provide. For more information, see When to Use Delegates Instead of Interfaces.

Another common use of callbacks is defining a custom comparison method and passing that delegate to a sort method. It allows the caller's code to become part of the sort algorithm. The following example method uses the Del type as a parameter:

```
C#
Copy Code
public void MethodWithCallback(int param1, int param2, Del callback) { callback("The number is: " + (param1 +
param2).ToString()); }
```

You can then pass the delegate created above to that method:

```
C#
Copy Code
MethodWithCallback(1, 2, handler);
```

and receive the following output to the console:

The number is: 3

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

Using the delegate as an abstraction, MethodWithCallback does not need to call the console directly—it does not have to be designed with a console in mind. What MethodWithCallback does is simply prepare a string and pass the string to another method. This is especially powerful since a delegated method can use any number of parameters.

When a delegate is constructed to wrap an instance method, the delegate references both the instance and the method. A delegate has no knowledge of the instance type aside from the method it wraps, so a delegate can refer to any type of object as long as there is a method on that object that matches the delegate signature. When a delegate is constructed to wrap a static method, it only references the method. Consider the following declarations:

```
C#
Copy Code
public class MethodClass { public void Method1(string message) { } public void Method2(string message) { } }
```

Along with the static DelegateMethod shown previously, we now have three methods that can be wrapped by a Del instance.

A delegate can call more than one method when invoked. This is referred to as multicasting. To add an extra method to the delegate's list of methods—the invocation list—simply requires adding two delegates using the addition or addition assignment operators ('+' or '+='). For example:

```
C#
Copy Code
MethodClass obj = new MethodClass(); Del d1 = obj.Method1; Del d2 = obj.Method2; Del d3 = DelegateMethod; //Both types of assignment are valid. Del allMethodsDelegate = d1 + d2; allMethodsDelegate += d3;
```

At this point allMethodsDelegate contains three methods in its invocation list—Method1, Method2, and DelegateMethod. The original three delegates, d1, d2, and d3, remain unchanged. When allMethodsDelegate is invoked, all three methods are called in order. If the delegate uses reference parameters, the reference is passed sequentially to each of the three methods in turn, and any changes by one method are visible to the next method. When any of the methods throws an exception that is not caught within the method, that exception is passed to the caller of the delegate and no subsequent methods in the invocation list are called. If the delegate has a return value and/or out parameters, it returns the return value and parameters of the last method invoked. To remove a method from the invocation list, use the decrement or decrement assignment operator ('-' or '-='). For example:

```
C#
Copy Code
//remove Method1 allMethodsDelegate -= d1; // copy AllMethodsDelegate while removing d2 Del oneMethodDelegate = allMethodsDelegate - d2;
```


Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

Because delegate types are derived from **System.Delegate**, the methods and properties defined by that class can be called on the delegate. For example, to find the number of methods in a delegate's invocation list, you may write:

C#
Copy Code
```
int invocationCount = d1.GetInvocationList().GetLength(0);
```

Delegates with more than one method in their invocation list derive from MulticastDelegate, which is a subclass of **System.Delegate**. The above code works in either case because both classes support **GetInvocationList**.

Multicast delegates are used extensively in event handling. Event source objects send event notifications to recipient objects that have registered to receive that event. To register for an event, the recipient creates a method designed to handle the event, then creates a delegate for that method and passes the delegate to the event source. The source calls the delegate when the event occurs. The delegate then calls the event handling method on the recipient, delivering the event data. The delegate type for a given event is defined by the event source. For more, see Events (C# Programming Guide).

Comparing delegates of two different types assigned at compile-time will result in a compilation error. If the delegate instances are statically of the type **System.Delegate**, then the comparison is allowed, but will return false at run time. For example:

C#
Copy Code
```
delegate void Delegate1(); delegate void Delegate2(); static void method(Delegate1 d, Delegate2 e, System.Delegate f)
{ // Compile-time error. //Console.WriteLine(d == e); // OK at compile-time. False if the run-time type of f //is not the same as that of d. System.Console.WriteLine(d == f); }
```
Anonymous Methods (C# Programming Guide)

In versions of C# previous to 2.0, the only way to declare a delegate was to use named methods. C# 2.0 introduces anonymous methods.

Creating anonymous methods is essentially a way to pass a code block as a delegate parameter. For example:

C#
Copy Code
```
// Create a handler for a click event button1.Click += delegate(System.Object o, System.EventArgs e) {
System.Windows.Forms.MessageBox.Show("Click!"); };
```

or

C#

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

Copy Code

```
// Create a delegate instance delegate void Del(int x); // Instantiate the delegate using an anonymous method Del d = delegate(int k) { /* ... */ };
```

By using anonymous methods, you reduce the coding overhead in instantiating delegates by eliminating the need to create a separate method.

For example, specifying a code block in the place of a delegate can be useful in a situation when having to create a method might seem an unnecessary overhead. A good example would be when launching a new thread. This class creates a thread and also contains the code that the thread executes, without the need for creating an additional method for the delegate.

C#

Copy Code

```
void StartThread() { System.Threading.Thread t1 = new System.Threading.Thread (delegate() { System.Console.Write("Hello, "); System.Console.WriteLine("World!"); }); t1.Start(); }
```

Remarks

The scope of the parameters of an anonymous method is the **anonymous-method-block**.

It is an error to have a jump statement, such as goto, break, or continue, inside the anonymous method block whose target is outside the block. It is also an error to have a jump statement, such as **goto**, **break**, or **continue**, outside the anonymous method block whose target is inside the block.

The local variables and parameters whose scope contain an anonymous method declaration are called outer or captured variables of the anonymous method. For example, in the following code segment, n is an outer variable:

C#

Copy Code

```
int n = 0; Del d = delegate() { System.Console.WriteLine("Copy #:{0}", ++n); };
```

Unlike local variables, the lifetime of the outer variable extends until the delegates that reference the anonymous methods are eligible for garbage collection. A reference to n is captured at the time the delegate is created.

An anonymous method cannot access the ref or out parameters of an outer scope.

No unsafe code can be accessed within the **anonymous-method-block**.

Example

The following example demonstrates the two ways of instantiating a delegate:

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

- Associating the delegate with an anonymous method.
- Associating the delegate with a named method (DoWork).

In each case, a message is displayed when the delegate is invoked.

C#
Copy Code

```
// Declare a delegate delegate void Printer(string s); class TestClass { static void Main() { // Instatiate the delegate type using an anonymous method: Printer p = delegate(string j) { System.Console.WriteLine(j); }; // Results from the anonymous delegate call: p("The delegate using the anonymous method is called."); // The delegate instantiation using a named method "DoWork": p = new Printer(TestClass.DoWork); // Results from the old style delegate call: p("The delegate using the named method is called."); } // The method associated with the named delegate: static void DoWork(string k) { System.Console.WriteLine(k); } }
```

Output

The delegate using the anonymous method is called.

The delegate using the named method is called.

When to Use Delegates Instead of Interfaces (C# Programming Guide)

Both delegates and interfaces allow a class designer to separate type declarations and implementation. A given interface can be inherited and implemented by any class or struct; a delegate can created for a method on any class, as long as the method fits the method signature for the delegate. An interface reference or a delegate can be used by an object with no knowledge of the class that implements the interface or delegate method. Given these similarities, when should a class designer use a delegate and when should they use an interface?

Use a delegate when:

- An eventing design pattern is used.
- It is desirable to encapsulate a static method.
- The caller has no need access other properties, methods, or interfaces on the object implementing the method.
- Easy composition is desired.
- A class may need more than one implementation of the method.

Use an interface when:

- There are a group of related methods that may be called.
- A class only needs one implementation of the method.

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

- The class using the interface will want to cast that interface to other interface or class types.
- The method being implemented is linked to the type or identity of the class: for example, comparison methods.

One good example of using a single-method interface instead of a delegate is IComparable or IComparable. **IComparable** declares the CompareTo method, which returns an integer specifying a less than, equal to, or greater than relationship between two objects of the same type. **IComparable** can be used as the basis of a sort algorithm, and while using a delegate comparison method as the basis of a sort algorithm would be valid, it is not ideal. Because the ability to compare belongs to the class, and the comparison algorithm doesn't change at run-time, a single-method interface is ideal.

How to: Combine Delegates (Multicast Delegates)(C# Programming Guide)

This example demonstrates how to compose multicast delegates. A useful property of delegate objects is that they can be assigned to one delegate instance to be multicast using the **+** operator. A composed delegate calls the two delegates it was composed from. Only delegates of the same type can be composed.

The **-** operator can be used to remove a component delegate from a composed delegate.

Example

C#
Copy Code

```
delegate void Del(string s); class TestClass { static void Hello(string s) { System.Console.WriteLine(" Hello, {0}!", s); } static void Goodbye(string s) { System.Console.WriteLine(" Goodbye, {0}!", s); } static void Main() { Del a, b, c, d; // Create the delegate object a that references // the method Hello: a = Hello; // Create the delegate object b that references // the method Goodbye: b = Goodbye; // The two delegates, a and b, are composed to form c: c = a + b; // Remove a from the composed delegate, leaving d, // which calls only the method Goodbye: d = c - a; System.Console.WriteLine("Invoking delegate a:"); a("A"); System.Console.WriteLine("Invoking delegate b:"); b("B"); System.Console.WriteLine("Invoking delegate c:"); c("C"); System.Console.WriteLine("Invoking delegate d:"); d("D"); } }
```

Output

```
Invoking delegate a: Hello, A! Invoking delegate b: Goodbye, B! Invoking delegate c: Hello, C! Goodbye, C! Invoking delegate d: Goodbye, D!
```

How to: Declare, Instantiate, and Use a Delegate (C# Programming Guide)

Delegates are declared as shown here:

C#
Copy Code

```
public delegate void Del<T>(T item); public void Notify(int i) { }
```

C#

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

Copy Code
```
Del<int> d1 = new Del<int>(Notify);
```

In C# 2.0, it is also possible to declare a delegate using this simplified syntax:

C#
Copy Code
```
Del<int> d2 = Notify;
```

The following example illustrates declaring, instantiating, and using a delegate. The BookDB class encapsulates a bookstore database that maintains a database of books. It exposes a method, ProcessPaperbackBooks, which finds all paperback books in the database and calls a delegate for each one. The **delegate** type used is called ProcessBookDelegate. The Test class uses this class to print out the titles and average price of the paperback books.

The use of delegates promotes good separation of functionality between the bookstore database and the client code. The client code has no knowledge of how the books are stored or how the bookstore code finds paperback books. The bookstore code has no knowledge of what processing is done on the paperback books after it finds them.

Example

C#
Copy Code
```
// A set of classes for handling a bookstore: namespace Bookstore { using System.Collections; // Describes a book in the book list: public struct Book { public string Title; // Title of the book. public string Author; // Author of the book. public decimal Price; // Price of the book. public bool Paperback; // Is it paperback? public Book(string title, string author, decimal price, bool paperBack) { Title = title; Author = author; Price = price; Paperback = paperBack; } } // Declare a delegate type for processing a book: public delegate void ProcessBookDelegate(Book book); // Maintains a book database. public class BookDB { // List of all books in the database: ArrayList list = new ArrayList(); // Add a book to the database: public void AddBook(string title, string author, decimal price, bool paperBack) { list.Add(new Book(title, author, price, paperBack)); } // Call a passed-in delegate on each paperback book to process it: public void ProcessPaperbackBooks(ProcessBookDelegate processBook) { foreach (Book b in list) { if (b.Paperback) // Calling the delegate: processBook(b); } } } } // Using the Bookstore classes: namespace BookTestClient { using Bookstore; // Class to total and average prices of books: class PriceTotaller { int countBooks = 0; decimal priceBooks = 0.0m; internal void AddBookToTotal(Book book) { countBooks += 1; priceBooks += book.Price; } internal decimal AveragePrice() { return priceBooks / countBooks; } } // Class to test the book database: class TestBookDB { // Print the title of the book. static void PrintTitle(Book b) { System.Console.WriteLine(" {0}", b.Title); } // Execution starts here. static void Main() { BookDB bookDB = new BookDB(); // Initialize the database with some books: AddBooks(bookDB); // Print all the titles of paperbacks: System.Console.WriteLine("Paperback Book Titles:"); // Create a new delegate object associated with the static // method Test.PrintTitle: bookDB.ProcessPaperbackBooks(PrintTitle); // Get the average price of a paperback by using // a PriceTotaller object: PriceTotaller totaller = new PriceTotaller(); // Create a new delegate object associated
```


Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

```
with the nonstatic // method AddBookToTotal on the object totaller:
bookDB.ProcessPaperbackBooks(totaller.AddBookToTotal); System.Console.WriteLine("Average Paperback Book Price: ${0:#.##}", totaller.AveragePrice()); } // Initialize the book database with some test books: static void AddBooks(BookDB bookDB) { bookDB.AddBook("The C Programming Language", "Brian W. Kernighan and Dennis M. Ritchie", 19.95m, true); bookDB.AddBook("The Unicode Standard 2.0", "The Unicode Consortium", 39.95m, true); bookDB.AddBook("The MS-DOS Encyclopedia", "Ray Duncan", 129.95m, false); bookDB.AddBook("Dogbert's Clues for the Clueless", "Scott Adams", 12.00m, true); } } }
```

Output

```
Paperback Book Titles: The C Programming Language The Unicode Standard 2.0 Dogbert's Clues for the Clueless
Average Paperback Book Price: $23.97
```

Robust Programming

- Declaring a delegate.

  The following statement:

  C#

  Copy Code

  ```
  public delegate void ProcessBookDelegate(Book book);
  ```

  declares a new delegate type. Each delegate type describes the number and types of the arguments, and the type of the return value of methods that it can encapsulate. Whenever a new set of argument types or return value type is needed, a new delegate type must be declared.

- Instantiating a delegate.

  Once a delegate type has been declared, a delegate object must be created and associated with a particular method. In the example above, this is done by passing the PrintTitle method to the ProcessPaperbackBooks method, like this:

  C#

  Copy Code

  ```
  bookDB.ProcessPaperbackBooks(PrintTitle);
  ```

  This creates a new delegate object associated with the static method Test.PrintTitle. Similarly, the non-static method AddBookToTotal on the object totaller is passed like this:

  C#

  Copy Code

  ```
  bookDB.ProcessPaperbackBooks(totaller.AddBookToTotal);
  ```


Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

In both cases a new delegate object is passed to the ProcessPaperbackBooks method.

Once a delegate is created, the method it is associated with never changes; delegate objects are immutable.

- Calling a delegate.

  Once a delegate object is created, the delegate object is typically passed to other code that will call the delegate. A delegate object is called by using the name of the delegate object, followed by the parenthesized arguments to be passed to the delegate. An example of a delegate call is:

C#

Copy Code

```
processBook(b);
```

A delegate can either be called synchronously, as in this example, or asynchronously by using **BeginInvoke** and **EndInvoke** methods.

118. Explaint about Events?

Events and Delegates

An event is a message sent by an object to signal the occurrence of an action. The action could be caused by user interaction, such as a mouse click, or it could be triggered by some other program logic. The object that raises the event is called the event sender. The object that captures the event and responds to it is called the event receiver.

In event communication, the event sender class does not know which object or method will receive (handle) the events it raises. What is needed is an intermediary (or pointer-like mechanism) between the source and the receiver. The .NET Framework defines a special type (Delegate) that provides the functionality of a function pointer.

A delegate is a class that can hold a reference to a method. Unlike other classes, a delegate class has a signature, and it can hold references only to methods that match its signature. A delegate is thus equivalent to a type-safe function pointer or a callback. While delegates have other uses, the discussion here focuses on the event handling functionality of delegates. A delegate declaration is sufficient to define a delegate class. The declaration supplies the signature of the delegate, and the common language runtime provides the implementation. The following example shows an event delegate declaration.

C#

Copy Code

```
public delegate void AlarmEventHandler(object sender, AlarmEventArgs e);
```

Visual Basic

Copy Code

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

```
Public Delegate Sub AlarmEventHandler(sender As Object, e As AlarmEventArgs)
```

The syntax is similar to that of a method declaration; however, the **delegate** keyword informs the compiler that AlarmEventHandler is a delegate type. By convention, event delegates in the .NET Framework have two parameters, the source that raised the event and the data for the event.

An instance of the AlarmEventHandler delegate can bind to any method that matches its signature, such as the AlarmRang method of the WakeMeUp class shown in the following example.

C#

Copy Code

```
public class WakeMeUp { // AlarmRang has the same signature as AlarmEventHandler. public void AlarmRang(object sender, AlarmEventArgs e) {...}; ... }
```

Visual Basic

Copy Code

```
Public Class WakeMeUp ' AlarmRang has the same signature as AlarmEventHandler. Public Sub AlarmRang(sender As Object, e As AlarmEventArgs) ... End Sub ... End Class
```

Custom event delegates are needed only when an event generates event data. Many events, including some user-interface events such as mouse clicks, do not generate event data. In such situations, the event delegate provided in the class library for the no-data event, System.EventHandler, is adequate. Its declaration follows.

C#

Copy Code

```
delegate void EventHandler(object sender, EventArgs e);
```

Visual Basic

Copy Code

```
Public Delegate Sub EventHandler(sender As Object, e As EventArgs)
```

Event delegates are multicast, which means that they can hold references to more than one event handling method. For details, see Delegate. Delegates allow for flexibility and fine-grain control in event handling. A delegate acts as an event dispatcher for the class that raises the event by maintaining a list of registered event handlers for the event.

119. How to: Connect Event Handler Methods to Events

To consume events defined in another class, you must define and register an event handler. The event handler must have the same method signature as the delegate declared for the event. You register your event handler by adding the handler to the event. After you have added your event handler to the event, the method is called whenever the class raises the event.

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

For a complete sample that illustrates raising and handling events, see How to: Raise and Consume Events.

To add an event handler method for an event

1. Define an event handler method with the same signature as the event delegate.

```
C#

Copy Code
public class WakeMeUp { // AlarmRang has the same signature as AlarmEventHandler. public void
AlarmRang(object sender, AlarmEventArgs e) {...}; ... }
```

```
Visual Basic

Copy Code
Public Class WakeMeUp ' AlarmRang has the same signature as AlarmEventHandler. Public Sub
AlarmRang(sender As Object, e As AlarmEventArgs) ... End Sub ... End Class
```

2. Create an instance of the delegate, using a reference to the event handler method. When the delegate instance is called, it in turn calls the event handler method.

```
C#

Copy Code
// Create an instance of WakeMeUp. WakeMeUp w = new WakeMeUp(); // Instantiate the event delegate.
AlarmEventHandler alhandler = new AlarmEventHandler(w.AlarmRang);
```

```
Visual Basic

Copy Code
' Create an instance of WakeMeUp. Dim w As New WakeMeUp() ' Instantiate the event delegate. Dim alhandler
As AlarmEventHandler = AddressOf w.AlarmRang
```

3. Add the delegate instance to the event. When the event is raised, the delegate instance and its associated event handler method is called.

```
C#

Copy Code
// Instantiate the event source. AlarmClock clock = new AlarmClock(); // Add the delegate instance to the event.
clock.Alarm += alhandler;
```


Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

```
Visual Basic

Copy Code
' Instantiate the event source. Dim clock As New AlarmClock() ' Add the delegate to the event. AddHandler
clock.Alarm, AddressOf w.AlarmRang
```

Consuming Events

To consume an event in an application, you must provide an event handler (an event-handling method) that executes program logic in response to the event and register the event handler with the event source. This process is referred to as event wiring. The visual designers for Windows Forms and Web Forms provide rapid application development (RAD) tools that simplify or hide the details of event wiring.

This topic describes the general pattern of handling events. For an overview of the event model in the .NET Framework, see Events and Delegates. For more information about the event model in Windows Forms, see How to: Consume Events in a Windows Forms Application. For more information about the event model in Web Forms, see How to: Consume Events in a Web Forms Application.

The Event Pattern

The details of event wiring differ in Windows Forms and Web Forms because of the different levels of support provided by different RAD tools. However, both scenarios follow the same event pattern, which has the following characteristics:

- A class that raises an event named *EventName* has the following member:

```
C#

Copy Code
public event EventNameEventHandler EventName;
```

```
Visual Basic

Copy Code
Public Event EventName As EventNameEventHandler
```

- The event delegate for the *EventName* event is *EventName***EventHandler**, with the following signature:

```
C#

Copy Code
public delegate void EventNameEventHandler(object sender, EventNameEventArgs e);
```


Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

Visual Basic

Copy Code

```
Public Delegate Sub EventNameEventHandler(sender As Object, e As EventNameEventArgs)
```

To consume the *EventName* event, your event handler must have the same signature as the event delegate:

C#

Copy Code

```
void EventHandler(object sender, EventNameEventArgs e) {}
```

Visual Basic

Copy Code

```
Sub EventHandler(sender As Object, e As EventNameEventArgs)
```

Note

An event delegate in the .NET Framework is named *EventName***EventHandler**, while the term event handler in the documentation refers to an event-handling method. The logic behind the naming scheme is that an *EventName***EventHandler** delegate points to the event handler (the method) that actually handles the event.

When an event does not have any associated data, the class raising the event uses System.EventHandler as the delegate and System.EventArgs for the event data. Events that have associated data use classes that derive from **EventArgs** for the event data type, and the corresponding event delegate type. For example, if you want to handle a **MouseUp** event in a Windows Forms application, the event data class is **MouseEventArgs** and the event delegate is **MouseEventHandler**. Note that several mouse events use a common class for event data and a common event delegate, so the naming scheme does not exactly match the convention described above. For the mouse events, your event handler must have the following signature:

C#

Copy Code

```
void Mouse_Moved(object sender, MouseEventArgs e){}
```

Visual Basic

Copy Code

```
Sub Mouse_Moved(sender As Object, e As MouseEventArgs)
```

The sender and event argument parameters supply additional details about the mouse event to the event handler. The sender object indicates what raised the event. The **MouseEventArgs** parameter provides details on the mouse movement that raised the event. Many event sources provide additional data for the event, and many event handlers use the event-specific data in processing the event. For an example that demonstrates raising and handling events with event-specific data, see How to: Raise and Consume Events.

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

120. How to: Consume Events in a Web Forms Application

A common scenario in Web Forms applications is to populate a Web page with controls, and then perform a specific action based on which control the user clicks. For example, a System.Web.UI.WebControls.Button control raises an event when the user clicks on it in the Web page. By handling the event, your application can perform the appropriate application logic for that button click.

For information about the Web Forms programming model, see Programming Web Forms.

To handle a button click event on a Web page

1. Create a Web Forms page (ASP.NET page) that has a Button control.

```
Copy Code
<asp:Button id = "Button" Text = "Click Me" runat = server/>
```

2. Define an event handler that matches the **Click** event delegate signature. The **Click** event uses the EventHandler class for the delegate type and the EventArgs class for the event data.

```
C#

Copy Code
void Button_Click(object sender, EventArgs e) {...}
```

```
Visual Basic

Copy Code
Sub Button_Click(sender As Object, e As EventArgs) ... End Sub
```

3. Set the **OnClick** attribute in the **Button** element to the event handler method.

```
Copy Code
<asp:Button id = "Button" OnClick = "Button_Click" Text = "Click Me" runat = server/>
```

Note

A Web Forms application developer can wire the event declaratively as shown without directly working with the delegate. The ASP.NET page framework generates code that creates an instance of **EventHandler** that references Button_Click and adds this delegate instance to the **Click** event of the **Button** instance.

Example

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

The following Web Forms page handles the **Click** event of **Button** to change the background color of **TextBox**. The elements in bold in this example show the event handler code and how the event handler is wired to the **Click** event of **Button**.

Security Note

This example has a text box that accepts user input, which is a potential security threat. By default, ASP.NET Web pages validate that user input does not include script or HTML elements. For more information, see Script Exploits Overview.

Copy Code

```
[C#] <html> <script language="C#" runat=server> private void Button_Click(object sender, EventArgs e){ Box.BackColor = System.Drawing.Color.LightGreen; } </script> <body> <form method="POST" action="Events.aspx" runat=server> Click the button, and notice the color of the text box.<br><br> <asp:TextBox id = "Box" Text = "Hello" BackColor = "Cyan" runat=server/> <br><br> <asp:Button id = "Button" OnClick = "Button_Click" Text = "Click Me" runat = server/> </form> </body> </html>
```

Visual Basic

Copy Code

```
<html> <script language="VB" runat=server> Private Sub Button_Click(sender As Object, e As EventArgs) Box.BackColor = System.Drawing.Color.LightGreen End Sub </script> <body> <form method="POST" action="Events.aspx" runat=server> Click the button, and notice the color of the text box.<br><br> <asp:TextBox id = "Box" Text = "Hello" BackColor = "Cyan" runat=server/> <br><br> <asp:Button id = "Button" OnClick = "Button_Click" Text = "Click Me" runat = server/> </form> </body> </html>
```

Compiling the Code

To see how event handling works in Web Forms, save the example page to a file with an .aspx extension (which indicates that the file is an ASP.NET page) and deploy it anywhere in your IIS virtual root directory.

How to: Consume Events in a Windows Forms Application

A common scenario in Windows Forms applications is to display a form with controls, and then perform a specific action based on which control the user clicks. For example, a Button control raises an event when the user clicks it in the form. By handling the event, your application can perform the appropriate application logic for that button click.

For more information about Windows Forms, see Getting Started with Windows Forms.

To handle a button click event on a Windows Form

1. Create a Windows Form that has a **Button** control.

   C#

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

```
Copy Code
private Button button;
```

2. Define an event handler that matches the **Click** event delegate signature. The **Click** event uses the EventHandler class for the delegate type and the EventArgs class for the event data.

```
C#

Copy Code
void Button_Click(object sender, EventArgs e) {...}
```

3. Add the event handler method to the **Click** event of the **Button**.

```
C#

Copy Code
button.Click += new EventHandler(this.Button_Click);
```

Example

The following code example handles the **Click** event of a **Button** to change the background color of **TextBox**. The elements in bold show the event handler and how it is wired to the **Click** event of the **Button**.

The code in this example was written without using a visual designer (such as Visual Studio 2005) and contains only essential programming elements. If you use a designer, it will generate additional code.

```
C#
Copy Code
using System;
using System.ComponentModel;
using System.Windows.Forms;
using System.Drawing;
public class MyForm : Form
{
        private TextBox box;
        private Button button;
        public MyForm() : base()
        {
         box = new TextBox();
         box.BackColor = System.Drawing.Color.Cyan;
         box.Size = new Size(100,100);
         box.Location = new Point(50,50);
         box.Text = "Hello"; button = new Button();
```


Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

```
            button.Location = new Point(50,100);
            button.Text = "Click Me";
           // To wire the event, create
           // a delegate instance and add it to the Click event.
            button.Click += new EventHandler(this.Button_Click);
            Controls.Add(box); Controls.Add(button);
            }
           // The event handler.
            private void Button_Click(object sender, EventArgs e)
             {
            box.BackColor = System.Drawing.Color.Green;
              }
            // The STAThreadAttribute indicates that Windows Forms uses the // single-threaded apartment model.
           [STAThreadAttribute]
           public static void Main(string[] args)
            {
              Application.Run(new MyForm());
           }
 }
```

Compiling the Code

Save the preceding code to a file (with a .cs extension for a C# file and .vb for Visual Basic 2005), compile, and execute. For example, if the source file is named WinEvents.cs (or WinEvents.vb), run the following command:

```
C#
Copy Code
csc /r:System.DLL /r:System.Windows.Forms.DLL /r:System.Drawing.DLL WinEvents.cs
```

Your executable file will be named WinEvents.exe.

Raising an Event

Event functionality is provided by three interrelated elements: a class that provides event data, an event delegate, and the class that raises the event. The .NET Framework has a convention for naming classes and methods related to events. If you want your class to raise an event named EventName, you need the following elements:

- A class that holds event data, named EventNameEventArgs. This class must derive from System.EventArgs.
- A delegate for the event, named EventNameEventHandler.
- A class that raises the event. This class must provide the event declaration (EventName) and a method that raises the event (OnEventName).

The event data class and the event delegate class might already be defined in the .NET Framework class library or in a third-party class library. In that case, you do not have to define those classes. For example, if your event does not use custom data, you can use System.EventArgs for your event data and System.EventHandler for your delegate.

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

You define an event member in your class using the event keyword. When the compiler encounters an event keyword in your class, it creates a private member such as:

```
Copy Code
private EventNameHandler eh = null;
```

The compiler also creates the two public methods add_EventName and remove_EventName. These methods are event hooks that allow delegates to be combined or removed from the event delegate eh. The details are hidden from the programmer.
Once you have defined your event implementation, you must determine when to raise the event. You raise the event by calling the protected OnEventName method in the class that defined the event, or in a derived class. The OnEventName method raises the event by invoking the delegates, passing in any event-specific data. The delegate methods for the event can perform actions for the event or process the event-specific data.
How to: Implement Events in Your Class

The following procedures describe how to implement an event in a class. The first procedure implements an event that does not have associated data; it uses the classes System.EventArgs and System.EventHandler for the event data and delegate handler. The second procedure implements an event with custom data; it defines custom classes for the event data and the event delegate handler.
For a complete sample that illustrates raising and handling events, see How to: Raise and Consume Events.
To implement an event without event-specific data

1. Define a public event member in your class. Set the type of the event member to a System.EventHandler delegate.

```
C#

Copy Code
public class Countdown { ... public event EventHandler CountdownCompleted; }
```

1. Provide a protected method in your class that raises the event. Name the method **On***EventName*. Raise the event within the method.

```
C#

Copy Code
public class Countdown { ... public event EventHandler CountdownCompleted; protected virtual void
OnCountdownCompleted(EventArgs e) { if (CountdownCompleted != null) CountdownCompleted(this, e); } }
```

2. Determine when to raise the event in your class. Call **On***EventName* to raise the event.

```
C#

Copy Code
public class Countdown { ... public void Decrement { internalCounter = internalCounter - 1; if (internalCounter ==
0) OnCountdownCompleted(new EventArgs()); } }
```

To implement an event with event-specific data

1. Define a class that provides data for the event. Name the class *EventName***Args**, derive the class from System.EventArgs, and add any event-specific members.

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

```
C#

Copy Code
public class AlarmEventArgs : EventArgs { private readonly int nrings = 0; private readonly bool snoozePressed = false; //Constructor. public AlarmEventArgs(bool snoozePressed, int nrings) { this.snoozePressed = snoozePressed; this.nrings = nrings; } //Properties. public string AlarmText { ... } public int NumRings { ... } public bool SnoozePressed{ ... } }
```

2. Declare a delegate for the event. Name the delegate *EventName***EventHandler**.

```
public delegate void AlarmEventHandler(object sender, AlarmEventArgs e);
```

3. Define a public event member named *EventName* in your class. Set the type of the event member to the event delegate type.

```
C#
public class AlarmClock { ... public event AlarmEventHandler Alarm; }
```

4. Define a protected method in your class that raises the event. Name the method **On***EventName*. Raise the event within the method.

```
C#
public class AlarmClock { ... public event AlarmHandler Alarm; protected virtual void OnAlarm(AlarmEventArgs e) { if (Alarm != null) Alarm(this, e); } }
```

5. Determine when to raise the event in your class. Call **On***EventName* to raise the event and pass in the event-specific data using *EventName***EventArgs**.

```
Public Class AlarmClock { ... public void Start() { ... System.Threading.Thread.Sleep(300); AlarmEventArgs e = new AlarmEventArgs(false, 0); OnAlarm(e); } }
```

How to: Raise and Consume Events

The following example program demonstrates raising an event in one class and handling the event in another class. The AlarmClock class defines the public event Alarm, and provides methods to raise the event. The AlarmEventArgs class derives from EventArgs and defines the data specific to an Alarm event. The WakeMeUp class defines the method AlarmRang, which handles an Alarm event. The AlarmDriver class uses the classes together, setting the AlarmRang method of WakeMeUp to handle the Alarm event of the AlarmClock.
This example program uses concepts described in detail in Events and Delegates and Raising an Event.

```
// EventSample.cs.
//
namespace EventSample
{
  using System;
  using System.ComponentModel;

  // Class that contains the data for
  // the alarm event. Derives from System.EventArgs.
  //
  public class AlarmEventArgs : EventArgs
  {
```


Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

```
    private readonly bool snoozePressed ;
    private readonly int nrings;

    //Constructor.
    //
    public AlarmEventArgs(bool snoozePressed, int nrings)
    {
      this.snoozePressed = snoozePressed;
      this.nrings = nrings;
    }

    // The NumRings property returns the number of rings
    // that the alarm clock has sounded when the alarm event
    // is generated.
    //
public int NumRings
    {
      get { return nrings;}
    }

    // The SnoozePressed property indicates whether the snooze
    // button is pressed on the alarm when the alarm event is generated.
    //
    public bool SnoozePressed
    {
      get {return snoozePressed;}
    }

    // The AlarmText property that contains the wake-up message.
    //
    public string AlarmText
    {
      get
      {
        if (snoozePressed)
        {
          return ("Wake Up!!! Snooze time is over.");
        }
        else
        {
          return ("Wake Up!");
        }
      }
    }
  }

  // Delegate declaration.
  //
  public delegate void AlarmEventHandler(object sender, AlarmEventArgs e);

  // The Alarm class that raises the alarm event.
  //
  public class AlarmClock
  {
    private bool snoozePressed = false;
```


Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

```
private int nrings = 0;
private bool stop = false;

// The Stop property indicates whether the
// alarm should be turned off.
//
public bool Stop
{
   get {return stop;}
   set {stop = value;}
}

// The SnoozePressed property indicates whether the snooze
// button is pressed on the alarm when the alarm event is generated.
//
public bool SnoozePressed
{
   get {return snoozePressed;}
   set {snoozePressed = value;}
}
// The event member that is of type AlarmEventHandler.
//
public event AlarmEventHandler Alarm;

// The protected OnAlarm method raises the event by invoking
// the delegates. The sender is always this, the current instance
// of the class.
//
protected virtual void OnAlarm(AlarmEventArgs e)
{
  AlarmEventHandler handler = Alarm;
  if (handler != null)
  {
     // Invokes the delegates.
     handler(this, e);
  }
}

// This alarm clock does not have
// a user interface.
// To simulate the alarm mechanism it has a loop
// that raises the alarm event at every iteration
// with a time delay of 300 milliseconds,
// if snooze is not pressed. If snooze is pressed,
// the time delay is 1000 milliseconds.
//
public void Start()
{
   for (;;)
   {
      nrings++;
      if (stop)
      {
         break;
      }
```


Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

```
        else if (snoozePressed)
        {
          System.Threading.Thread.Sleep(1000);
          {
            AlarmEventArgs e = new AlarmEventArgs(snoozePressed,
              nrings);
            OnAlarm(e);
          }
        }
        else
        {
          System.Threading.Thread.Sleep(300);
          AlarmEventArgs e = new AlarmEventArgs(snoozePressed,
            nrings);
          OnAlarm(e);
        }
      }
    }
}

// The WakeMeUp class has a method AlarmRang that handles the
// alarm event.
//
public class WakeMeUp
{

  public void AlarmRang(object sender, AlarmEventArgs e)
  {

    Console.WriteLine(e.AlarmText +"\n");

    if (!(e.SnoozePressed))
    {
      if (e.NumRings % 10 == 0)
      {
        Console.WriteLine(" Let alarm ring? Enter Y");
        Console.WriteLine(" Press Snooze? Enter N");
        Console.WriteLine(" Stop Alarm? Enter Q");
        String input = Console.ReadLine();

        if (input.Equals("Y") ||input.Equals("y")) return;

        else if (input.Equals("N") || input.Equals("n"))
        {
          ((AlarmClock)sender).SnoozePressed = true;
          return;
        }
        else
        {
          ((AlarmClock)sender).Stop = true;
          return;
        }
      }
    }
```


Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

```
        else
        {
          Console.WriteLine(" Let alarm ring? Enter Y");
          Console.WriteLine(" Stop Alarm? Enter Q");
          String input = Console.ReadLine();
          if (input.Equals("Y") || input.Equals("y")) return;
          else
          {
            ((AlarmClock)sender).Stop = true;
            return;
          }
        }
      }
    }


    // The driver class that hooks up the event handling method of
    // WakeMeUp to the alarm event of an Alarm object using a delegate.
    // In a forms-based application, the driver class is the
    // form.
    //
    public class AlarmDriver
    {
      public static void Main (string[] args)
      {
        // Instantiates the event receiver.
        WakeMeUp w= new WakeMeUp();

        // Instantiates the event source.
        AlarmClock clock = new AlarmClock();

        // Wires the AlarmRang method to the Alarm event.
        clock.Alarm += new AlarmEventHandler(w.AlarmRang);

        clock.Start();
      }
    }

}
```

How to: Handle Multiple Events Using Event Properties

To use event properties (custom events in Visual Basic 2005), you define the event properties in the class that raises the events, and then set the delegates for the event properties in classes that handle the events. To implement multiple event properties in a class, the class must internally store and maintain the delegate defined for each event. A typical approach is to implement a delegate collection that is indexed by an event key.

To store the delegates for each event, you can use the EventHandlerList class, or implement your own collection. The collection class must provide methods for setting, accessing, and retrieving the event handler delegate based on the event key. For example, you could use a Hashtable class, or derive a custom class from the DictionaryBase class. The implementation details of the delegate collection do not need to be exposed outside your class.

Each event property within the class defines an add accessor method and a remove accessor method. The add accessor for an event property adds the input delegate instance to the delegate collection. The remove accessor for an event property removes the input delegate instance from the delegate collection. The event property accessors use the predefined key for the event property to add and remove instances from the delegate collection.

To handle multiple events using event properties

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

- Define a delegate collection within the class that raises the events.
- Define a key for each event.
- Define the event properties in the class that raises the events.
- Use the delegate collection to implement the add and remove accessor methods for the event properties.
- Use the public event properties to add and remove event handler delegates in the classes that handle the events.

Example
The following C# example implements the event properties MouseDown and MouseUp, using an EventHandlerList to store each event's delegate. The keywords of the event property constructs are in bold type.

```
// The class SampleControl defines two event properties, MouseUp and MouseDown.
class SampleControl: Component {
  // :
  // Define other control methods and properties.
  // :

  // Define the delegate collection.
  protected EventHandlerList listEventDelegates = new EventHandlerList();

  // Define a unique key for each event.
  static readonly object mouseDownEventKey = new object();
  static readonly object mouseUpEventKey = new object();

  // Define the MouseDown event property.
  public event MouseEventHandler MouseDown {
    // Add the input delegate to the collection.
    add { listEventDelegates.AddHandler(mouseDownEventKey, value); }
    // Remove the input delegate from the collection.
    remove { listEventDelegates.RemoveHandler(mouseDownEventKey, value); }
  }

  // Define the MouseUp event property.
  public event MouseEventHandler MouseUp {
    // Add the input delegate to the collection.
    add { listEventDelegates.AddHandler(mouseUpEventKey, value); }
    // Remove the input delegate from the collection.
    remove { listEventDelegates.RemoveHandler(mouseUpEventKey, value); }
  }
}
```

121. What's the difference between localization and globalization?

Globalization: The process of developing a program core whose features and code design are not solely based on a single language or locale. Instead, their design is developed for the input, display, and output of a defined set of Unicode-supported language scripts and data related to specific locales.
Localization: The process of adapting a program for a specific local market, which includes translating the user interface, resizing dialog boxes, customizing features (if necessary), and testing results to ensure that the program still works. You can visualize globalization as more of architecture decisions. While localization is adapting your content to local market. Localization phase occurs before globalization phase

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

122. Difference between primitive types, ref types and value types?

Primitive: Data types that can be mapped directly to types that exist in the base class library are called Primitive. Example sbyte (System.SByte)* byte (System.Byte) ,short (System.Int16) ,ushort (System.UInt16)* ,int (System.Int32) ,uint (System.UInt32)* ,long (System.Int64) ,ulong (Sysem.UInt64)* ,char (System.Char),float (System.Single) ,double (System.Double) ,bool (System.Boolean) ,decimal (System.Decimal) ,**string** (System.String) ,**object** (System.Object)

Value Types :

Value types inherit from the System.ValueType class, which in turn, inherits from System.Object. However, you can not inherit directly from the System.ValueType class. If you try to inherit explicitly from System.Value, you'll get the **C3838** compiler error. Value types have several special properties:

- Value types are stored on the stack.
- Value type objects have two representations: an unboxed form and a boxed form.
- Value types are accessed directly which means you don't need to use the new operator.
- Value types are managed types, they are initialised to 0 when they are created.
- Value type instances are not under the control of the Garbage Collector.
- Value types are implicitly sealed, which means that no class can derive from them.
- In C#, structs are always value types and allocated on the stack.
- Examples : sbyte (System.SByte)* ,byte (System.Byte) ,short (System.Int16) ,ushort (System.UInt16)* ,int (System.Int32) ,uint (System.UInt32)* ,long (System.Int64) ,ulong (Sysem.UInt64)* ,char (System.Char),float (System.Single) ,double (System.Double) ,bool (System.Boolean) ,decimal (System.Decimal) ,All variables defined from a struct data type ,All Enumerations (enum) types

Reference Type

Reference types inherit directly from System.Object, they offer many advantages over Value types:

- Reference types are stored on the managed heap, thus they are under the control of Garbage Collector.
- Two or more reference type variables can refer to a single object in the heap, allowing operations on one variable to affect the object referenced by the other variable.

The variable representing the instance contains a pointer to the instance of the class, it is dereferenced first if you want to access any of its members. In C#, Classes are always reference types and created on the managed heap.

123. Difference between gettype() and typeof.

The typeof operator is used to obtain the System.Type object for a type. A typeof expression takes the form:
typeof (type)
where: type The type for which the System.Type object is obtained.
Remarks

- The typeof operator cannot be overloaded.
- To obtain the run-time type of an expression, you can use the .NET Framework method GetType.

GetType
The **Type** instance that represents the exact runtime type of the current instance

```
MyBaseClass myBase = new MyBaseClass();
    MyDerivedClass myDerived = new MyDerivedClass();
    object o = myDerived;
    MyBaseClass b = myDerived;
```


Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

```
        Console.WriteLine("mybase: Type is {0}", myBase.GetType());
        Console.WriteLine("myDerived: Type is {0}", myDerived.GetType());
        Console.WriteLine("object o = myDerived: Type is {0}", o.GetType());
        Console.WriteLine("MyBaseClass b = myDerived: Type is {0}", b.GetType());
    /*

    This code produces the following output.

    mybase: Type is MyBaseClass
    myDerived: Type is MyDerivedClass
    object o = myDerived: Type is MyDerivedClass
    MyBaseClass b = myDerived: Type is MyDerivedClass

    */
```

124. What is Microsoft SharePoint Portal Server?

SharePoint Portal Server is a portal server that connects people, teams, and knowledge across business processes. SharePoint Portal Server integrates information from various systems into one secure solution through single sign-on and enterprise application integration capabilities. It provides flexible deployment and management tools, and facilitates end-to-end collaboration through data aggregation, organization, and searching. SharePoint Portal Server also enables users to quickly find relevant information through customization and personalization of portal content and layout as well as through audience targeting.

125. What is Microsoft Windows SharePoint Services?

Windows SharePoint Services is the solution that enables you to create Web sites for information sharing and document collaboration. Windows SharePoint Services " a key piece of the information worker infrastructure delivered in Microsoft Windows Server 2003 " provides additional functionality to the Microsoft Office system and other desktop applications, and it serves as a platform for application development. Office SharePoint Server 2007 builds on top of Windows SharePoint Services 3.0 to provide additional capabilities including collaboration, portal, search, enterprise content management, business process and forms, and business intelligence.

126. What is JQuery

JQuery is a light weight JavaScript library which provides fast and easy way of HTML DOM traversing and manipulation, its event handling, its client side animations, etc. One of the greatest features of jQuery is that jQuery supports an efficient way to implement AJAX applications because of its light weight nature and make normalize and efficient web programs.

The advantages of using jQuery are:

1. JavaScript enhancement without the overhead of learning new syntax

2. Ability to keep the code simple, clear, readable and reusable

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

3. Eradication of the requirement of writing repetitious and complex loops and DOM scripting library calls

127. What is difference between static and singleton classes?

You want to store common data that is only needed in one location, using a **singleton** or static class. Save state between usages and store some caches. The object must be initialized only once and shared. Here we discuss the singleton pattern and static classes from the C# language with examples.

Singletons can be used differently than static classes.

This design pattern can be used with interfaces.

Singletons can be used as parameters to methods.

Static classes

Used for global data. You can use static classes to store single-instance, global data. The class will usually be initialized lazily, at the last possible moment. However, you lose control over the exact behavior and static constructors are slow.

Advantages of singletons

Singletons preserve the conventional class approach, and don't require that you use the static keyword everywhere. They may be more demanding to implement at first, but will greatly simplify the architecture of your program. Unlike static classes, we can use singletons as **parameters** or objects.

=== Using singleton as parameter (C#) ===

```
//We want to call a function with this structure as an object.

// Get a reference from the Instance property on the singleton.

SiteStructure site = SiteStructure.Instance;

OtherFunction(site); // Use singleton as parameter.
```

128. Explain Advantages of WCF.

- WCF has support for multiple protocols (TCP/IP, HTTP, PIPE, MSMQ etc.).
- WCF can be hosted outside if IIS ie: can be hosted in managed windows applications, a windows service, and WAS (Windows Process Activation Service).
- WCF provides customisation such as event hooks into service start / end / init / custom global error Handling etc. etc.
- WCF services supports standard web services ws-*. wsHttpBinding
- WCF uses http endpoint for web services.
- It uses schemas and contracts instead of classes & types to define and implement communication.
- WCF services expose endpoints that clients and services use to exchange messages. Each endpoint consists of an address, a binding, and a contract.

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

- he main three elements of WCF architecture are : Address, Binding & Contract.
- Address :It specifies where service is located and format is network protocol specific e.g. http or tcp.
- Binding :It specifies transport protocol, security requirements & message encoding to be used by client & service for communication.
- Contract :It defines what the service can do. WCF service publish multiple types of contracts like service contract, message contract, data contract etc.
- Currently WCF supports three message patterns :

One way messaging : client sends a message to service without expecting a response back

Request Response : client sends a message and waits for reply

Duplex Messaging : client & service send message to each other without the synchronization as required in request – response.

Some of the bindings available are :

1. basicHttpBinding : HTTP protocol binding confirming to WS-I profile specification.

2. wsHttpBinding : confirming to WS-* protocol.

3. NetNamedPipeBinding : for connecting endpoints on same machine.

4. NetMsmqBinding : uses queued message connections.

5. NetTcpBinding : optimized binding for cross machine communication

Custom Binding can be created to handle more complex requirements.

The main features of any binding are :

1. Interoperability Type : kind of integration possible like WS, .NET, peer, MSMQ, etc.

2. Security : Level of security e.g. Transport, Message or Mixed.

3. Encoding : Type of encoding supported e.g. Text, Binary, MTOM.

SECURITY

Securing Services involves four aspects mainly :

1.Authentication: Who the client is and whether it is allowed to communicate with service.

2.Confidentiality: Encrypting communication between client and service.

3.Integrity: To make sure that the communication is tamper proof.

4.Authorization: The access or execution rights of the client with respect to the service.

The various modes of security supported in WCF to implement above requirements are :

1.Transport Mode: The underlying transport protocol like http take care of all the above requirements by default.

2.Message Mode: In this mode all the data required to satisfy above requirements flow as part of message headers.

3.Hybrid Mode: In the mode Confidentiality & Integrity requirements are taken care by Transport mode while Authentication & Authorization are implemented using Message Mode. This mode is also called 'Transport with Message Credentials'.

There are two additional modes that are specific to two bindings : the 'transport-credentials only' mode found on the BasicHttpBinding and the 'both' mode found on the NetMsmqBinding.

129. What is dependency Injection?

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

The dependency injection pattern, also knows as Inversion of Control, is one of the most popular design paradigms today. It facilitates the design and implementation of loosely coupled, reusable, and testable objects in your software designs by removing dependencies that often inhibit reuse. Dependency injection can help you design your applications so that the architecture links the components rather than the components linking themselves. Dependency injection eliminates tight coupling between objects to make both the objects and applications that use them more flexible, reusable, and easier to test. It facilitates the creation of loosely coupled objects and their dependencies. The basic idea behind Dependency Injection is that you should isolate the implementation of an object from the construction of objects on which it depends. Dependency Injection is a form of the Inversion of Control Pattern where a factory object carries the responsibility for object creation and linking. The factory object ensures loose coupling between the objects and promotes seamless testability.

Advantages and Disadvantages of Dependency Injection

The primary advantages of dependency injection are:

- Loose coupling
- Centralized configuration
- Easily testable

Code becomes more testable because it abstracts and isolates class dependencies.

However, the primary drawback of dependency injection is that wiring instances together can become a nightmare if there are too many instances and many dependencies that need to be addressed.

Types of Dependency Injection

There are three common forms of dependency injection:

1. Constructor Injection
2. Setter Injection
3. Interface-based injection

Constructor injection uses parameters to inject dependencies. In setter injection, you use setter methods to inject the object's dependencies. Finally, in interface-based injection, you design an interface to inject dependencies. The following section shows how to implement each of these dependency injection forms and discusses the pros and cons of each.

```
public class BusinessFacade
  {
    private IBusinessLogic businessLogic;
    public BusinessFacade(IBusinessLogic businessLogic)
    {
      this.businessLogic = businessLogic;
    }
  }
```

Implementing Setter Injection

Setter injection uses properties to inject the dependencies, which lets you create and use resources as late as possible. It's more flexible than constructor injection because you can use it to change the dependency of one object on another without having to create a new instance of the class or making any changes to its constructor. Further, the setters can have meaningful, self-descriptive names that simplify understanding and using them. Here's an example that adds a property to the BusinessFacade class which you can use to inject the dependency.

The following is now our BusinessFacade class with the said property.

```
  public class BusinessFacade
   {
    private IBusinessLogic businessLogic;

    public IBusinessLogic BusinessLogic
      {
       get
```


Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

```
        {
          return businessLogic;
        }

          set
        {
          businessLogic = value;
        }
      }
    }
```

The following code snippet illustrates to implement setter injection using the BusinessFacade class shown above.

```
IBusinessLogic productBL = new ProductBL();
BusinessFacade businessFacade = new BusinessFacade();
businessFacade.BusinessLogic = productBL;
```

Implementing Interface Injection
You accomplish the last type of dependency injection technique, interface injection, by using a common interface that other classes need to implement to inject dependencies. The following code shows an example in which the classes use the IBusinessLogic interface as a base contract to inject an instance of any of the business logic classes (ProductBL or CustomerBL) into the BusinessFacade class. Both the business logic classes ProductBL and CustomerBL implement the IBusinessLogic interface:

```
interface IBusinessLogic
{
  //Some code
}

class ProductBL : IBusinessLogic
{
  //Some code
}

class CustomerBL : IBusinessLogic
{
  //Some code
}

class BusinessFacade : IBusinessFacade
{
   private IBusinessLogic businessLogic;
   public void SetBLObject(IBusinessLogic businessLogic)
   {
      this.businessLogic = businessLogic;
   }
}
```

130. What is difference between STA & MTA?

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

Apartment threading is an automatic thread-safety regime, closely allied to COM – Microsoft's legacy Component Object Model. While .NET largely breaks free of legacy threading models, there are times when it still crops up because of the need to interoperate with older APIs. Apartment threading is most relevant to Windows Forms, because much of Windows Forms uses or wraps the long-standing Win32 API – complete with its apartment heritage.

An apartment is a logical "container" for threads. Apartments come in two sizes – "single" and "multi". A single-threaded apartment contains just one thread; multi-threaded apartments can contain any number of threads. The single-threaded model is the more common and interoperable of the two.

As well as containing threads, apartments contain objects. When an object is created within an apartment, it stays there all its life, forever house-bound along with the resident thread(s). This is similar to an object being contained within a .NET **synchronization context**, except that a synchronization context does not own or contain threads. Any thread can call upon an object in any synchronization context – subject to waiting for the exclusive lock. But objects contained within an apartment can only be called upon by a thread within the apartment.

Imagine a library, where each book represents an object. Borrowing is not permitted – books created in the library stay there for life. Furthermore, let's use a person to represent a thread.

A synchronization context library allows any person to enter, as long as only one person enters at a time. Any more, and a queue forms outside the library.

An apartment library has resident staff – a single librarian for a single-threaded library, and whole team for a multi-threaded library. No-one is allowed in other than members of staff – a patron wanting to perform research must signal a librarian, then ask the librarian to do the job! Signaling the librarian is called *marshalling*– the patron *marshals* the method call over to a member of staff (or, the member of staff!) Marshalling is automatic, and is implemented at the librarian-end via a message pump – in Windows Forms, this is the mechanism that constantly checks for keyboard and mouse events from the operating system. If messages arrive too quickly to be processed, they enter a message queue, so they can be processed in the order they arrive.

131. What is distributed transactions?

A distributed transaction spans multiple data sources. Distributed transactions enable you to incorporate several distinct operations, which occur on different systems, into an atomic action that either succeeds or fails completely.

Creating Distributed Transactions:

The .NET Framework 2.0 includes the *System.Transactions* namespace, which provides extensive support for distributed transactions across a range of transaction managers, including data sources and message queues.
The *System.Transactions* namespace defines the *TransactionScope* class, which enables you to create and manage distributed transactions.

To create and use distributed transactions, create a *TransactionScope* object, and specify whether you want to create a new transaction context or enlist in an existing transaction context. You can also exclude operations from a transaction context if appropriate.

You can open multiple database connections within the same transaction scope. The transaction scope decides whether to create a local transaction or a distributed transaction. The transaction scope, automatically promotes a local transaction to a distributed transaction if necessary, based on the following rules:

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

- When you create a *TransactionScope* object, it initially creates a local, lightweight transaction. Lightweight transactions are more efficient than distributed transactions because they do not incur the overhead of the Microsoft Distributed Transaction Coordinator (DTC).
- If the first connection that you open in a transaction scope is to a SQL Server 2005 database, the connection enlists in the local transaction. The resource manager for SQL Server 2005 works with the *System.Transactions* namespace and supports automatic promotion of local transactions to distributed transactions. Therefore, the transaction scope is able to defer creating a distributed transaction unless and until it becomes necessary later.
- If the first connection that you open in a transaction scope is to anything other than a SQL Server 2005 database, the transaction scope promotes the local transaction to a distributed transaction immediately. This immediate promotion occurs because the resource managers for these other databases do not support automatic promotion of local transactions to distributed transactions.
- When you open subsequent connections in the transaction scope, the transaction scope promotes the transaction to a distributed transaction, regardless of the type of the database.

Steps of creating distributed transaction:

1. Instantiate a *TransactionScope* object.
2. Open a connection with the database.
3. Perform your database operations (insert, update & delete).
4. If your operations completed successfully, mark your transaction as complete.
5. Dispose The *TransactionScope* object.

If all the update operations succeed in a transaction scope, call the *Complete* method on the *TransactionScope* object to indicate that the transaction completed successfully. To terminate a transaction, call the *Dispose* method on the *TransactionScope* object. When you dispose a *TransactionScope*, the transaction is either committed or rolled back, depending on whether you called the *Complete* method:

- If you called the *Complete* method on the *TransactionScope* object before its disposal, the transaction manager commits the transaction.
- If you did not call the *Complete* method on the *TransactionScope* object before its disposal, the transaction manager rolls back the transaction.

TransactionScope How To:

A transaction scope defines a block of code that participates in a transaction. If the code block completes successfully, the transaction manager commits the transaction. Otherwise, the transaction manager rolls back the transaction. Bellow is a guide lines on how you can create and use a *TransactionScope* instance:

1. Add a reference to the *System.Transactions* assembly.
2. Import the *System.Transactions* namespace into your application.
3. If you want to specify the nesting behavior for the transaction, declare a *TransactionScopeOption* variable, and assign it a suitable value.
4. If you want to specify the isolation level or timeout for the transaction, create a *TransactionOptions* instance, and set its *IsolationLevel* and *TimeOut*properties.
5. Create a new *TransactionScope* object in a *using statement* (a *Using statement in Microsoft Visual Basic*). Pass a *TransactionScopeOption* variable and a *TransactionOptions* object into the constructor, if appropriate.
6. In the using block (Using block in Visual Basic), open connections to each database that you need to update, and perform update operations as required by your application. If all updates succeed, call the *Complete* method on the *TransactionScope* object
7. Close the using block (Using block in Visual Basic) to dispose the *TransactionScope* object. If the transaction completed successfully, the transaction manager commits the transaction. Otherwise, the transaction manager rolls back the transaction.

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

Listing 1: TransactionScope C#:

```
using System.Transactions;
...
TransactionOptions options = new TransactionOptions();
options.IsolationLevel = System.Transactions.IsolationLevel.ReadCommitted;
options.Timeout = new TimeSpan(0, 2, 0);
using (TransactionScope scope = new TransactionScope(TransactionScopeOption.Required, options))
{
        using (SqlConnection connA = new SqlConnection(connStringA))
          {
                using (SqlCommand cmdA = new SqlCommand(sqlStmtA, connA))
                {
                        int rowsAffectedA = cmdA.ExecuteNonQuery();
                        if (rowsAffectedA > 0)
                        {
                                using (SqlConnection connB = new SqlConnection(connStringB))
                                {
                                        using (SqlCommand cmdB = new SqlCommand(sqlStmtB, connB))
                                        {
                                                int rowsAffectedB = cmdB.ExecuteNonQuery();
                                                if (rowsAffectedB > 0)
                                                {
                                                        transactionScope.Complete();
                                                }
                                        } // Dispose the second command object.
                                } // Dispose (close) the second connection.
                        }
                } // Dispose the first command object.
        } // Dispose (close) the first connection.
} // Dispose TransactionScope object, to commit or rollback transaction.
```

132. How does a database index work?

An index

- is sorted by key values, (that need not be the same as those of the table)
- is small, has just a few columns of the table.
- refers for a key value to the right block within the table.
- speeds up reading a row, when you know the right search arguments.

Database Index Tips

- Put the most unique data element first in the index, the element that has the biggest veriety of values. The index will find the correct page faster.
- Keep indexes small. It's better to have an index on just zip code or postal code, rather than postal code & country. The smaller the index, the better the response time.
- For high frequency functions (thousands of times per day) it can be wise to have a very large index, so the system does not even need the table for the read function.
- For small tables an index is disadvantageous. For any function the system would be better off by scanning the whole table. An index would only slow down.

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

- Index note:
- An index slows down additions, modifications and deletes. It's not just the table that needs an update, but the index as well. So, preferably, add an index for values that are often used for a search, but that do not change much. An index on bank account number is better than one on balance.

133. Give some threading best practices?

Multithreading requires careful programming. For most tasks, you can reduce complexity by queuing requests for execution by thread pool threads. This topic addresses more difficult situations, such as coordinating the work of multiple threads, or handling threads that block.

Deadlocks and Race Conditions

Multithreading solves problems with throughput and responsiveness, but in doing so it introduces new problems: deadlocks and race conditions.

Deadlocks

A deadlock occurs when each of two threads tries to lock a resource the other has already locked. Neither thread can make any further progress.

Many methods of the managed threading classes provide time-outs to help you detect deadlocks. For example, the following code attempts to acquire a lock on the current instance. If the lock is not obtained in 300 milliseconds, Monitor.TryEnter returns false.

```
if (Monitor.TryEnter(this, 300)) {
    try {
        // Place code protected by the Monitor here.
    }
    finally {
        Monitor.Exit(this);
    }
}
else {
    // Code to execute if the attempt times out.
}
```


Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

Race Conditions

A race condition is a bug that occurs when the outcome of a program depends on which of two or more threads reaches a particular block of code first. Running the program many times produces different results, and the result of any given run cannot be predicted.

A simple example of a race condition is incrementing a field. Suppose a class has a private static field (Shared in Visual Basic) that is incremented every time an instance of the class is created, using code such as objCt++; (C#) or objCt += 1 (Visual Basic). This operation requires loading the value from objCt into a register, incrementing the value, and storing it in objCt.

In a multithreaded application, a thread that has loaded and incremented the value might be preempted by another thread which performs all three steps; when the first thread resumes execution and stores its value, it overwrites objCt without taking into account the fact that the value has changed in the interim.

This particular race condition is easily avoided by using methods of the Interlocked class, such as Interlocked.Increment. To read about other techniques for synchronizing data among multiple threads, see Synchronizing Data for Multithreading.

Race conditions can also occur when you synchronize the activities of multiple threads. Whenever you write a line of code, you must consider what might happen if a thread were preempted before executing the line (or before any of the individual machine instructions that make up the line), and another thread overtook it.

Number of Processors

Multithreading solves different problems for the single-processor computers that run most end-user software, and the multiprocessor computers typically used as servers.

Single-Processor Computers

Multithreading provides greater responsiveness to the computer user, and uses idle time for background tasks. If you use multithreading on a single-processor computer:

Only one thread runs at any instant.

A background thread executes only when the main user thread is idle. A foreground thread that executes constantly starves background threads of processor time.

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

When you call the Thread.Start method on a thread, that thread does not start executing until the current thread yields or is preempted by the operating system.

Race conditions typically occur because the programmer did not anticipate the fact that a thread can be preempted at an awkward moment, sometimes allowing another thread to reach a code block first.

Multiprocessor Computers

Multithreading provides greater throughput. Ten processors can do ten times the work of one, but only if the work is divided so that all ten can be working at once; threads provide an easy way to divide the work and exploit the extra processing power. If you use multithreading on a multiprocessor computer:

The number of threads that can execute concurrently is limited by the number of processors.

A background thread executes only when the number of foreground threads executing is smaller than the number of processors.

When you call the Thread.Start method on a thread, that thread might or might not start executing immediately, depending on the number of processors and the number of threads currently waiting to execute.

Race conditions can occur not only because threads are preempted unexpectedly, but because two threads executing on different processors might be racing to reach the same code block.

Static Members and Static Constructors

A class is not initialized until its class constructor (static constructor in C#, Shared Sub New in Visual Basic) has finished running. To prevent the execution of code on a type that is not initialized, the common language runtime blocks all calls from other threads to static members of the class (Shared members in Visual Basic) until the class constructor has finished running.

For example, if a class constructor starts a new thread, and the thread procedure calls a static member of the class, the new thread blocks until the class constructor completes.

This applies to any type that can have a static constructor.

General Recommendations

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

Consider the following guidelines when using multiple threads:

Don't use Thread.Abort to terminate other threads. Calling Abort on another thread is akin to throwing an exception on that thread, without knowing what point that thread has reached in its processing.

Don't use Thread.Suspend and Thread.Resume to synchronize the activities of multiple threads. Do use Mutex, ManualResetEvent, AutoResetEvent, and Monitor.

Don't control the execution of worker threads from your main program (using events, for example). Instead, design your program so that worker threads are responsible for waiting until work is available, executing it, and notifying other parts of your program when finished. If your worker threads do not block, consider using thread pool threads. Monitor.PulseAll is useful in situations where worker threads block.

Don't use types as lock objects. That is, avoid code such as lock(typeof(X)) in C# or SyncLock(GetType(X)) in Visual Basic, or the use of Monitor.Enter with Type objects. For a given type, there is only one instance of System.Type per application domain. If the type you take a lock on is public, code other than your own can take locks on it, leading to deadlocks. For additional issues, see Reliability Best Practices.

Use caution when locking on instances, for example lock(this) in C# or SyncLock(Me) in Visual Basic. If other code in your application, external to the type, takes a lock on the object, deadlocks could occur.

Do ensure that a thread that has entered a monitor always leaves that monitor, even if an exception occurs while the thread is in the monitor. The C# lock statement and the Visual Basic SyncLock statement provide this behavior automatically, employing a finally block to ensure that Monitor.Exit is called. If you cannot ensure that Exit will be called, consider changing your design to use Mutex. A mutex is automatically released when the thread that currently owns it terminates.

Do use multiple threads for tasks that require different resources, and avoid assigning multiple threads to a single resource. For example, any task involving I/O benefits from having its own thread, because that thread will block during I/O operations and thus allow other threads to execute. User input is another resource that benefits from a dedicated thread. On a single-processor computer, a task that involves intensive computation coexists with user input and with tasks that involve I/O, but multiple computation-intensive tasks contend with each other.

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

Consider using methods of the Interlocked class for simple state changes, instead of using the lock statement (SyncLock in Visual Basic). The lock statement is a good general-purpose tool, but the Interlocked class provides better performance for updates that must be atomic. Internally, it executes a single lock prefix if there is no contention. In code reviews, watch for code like that shown in the following examples. In the first example, a state variable is incremented:

```
lock(lockObject)
{
   myField++;
}
```

You can improve performance by using the Increment method instead of the lock statement, as follows:

```
System.Threading.Interlocked.Increment(myField);
```

In the second example, a reference type variable is updated only if it is a null reference (Nothing in Visual Basic).

```
if (x == null)
{
   lock (lockObject)
   {
      if (x == null)
      {
         x = y;
      }
   }
}
```

Performance can be improved by using the CompareExchange method instead, as follows:

```
System.Threading.Interlocked.CompareExchange(ref x, y, null);
```

Recommendations for Class Libraries

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

Consider the following guidelines when designing class libraries for multithreading:

Avoid the need for synchronization, if possible. This is especially true for heavily used code. For example, an algorithm might be adjusted to tolerate a race condition rather than eliminate it. Unnecessary synchronization decreases performance and creates the possibility of deadlocks and race conditions.

Make static data (Shared in Visual Basic) thread safe by default.

Do not make instance data thread safe by default. Adding locks to create thread-safe code decreases performance, increases lock contention, and creates the possibility for deadlocks to occur. In common application models, only one thread at a time executes user code, which minimizes the need for thread safety. For this reason, the .NET Framework class libraries are not thread safe by default.

Avoid providing static methods that alter static state. In common server scenarios, static state is shared across requests, which means multiple threads can execute that code at the same time. This opens up the possibility of threading bugs. Consider using a design pattern that encapsulates data into instances that are not shared across requests. Furthermore, if static data are synchronized, calls between static methods that alter state can result in deadlocks or redundant synchronization, adversely affecting performance.

134. What is object pooling?

Object Pooling is something that tries to keep a pool of objects in memory to be re-used later and hence it will reduce the load of object creation to a great extent. The example is for an Employee object, but you can make it general by using Object base class.

We are going to use Factory pattern for this purpose. We will have a factory method, which will take care about the creation of objects. Whenever there is a request for a new object, the factory method will look into the object pool (we use Queue object). If there is any object available within the allowed limit, it will return the object (value object), otherwise a new object will be created and give you back.

The below code is just an example to give you an idea, and is neither tested nor error-proof. You can modify it as you wish: up-to your creativity.

Code 1: Object Pool and Employee class.

```
using System;
```


Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

```
using System.Collections;

namespace ObjectPooling
{
   class Factory
   {
      // Maximum objects allowed!
      private static int _PoolMaxSize = 2;

      // My Collection Pool
      private static readonly Queue objPool = new Queue(_PoolMaxSize);

      public Employee GetEmployee()
      {
         Employee oEmployee;

         // check from the collection pool. If exists return object else create new
         if (Employee.Counter >= _PoolMaxSize && objPool.Count>0)
         {
            // Retrieve from pool
            oEmployee = RetrieveFromPool();
         }
         else
         {
            oEmployee = GetNewEmployee();
         }
         return oEmployee;
      }

      private Employee GetNewEmployee()
      {
         // Creates a new employee
```


Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

```
        Employee oEmp = new Employee();
        objPool.Enqueue(oEmp);
        return oEmp;
    }

    protected Employee RetrieveFromPool()
    {
        Employee oEmp;

        // if there is any objects in my collection
        if (objPool.Count>0)
        {
            oEmp = (Employee)objPool.Dequeue();
            Employee.Counter--;
        }
        else
        {
            // return a new object
            oEmp = new Employee();
        }
        return oEmp;
    }
}

class Employee
{
    public static int Counter = 0;
    public Employee()
    {
        ++Counter;
    }
```


Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

```
        private string _Firstname;
        public string Firstname
        {
            get
            {
                return _Firstname;
            }
            set
            {
                _Firstname = value;
            }
        }
    }
}
```

Code 2: How to use it?

```
private void button1_Click(object sender, EventArgs e)
{
    Factory fa = new Factory();

    Employee myEmp = fa.GetEmployee();
    Console.WriteLine("First object");

    Employee myEmp1 = fa.GetEmployee();
    Console.WriteLine("Second object")
}
```

135. Static and Dynamic Assembly.

Assemblies can be static or dynamic. Static assemblies can include .NET Framework types (interfaces and classes), as well as resources for the assembly (bitmaps, JPEG files, resource files, and so on). Static assemblies are stored on disk in PE files. You can also use the .NET Framework to

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

create dynamic assemblies, which are run directly from memory and are not saved to disk before execution. You can save dynamic assemblies to disk after they have executed. You can create dynamic assembly using System.Reflection.Emit namespace.

136. Why we have to consider object in lock statement as private?
Please send your answer to neeraj.kaushik@live.in

137. How can we make class immutable? Give Example
Please send your answer to neeraj.kaushik@live.in

138. How to override Equal, GetHashCode?
Please send your answer to neeraj.kaushik@live.in

139. What is SSL and how can we implement it in our asp.net website?
Please send your answer to neeraj.kaushik@live.in

140. What is difference between runtime polymorphism and compile time polymorphism?
Please send your answer to neeraj.kaushik@live.in

141. What is difference between real proxy and transparent proxy?
Please send your answer to neeraj.kaushik@live.in

142. What is prologue code?
Please send your answer to neeraj.kaushik@live.in

143. Explain string class or what is behind the scene of string?
Please send your answer to neeraj.kaushik@live.in

144. What is Encapsulation, polymorphism and abstraction? Give real time examples?
Please send your answer to neeraj.kaushik@live.in

145. Why we use command object?
Please send your answer to neeraj.kaushik@live.in

146. What is Satellite Assembly? How application knows which assembly should be loaded for particular culture?

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.

Please send your answer to neeraj.kaushik@live.in

147. Http request and response life cycle in ASP.Net?

Please send your answer to neeraj.kaushik@live.in

Disclaimer: Some of answers and code examples are from different sources and may be incomplete.